Lecture Notes in Computer Science 1628

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Rachid Guerraoui (Ed.)

ECOOP '99 – Object-Oriented Programming

13th European Conference
Lisbon, Portugal, June 14-18, 1999
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Rachid Guerraoui
Swiss Federal Institute of Technology
Computer Science Department
CH-1015 Lausanne, Switzerland
E-mail: Rachid.Guerraoui@epfl.ch

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Object-oriented programming : 13th European conference ; proceedings / ECOOP '99, Lisbon, Portugal, June 14 - 18, 1999. Rachid Guerraoui (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1628)
ISBN 3-540-66156-5

CR Subject Classification (1998): D.1-3, H.2, F.3, C.2, K.4

ISSN 0302-9743
ISBN 3-540-66156-5 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1999
Printed in Germany

Typesetting: Camera-ready by author
SPIN 10703317 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

"My tailor is Object-Oriented". Most software systems that have been built recently are claimed to be Object-Oriented. Even older software systems that are still in commercial use have been upgraded with some OO flavors. The range of areas where OO can be viewed as a "must-have" feature seems to be as large as the number of fields in computer science. If we stick to one of the original views of OO, that is, to create cost-effective software solutions through modeling physical abstractions, the application of OO to any field of computer science does indeed make sense. There are OO programming languages, OO operating systems, OO databases, OO specifications, OO methodologies, etc. So what does a conference on Object-Oriented Programming really mean? I honestly don't know. What I do know is that, since its creation in 1987, ECOOP has been attracting a large number of contributions, and ECOOP conferences have ended up with high-quality technical programs, featuring interesting mixtures of theory and practice.

Among the 183 initial submissions to ECOOP'99, 20 papers were selected for inclusion in the technical program of the conference. Every paper was reviewed by three to five referees. The selection of papers was carried out during a two-day program committee meeting at the Swiss Federal Institute of Technology in Lausanne. Papers were judged according to their originality, presentation quality, and relevance to the conference topics. The accepted papers cover various subjects such as programming languages, types, distribution, and formal specifications.

Beside the 20 papers selected by the program committee, this volume also contains four invited papers. Three of the invited papers are from ECOOP'99 keynote speakers, C.A.R. Hoare, B. Liskov, and J. Waldo. The fourth paper is from the banquet speaker at ECOOP'98: A.P. Black.

I would like to express my deepest appreciation to the authors of submitted papers, the program committee members, the external referees, Romain Boichat for smoothly handling the paper printing process, and Richard van de Stadt for his excellent job in managing the electronic submissions of papers and reviews. I would also like to thank the numerous people who have been involved in the organization of ECOOP'99 and, in particular, the General Organizing Chair, Vasco Vasconcelos, the Tutorial Chair, Rui Oliveira, the Workshop Chair, Ana Moreira, the Panel Chair, Luís Caires, the Demonstration Chair, António Rito Silva, the Poster Chair, Carlos Baquero, the Exhibit Chair, Mário J. Silva, and all the volunteers for their tremendous work.

April 1999

Rachid Guerraoui
ECOOP'99 Program Chair

Organization

ECOOP'99 was organized by the Department of Computer Science of the University of Lisbon, under the auspices of AITO (Association Internationale pour les Technologies Objets).

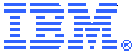
Executive Committee

Organizing Chair:	Vasco T. Vasconcelos (Universidade de Lisboa)
Program Chair:	Rachid Guerraoui (Swiss Federal Institute of Technology)
Tutorials:	Rui Oliveira (Universidade do Minho)
Workshops:	Ana Moreira (Universidade Nova de Lisboa)
Panels:	Luís Caires (Universidade Nova de Lisboa)
Demonstrations:	António Rito Silva (INESC/IST)
Posters:	Carlos Baquero (Universidade do Minho)
Exhibits:	Mário J. Silva (Universidade de Lisboa)
Submission Site:	Richard van de Stadt (University of Twente)

Sponsoring Institutions and Companies

AITO (Association Internationale pour les Technologies Objets)
http://iamwww.unibe.ch/ECOOP/AITO/

Air Portugal
http://www.tap.pt/en/index1.html



IBM
http://www.ibm.com/

Program Committee

Mehmet Akşit (University of Twente)
Suad Alagić (Wichita State University)
Paulo Sérgio Almeida (Universidade do Minho)
Elisa Bertino (University of Milan)
Jean-Pierre Briot (Laboratoire d'Informatique de Paris 6)
Alberto Coen-Porisini (Politecnico di Milano)
Pierre Cointe (Ecole des Mines de Nantes)
Klaus Dittrich (University of Zurich)
Erich Gamma (Object Technology International)
Yossi Gil (Technion, Haifa)
Rachid Guerraoui (Swiss Federal Institute of Technology)
Gorel Hedin (Lund University)
Kohei Honda (Queen Mary and Westfield College)
Mehdi Jazayeri (Vienna University of Technology)
Eric Jul (University of Copenhagen)
Karl Lieberherr (Northeastern University)
Jørgen Lindskov Knudsen (University of Aarhus)
Klaus-Peter Löhr (University of Berlin)
Cristina Lopes (Xerox Palo Alto Research Center)
Satoshi Matsuoka (Tokyo Institute of Technology)
Hanspeter Mössenböck (University of Linz)
Oscar Nierstrasz (University of Berne)
Linda Northrop (Carnegie Mellon University)
Jens Palsberg (Purdue University)
Markku Sakkinen (University of Jyväskylä)
Santosh Shrivastava (University of Newcastle)
Clemens Szyperski (Queensland University of Technology)

Referees

Franz Achermann
Ulf Asklund
Isabelle Attali
Dave Baken
Carlos Baquero
Luciano Baresi
Gilles Barthe
Andreas Behm
Klaas van den Berg
Luis Blando
Gűnther Blaschek
Mario A. Bochicchio
Boris Bokowski
Lars Bratthall
Mathias Braux
Pim van den Broek
Gerald Brose
Kim Bruce
M.C. Little
Massimo Cafaro
Luca Cardelli
Juan Carlos Cruz
Denis Caromel
Giuseppe Castagna
Silvana Castano
S.J. Caughey
Steve Caughey
Craig Chambers
Shigeru Chiba
Philippe Codognet
Jean-Louis Colaço
Aino Cornils
Gianpaolo Cugola
Markus Dahm
Jean-Daniel Fekete
Philippe Darche
Serge Demeyer
Ruxandra Domenig
Anne Doucet
Rémi Douence
Karel Driesen
Sophia Drossopoulou
Stéphane Ducasse
Roland Ducournau
Andrew Duncan
Susan Eisenbach
Marc Evers
Matthias Felleisen
Luís Ferreira Pires
Rémy Foisel
Bertil Folliot
Marko Forsell
Hans Fritschi
Svend Frolund
Nobuhisa Fujinami
Jean-Marc Geib
Andreas Geppert
David Gitchell
Giovanna Guerrini
Zahia Guessoum
Christian Heide Damm
Roger Henriksson
Chris Ho-Stuart
Urs Hölzle
Markus Hof
Atsushi Igarashi
Anders Ive
Jean-Marc Jézéquel
Dirk Jonscher
Gerti Kappel
Wayne Kelly
Gregor Kiczales
Yechiel Kimchi
Graham Kirby
Masaru Kitsuregawa
Fabrice Kordon
Kai Koskimies
Svetlana Kouznetsova
Kresten Krab Thorup
Phillip Kutter
John Lamping
Doug Lea
Gary Leavens
Thomas Ledoux
Ole Lehrmann Madsen
Mauri Leppänen
Xavier Leroy
M.C. Little

Reed Little
David Lorenz
Munenori Maeda
Eva Magnusson
Brahim Mammass
Dino Mandrioli
Klaus Marius Hansen
Hidehiko Masuhara
Kai-Uwe Mắtzel
Walter Merlat
Isabella Merlo
Mira Mezini
Philippe Mulet
Amedeo Napoli
Susumu Nishimura
José Nuno Oliveira
Martin Odersky
Lennart Ohlsson
Rui Oliveira
Tamiya Onodera
José Orlando Pereira
Alessandro Orso
Johan Ovlinger
Marc Pantel
Francois Pennaneach
Jean-François Perrot
Jonas Persson
Patrik Persson
Dick Quartel
Christian Queinnec
Stefan Rausch-Schott
Arend Rensink
Werner Retschitzegger
Tamar Richner
Dženan Ridjanović
Matthias Rieger
Dirk Riehle
Helena Rodrigues
Paul Roe
Henrik Røn
Houari A. Sahraoui
Johannes Sametinger
Prahladavaradan Sampath
Jean-Guy Schneider
Martin Schönhoff
Marten van Sinderen
Jonas Skeppstedt
André Spiegel
Martin Steffen
Christoph Steindl
Mario Südholt
Peter Sweeney
Toshiyuki Takahashi
Jean-Pierre Talpin
Bedir Tekinerdoğan
Josef Templ
Michael Thomsen
Sander Tichelaar
Dimitrios Tombros
Mads Torgersen
Anca Vaduva
Vasco Vasconcelos
Athanasios Vavouras
Jari Veijalainen
Juha Vihavainen
Jan Vitek
Ken Wakita
Dan Wallach
Andre Weinand
Stuart Wheater
Mikal Ziane
Job Zwiers

Contents

Invited Paper 1

A Trace Model for Pointers and Objects 1
C.A.R Hoare (Oxford University)
J. He (United Nations University)

Mixins

Synthesizing Objects 18
Krzysztof Czarnecki (DaimlerChrysler AG)
Ulrich W. Eisenecker (University of Applied Sciences, Heidelberg)

A Core Calculus of Classes and Mixins 43
Viviana Bono (University of Torino)
Amit Patel and Vitaly Shmatikov (Stanford University)

Propagating Class and Method Combination 67
Erik Ernst (University of Aarhus)

Debugging and Garbage Collection

A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks ... 92
Sylvia Dieckman and Urs Hölzle (University of California, Santa Barbara)

Visualizing Reference Patterns for Solving Memory Leaks in Java 116
Wim De Pauw and Gary Sevitski (IBM T.J. Watson Research Center)

Dynamic Query-Based Debugging 135
Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh (University of California, Santa Barbara)

Type Checking

Foundations for Virtual Types 161
Atsushi Igarashi and Benjamin C. Pierce (University of Pennsylvania)

Unifying Genericity 186
Kresten Krab Thorup and Mads Torgersen (University of Aarhus)

An Object-Oriented Effects System 205
Aaron Greenhouse (Carnegie Mellon University)
John Boyland (University of Wisconsin-Milwaukee)

Invited Paper 2

Providing Persistent Objects in Distributed Systems 230
Barbara Liskov, Miguel Castro, Liuba Shrira and Atul Adya (Massachusetts Institute of Technology)

Virtual and Multi-methods

Inlining of Virtual Methods 258
David Detlefs and Ole Agesen (Sun Microsystems Laboratories)

Modular Statically Typed Multimethods 279
Todd Millstein and Craig Chambers (University of Washington)

Multi-method Dispatch Using Multiple Row Displacement 304
Candy Pang, Wade Holst, Yuri Leontiev and Duane Szafron (University of Alberta)

Adaptive Programming

Internal Iteration Externalized 329
Thomas Kühne (Staffordshire University)

Type-Safe Delegation for Run-Time Component Adaptation 351
Günter Kniesel (University of Bonn)

Towards Automatic Specialization of Java Programs 367
Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel and Gilles Muller (IRISA, Rennes)

Classification and Inheritance

Wide Classes 391
Manuel Serrano (University of Nice Sophia-Antipolis)

An Approach to Classify Semi-structured Objects 416
Elisa Bertino (University of Milan)
Giovanna Guerrini and Isabella Merlo (University of Genova)
Marco Mesiti (Bell Communications Research)

Invited Paper 3

Object-Oriented Programming on the Network ... 441
Jim Waldo (Sun Microsystems, Inc.)

Distributed Objects

Providing Fine-Grained Access Control for Java Programs ... 449
Raju Pandey and Brant Hashii (University of California, Davis)

Formal Specification and Prototyping of CORBA Systems ... 474
Rémi Bastide, Ousmane Sy and Philippe Palanque (University of Toulouse)

A Process Algebraic Specification of the New Asynchronous CORBA Messaging Service ... 495
Mauro Gaspari and Gianluigi Zavattaro (University of Bologna)

Invited Paper 4

Object-Oriented Programming: Regaining the Excitement ... 519
Andrew P. Black (Oregon Graduate Institute of Science & Technology)

Author Index ... 529

A Trace Model for Pointers and Objects

C.A.R. Hoare[1] and He Jifeng[2]

[1] Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD
tony.hoare@comlab.ox.ac.uk
[2] United Nations University
International Institute for Software Technology
P.O. Box 3058, Macau
jifeng@iist.unu.edu

Abstract. Object-oriented programs [Dahl, Goldberg, Meyer] are notoriously prone to the following kinds of error, which could lead to increasingly severe problems in the presence of tasking

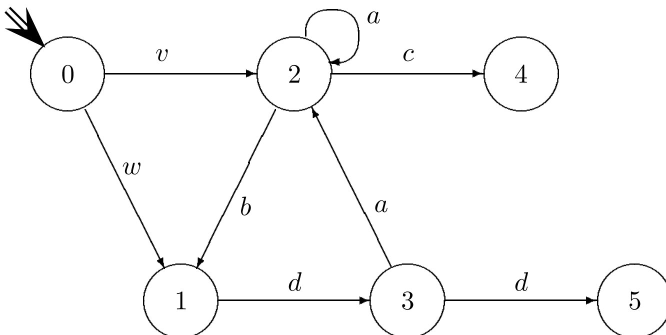
1. Following a null pointer
2. Deletion of an accessible object
3. Failure to delete an inaccessible object
4. Interference due to equality of pointers
5. Inhibition of optimisation due to fear of (4)

Type disciplines and object classes are a great help in avoiding these errors. Stronger protection may be obtainable with the help of assertions, particularly invariants, which are intended to be true before and after each call of a method that updates the structure of the heap. This note introduces a mathematical model and language for the formulation of assertions about objects and pointers, and suggests that a graphical calculus [Curtis, Lowe] may help in reasoning about program correctness. It deals with both garbage-collected heaps and the other kind. The theory is based on a trace model of graphs, using ideas from process algebra; and our development seeks to exploit this analogy as a unifying principle.

1 Introduction: The Graph Model

Figure 1.0 shows a rooted edge-labelled graph. Its **nodes** are represented by circles and its **edges** by arrows from one node to another. The letter drawn next to each arrow is its **label**. The set of allowed labels is called the **alphabet** of the graph. A double-shafted arrow singles out a particular node as the **root** of the graph.

Figure 1.0 (Rooted edge-labelled graph)



□

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 1–18, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Such a graph can be defined less graphically as a tuple

$$G = (A_G, N_G, E_G, \mathrm{root}_G),$$

where A_G is the alphabet of labels
N_G is the set of nodes
E_G is the set of edges with their labels, i.e., a subset of $N_G \times A_G \times N_G$
root_G is the node selected as the root

We use variables G, G' to stand for graphs, l, m, n to stand for nodes, x, y, z to stand for general labels and s, t, u to stand for sequences of labels (traces). We write $l \xrightarrow[G]{x} m$ to mean $(l, x, m) \in E_G$. Where only one graph is in question, we omit the subscript G. The smallest graph (called 0_A) with given alphabet A consists only of the root node, with no edges, i.e. $(A, \{\mathrm{root}\}, \{\}, \mathrm{root})$. Another small but interesting graph is $1_A =_{df} \{(A, \{\mathrm{root}\}, (\{\mathrm{root}\} \times A \times \{\mathrm{root}\}), \mathrm{root})$.

Example 1.1 (Tuple from graph) The graph of Figure 1.0 is coded as the mathematical structure defined by the following equations

$$\begin{aligned} A &= \{v, w, a, b, c, d\} \\ N &= \{0, 1, 2, 3, 4, 5\} \\ E &= \{(0, v, 2), (0, w, 1), (2, a, 2), (2, b, 1), (1, d, 3), (2, c, 4), (3, a, 2), (3, d, 5)\} \\ \mathrm{root} &= 0 \end{aligned}$$

□

The awkward feature of this encoding is the arbitrary selection of the first six natural numbers to serve as the nodes. Any other six distinct values would have done just as well. We are only interested in properties of graphs that are preserved by one-one transformations (isomorphisms) of the node-set. The use of isomorphism in place of mathematical equality is an inconvenience. We aim to avoid it by constructing a canonical representation for the nodes of a graph. For this, we will have to restrict the theory to graphs satisfying certain healthiness conditions.

Rooted edge-labelled graphs are useful in the study of many branches of computing science, of which data diagrams and heap storage are relevant to object-oriented programming.

Example 1.2 (Automata theory) A graph defines the behaviour of an automaton. The nodes stand for states, with the root as the initial state. The labels stand for events, and the presence in E of an edge $l \xrightarrow{x} m$ means that event x happens as the automaton passes from state l to state m. □

Example 1.3 (Data diagrams) In a data diagram, a node stands for a set of values, e.g., a type or a class of objects. The labels stand for functions, and the presence of an edge $l \xrightarrow{x} m$ means that x maps values of type l to results of type m. The root is somewhat artificial: the labels on arrows leading from the root can be regarded as the names of the types that they point to. □

Example 1.4 (Control flow) In a control flow graph, the nodes represent basic blocks, i.e sections of program code with no internal label. The edge $l \xrightarrow{x} m$ represents the presence in block l of a jump to a label x which is placed at the beginning of block m. The root is the main block of the program. The same analysis applies when the jumps are procedure calls and the nodes are procedure bodies. □

Example 1.5 (Heap storage) A graph can describe the instantaneous content of the entire heap at a particular point in the execution of an object-oriented program. The nodes stand for the objects, and the labels are the names for the attributes. An edge $l \xrightarrow{x} m$ means that m is the value of the x-attribute of the object l. □

When used to model objects and heaps, the labelled graph is both simple and general, in that it allows more complex concerns to be treated separately. For example,

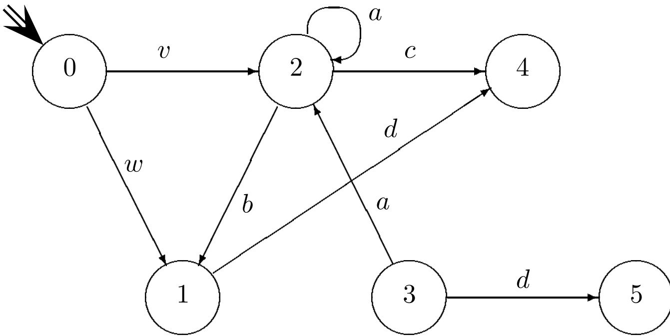
1. Simple values (e.g., like 5, which is printable) can be treated in the usual way as sinks of the graph, i.e. as nodes from which no pointer can ever point. A method local to an object can be similarly represented as a value of one of its attributes.
2. The labels on pointers from the unique root represent the directly accessible program variables. There is no restriction on pointing from the heap into declared program workspace; such pointers are often used in legacy code for cyclic representations of chains, even if their use is deprecated or forbidden in higher level languages.
3. Absence of a pointer from an object in which space has been allocated for it is often represented by filling the space with a **nil** value. The model allows this; another representation permitted by our model is to introduce a special **nil** object, with special properties, e.g. all arrows from it lead back to itself.
4. The model describes the statics and dynamics of object storage, and is quite independent of the class declarations and inheritance structure of the source language in which a program has been written. In fact, the relationship between the run-time heap and a data diagram is a special case of an invariant assertion, that remains true throughout the execution of the program. The invariant is elegantly formalised with the aid of graph homomorphisms, as described in Definition 1.10.

The main operation for updating the value of the heap is written $l \rightarrow a := m$. It causes the a-labelled arrow whose tail rests at node l to point to node m, instead of what it pointed to before. The operation changes only the edges of the graph, leaving the nodes, the alphabet, and the root unchanged.

Definition 1.6 (Pointer swing)

$$(l \rightarrow a := m) \;:=_{df}\; (E := (E - \{l\} \times \{a\} \times N) \cup \{(l, a, m)\}),\ \text{where}\, l, m \in N \square$$

Example 1.7 (Pointer swing) After execution of $1 \rightarrow d := 4$, the graph of Figure 1.0 would appear, as follows



□

Further operations are needed for deleting an edge and for creating a new node. Node creation introduces an arbitrary new object into the node-set and swings a pointer to point to it. Deletion of a node is more problematic, and will be treated later.

Definition 1.8 (Edge deletion, Node creation)

$$(l \rightarrow a := \mathbf{nil}) \;\; =_{df} \; E := E - \{l\} \times \{a\} \times N$$
$$(l \rightarrow a := \mathbf{new}) =_{df} N := N + \{m\};\; l \rightarrow a := m, \text{ where } m \;\tilde{\in}\; N$$

□

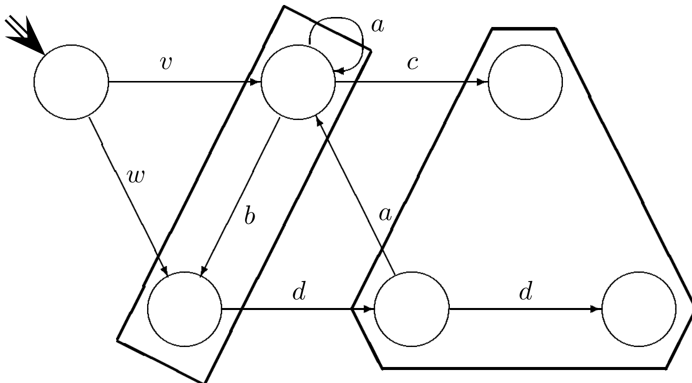
There are two problems with the above definitions of operations on the heap. The first is that an object-oriented program has no means of directly naming the objects l and m. These references have to be made indirectly by quoting the sequence of labels on a path which leads from the root to the desired object. Thus the assignment in Example 1.7 might have been written

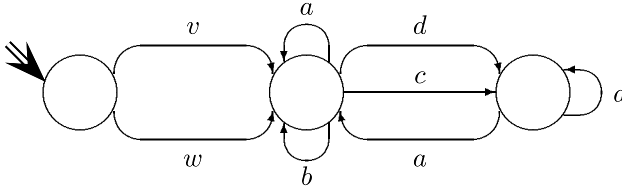
$$w \rightarrow d \;:=\; w \rightarrow d \rightarrow a \rightarrow c$$

The second problem is that, after the assignment, two of the nodes (3 and 5) have become inaccessible: the program will never again be able to refer to those nodes by any path. In a garbage-collected heap, such nodes are subject to disappearance at any time. In a non-collected heap they could represent a storage leak. Our trace model of object-orientation will solve all these problems, with the help of a canonical representation of the graph.

When a graph is used as a data diagram it specifies the classes of object to which each variable and attribute is allowed to point. A compiler can therefore allocate to each object only just enough store to hold all its permitted attributes. The compiler will also check all the operations of a program to ensure that all the rules have been observed. As a result, at all times during execution it is possible to ascribe each object in the heap to a node in the data diagram representing the object class to which it belongs. This can be pictured by drawing a polygon around all nodes belonging to the same particular class. Each polygon is then contracted to a single node, dragging the heads and tails of the arrows with it. The result will be a data diagram, which will match the intended structure of class declarations.

Figure 1.9 (Object classes)





□

The informal description of the transformation of a heap structure to a class diagram is formalised in the mathematical definition of a homomorphism. This is a function from the nodes of one graph to the nodes of another that preserves the root and the labels on the edges.

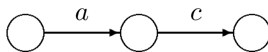
Definition 1.10 (Homomorphism) Let $G = (A, N, E, \text{root})$ and $G' = (A', N', E',$ $\text{root}')$. Let f be a total function from N to N'. The triple $f : G \to G'$ is called a homomorphism if $A \subseteq A'$, and for all x in A

1. $f(\text{root}) = \text{root}'$
2. $m \xrightarrow[G]{x} n$ implies $f(m) \xrightarrow[G']{x} f(n)$, for all x in A. □

Examples 1.11 (Homomorphisms) From every graph G with alphabet A, there is just one homomorphism to 1_A; from 0_A to G, there are as many homomorphisms as nodes in G. □

Homomorphisms can also be used to define the relationship between a subclass and its parent class in a class hierarchy [Cardelli, Cook]. For this, we will later introduce a method of reducing the alphabet of labels to match that of the target of the homomorphism. Multiple inheritance is simply modelled by asserting the existence of more than one homomorphism from the heap to several different data diagrams. Different languages enforce differing conventions and rules, to ensure that the invariance of such assertions at run time is checkable by compiler. Our theory is claimed to be sufficiently expressive to describe all such checkable rules in any language. It can also formulate much more general assertions, whose truth cannot be checked at compile time, but only at run time or by proof.

Another important role for a homomorphism is to select from a large graph a smaller subgraph for detailed consideration. The shape of the subgraph is specified by the source of the homomorphism, and the target specifies which particular subgraph of that shape is selected. For example, consider the graph



A subgraph of this shape occurs just twice in Figure 1.0; there is only one injective homomorphism from it into the Figure, and one that is non-injective. This kind of subgraph homomorphism has to be redefined to allow for absence of a root.

The remaining role of the homomorphism is to define the concept of an isomorphism of graphs, and so specify what it means for two graphs with different node sets to be essentially the same.

Definition 1.12 (Isomorphism) Let $f : G \to G'$ be a homomorphism. This is said to be an isomorphism if f is invertible, and $f^{-1} : G' \to G$ is also a homomorphism. G and G' are isomorphic if there is an isomorphism from one to the other. □

This rather indirect definition represents the very simple intuitive idea of laying one graph on top of another, and ensuring that it has nodes and edges and labels in all the same places. Like congruent triangles in geometry, they are just two copies of the same graph!

2 The Trace Model

The problem of inaccessible objects is the same as that of inaccessible states in automata theory; and the solution that we adopt is the same: calculate the language of traces that are generated by the graph. A trace of an automaton is a sequence of consecutive events that can occur during its evolution. A trace can be read from the graph by starting at node l and following a path of consecutive edges leading from each node to the next, along a path of directed edges. The trace is extracted as the sequence of labels encountered on the path up to its last node m. The existence of such a trace s is denoted $l \xrightarrow{s} m$. A formal definition uses recursion on the length of the trace.

Definition 2.0 (Traces)

$$\begin{array}{lll} l \xrightarrow{<>} m & \textbf{iff} & l = m \\ l \xrightarrow{<a>} m & \textbf{iff} & (l, a, m) \in E \\ l \xrightarrow{s\frown t} m & \textbf{iff} & \exists n \bullet l \xrightarrow{s} n \wedge n \xrightarrow{t} m \\ l \xrightarrow{*} m & =_{df} & \{s \mid l \xrightarrow{s} m\} \\ \text{traces}(l) & =_{df} & \text{root} \xrightarrow{*} l \end{array}$$

□

Example 2.1 (Figure 1.0) From the graph of Figure 1.0 the sets of traces of each of the six nodes are given by the following six regular expressions

$$\begin{array}{l} n_0 = \varepsilon \\ n_1 = w + n_2\, b \\ n_2 = v + n_2\, a + n_3\, a \\ n_3 = n_1 d \\ n_4 = n_2 c \\ n_5 = n_3 d \end{array}$$

□

In the canonical trace model of a graph, each node l is represented by the set traces(l), containing all traces on paths to it from the root. The set of nodes is therefore a family N of sets of traces ($N \subseteq \mathbf{P}A^*$). The labelled edges and the root of the graph can be defined in terms of this family.

Definition 2.2 (Canonical representation)

$$\begin{array}{rcl} \text{Let } G & = & (A, N, E, r) \\ \widehat{N} & =_{df} & \{\text{traces}(n) \mid n \in N\} \\ \widehat{E} & =_{df} & \{l \xrightarrow{x} m \mid l, m \in \widehat{N} \wedge l^\frown < x > \subseteq m\} \\ \widehat{r} & =_{df} & \text{the unique } n \text{ in } \widehat{N} \text{ containing } <>. \\ \widehat{G} & =_{df} & (A, \widehat{N}, \widehat{E}, \widehat{r}) \end{array}$$

□

Theorem 2.3

For all l, m, $n \in \widehat{N}$ and $X \subseteq A^*$

(1) $(l^\frown X) \subseteq m$ **iff** $X \subseteq (l \xrightarrow[G]{*} m)$

(2) $(l \xrightarrow[G]{*} m)^\frown(m \xrightarrow[G]{*} n) \subseteq (l \xrightarrow[G]{*} n)$

(3) $(\widehat{r} \xrightarrow[G]{*} m) \;=\; m$

Proof: (1) From the fact that for all $s \in A^*$

$$(l ^\frown s) \;\subseteq\; m \quad \textbf{iff} \quad l \xrightarrow[G]{s} m$$

(2) From the associativity of the catenation operator and the Galois connection (1)

(3)

$$\begin{array}{ll}
 & X \subseteq LHS \\
\equiv \{(1)\} & \widehat{r} ^\frown X \subseteq m \\
\Rightarrow \{<>\in \widehat{r}\} & X \subseteq RHS \\
\equiv \{\text{let } m = \text{traces}(n)\} & \forall s \in X \bullet \; (\text{root} \xrightarrow[G]{s} n) \\
\Rightarrow \{\forall t \in \widehat{r} \bullet \; (\text{root} \xrightarrow[G]{t} \text{root})\} & \forall t \in \widehat{r},\, s \in X \bullet \; (\text{root} \xrightarrow[G]{t ^\frown s} n) \\
\equiv \{\text{def of traces}\} & (\widehat{r} ^\frown X) \subseteq \text{traces}(n) = m \\
\equiv \{(1)\} & X \subseteq LHS
\end{array}$$

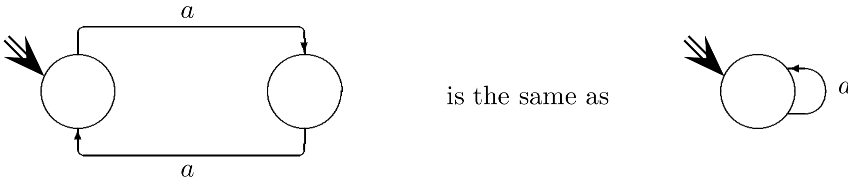
□

In the theory of deterministic automata, the language generated by the automaton is just the union of the set of traces of all its states

$$\text{language } (G) = \bigcup \{\text{traces}(l) \mid l \in N_G\}$$

The great advantage of this is that an inaccessible state has no traces at all, and so makes no contribution to the language. Two automata are therefore decreed to be identical if they have the same language.

Example 2.4 (Identical automata)



because in both cases the language is $\{a\}^*$ □

For automata, the purpose of this identification is to allow automatic minimisation of the number of states needed to generate or recognise a specified language. But in object-oriented programming, such identification of objects would be wholly inappropriate. The reason is that the pointer swinging operation (not considered by automata theory) distinguishes graphs which automata theory says should be the same.

Example 2.5 (after swing) After the assignment $a \rightarrow a \;:=$ **nil**, the two graphs of example 2.4 now look like

□

Even in automata theory, these two graphs are distinct. For this reason, we cannot model a heap simple as a set of traces, and we have to go up one level in complexity to model it as a **set** of sets of traces, as shown in the definition of $\widehat{N}$.

Nevertheless, there are many interesting analogies between our trace model the process algebra of non-deterministic automata. For example, as in CSP [Hoare], the entire set of valid traces is prefix-closed.

Theorem 2.6 (Prefix closure) $\bigcup \widehat{N}$ is non-empty; and if it contains $s^\frown t$, it also contains s. □

An important property of the $\widehat{\ }$ operator, transforming a graph to its canonical representation, is that it leaves unchanged an argument that is already canonical.

Theorem 2.7 (Idempotence)

$$\widehat{\widehat{G}} = \widehat{G}$$

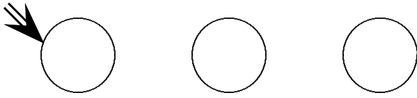
Proof

$$\begin{aligned}
& \text{traces}(\text{traces}(n)) \\
= & \{\text{def of traces}\} \\
& \text{traces}(\text{root}) \xrightarrow[\widehat{G}]{*} (\text{traces}(n)) \\
= & \{\text{Theorem 2.3(3)}\} \\
& \text{traces}(n)
\end{aligned}$$

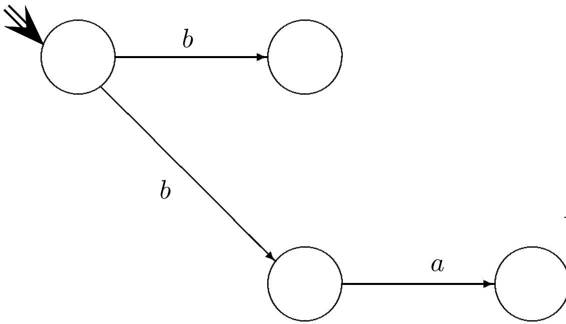
□

Note that this is an equality, not just an isomorphism. But the claim that the result is canonical for all graphs is not justified: there are G such that $\widehat{G}$ is not even isomorphic to G.

Counterexamples 2.8 (Disappearing nodes)



$\widehat{N} = \{\{<>\}, \{\}\}$



$\widehat{N} = \{\{<>\}, \{< b >\}, \{< b, a >\}\}$

□

In choosing to study the canonical representation, we exclude such counterexamples from consideration. The remainder of this section will define and justify the exclusion.

An important property in object-oriented programming is that each name should uniquely denote a single object. This is assured if the graph is deterministic.

Definition 2.9 (Determinism) A graph is deterministic if for each l and x there is at most one m such that $l \xrightarrow{x} m$. This is necessary for determinism of the corresponding automaton. In a data diagram, determinism permits automatic resolution of the polymorphic use of the same label to denote different functions on different data types. In object-oriented programming, it states the obvious fact that each attribute of each object can have only one value. □

If the original graph is deterministic, its canonical node-set $\widehat{N}$ satisfies an additional property familiar from process algebra — it is a bisimulation [Milner].

Definition 2.10 (Bisimulation) A family N of sets of traces is a bisimulation if it is a partial equivalence which is respected by trace extension. More formally, for all p, q in N

$$\begin{array}{lll} p = q & \vee & p \cap q = \{\} \\ s, t \in p & \wedge & t^\frown u \in q \;\Rightarrow\; s^\frown u \in q \end{array}$$

□

Determinism ensures that any two distinct objects will have distinct trace sets, except in the extreme case that both have empty trace sets. Such objects can never be accessed by a program, so they might as well not exist.

Definition 2.11 (Accessibility) A node n is accessible if traces(n) is non-empty. A graph is accessible if all its nodes are, i.e. $\{\} \;\tilde{\in}\; \widehat{N}$. For automata, inaccessible nodes represent unreachable states, which can and should be ignored in any comparison between them. In a heap, they represent unusable storage, which can and should be garbage-collected (or otherwise explicitly deleted). □

At last we can show that we can model all the graphs that we are interested in by simply considering canonical graphs; furthermore, we can assume that N is always a prefix-closed bisimulation.

Theorem 2.12 (Representability) If G is deterministic and accessible, it is isomorphic to $\widehat{G}$

Proof Let $f(n) =_{df}$ traces(n) for all $n \in N$. Because G is deterministic

$$(n \neq m) \;\Rightarrow\; (\text{traces}(n) \neq \text{traces}(m))$$

Futhermore we have

$$\begin{array}{l} \quad f(n) \xrightarrow[G]{x} f(m) \\ \equiv \{\text{def of } \widehat{E}\} \\ \quad \text{traces}(n)^\frown < x > \subseteq \text{traces}(m) \\ \equiv \{\text{traces}(n) \neq \{\} \text{ and } G \text{ is deterministic}\} \\ \quad n \xrightarrow[G]{x} m \end{array}$$

□

We can now solve the problems of graph representation left open in the previous section: objects will be named by traces, and inaccessible objects will disappear. We will assume that the heap G is at all times held in its canonical representation; and redefine each operation of object-oriented programming as an operation on the trace sets N.

Edge deletion $t \rightarrow x :=$ **nil** now has to remove not only the single edge x, but also all traces that include this edge. Every such trace must begin with a trace of the object t itself, i.e. a trace which is equivalent to t by the equivalence N. The trace to be removed must of course contain an occurrence of $< x >$, the edge to be removed. It ends with a trace leading to some other node n in N. The traces removed from n are therefore exactly defined by the set

$$[t]^\frown < x >^\frown (t^\frown < x > \xrightarrow{*} n)$$

We use the usual square bracket notation $[t]_N$ that contains t to denote the equivalence class (or more simply just $[t]$). Of course, x may occur more than once in the trace, either before or after the occurrence shown explicitly above. In the following definition, the removal of the edge is followed by removal of any set that becomes empty – a simple mathematical implementation of garbage-collection.

Re-definition 2.13 (Edge deletion, Node creation)

$$\begin{array}{ll}(t \rightarrow x := \mathbf{nil}) & =_{df} N := \{n - [t]^\frown < x >^\frown (t^\frown < x > \xrightarrow{*} n) \mid n \in N\} - \{\{\}\} \\ (t \rightarrow x := \mathbf{new}) & =_{df} t \rightarrow x := \mathbf{nil};\ \ N := N + \{[t]^\frown < x >\}\end{array}$$

□

Unfortunately, pointer swing is even more complicated than this. We consider first the effect of $t^\frown < y > := s$, in the case where y is a **new** label, occurring nowhere else in the graph. The question now is, what are all the new traces introduced as a result of insertion of this new and freshly labelled edge? As before, every such trace must start with a trace from $[t]$, followed by the first occurrence of y. But now we must consider explicitly the possibility that the new edge occurs many times in a loop. The trace that completes the loop from the head of y back to its tail must be a path leading from s to t in the original graph, i.e. a member of $(s \xrightarrow{*} t)$. After any number of repetitions of $((s \xrightarrow{*} t)^\frown < y >)$, the new trace concludes with a path from s to some node n. The traces added to an arbitrary equivalence class n are exactly defined by the set

$$[t]^\frown < y >^\frown ((s \xrightarrow{*} t)^\frown < y >)^{*\frown} (s \xrightarrow{*} n)$$

Note that in many cases $(s \xrightarrow{*} n)$ will be empty, because there is no path from s to n. Then by definition of $^\frown$ between sets, the whole of the set described above is empty, and no new traces are added to n.

After inserting these new traces, it is permissible and necessary to remove the original edge x from the original graph and from the newly added traces too. Finally, the freshly named new edge y can be safely renamed as x.

Re-definition 2.14 (Pointer swing)

$$\begin{array}{ll}(t \rightarrow x := s) & =_{df} \ \text{let } y \text{ be a fresh label in} \\ & N := \{n + [t]^\frown < y >^\frown ((s \xrightarrow{*} t)^\frown < y >)^{*\frown} (s \xrightarrow{*} n) \mid n \in N\}; \\ & t \rightarrow x := \mathbf{nil};\ \ \text{rename } y \text{ to } x\end{array}$$

□

Note that it is **not** permissible to delete the edge x before adding the new edge: this could make inaccessible some of the objects that need to be retained because they are accessible through s. The problem is clearly revealed in the simplest case: $t \rightarrow x := t \rightarrow x$. The necessary complexity of the pointer swing is a serious, perhaps a crippling disadvantage of the trace model of pointers and objects.

3 Applications

The purpose of our investigations is not just to contribute towards a fully abstract denotational semantics for an object-oriented programming language. We also wish to provide assistance in reasoning about the correctness of such programs, and to clarify the conditions under which they can be validly optimised. Both objectives can be met with the aid of assertions, which describe useful properties of the values of variables at appropriate times in the execution of the program. To formulate clear assertions (unclear ones do not help), we need an expressive language; and to prove the resulting verification conditions, we need a toolkit of powerful theorems. This section makes a start on satisfying both these needs.

Two important properties of an individual node are defined as follows. It is acyclic if the only path leading from itself to itself is the empty path.

$$n \text{ is acyclic } =_{df} (n \stackrel{*}{\longrightarrow} n) = \{<>\}$$

A graph is acyclic if all its nodes are. A node is a sink if there is no node accessible from it except itself

$$n \text{ is a sink } =_{df} \forall m \bullet n \stackrel{*}{\longrightarrow} m \subseteq \{<>\}$$

These definitions can be qualified by a subset B of the alphabet, e.g.

$$n \text{ is a } B\text{-sink } =_{df} \forall m \bullet (n \stackrel{*}{\longrightarrow} m) \cap B^* \subseteq \{<>\}$$

Two important relationships between nodes are connection and dominance. Connection is defined by the existence of a path between the nodes; and this path may be required to use only labels from B

$$m \stackrel{B}{\Longrightarrow} n =_{df} (n \stackrel{*}{\longrightarrow} m) \cap B^* \neq \{\}$$

$\stackrel{B}{\Longrightarrow}$ is clearly a pre-order, i.e., transitive, and reflexive, but it is antisymmetric only in acyclic graphs. The root is the bottom of any accessible graph. The superscript B is omitted when it is the whole alphabet of the graph under discussion. The relation of dominance between objects is stronger than connection. One object l in a graph dominates an object m if every path to m leads through l

$$l \sqsubseteq m =_{df} l^\frown(l \stackrel{*}{\rightarrow} m) = m$$

Deletion of a dominating object makes a dominated object inaccessible. So this relationship is very important in proving that a graph remains accessible and/or acyclic after a pointer swing. Its properties are similar to those of the prefix ordering over simple traces.

Theorem 3.0 (Dominance ordering) $\sqsubseteq$ is a partial order with the root as a bottom and the empty set as its top. The dominators of any node are totally ordered, i.e.

$$\text{if } l \sqsubseteq n \text{ and } m \sqsubseteq n \text{ then } l \sqsubseteq m \text{ or } m \sqsubseteq l$$

Proof see appendix. □

For non-empty nodes, dominance implies connection. If a node has only one trace, then every node that connects to it will dominate it. If all nodes have this property, the graph is called a divergent tree — divergent because all its pointers point away from the root and towards its sinks (i.e. the leaves).

In a language without garbage-collection, there is a grave danger that a pointer swing will leave an object that has no other pointer pointing to it. Such an object can never again be accessed by the program, and the storage space that it occupies will never be reused. This phenomenon is known as a space leak. In order to prevent it, the programmer must accept the obligation to ensure that a certain precondition is satisfied before each pointer swing $s \rightarrow x := t$. The relevant precondition is expressed as non-dominance

$$\neg s \sqsubseteq s^\frown < x >$$

In a language without garbage-collection, the only way in which heap storage can be recovered for reuse is by an explicit command in the program, declaring that a particular object will never be accessed again. We will treat the simplest form of atomic deletion, as for example the delete command in PASCAL. This command must be given at the same time that the last pointer to the object is deleted by $s \rightarrow x :=$ **nil**. The precondition of such a deletion is the opposite of that for an assignment

$$s \sqsubseteq s^\frown < x >$$

In fact, a stronger precondition is necessary. All the objects accessible through $s \ ^\frown < x >$ must be accessible through some other object as well (otherwise their space would leak anyway). The full precondition for deletion is

$$\forall y \bullet \ (s^\frown < x > \sqsubseteq s^\frown < x >^\frown y \textbf{ iff } y = <>).$$

The complexity of these preconditions may explain why control of space leaks is a difficult problem in practice.

A heap as represented in the store of a computer must be described as a single variable, even though its value is of great size and complexity. Any pointer in the heap can at any time be swung to point to any other object whatsoever. To control this complexity, a programmer usually constrains the use of a heap in a highly disciplined way. The heap is understood to be split into a number of component subgraphs, satisfying invariant properties that limit the connections within and between the components. A component of a graph can readily be selected in two ways: by restricting the alphabet, or by concentration on a single branch.

Definition 3.1 (Subgraphs)

$$N \upharpoonright B =_{df} \{n \cap B^* \mid n \in N\} - \{\{\}\}$$

$N \upharpoonright B$ is a canonical graph with alphabet B, containing just those objects nameable by chains of labels drawn wholly from B.

$$N/n = \{n \xrightarrow{*} m \mid m \in N\} \ - \ \{\{\}\}, \text{ where } n \in N$$

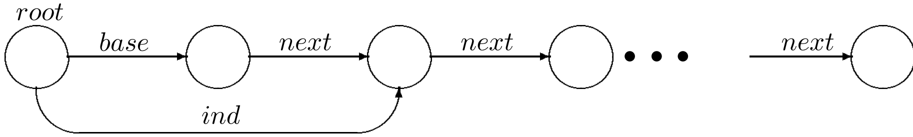
N/n is a canonical graph, isomorphic to the subgraph of all nodes accessible from n. It consists of just that part of the heap that is seen by a method local to n. □

These subgraph operations obey laws identical to those found in process algebra

$$\begin{aligned}
N \upharpoonright A &= N, \quad \text{if } A \text{ is the alphabet of the graph} \\
(N \upharpoonright B) \upharpoonright C &= N \upharpoonright (B \cap C) \\
N \upharpoonright \{\} &= 0_{\{\}} \\
N/ <> &= N \\
(N/s)/t &= N/(s^\frown t)
\end{aligned}$$

The purpose of this paper has been to provide a conceptual framework for formalisation of invariant assertions about data structures represented as objects in a heap. Class and type declarations serve the same purpose; they are also carefully designed to enjoy the additional advantage that their validity can be checked by compiler. Assertions have more expressive power, but they can be tested only at run time, and they can be validated only by proof. In the remainder of this section we explore the power of the trace model in the formulation of assertions, and suggest that a diagram may be helpful in visualising them.

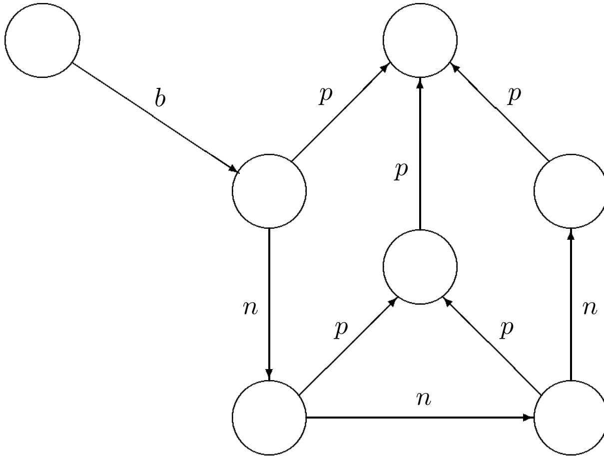
Definition 3.2 (Chain) Consider a pair of labels $B = \{base, next\}$. This defines a chain if $C = (N/base) \restriction \{next\}$ is invariantly acyclic.



A variable *ind* is an index into this chain if invariantly $base \overset{next}{\Longrightarrow} ind$. A last-pointer is an index that always points to a *next*-sink. A final segment of the chain, chopped off at a given index, is C/ind. □

A chain is often used to scan a set of objects of interest. A good example is a convergent tree — convergent because the pointers point away from the leaves towards the root (a sink). Without a chain through them, the leaves (and indeed the whole tree) would be inaccessible.

Figure 3.3 (Convergent tree)



The attribute p points from an offspring to its parent in the tree; the attribute n constructs the leaf chain starting at a declared base variable b. □

We wish to formalise the properties shared by all such trees, without restriction on size or shape, and without defining which other attributes besides p and n may be pointing to or from the nodes. Let us first confine attention to the subgraph T

of interest

$$T =_{df} (N/b) \upharpoonright \{p, n\}$$

The aim is to formulate the desired invariant properties of T as a conjunction of simple conditions that can be checked separately, and reused in different combinations for different purposes. The first condition has already been given a formal definition

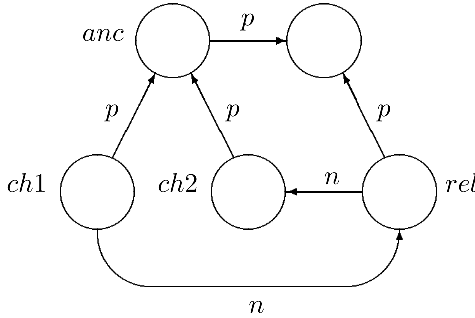
1. T is acyclic

2. Every object on the chain has a parent: if $j \stackrel{n}{\longrightarrow} k$ then $(\exists l \bullet k \stackrel{p}{\rightarrow} l) \wedge (\exists l \bullet j \stackrel{p}{\rightarrow} l)$

3. No parent is an object on the chain: if $l \stackrel{p}{\longrightarrow} m$ then $(\neg\, k \stackrel{n}{\rightarrow} m) \wedge (\neg\, m \stackrel{n}{\rightarrow} k)$

4. Any two nodes on the tree share a common ancestor: $\forall j, k\ \exists l \bullet j \stackrel{p}{\Rightarrow} l \wedge k \stackrel{p}{\Rightarrow} l$

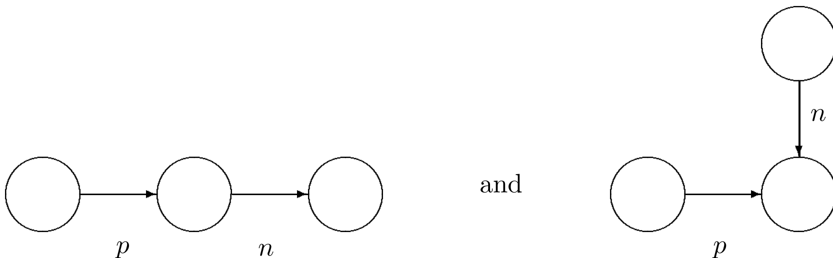
There is one more property that is usually desired of a leaf-chain: it should visit the leaves in some reasonable order, for example, close relatives should appear close in the chain. In particular, the following picture should **not** appear in the graph.



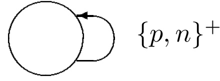
Note that *ch1* and *ch2* are more closely related to each other than to *rel*, which therefore should not separate them in the chain. The requirement is formalised

5. If $ch1 \stackrel{p}{\Rightarrow} anc \wedge ch2 \stackrel{p}{\Rightarrow} anc \wedge ch1 \stackrel{n}{\Rightarrow} rel \wedge rel \stackrel{n}{\Rightarrow} ch2$ then $rel \stackrel{p}{\Rightarrow} anc$.

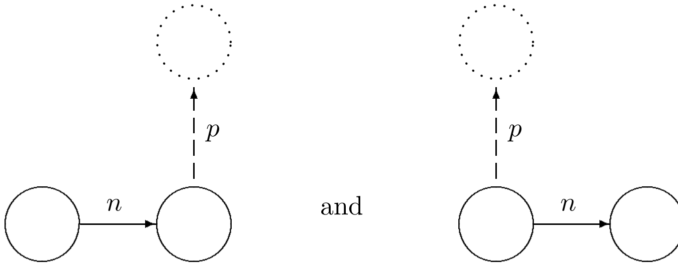
These invariants are expressed in the predicate calculus, using variables that have either implicit quantification over all traces in $\bigcup T$, or explicit existential quantification. The invariants can also be conveniently represented pictorially in the graphical calculus [Curtis and Lowe]. The simpler invariants directly prohibit occurrence in the heap of any subgraph of a certain shape. For example, condition (2) prohibits any occurrence of the two shapes



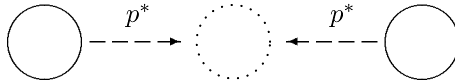
More formally, there is no homomorphism from either of these graphs into the heap. The acyclic condition (1) can be pictured by using a single arrow to represent a complete (non-empty) trace drawn from the specified alphabet; the following is prohibited



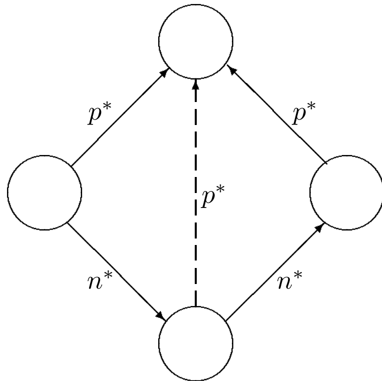
The more complicated invariants take the form of an implication, whose consequent has existentially quantified variables. These are drawn as dotted lines rather than the solid lines that represent variables universally quantified over the whole formula. So the condition (2) would be drawn



The condition (4) combines this convention with the path convention



The fifth condition is the most elaborate



The meaning of the dotted line convention illustrated above can be formalised, again in terms of graph homomorphisms. The picture states that every homomorphism from the graph drawn as solid lines and nodes can be extended to a homomorphism from the whole picture, including dotted lines and nodes. The extension must not

change the mapping of any of the solid components. Even more formally, the diagram defines an obvious injective homomorphism $j : solid \rightarrow diagram$ from its solid components to the whole diagram. It states that for all $h : solid \rightarrow T$ there exists an $h' : diag \rightarrow T$ such that $h = j; h'$ (h factors through j). In plainer words, perhaps the programmer's instinct to draw pictures when manipulating pointers can be justified by appeal to higher mathematics.

4 Conclusion

The ideas reported in this paper have not been pursued to any conclusion. Perhaps, in view of the difficulties described at the end of section 2, they never will be. Their interest is mainly as an example of the construction of a generic mathematical model to help in formalisation of assertions about interesting and useful data structures. Such assertions can be helpful in designing and maintaining complicated class libraries, and in testing the results of changes, even if they are never used for explicit program proof.

Other published approaches to reasoning about pointer structures have been much better worked out. An early definition of a tree-structured machine with an equivalence relation for sharing was given in [Landin]. The closest in spirit to the trace model is described in [Morris] and applied to the proof of an ingenious graph marking algorithm. A similar approach using nice algebraic laws was taken in [Nelson]. Another promising approach [Möller] exploits the proof obligation as a driver for the design of the algorithm in a functional style. It models each label as a function from addresses to values or other addresses contained in the addressed location. Other authors too have been deterred by the complexity of sharing structures introduced by pointer swing [Suzuki].

Acknowledgements For useful comments on earlier drafts we thank Frances Page, Bernhard Möller, Manfred Broy, Jay Misra, Paul Rudin, Ralph Steinbrüggen, Zhou Yu Qian.

References

1. M. Abadi and L. Cardelli. A theory of objects. Springer (1998).
2. L. Cardelli. A semantics of multiple inheritance. Information and Computation 76: 138–164 (1988).
3. W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. Information and Computation 114(2), 329–350 (1994).
4. S. Curtis and G. Lowe, A graphical calculus. In B. Möller (ed) Mathematics of Program Construction LNCS 947 Springer (1995)
5. O. Dahl and K. Nygaard. Simula, an Algol-based simulation language. Communications of the ACM 9(9) 671–678 (1966).
6. A. Goldberg and D. Robson. Smalltalk-80. The language and its implementation. Addison-Wesley (1983).
7. C.A.R. Hoare, Communicating Sequential Processes. Prentice-Hall (1985).
8. S.N. Kamin and U.S. Reddy. Two semantic models of object-oriented languages. In C.A. Gunter and J.C. Mitchell (eds): Theoretical Aspects of Object-Oriented Programming, 463–495, MIT Press, (1994).
9. P.J. Landin, A correspondence between ALGOL 60 and Church's lambda-notation Part 1. Communications ACM 8.2 (1965) 89-101
10. B. Meyer. Object-oriented Software Construction, Prentice-Hall second edition (1997).
11. R. Milner. Communication and Concurrency, Prentice Hall (1987)
12. B. Möller, Towards pointer algebra. Science of Computer Programming 21 (1993), 57-90.
13. B. Möller, Calculating with pointer structures. Proceedings of Mathematics for Software Construction, Chapman and Hall (1997), 24-48.

14. J.M.Morris, A general axiom of assignment, Assignment and linked data structure, A proof of the Schorr-Waite algorithm. In M Broy and G. Schmidt (eds.) Theoretical Foundations of Programming Methodology, 25-51, Reidel 1982 (Proceedings of the 1981 Marktoberdorf Summer School).
15. G. Nelson, Verifying reachability invariants of linked structures. Proceedings of POPL (1983), ACM Press, 38-47.
16. N. Suzuki, Analysis of pointer rotation. Communications ACM vol 25 No 5, May (1982), 330-335.

Appendix

Proof of Theorem 3.0

First we are going to show that dominance is a partial order.

(reflexive)

$$\begin{aligned} & l \\ = & \{s\hat{}<> \;=\; s\} \\ & l\hat{}\{<>\} \\ \subseteq & \{<> \in (l \xrightarrow{*} l)\} \\ & l\hat{}(l \xrightarrow{*} l) \\ \subseteq & \{\text{Theorem 2.3.(1)}\} \\ & l \end{aligned}$$

(antisymmetric) Assume that $l \sqsubseteq m$ and $m \sqsubseteq l$. If $l = \{\}$ then

$$m \;=\; l\hat{}(l \xrightarrow{*} m) \;=\; \{\}\hat{}(l \xrightarrow{*} m) \;=\; \{\} \;=\; l$$

Assume that $l \neq \{\}$. From the fact that

$$l \;=\; l\hat{}(l \xrightarrow{*} m)\hat{}(m \xrightarrow{*} l)$$

and $l \neq \{\}$ we conclude that

$$\begin{aligned} & <> \in (l \xrightarrow{*} m)\hat{}(m \xrightarrow{*} l) \\ \equiv & \{s\hat{}t = <> \;\textbf{iff}\; s = t = <>\} \\ & <> \in (l \xrightarrow{*} m) \cap (m \xrightarrow{*} l) \\ \Rightarrow & \{l \;=\; m\hat{}(m \xrightarrow{*} l) \text{ and } m \;=\; l\hat{}(l \xrightarrow{*} m)\} \\ & (l \subseteq m) \;\wedge\; (m \subseteq l) \\ \equiv & \; l = m \end{aligned}$$

(transitive) Assume that $l \sqsubseteq m$ and $m \sqsubseteq n$.

$$\begin{aligned} & n \\ \supseteq & \{\text{Theorem 2.3(1)}\} \\ & l\hat{}(l \xrightarrow{*} n) \\ \supseteq & \{\text{Theorem 2.3(2)}\} \\ & l\hat{}(l \xrightarrow{*} m)\hat{}(m \xrightarrow{*} n) \\ = & \{l \sqsubseteq m \text{ and } m \;\sqsubseteq\; n\} \\ & n \end{aligned}$$

Let n be a non-empty node. Assume that

$$l \sqsubseteq n \text{ and } m \sqsubseteq n$$

For any subset X of A^* we define $sht(X)$ as the set of shortest traces of X

$$sht(X) \;=_{df}\; \{s \in X \mid \forall t \in X \bullet t \leq s \Rightarrow t = s\}$$

From the fact that $n \neq \{\}$ we conclude that neither $sht(l)$ nor $sht(m)$ is empty. Consider the following cases:

(1) $sht(l) \cap sht(m) \neq \{\}$: From the determinacy it follows that

$$l = m$$

(2) $sht(l) \cap sht(m) = \{\}$: From the assumption that $l \sqsubseteq n$ and $m \sqsubseteq n$ it follows that

$$\forall u \in sht(l) \exists v \in sht(m) \bullet (u \leq v \;\vee\; v \leq u)$$

and

$$\forall v \in sht(m) \exists u \in sht(l) \bullet (u \leq v \;\vee\; v \leq u)$$

(2a) $\forall u \in sht(l) \exists v \in sht(m) \bullet v \leq u$: From the bisimulation property it follows that

$$m \sqsubseteq l$$

(2b) $\forall v \in sht(m) \exists u \in sht(l) \bullet u \leq v$: In this case we have

$$l \sqsubseteq m$$

(2c) There exist u, $\hat{u} \in sht(l)$ and v, $\hat{v} \in sht(m)$ such that

$$u < v \text{ and } \hat{v} < \hat{u}$$

Let

$$j =_{df} min\{length(s) \mid s \in sht(l \xrightarrow{*} n)\}$$

$$k =_{df} min\{length(t) \mid t \in sht(m \xrightarrow{*} n)\}$$

From $u \leq v$ we conclude that $j > k$, and from $\hat{v} < \hat{u}$ we have $j < k$, which leads to contradiction.

From the above case analysis we conclude that

$$l \sqsubseteq m \quad \text{or} \quad m \sqsubseteq l \qquad \square$$

Synthesizing Objects

Krzysztof Czarnecki[1] and Ulrich W. Eisenecker[2]

[1]DaimlerChrysler AG Research and Technology, Ulm, Germany
czarnecki@acm.org

[2]University of Applied Sciences Heidelberg, Germany
ulrich.eisenecker@t-online.de

Abstract. This paper argues that the current OO technology does not support reuse and configurability in an effective way. This problem can be addressed by augmenting OO analysis and design with feature modeling and by applying generative implementation techniques. Feature modeling allows capturing the variability of domain concepts. Concrete concept instances can then be synthesized from abstract specifications.

Using a simple example of a configurable list component, we demonstrate the application of feature modeling and how to implement a feature model as a generator. We introduce the concepts of configuration repositories and configuration generators and show how to implement them using object-oriented, generic, and generative language mechanisms. The configuration generator utilizes C++ template metaprogramming, which enables its execution at compile-time.

1 Introduction

In the early days of OO, there used to be the belief that objects are reusable by their very nature and that reusable OO software simply “falls out” as a byproduct of application development. Today, the OO community widely recognizes that nothing could be further from the truth. Reusable OO software has to be carefully engineered and engineering *for reuse* requires a substantial investment. One of the weaknesses of current OO Analysis and Design (OOA/D) is the inadequate support for variability modeling. As suggested by the work in the reuse community (e.g. [CN98, GFA98, Cza98]), this problem can be addressed by augmenting OOA/D with *feature modeling*. Feature modeling was originally introduced in the Feature-Oriented Domain Analysis (FODA) method [KCH+90] as a technique for modeling commonalities and variabilities within a domain. Since reusable models may contain a significant amount of variability, rigid class hierarchies are inappropriate for implementing such models. They are more adequately implemented as flexible, highly parameterized component classes. Concrete component configurations can be synthesized from abstract descriptions by a *configuration generator*. An important aspect of such designs is the separation between the components and the configuration knowledge, which is achieved using a *configuration repository*. We demonstrate these concepts using a concrete example and also show how to implement them using object-oriented and generic language mechanisms in C++. The configuration generator utilizes C++ template metaprogramming, which enables its execution at compile-time.

Rachid Guerraoui (Ed.): ECOOP’99, LNCS 1628, pp. 18-42, 1999.
© Springer-Verlag Berlin Heidelberg 1999

The rest of the paper is organized as follows. Section 2 discusses the weaknesses of OOA/D in the context of reuse and configurability. Section 3 introduces the concept of feature models. Section 4 makes the connection between feature models and object synthesis. Section 5 contains a concrete example starting with a feature diagram and concluding with a configuration generator. Section 6 lists extensions not included in the example due to space limitations. Section 7 discusses two libraries implemented using the techniques from this paper. Section 8 discusses related work. Section 9 concludes by making the connection to active libraries and Generative Programming.

2 Problems of Object Technology in the Context of Software Reuse

Two important areas of OO technology addressing reuse are frameworks and design patterns. A framework embodies an abstract design for a *family* of related systems in the form of collaborating classes. Similarly, design patterns provide reusable solutions to recurring design problems across different systems. Patterns, as a documentation form, also proved useful in capturing reusable solutions in other areas such as analysis, architecture, and organizational issues. Unfortunately, only very few OOA/D methods provide any support for the development of frameworks.[1] Similarly, there is little systematic support in both finding and applying patterns.

Most OOA/D methods focus on developing single systems rather than families of systems. Given this goal, these methods are inadequate for developing reusable software, which requires focusing on classes of systems rather than single systems. A comparison between Domain Engineering methods (e.g. ODM [SCK+96] or FODA [KCH+90]), which are designed for engineering system families, and OOA/D methods reveals the following deficiencies of the latter:

- *No distinction between engineering for reuse and engineering with reuse*: Taking reuse into account requires splitting the OO software engineering process into engineering *for reuse* (i.e. Domain Engineering) and engineering *with reuse* (i.e. Application Engineering). OOA/D methods come closest to Application Engineering, with the difference that Application Engineering focuses on reusing available assets produced during Domain Engineering.
- *No domain scoping phase*: Since OOA/D methods focus on engineering single systems, they lack a domain scoping phase, where the target class of systems is selected. Also, OOA/D focuses on satisfying “the customer” of a single system rather than analyzing and satisfying *stakeholders* (including potential customers) of a class of systems.
- *Inadequate modeling of variability*: The only kind of variability modeled in current OOA/D is intra-application variability, e.g. variability of certain objects

[1] As of writing, OOram [Ree96] is the only OOA/D method known to the authors which truly recognizes the need for a specialized engineering process for reuse. The method includes a domain scoping activity involving the analysis of different classes of consumers. However, the method does not incorporate feature modeling.

over time and the use of different variants of an object at different locations within an application. Domain Engineering, on the other hand, focuses on variability across different systems in a domain for different users and usage contexts. Since modeling variability is fundamental to Domain Engineering, Domain Engineering methods provide specialized notations for expressing variability.

Thus, a general problem of all OOA/D methods is inadequate modeling of variability. Although the various modeling techniques used in OOA/D methods support variability mechanisms (e.g. inheritance, aggregation, and static parameterization), OOA/D methods do not include an abstract and concise model of commonality, variability, and dependencies. There are several reasons for providing such a model:

- Since the same variability may be implemented using different variability mechanisms in different models, we need a more abstract representation of variability (cf. Section 3).
- The user of reusable software needs an explicit and concise representation of available features and variability.
- The developer of reusable software needs to be able to answer the question: why is a certain feature or variation point included in the reusable software?

The lack of domain scoping and explicit variability modeling may cause two serious problems:

- relevant features and variation points are missing; and
- many features and variation points are included but never used; this causes unnecessary complexity and cost (both development and maintenance cost).

Covering the right features and variation points requires a careful balancing between current and future needs. Thus, we need an explicit model that summarizes the features and the variation points and includes the rationale and the stakeholders for each of them. In Domain Engineering, this role is played by a *feature model*. A feature model captures the reusability and configurability aspect of reusable software.

3 Feature Models

Domain concepts that we try to model in reusable software are inherently complex. Even the implementation of a simple container object requires a multitude of design decisions, e.g. type of elements, what kind of iterators it provides, ownership (i.e. whether the container keeps the original elements or their copies and whether it is responsible for deallocating the elements or not), memory allocation (on the stack, on the heap, in a persistent store, etc.), memory management (e.g. whether growing is possible or not), error detection (e.g. whether bounds checking is done or not), synchronization of concurrent access, etc. If the container is to be reusable, many of these decisions have to be changeable since different usage contexts will have different requirements. The changeable design decisions span a design space containing *variation points* [JGJ98] at which different design alternatives and options may be selected.

In Domain Engineering, each of the alternative design decisions is represented by a *feature*.[2] Following the conceptual modeling perspective, a feature is defined as an important property of a concept. For example, the features of a *list* may include *ordered*, *singly linked*, *keeps track of its size*, *can contain elements of different types*, etc. In the context of Domain Engineering, features represent reusable, configurable requirements and each feature has to make a difference to someone, e.g. a stakeholder or a client program. For example, when we build an order processing system, one of the features of the pricing component could be *aging pricing strategy*, e.g. you pay less for older merchandise. This pricing strategy might be particularly interesting to companies selling perishable goods.

Features are organized into *feature diagrams* [KCH+90], which reveal the kinds of variability contained in the design space. An example of a feature diagram is shown in Fig. 1. It describes a simple model of a car. The root of the diagram represents the concept *car*. The remaining nodes are features. The diagram contains four kinds of features (see [Cza98] for other kinds of features and a full description of the feature diagram notation):[3]

- *Mandatory features*: Mandatory features are pointed to by simple edges ending with a filled circle. The features *car body*, *transmission*, and *engine* are mandatory and thus part of any car.
- *Optional features*: Optional features are pointed to by simple edges ending with an empty circle, e.g. *pulls trailer*. A car may pull a trailer or not.
- *Alternative features*: Alternative features are pointed to by edges connected by an arc, e.g. *automatic* and *manual*. Thus, a car may have an automatic or manual transmission.
- *Or-features*: Or-features are pointed to by edges connected by a filled arc, e.g. electric and gasoline. Thus, a car may have an electric engine, a gasoline engine, or both.

A feature diagram is usually accompanied by additional information, such as short semantic description of each feature, rationale for each feature, stakeholders and client programs interested in each feature, examples of systems with a given feature, constraints between features (e.g. which feature combinations are illegal and which features imply the selection of which other features), default dependency rules (i.e. which feature suggests the selection of which other features), availability sites (i.e. where, when, and to whom a feature is available), binding modes (e.g. whether a feature is bound dynamically or statically), open/closed attributes (i.e. whether new subfeatures are expected), and priorities (i.e. how important is a feature). All this information together constitutes a *feature model* (see [Cza98] for a detailed description).

---

[2] Features and feature modeling have been propagated by the Feature-Oriented Domain-Analysis (FODA) method [KCH+90].

[3] Or-features are not part of the notation in [KCH+90]. They were introduced in [Cza98].

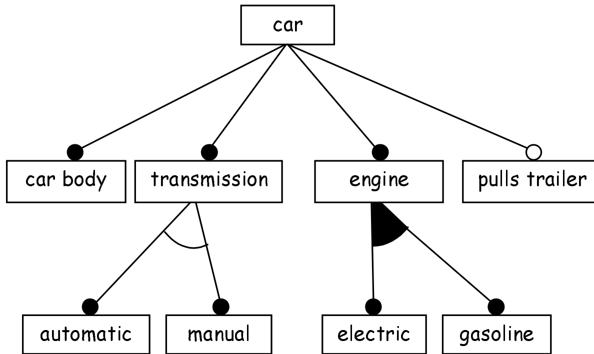


Fig. 1. A sample feature diagram of a car

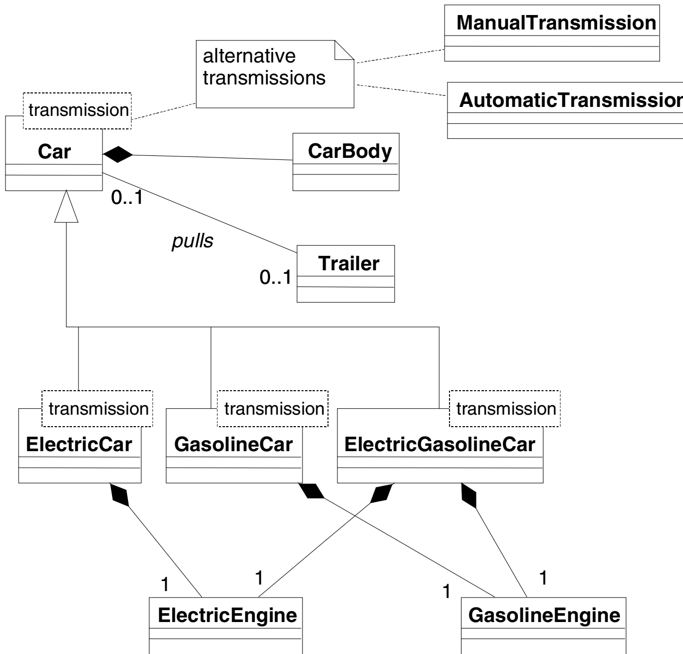


Fig. 2. One possible implementation of the simple car from Fig. 1 using a UML class diagram

A feature model describes the configurability aspect of a concept at a high level of abstraction. Indeed, feature models are more abstract than object diagrams. Fig. 2 shows *one possible implementation* of the car feature diagram in UML. In order to be able to represent our car as an UML class diagram, we had to make a number of concrete decisions about which variability mechanisms to use. The implementation in Fig. 2 uses static parameterization for the transmission, inheritance for the engine, and

an association with the cardinality 0..1 for the trailer. Of course, other implementation choices are also possible. For example, we could also use inheritance or dynamic parameterization for the transmission.

The fundamental flaw of the OOA/D methodology in the context of software reuse is to start modeling in terms of variability implementation mechanisms such as inheritance, dynamic parameterization, and static parameterization rather than more abstract variability representations.

For this reason, as suggested by various researchers [CN98, GFA98, Cza98], we use feature models in addition to other OO models.

4 Feature Models and Object Synthesis

We can implement the functionality represented by a feature model using a set of *implementation components*. The idea is that the implementation components can be configured to yield the different systems covered by a feature model. We can utilize different technologies for implementing the components. One possibility is to use objects.

As stated, object-oriented designs support variability using different mechanisms and techniques. Nearly every design pattern listed in [GHJV95] is about making some part of a design variable, e.g. *bridge* lets you vary the implementation of an object; *strategy* turns an algorithm into a parameter; *state* allows you to vary behavior depending on the state; *template method* provides a way to vary computation steps while keeping the algorithm structure constant. Most of the standard implementations of these patterns utilize dynamic parameterization allowing parameters to vary at runtime. However, reusable models are also full of static variation points, i.e. ones that vary from application to application rather than within one application at runtime. Such variation points are better implemented using static parameterization. We will see concrete examples in Section 5.2.

What is the relationship between features and the implementation components? In this context, we differentiate between three kinds of features:

- *Concrete features*: A concrete feature is directly implemented by one component, e.g. sorting can be directly implemented by a sorting component.
- *Aspect features*: An aspect feature is implemented as an *aspect* in the sense of Aspect-Oriented Programming [KLM+97]. An *aspect* is a kind of modularity which affects many other components, e.g. a declarative description of the synchronization of a number of components. Aspects are usually implemented using an aspect weaver, i.e. a language processor which automatically implements the aspect by coordinating (e.g. merging or interpretatively scheduling) the component code and the aspect code.
- *Abstract features*: Abstract features do not have direct implementations whatsoever. They are implemented by an appropriate combination of components and aspects. Examples of abstract features are performance requirements, such as optimize for speed or space or accuracy.

We say that features make up the *problem space*, whereas the implementation components and aspects constitute the *solution space*. Both spaces have different structures: abstract features have no directly corresponding components or aspects in the solution space, and there may be some "implementation-detail" components and aspects that have no direct correspondence in the problem space. Both spaces are also driven by different, usually conflicting goals: The problem space consists of high-level concepts and features which application programmers would like to work with, while the components in the solution space are designed

- as elementary components combinable in as many ways as possible,
- to avoid any code duplication, and possibly
- to be reusable across many product lines.

The separation between problem and solution space allows us to satisfy both the problem space goals and the solution space goals. It also promotes software evolution since both spaces may be modified (to a certain degree) independently.

The mapping between the problem and the solution space is facilitated by *configuration knowledge*, which consists of

- *constraints* specifying which feature combinations are illegal and which features require the selection of which other features,
- *default dependency rules* specifying which feature suggests the selection of which other features, and
- *mapping rules* specifying which feature combinations require which combinations of components and aspects.

Since some of the variation points in the problem space are static, we need a way to evaluate the configuration knowledge and compose the appropriate components and aspects at compile time. What does this mean if we use objects to implement the solution space? We need a metaprogramming facility which synthesizes objects according to abstract featural descriptions at compile time. As with any new programming concept, direct support in a programming language is desirable. Surprisingly, the concepts outlined above can be implemented utilizing the OO and generic features of C++. We demonstrate the necessary techniques in the following section.

5 Example: Synthesizing a List Container

5.1 Feature Model

Suppose we want to develop a reusable singly linked list. First, we need to analyze the requirements different applications may have for a list in areas such as type of elements, element traversal, storage layout, ownership, memory allocation and memory management, error detection, synchronization of concurrent access, etc. We document the variable features in different areas using feature diagrams. Finally, we should prioritize the features according to project goals (e.g. target customers and applications). For the purpose of this presentation, we show the implementation of a

list covering the features shown in Fig. 3. ElementType is the type of the elements stored in the list and is a free parameter (i.e. any type can be substituted for ElementType), as indicated by the square brackets. Ownership indicates whether the list keeps references to the original elements and is not responsible for element deallocation (i.e. external reference), or keeps references and is responsible for element deallocation (i.e. owned reference), or keeps copies of the original elements and is responsible for their allocation and deallocation (i.e. copy). Morphology describes whether the list is monomorphic (i.e. all elements have the same type) or polymorphic (i.e. can contain elements of different types). Each list element may also contain a length counter allowing for a length operation of a constant time complexity. LengthType is the type of the counter. Finally, the list may optionally trace its operation, e.g. by logging operation calls to the console. (Of course, this diagram could be extended with further features.)

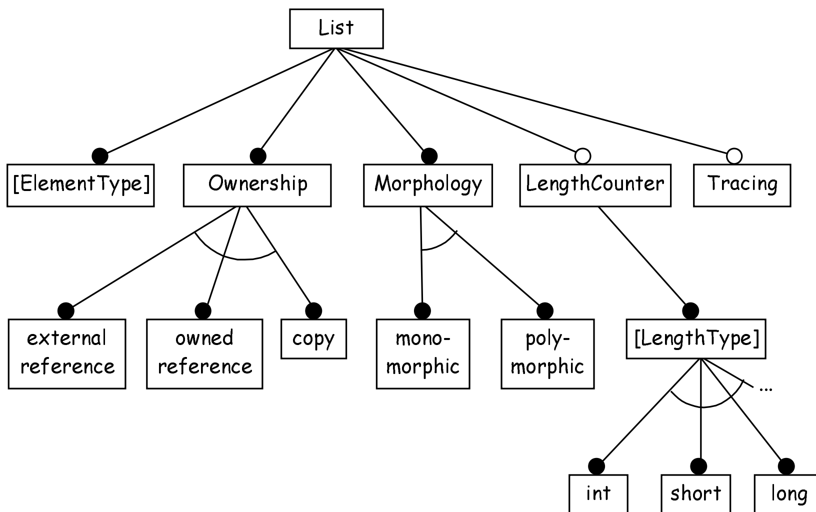


Fig. 3 Feature diagram of a simple list container

5.2 Implementation Components

As stated, we can apply different OO techniques to implement the variability contained in the list feature diagram. Here we show an implementation using static parameterization (including parameterized inheritance). Our implementation consists of the following components:

- PtrList, which implements the basic list functionality including the accessing operations for the head (head() and setHead()) and tail (tail() and setTail());
- LenList, which is a wrapper for adding a length counter to a list;
- TracedList, which is a wrapper for adding tracing to a list.

Additionally, there are three sets of small components for implementing ownership and morphology:

- *destroyers*, which deallocate memory;
- *type checkers*, which check the type of the elements added to the list;
- *copiers*, which copy the elements.

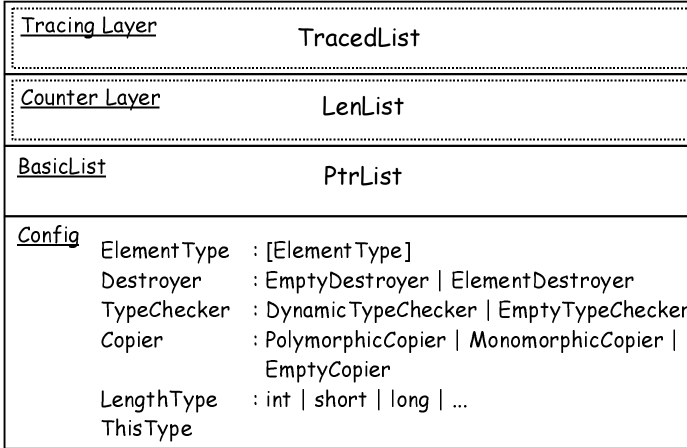


Fig. 4 Target architecture of the list container

```
List            : TracedList[OptCounterList] | OptCounterList
OptCounterList  : LenList[BasicList] | BasicList
BasicList       : PtrList[Config]
Config          :
  ElementType   : [ElementType]
  Destroyer     : EmptyDestroyer | ElementDestroyer
  TypeChecker   : DynamicTypeChecker | EmptyTypeChecker
  Copier        : PolymorphicCopier | MonomorphicCopier | EmptyCopier
  LengthType    : int | short | long | ...
  ReturnType //the final list type
```

Fig. 5 GenVoca grammar of the list container

The implementation components constitute a layered architecture (or a GenVoca architecture [BO92]) shown in Fig. 4. Each rectangle represents a layer. A layer drawn on top of another layer refines the latter by adding new functionality. In general, a layer may contain more than one alternative component. A component from one layer takes another component from the layer below as its parameter, e.g. `LenList` may take `PtrList` as its parameter. A layer containing a dashed rectangle is optional (see [Cza98] for details of this notation). The bottom layer is referred to as a *configuration repository* (abbr. `Config`). A configuration repository provides types needed by the other layers under standardized names. It works as a kind of registry, which components may use to retrieve configuration information from and also to exchange such information among each other. A configuration repository allows us to separate the configuration knowledge from the components. It minimizes dependencies

between components since the components do not exchange types and constants directly, but through the repository. As we show later, we implement a configuration repository as a so-called traits class [Mye95]. The use of trait classes as configuration repositories was first proposed in [Eis96].

Alternatively, the list architecture can be described as a so-called GenVoca grammar (see Fig. 5). The vertical bar separates alternatives and parameters are enclosed in square brackets, e.g. `List` is either `TracedList` parameterized by `OptCounterList` or `OptCounterList`. The names exported by the configuration repository `Config` are listed below it. `ReturnType` is the final type of a configured list.

We can implement `PtrList` in C++ as follows:[4]

```
template<class Config_>
class PtrList
{
public:
   //make Config available as a member type
   typedef Config_ Config;

private:
   //retrieve needed types from the configuration repository
   typedef typename Config::ElementType ElementType;
   typedef typename Config::SetHeadElementType SetHeadElementType;
   typedef typename Config::ReturnType ReturnType;

   typedef typename Config::Destroyer Destroyer;
   typedef typename Config::TypeChecker TypeChecker;
   typedef typename Config::Copier Copier;

public:
   PtrList(SetHeadElementType& h, ReturnType *t = 0) :
      head_(0), tail_(t)
   {  setHead(h); }

   ~PtrList()
   {  Destroyer::destroy(head_); }

   void setHead(SetHeadElementType& h)
   {  TypeChecker::check(h);
      head_ = Copier::copy(h);
   }

   ElementType& head()
   {  return *head_; }

   void setTail(ReturnType *t)
   {  tail_ = t; }

   ReturnType *tail() const
   {  return tail_; }

private:
   ElementType* head_;
   ReturnType *tail_;
```

[4] The keyword `typename` is required by ANSI C++ to tell the compiler that a member of a template parameter is expected to be a type.

```
};
```

PtrList has two instance variables: head_ and tail_. head_ points to the head element and tail_ points to the rest of the list. Please note that the type of tail_ is ReturnType, which is the final type of the list. We cannot use PtrList as the type of tail_ since we will derive a list with counter and a list with tracing from PtrList. Whenever we derive classes from PtrList and want to create instances of the *most refined type*, tail_ has to be of the most refined type. Since this type is unknown in PtrList, PtrList retrieves it from the configuration repository (which is passed to PtrList as the parameter Config_).[5]

The next interesting point about PtrList is that methods setting the head (i.e. the constructor and setHead()) use the type SetHeadElementType&. This type should be either ElementType& or const ElementType&, depending whether the list stores references to elements or copies of elements. Since this is unknown in PtrList, PtrList retrieves SetHeadElementType from the configuration repository.

Finally, PtrList delegates some of its work to other components: The destructor delegates its job to the type name Destroyer, which is retrieved from the configuration repository. Similarly, setHead() delegates type checking and copying to TypeChecker and Copier, respectively. The type names Destroyer, TypeChecker, and Copier may point to different components, as specified in Fig. 5.

We have two destroyer components: ElementDestroyer and EmptyDestroyer. ElementDestroyer deletes an element and is used if a list keeps element copies or owned references. It is implemented as a struct rather than a class since it defines only one public operation and struct members are public by default:

```
template<class ElementType>
struct ElementDestroyer
{  static void destroy(ElementType *e)
   {  delete e; }
};
```

EmptyDestroyer is used if a list keeps external references to the original elements. EmptyDestroyer does nothing. Since its destroy() method is implemented inline, an optimizing compiler will remove any calls to this method.

```
template<class ElementType>
struct EmptyDestroyer
{  static void destroy(ElementType *e)
   {} //do nothing
};
```

DynamicTypeChecker is used to assure that a monomorphic list contains elements of one type only:

```
template<class ElementType>
struct DynamicTypeChecker
{  static void check(const ElementType& e)
```

[5] Our example demonstrates how configuration repositories can help in typing recursive classes, i.e. classes that are used directly or indirectly in their own definition (e.g. a list).

```
   {  assert(typeid(e)==typeid(ElementType)); }
};
```

`EmptyTypeChecker` is used in a polymorphic list and it does nothing:

```
template<class ElementType>
struct EmptyTypeChecker
{  static void check(const ElementType& e)
   {}
};
```

`PolymorphicCopier` copies an element by calling the virtual function `clone()`:

```
template<class ElementType>
struct PolymorphicCopier
{  static ElementType* copy(const ElementType& e)
   {  return e.clone(); } //call a virtual clone()
};
```

`MonomorphicCopier` copies an element by calling its copy constructor:

```
template<class ElementType>
struct MonomorphicCopier
{  static ElementType* copy(const ElementType& e)
   {  return new ElementType(e); } //call copy constructor
};
```

Finally, `EmptyCopier` simply returns the original element:

```
template<class ElementType>
struct EmptyCopier
{  static ElementType* copy(ElementType& e) //pass by non-const
                                            //reference!
   {  return &e; } //simply return the original
};
```

Next, we need the two wrappers `LenList` and `TracedList` implementing length counter and tracing, respectively. `LenList` is implemented as an *inheritance-based wrapper*, i.e. a template class derived from its parameter.[6] It overrides the method `setTail()` to keep track of the list length and adds the method `length()`. Please note that the component retrieves the configuration repository from its parameter:

```
template<class BaseList>
class LenList : public BaseList
{
public:
   //retrieve the configuration repository
   typedef typename BaseList::Config Config;

private:
   //retrieve the necessary types from the repository
   typedef typename Config::ElementType ElementType;
   typedef typename Config::SetHeadElementType SetHeadElementType;
   typedef typename Config::ReturnType ReturnType;
   typedef typename Config::LengthType LengthType;

public:
   LenList(SetHeadElementType& h, ReturnType *t = 0) :
      BaseList(h,t), length_(computedLength())
   {}

   void setTail(ReturnType *t)
```

---

[6] Inheritance-based wrappers are useful whenever we only want to override few methods and inherit the remaining ones. Otherwise, we can use *aggregation-based wrappers*.

```
   {  BaseList::setTail(t);
      length_ = computedLength();
   }

   const LengthType& length() const
   {  return length_; }

private:
   LengthType computedLength() const
   {  return tail()  ? tail()->length()+1
                     : 1;
   }

   LengthType length_;
};
```

`TracedList` is also implemented as an inheritance-based wrapper:

```
template<class BaseList>
class TracedList : public BaseList
{
public:
   typedef typename BaseList::Config Config;

private:
   typedef typename Config::ElementType ElementType;
   typedef typename Config::SetHeadElementType SetHeadElementType;
   typedef typename Config::ReturnType ReturnType;

public:
   TracedList(SetHeadElementType& h, ReturnType *t = 0) :
      BaseList(h,t)
   { }

   void setHead(SetHeadElementType& h)
   {  cout << "setHead(" << h << ")"<< endl;
      BaseList::setHead(h);
   }

   ElementType& head()
   {  cout << "head()"<< endl;
      return BaseList::head();
   }

   void setTail(ReturnType *t)
   {  cout << "setTail(t)"<< endl;
      BaseList::setTail(t);
   }
};
```

5.3 Composing List Components

Now that we have created all components the question is how to produce a concrete configuration, e.g. a monomorphic list keeping copies of elements of type `Person` and providing a length counter and tracing. For this purpose we have to define a configuration repository with the necessary configuration information (see Fig. 6).

```
struct TracedCopyMonoBaseLenListConfig
{
   //provide the different type names required by the components
   typedef Person                              ElementType;
   typedef const ElementType                   SetHeadElementType;
   typedef int                                 LengthType;
   typedef ElementDestroyer<ElementType>       Destroyer;
   typedef DynamicTypeChecker<ElementType>     TypeChecker;
   typedef MonomorphicCopier<ElementType>      Copier;

   //wrap PtrList into LenList and TracedList
   typedef
      TracedList<
        LenList<
          PtrList<TracedCopyMonoBaseLenListConfig> > > ReturnType;
};
//define a short name for our list
typedef TracedCopyMonoBaseLenListConfig::ReturnType MyList;
```

Fig. 6 Sample configuration repository

TracedCopyMonoBaseLenListConfig is implemented as a so-called *traits class* [Mye95], i.e. a class aggregating a number of types and constants to be passed to a template as a parameter. MyList contains the type we want to produce. The gray boxes and arrows visualize the flow of types from the configuration repository to PtrList to LenList and, finally, to TracedList. Among those types is ReturnType, which represents the final list type. The list components can retrieve it from the repository. It is interesting to note the circularity involving ReturnType: The components are actually composed in the configuration repository and the repository is passed to the components.

It is also worth noting that, since all variation points were static, we used static parameterization and inlining to avoid the unnecessary cost of virtual function calls and the cost of function calls for small functions. Templates and inlining allow composing code fragments without any runtime cost. As a result, the composed types are as efficient as manually coded concrete variants.

Without counting ElementType and LengthType, the feature diagram in Fig. 3 defines 24 different list configurations. We can define a configuration repository for each of them, e.g. a polymorphic list keeping external references to original elements of type Person or derived types:

```
struct RefPolyBaseListConfig
{
   typedef Person                              ElementType;
   typedef ElementType                         SetHeadElementType;
   typedef EmptyDestroyer<ElementType>         Destroyer;
   typedef EmptyTypeChecker<ElementType>       TypeChecker;
   typedef EmptyCopier<ElementType>            Copier;

   typedef PtrList<RefPolyBaseListConfig>      ReturnType;
};
typedef RefPolyBaseListConfig::ReturnType RefPolyBaseList;
```

Writing down the configuration repositories for all 24 list variants (we would parameterize ElementType and LengthType for this purpose) is rather tedious.

The situation is worse if we have concepts with more variable features. For example, we implemented a configurable matrix component, which covers 1840 different matrix variants (see Section 7). Writing down all configuration repositories in this case is impracticable. An alternative would be to let the application programmer write the configuration repositories for the types he or she needs. This approach has its drawbacks. Writing the lengthy configuration repositories is error prone and tedious. Furthermore, configuration repositories mention implementation detail which is not relevant at the level of the feature diagram in Fig. 3. For example, we have to explicitly define `SetHeadElementType`, although it is not part of the feature diagram. Similarly, selecting the appropriate destroyer, type checker, and copier is an implementation detail. These choices, although they automatically follow from the selected abstract features, have to be programmed manually! A much better solution is to generate the configuration repositories from abstract specifications. This is described in the following section.

5.4 Generating Lists from Abstract Specifications

Our goal now is to generate list configurations from abstract specifications, i.e. sets of features defined in Fig. 3. This requires writing some code which is executed at compile time. We can use *template metaprogramming* for this purpose [Vel95a, CE98, Cza98]. Template metaprograms consist of class templates operating on numbers and/or types as data.[7] Algorithms are expressed using template recursion as a looping construct and class template specialization as a conditional construct. Template recursion involves the direct or indirect use of a class template in the construction of its own member type or member constant. In other words, C++ templates constitute a Turing-complete sublanguage of C++, which is interpreted by the compiler at compile time. As an example, consider a template which computes the factorial of a non-negative integral number:

```
template<int n>
struct Factorial
{  enum { RET = Factorial<n-1>::RET * n };
};

//the following template specialization terminates the recursion
template<>
struct Factorial<0>
{  enum { RET = 1 };
};
```

We can use this class template as follows:

```
void main()
{  cout << Factorial<7>::RET << endl; //prints 5040
}
```

---

[7] The ability to use the template instantiation process to perform compile-time computations was observed by Erwin Unruh. He wrote a small C++ program generating prime numbers at compile time, which the compiler would output as a sequence of warnings. This program [Unr94] was circulated as a “curiosity” during an ANSI C++ standardization committee meeting in 1994.

The important point about this program is that `Factorial<7>` is instantiated at compile time. During the instantiation, the compiler also determines the value of `Factorial<7>::RET` (`RET` is an abbreviation for `RETURN`). Thus, the code generated for this `main()` function by the C++ compiler is the same as the code generated for the following `main()`:

```
void main()
{  cout << 5040 << endl; //prints 5040
}
```

We can regard `Factorial<>` as a *metafunction* which is evaluated at compile time. `Factorial<>` is a metafunction since, at compilation time, it computes constant data of a program which has not been generated yet. Template metafunctions can also take types as parameters and return types. The following metafunction takes a Boolean and two types as its parameters and returns a type:[8]

```
template<bool cond, class ThenType, class ElseType>
struct IF
{  typedef ThenType RET;
};

template<class ThenType, class ElseType>
struct IF<false, ThenType, ElseType>
{  typedef ElseType RET;
};
```

This function corresponds to an if statement: it has a condition parameter, a "then" parameter, and an "else" parameter. If the condition is `true`, it returns `ThenType` in `RET`. This is encoded in the base definition of the template. If the condition is `false`, it returns `ElseType` in `RET`. Thus, this metafunction can be viewed as a meta-control statement. Implementing other meta-control statements such as switch and looping constructs is also possible [CE98]. Indeed, even a simple Lisp was implemented using these techniques [CE98, Cza98].

We can use `IF<>` to implement a metafunction which takes a number of flags representing the features from Fig. 3 and returns a ready-to-use list type. First, we define the flags representing the list features:

```
enum Ownership {ext_ref, own_ref, cp};
enum Morphology {mono, poly};
enum CounterFlag{with_counter, no_counter};
enum TracingFlag{with_tracing, no_tracing};
```

Next, we present the metafunction `LIST_GENERATOR<>` implementing a *configuration generator*, which takes an abstract description of a list and generates a ready-to-use list type. Using `LIST_GENERATOR<>`, we can declare a monomorphic list keeping copies of elements of type `Person` and providing a length counter and tracing as follows:

```
LIST_GENERATOR<Person, cp, mono, with_counter, with_tracing>::RET list1;
```

A polymorphic list keeping external references to original elements of type `Person` or derived types can be declared as follows:

```
LIST_GENERATOR<Person, ext_ref, poly>::RET list2;
```

---

[8] A different `IF<>` implementation which does not require partial template specialization is described in [Cza98].

Here is the definition of the metafunction:

```
template<
   class ElementType_ = int,
   Ownership ownership = cp,
   Morphology morphology = mono,
   CounterFlag counterFlag = no_counter,
   TracingFlag tracingFlag = no_tracing,
   class LengthType_ = int
>
class LIST_GENERATOR
{
public:
   typedef LIST_GENERATOR<
      ElementType_,
      ownership,
      morphology,
      counterFlag,
      tracingFlag,
      LengthType_> Generator;

private:
   enum {
      isCopy      = ownership==cp,
      isOwnRef    = ownership==own_ref,
      isMono      = morphology==mono,
      hasCounter  = counterFlag==with_counter,
      doesTracing = tracingFlag==with_tracing };

   typedef
      IF<isCopy || isOwnRef,
            ElementDestroyer<ElementType_>,
            EmptyDestroyer<ElementType_>
      >::RET Destroyer_;

   typedef
      IF<isMono,
            DynamicTypeChecker<ElementType_>,
            EmptyTypeChecker<ElementType_>
      >::RET TypeChecker_;

   typedef
      IF<isCopy,
            IF<isMono,
                  MonomorphicCopier<ElementType_>,
                  PolymorphicCopier<ElementType_> >::RET,
            EmptyCopier<ElementType_>
      >::RET Copier_;

   typedef
      IF<isCopy,
            const ElementType_,
            ElementType_
      >::RET SetHeadElementType_;

   typedef PtrList<Generator> List;

   typedef
      IF<hasCounter,
            LenList<List>,
            List
      >::RET List_with_counter_or_not;

   typedef
```

```
      IF<doesTracing,
            TracedList<List_with_counter_or_not>,
            List_with_counter_or_not
      >::RET List_with_tracing_or_not;
public:
   typedef List_with_tracing_or_not RET;

   struct Config
   {
      typedef ElementType_         ElementType;
      typedef SetHeadElementType_  SetHeadElementType;
      typedef Destroyer_           Destroyer;
      typedef TypeChecker_         TypeChecker;
      typedef Copier_              Copier;
      typedef LengthType_          LengthType;
      typedef RET                  ReturnType;
   };
};
```

`LIST_GENERATOR<>` evaluates the input flags, computes the types for the configuration repository, wraps `PtrList` (if necessary), and returns the final list type in `RET`. The last part of `LIST_GENERATOR<>` is the configuration repository. Please note that we pass `Generator` to `PtrList` as parameter. Since `Config` is a member of `Generator`, `PtrList` can retrieve `Config` from `Generator`. For this reason, we need to slightly modify two lines of `PtrList` (the modifications are highlighted):

```
template<class Generator>
class PtrList
{
public:
   typedef typename Generator::Config Config;
//the rest as previously
//...
```

An important aspect of the separation between the problem-space-oriented feature description and the implementation components is that we can make useful extensions to the components (e.g. modify the component structure or add new components) without the need to change existing client code. This is certainly possible as long as the abstract feature space can still be mapped on the new components. Moreover, we can even make certain extensions to the feature space without the need to change existing client code. For example, we could append new parameters, e.g. memory allocation, to the parameter list expected by the generator. By choosing appropriate defaults for the new parameters, existing calls to the generator will still work properly. Similarly, if we model all features as template structs, we can also add new nested features (cf. the following section).

6 Extensions

The previous section demonstrated the basic techniques using a very simple example. Applying these techniques to larger problems requires several extensions:

- *Nested features*: Our sample generator expects a flat list of features although feature diagrams are trees. This was acceptable for this small example, but in general configuration generators accept tree-like structures. We can represent tree-like feature structures in C++ using types and templates. For example, we

could model `counterFlag` as a type parameter with the values `no_counter` and `with_counter<>`. `no_counter` would be a struct and `with_counter<>` a template struct expecting `LengthType` as its parameter.

- *Multistage configuration generators*: Large feature models may contain many constraints and default dependency rules. In this case, a configuration generator consists of several stages: specification completion stage (computes defaults for the unspecified features based on default dependency rules and constraints), feature combination checking stage (checks whether the feature combinations satisfy the constraints), and component assembly stage (assembles components into the final type). The dependency rules and constraints are specified using a kind of decision tables. For this reason, we implemented a table evaluation metafunction, which allows us to directly type in the tables in the C++ source code [Cza98, Kna98]. This function utilizes both `IF<>` and template recursion.
- *Nested configuration repositories*: Avoiding name clashes in the configuration repository may require introducing separate name scopes within the repository. For example, two different components retrieve the type name `ElementType`, but each of them should be supplied a different type. This can be resolved by providing a separate name scope for each of these components in the repository. We can model such nested name scopes in C++ as nested classes.
- *Metafunctions as part of configuration repositories*: Sometimes one component is used more than once in a configuration and each instance needs different configuration parameters. In this case, the component does not retrieve the required type from the configuration repository directly, but it retrieves a metafunction. Each instance can then supply a different parameter to the metafunction to compute the needed type. In C++, class templates can be defined as members of other classes. This way it is possible to pass around a metafunction as a type, which corresponds to the idea of higher-order metafunctions.
- *Configuration repository as a part of the generated type*: Each component exports the configuration repository under the name `Config`. Thus, we can also retrieve `Config` from the final type. e.g. `MyList::Config`. This feature can be used by other generators, e.g. for generating customized algorithms operating on the generated types. An algorithm generator can retrieve the properties of a type from its `Config` (e.g. `ElementType`, `Ownership`, etc.) and use this information to generate optimized algorithms. For example, we have used the configuration repository of a matrix type in order to generate optimized matrix operation code using expression templates [Cza98].
- *Traits templates for encoding metainformation*: Metainformation about types can be encoded as *traits templates* [Mye95], which basically correspond to metafunctions taking types as parameter and returning their properties. For example, our polymorphic copier in Section 5.2 assumes that the element type provides the virtual `clone()` method. If a particular element type does not provide this method, we can use an adapter. In this case, we could use a traits template on element type to retrieve the appropriate (user-provided) adapter.

Furthermore, we could use a traits template to make sure that `DynamicTypeChecker` is used only for types having a virtual function table.

We used the above extensions in the implementation of two applications described in the following section.

The generator approach described here can be also used to synthesize frameworks. Frameworks can be modeled as compositions of collaborations and the latter can be implemented as *mixin layers* [SB98]. In C++, a mixin layer may be implemented as a class containing a number of nested classes. Each of the nested classes implements a particular role. The nested classes can inherit from their parameters, so that they can be used to extend other classes. A mixin layer takes another layer as its parameter, accesses the classes nested in the parameter and uses them as superclasses of some of its own nested classes (see [SB98] for details). Just as we used our configuration generator to compose parameterized classes, we can also use it to compose mixin layers.

7 Applications

The techniques described in previous sections were use to develop two medium size libraries demonstrating their applicability to generating efficient abstract data types and algorithms:

- *Generative Matrix Computation Library* (GMCL) [GMCL, Cza98, Neu98] contains a matrix generator (in the style of our list generator from Section 5.4) able to generate matrices with a selected combination of features such as element types (real numbers), density (dense and sparse), storage formats (row- and column-wise, several sparse formats), memory allocation (dynamic and static), error checking (bounds, compatibility, memory allocation), and operations (addition, subtraction, multiplication). GMCL contains another generator for generating efficient implementations of matrix expressions (e.g. "(A+B)*(C+D)"). The expression generator is based on expression templates [Vel95b]. It reads out the properties of the operands from their configuration repositories in order to generate optimized code. The C++ implementation of the matrix component comprises 7500 lines of C++ code (6000 lines for the configuration generator and the matrix components and 1500 lines for the operations). The matrix configuration feature diagram covers more than 1840 different kinds of matrices. Despite the large number of provided matrix variants, the performance of the generated code is comparable with the performance of manually coded variants. This is achieved by the exclusive use of static binding, which is often combined with inlining.
- *Generative matrix factorization library* [Kna98] contains a generator synthesizing different instances of the LU factorization algorithm family (e.g. Gauss, Cholesky, LDL^T) with different pivoting strategies (e.g. partial, full, symmetric, diagonal) and for different matrix shapes. The different parts of the algorithms are implemented as methods of class templates organized in a layered architecture. The templates are configured by a configuration generator.

8 Related Work

Variability modeling The need for variability modeling in framework design has been recognized in the form of "hot spots" [Pre95]. Hot spots represent variation points. Unfortunately, they are not supported in current OOA/D methods. Furthermore, they make no distinction between different kinds of variation points (e.g. dimensions, dimensions with optional features, extension points, etc. [Cza98]) and do not model the information contained in feature models. Reuse-Driven Software Engineering Business (RSEB) [JGJ98] extends UML with the concept of variation points and defines a reuse-driven development process. However, it still lacks feature modeling. Feature modeling is the corner stone of Domain Engineering (DE) and efforts aimed at integrating OO and DE methods [CN98, GFA98, Cza98] augment OO modeling techniques with feature modeling.

Layered Designs and Fragment-Based Component Models GenVoca is a layered architecture model based on parameterized layers of refinement [BO92]. Recent work by Smaragdakis and Batory [SB98] views GenVoca layers as so-called *mixin layers*, i.e. layers containing classes whose superclasses are parameters. Parameterized inheritance has also been used to express collaboration-based designs [VHN96]. The technique of exchanging types between components at compile time is extensively used in the Standard Template Library (STL) [MS96]. Fragment-based designs have been studied in the context of OO, among others, in [Pre97, Mez97, ML98]. Our work extends these approaches with configuration repositories, which among others provide an effective approach for typing synthesized recursive classes. Furthermore, our configuration generator is capable of synthesizing layered and fragment-based designs from abstract specifications.

Metaprogramming There is a large body of work on static metaprogramming for code composition. Most of this work is has been done in the context of procedural languages such as C (e.g. [SG97, Eng97]). There are also examples of static metaprogramming systems for C++, e.g. Open C++ [Chi95] and MPC++ [IHS+96]. All of these systems require special language extensions. Our approach, on the other hand, uses standard C++ language features and thus is widely available. Template metaprogramming has been used to develop a number of libraries including Blitz++ [Vel97], POOMA [POOMA], and MTL [SL98]. Unfortunately, it lacks debugging and error-reporting facilities. An example of a commercial metaprogramming environment supporting the full development cycle is Intentional Programming (IP) [Sim96]. IP is currently under development at Microsoft Research.

9 Conclusions and Outlook

The development of truly reusable software requires the parameterization of a multitude of design decisions. We showed that modeling the variability of domain concepts is inadequately supported by current OO modeling notations. We also showed that this problem can be addressed by using feature models in addition to OO models. Furthermore, we demonstrated that the implementation of feature models requires three ingredients: domain-level interface (e.g. a domain-specific language, a domain-specific GUI, etc.) allowing the application programmer to describe concepts

at the abstract domain level, configuration knowledge mapping between abstract descriptions and concrete component configurations, and elementary implementation components, which can be configured in a vast number of ways. We showed that the design of the problem and the solution space starts with feature modeling and the solution space is structured according to some appropriate architecture (in our example, we used a layered architecture). We also found it important to have a direct programming language support for these concepts. We showed concrete examples in C++; however, the concepts are not limited to C++. Indeed, the examples demonstrate the importance of parameterized inheritance in avoiding rigid inheritance hierarchies and the value of generic, STL-style techniques for implementing efficient and highly configurable components. The newly introduced concept of configuration repositories allows the separation between the components and the configuration knowledge and also facilitates an efficient approach to typing recursive classes. Finally, the built-in metaprogramming capabilities of C++ allow the configuration generators to be part of the same library as the implementation components. The presented example concentrated on static configuration and binding, but similar designs based on dynamic configuration and binding can be implemented in C++ and in other languages (e.g. Smalltalk, CLOS, Java[9]). Indeed, we want to parameterize configuration time and binding time. The latter is easily done in C++ [Eis97], but the former requires the ability to write metacode which can be executed both by the compiler and at runtime – a feature not supported by current languages. One of the conclusions of our work is the need for industrial strength metaprogramming environments. Template metaprogramming (TM) has the important advantage that is readily available to users as a built-in part of C++. But since TM is a child of accident rather than the result of conscious language design, it suffers from several deficiencies in the areas of debugging and error-reporting, code readability, long compilation times, various compiler limits, and portability [Cza98, Neu98, Kna98]. Currently, the size of template metaprograms is limited by compiler limits, compilation times, and debugging problems. However, compiler limits and portability problems will decrease as more and more compiler vendors adopt the new C++ ISO standard. Adequate metaprogramming support opens new possibilities to raise “the level” of programming using domain-specific abstractions. For example, domain-specific abstractions in Intentional Programming [Sim96] are active at any time (including programming and compilation time) and generate efficient, optimized code. This perspective forces us to redefine the conventional interaction between compilers, libraries, and applications and to acknowledge the need for *active libraries* [CEG+98], which “are not passive collections of routines or objects, as are traditional libraries, but take an active role in generating code. Active libraries provide abstractions and can optimize those abstractions themselves. They may generate components, specialize algorithms, optimize code, automatically configure and tune themselves for a target machine, and check source code for correctness. They may also describe themselves to tools such as profilers and debuggers in an intelligible way.” The effective application of metaprogramming in software engineering requires new analysis and design

---

[9] Java does not support parameterized inheritance. Therefore, whenever we need a parameterized relationship in Java, we have to use dynamic parameterization.

approaches. The integration of modeling and implementation technologies based on domain-specific abstractions and metaprogramming into a coherent paradigm is the goal of the emerging area of Generative Programming [CE99, CEG+98, Eis97].

Note: The source code for the list example is available at http://nero.prakinf.tu-ilmenau.de/~czarn/ecoop99

References

[AM97] M. Aksit and S. Matsuoka, (Eds.). *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP '97)*, Springer-Verlag 1997

[BO92] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. In *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, October 1992, pp. 355-398

[CE98] K. Czarnecki and U. Eisenecker. Template-Metaprogramming, http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm

[CE99] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications.* To appear, Addison-Wesley, 1999

[CEG+98] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative Programming and Active Libraries. Submitted for publication, 1998

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'95), ACM SIGPLAN Notices*, vol. 30, no. 10, 1995, pp. 285-299, http://www.softlab.is.tsukuba.ac.jp/~chiba/openc++.html

[CN98] S. Cohen and L. M. Northrop. Object-Oriented Technology and Domain Analysis. In [DP98], pp. 86-93

[Cza98] K. Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Ph.D. thesis, Technische Universität Ilmenau, Germany, 1998, see http://nero.prakinf.tu-ilmenau.de/~czarn/

[DP98] P. Devanbu and J. Poulin, (Eds.). *Proceedings of the Fifth International Conference on Software Reuse (Victoria, Canada, June 1998).* IEEE Computer Society Press, 1998

[Eis96] U. Eisenecker. Generatives Programmieren mit C++. In *OBJEKTspektrum*, No. 6, November/December 1996, pp. 79-84

[Eis97] U. Eisenecker. Generative Programming (GP) with C++. In *Proceedings of Modular Programming Languages (JMLC'97, Linz, Austria, March 1997)*, H. Mössenböck, (Ed.), Springer-Verlag, Heidelberg 1997, pp. 351-365

[Eng97] D. R. Engler. Incorporating application semantics and control into compilation. In *Proceedings USENIX Conference on Domain-Specific Languages (DSL'97)*, 1997

[GFA98] M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In [DP98], pp. 76-85, see http://www.intecs.it

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[GMCL] Homepage of the Generative Matrix Computation Library at http://nero.prakinf.tu-ilmenau.de/~czarn/gmcl/

[Gog96] J. A. Goguen. Parameterized Programming and Software Architecture. In *Proceedings of the Fourth International Conference on Software Reuse*, April 23-26, Orlando, Florida. IEEE Computer Society Press, Los Alamitos, California, 1996, pp. 2-10

[IHS+96] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and Implementation of Metalevel Architecture in C++ – MPC++ approach. In *Proceedings of Reflection'96*, 1996

[JGJ98] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Longman, May 1997

[Jul98] E. Jul, (Ed.). *Proceedings of the 12th European Conference Object-Oriented Programming (ECOOP'98)*, LNCS 1445, Springer-Verlag, 1998

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In [AM97], pp. 220-242

[Kna98] J. Knaupp. Algorithm Generators: A First Experience, see http://nero.prakinf.tu-ilmenau.de/~czarn/generate/stja98/knaupp.zip

[Mez97] M. Mezini. Dynamic Object Evolution Without Name Collisions. In [AM97], pp. 190-219

[ML98] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proceedings of the Conference on Object-Oriented Programming Languages and Applications (OOPSLA '98)*, 1998

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, Massachusetts, 1996

[Mye95] N.C. Myers. Traits: a new and useful template technique. In *C++ Report*, June 1995, see http://www.cantrip.org/traits.html

[Neu98] T. Neubert. Anwendung von generativen Programmiertechniken am Beispiel der Matrixalgebra. Diplomarbeit, Technische Universität Chemnitz, 1998, also see [GMCL]

[POOMA] POOMA: Parallel Object-Oriented Methods and Applications. A framework for scientific computing applications on parallel computers. Available at http://www.acl.lanl.gov/pooma

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995

[Pre97] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In [AM97], pp. 419-443

[Ree96] T. Reenskaug with P. Wold and O.A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning, 1996

[SB98] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In [Jul98], pp. 550-570

[SCK+96] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang. Organization Domain Modeling (ODM) Guidebook, Version 2.0. Informal Technical Report for STARS, STARS-VC-A025/001/00, June 14, 1996, see http://direct.asset.com

[SG97] J. Stichnoth and T. Gross. Code composition as an implementation language for compilers. In *Proceedings USENIX Conference on Domain-Specific Languages (DSL'97)*, 1997

[Sim96] C. Simonyi. Intentional Programming — Innovation in the Legacy Age. Position paper presented at IFIP WG 2.1 meeting, June 4, 1996, see http://www.research.microsoft.com/research/ip/

[SL98] J. G. Siek and A. Lumsdaine. A Rational Approach to Portable High Performance: The Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (FAST) Library. In Proceedings of the ECOOP'98 Workshop on Parallel Object-Oriented Computing (POOSC'98), 1998, see http://www.lsc.nd.edu/

[Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994

[Unr94] E. Unruh. Prime number computation. ANSI X3J16-94-0075/ISO WG21-462, 1994

[Vel95a] T. Veldhuizen. Using C++ template metaprograms. In *C++ Report*, vol. 7, no. 4, May 1995, pp. 36-43, see http://monet.uwaterloo.ca/blitz/

[Vel95b] T. Veldhuizen. Expression Templates. In *C++ Report*, vol. 7 no. 5, June 1995, pp. 26-31, see http://monet.uwaterloo.ca/blitz/

[Vel97] T. Veldhuizen. Scientific Computing: C++ versus Fortran. In *Dr. Dobb's Journal*, November 1997, pp. 34-41, see http://monet.uwaterloo.ca/blitz/

[VHN96] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, 1996, pp. 359-369

A Core Calculus of Classes and Mixins

Viviana Bono[1], Amit Patel[2], and Vitaly Shmatikov[2]

[1] Dipartimento di Informatica dell'Università di Torino, C.so Svizzera 185
10149 Torino, Italy, bono@di.unito.it
(currently at the School of Computer Science, The University of Birmingham
Birmingham B15 2TT, United Kingdom
v.bono@cs.bham.ac.uk)

[2] Computer Science Department, Stanford University
Stanford, CA 94305-9045, U.S.A., {amitp,shmat}@cs.stanford.edu

Abstract. We develop an imperative calculus that provides a formal model for both single and mixin inheritance. By introducing classes and mixins as the basic object-oriented constructs in a λ-calculus with records and references, we obtain a system with an intuitive operational semantics. New classes are produced by applying mixins to superclasses. Objects are represented by records and produced by instantiating classes. The type system for objects uses only functional, record, and reference types, and there is a clean separation between subtyping and inheritance.
Keywords: Object-oriented language, mixin, class, inheritance, calculus, operational semantics, type system.

1 Introduction

Mixins (classes parameterized over superclasses) have become a focus of active research both in the software engineering [43, 41, 25] and programming language design [11, 12, 10, 35, 30] communities. Mixin inheritance has been shown to be an expressive alternative to multiple inheritance and a powerful tool for implementing reusable class hierarchies. However, there has been a dearth of formal calculi to provide a theoretical foundation for mixin inheritance and, in particular, few attempts have been made to use mixins as the basic inheritance construct in the core calculus. Although mixin inheritance is easy to formalize in an untyped setting, static type checking of mixins at the time of declaration (as opposed to the time of mixin use) is more difficult. In addition, many approaches to mixins do not address the modular construction of objects, including initialization of fields.

While popular object-oriented languages such as C++ [42] and JAVA [3] are overwhelmingly class-based, most previous core calculi for object-oriented languages were based on objects. In our framework, classes and mixins are basic constructs. The decision to directly include classes in a core calculus reflects many years of struggle with object-based calculi. In simple terms, there is a fundamental conflict between inheritance and subtyping of object types [20, 14, 6, 28]. Our calculus resolves this conflict by supporting class inheritance without class subtyping and object subtyping without object extension. The separation between

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 43–66, 1999.
© Springer-Verlag Berlin Heidelberg 1999

inheritance (an operation associated with classes) and run-time manipulation of objects allows us to represent objects by records and keep the type system for objects simple, involving only functional, record, and reference types. In particular, we do not need polymorphic object types or recursive *MyType*.

An important advantage of our type system is that it gives types to mixin declarations and mixin applications separately. The actual class to which the mixin is applied may have a "richer" type than that expected by the mixin. For example, it may have more methods, or the types of its methods may be subtypes of those assumed when typing the mixin. This facilitates modular development of class hierarchies, promotes reuse of mixins, and enables the programmer to use a single mixin to add the same functionality to a wide variety of classes. Name clashes between mixins and classes to which they are applied are detected and resolved at the time of mixin application.

We discuss design motivations and tradeoffs, and give a brief overview of the core calculus in section 2. We then present the syntax of the calculus (section 3), its operational semantics (section 4), and the type system (section 5). Finally, we compare our calculus with other object-oriented calculi and indicate directions for future research.

A simpler version of the calculus described in this paper was presented at MFPS '99 [7]. The calculus of [7] supports conventional single inheritance instead of mixins.

2 Design of the Core Calculus

In this section, we present our design motivations, discuss tradeoffs involved to designing calculi for object-oriented languages, give a short overview of our calculus, and present an example illustrating mixin usage.

2.1 Design Motivations

Our goal is to design a simple class-based calculus that correctly models the basic features of popular class-based languages and reflects modular programming techniques commonly employed by working programmers. Modular program development in a class-based language involves minimizing code dependencies such as those between a superclass and its subclasses, and between a class implementation and object users. Our calculus minimizes dependencies by directly supporting data encapsulation, mixin inheritance, structural subtyping, and modular object creation.

Data encapsulation. We use the C++ terminology (private, protected, and public) for levels of encapsulation. Unlike C++ and some approaches to encapsulation in object calculi such as existential types, our levels of encapsulation describe *visibility*, and not merely *accessibility*. For example, in our calculus even the names of private items are invisible outside the class in which they are defined. We believe that this is a better approach since *no* information about data representation is revealed — not even the number and names of fields. One of the

benefits of using visibility-based encapsulation is that no conflicts arise if both the superclass and the subclass declare a private field of the same name. Among other advantages, this allows the same mixin to be applied twice (see the example in section 2.4).

Mixin inheritance. A *mixin* is a class definition parameterized over the superclass. The decomposition of ordinary inheritance into mixins plus mixin application is similar to the decomposition of let binding into functions plus function application. A mixin can be viewed as a function that takes a class and derives a new subclass from it. The same mixin can be applied to many classes, obtaining a family of subclasses with the same set of methods added and/or replaced. By providing an abstraction mechanism for inheritance, mixins remove the dependency of the subclass on the superclass, enabling modular development of class hierarchies — *e.g.*, a subclass can be implemented before its superclass has been implemented. Mixin inheritance can be used to model single inheritance and many common forms of multiple inheritance [11, 9].

Mixins were first introduced in the Flavors system [38] and CLOS [33], although as a programming idiom rather than a formal language construct. Our calculus is an attempt to formalize mixins as the basic mechanism underlying all inheritance. To ensure that mixin inheritance can be statically type checked, our calculus employs constrained parameterization. From each mixin definition the type system infers a constraint specifying to which classes the mixin may be applied so that the resulting subclass is type-safe. The constraint includes both positive (which methods the class must contain) and negative (which methods the class may not contain) information. The actual class to which the mixin is applied does not have to match the constraint exactly. It may have more methods than required by the positive part of the constraint, and the types of the required methods may be different from those specified by the constraint as long as the resulting subclass is type-safe.

We believe that new and redefined methods should be distinguished in the mixin implementation. From the implementor's viewpoint, a new method may have arbitrary behavior, while the behavior of a redefined method must be "compatible" with that of the old method it replaces. Having this distinction in the syntax of our calculus helps mixin implementors avoid unintentional redefinitions of superclass methods and facilitates generation of the mixin's superclass constraint (see section 4). It also helps resolve name clashes when the mixin is applied. Suppose the mixin and the class to which it is applied both define a method with the same name. If the mixin method is marked as *redefined*, then it is put in the resulting subclass (subject to type compatibility with the replaced method). If the mixin method is marked as *new*, the type system signals an error.

Structural subtyping. As in most popular object-oriented languages, objects in our calculus can only be created by instantiating a class. In contrast to C++, where an object's type is related to the class from which it was instantiated and subtyping relations apply only to object instantiated from the same class hierarchy, we made a deliberate design decision to use *structural subtyping* in

order to remove the dependency of object users on class implementation. Objects created from unrelated classes can be substituted for each other if their types satisfy the subtyping relation.

Modular object construction. Class hierarchies in a well-designed object-oriented program must not be fragile: if a superclass implementation changes but the specification remains intact, the implementors of subclasses should not have to rewrite subclass implementations. This is only possible if object creation is modular. In particular, a subclass implementation should not be responsible for initializing inherited fields when a new object is created, since some of the inherited fields may be private and thus invisible to the subclass. Also, the definitions of inherited fields may change when the class hierarchy changes, making the subclass implementation invalid. Instead, the object construction system should call a class *constructor* to provide initial values only for that class's fields, call the superclass constructor to provide initial values for the superclass fields, and so on for each ancestor class. This approach is used in many object-oriented programming languages, including C++ and JAVA.

Unlike many theoretical calculi for object-oriented languages, our calculus directly supports modular object construction. The mixin implementor only writes the local constructor for his own mixin. Mixin applications are reduced to generator functions which call all constructors in the inheritance chain in correct order, producing a fully initialized object (see section 4).

2.2 Design Tradeoffs

In this section, we explain the design decisions and tradeoffs chosen in our calculus. Our goal was to sacrifice as little expressive power as possible while keeping the type system simple and free of complicated types such as polymorphic object types and recursive *MyType*.

Classes. Even in purely object-based calculi, the conflict between inheritance and subtyping usually requires that two sorts of objects be distinguished [28]. "Prototype objects" do not support full subtyping but can be extended with new methods and fields and/or have their methods redefined. "Proper objects" support both depth and width subtyping but are not extensible. Without this distinction, special types with extra information are required to avoid adding a method to an object in which a method with the same name is hidden as a consequence of subtyping (*e.g.*, labeled types of [6]). In our calculus, the class construct plays the role of a "prototype" (extensible but not subtypable), while objects — represented by records of methods — are subtypable but not extensible.

Objects. Records are an intuitive way to model objects since both are collections of name/value pairs. The records-as-objects approach was in fact developed in the pioneering work on object-oriented calculi [19], in which inheritance was modeled by record subtyping. Unlike records, however, object methods should be able to modify fields and invoke sibling methods [21]. To be capable of updating the object's internal state, methods must be functions of the host object

(*self*). Therefore, objects must be *recursive* records. Moreover, *self* must be appropriately updated when a method is inherited, since new methods and fields may have been added and/or old ones redefined in the new host object. In our calculus, reduction rules produce class generators that are carefully designed so that methods are given a (recursive) reference to *self* only after inheritance has been resolved and all methods and fields contained in the host object are known.

Object updates. If all object updates are imperative, *self* can be bound to the host object when the object is instantiated from the class. We refer to this approach as *early self* binding. Self then always refers to the same record, which is modified imperatively in place by the object's methods. The main advantage of early binding is that the fixed-point operator (which gives the object's methods reference to *self*) has to be applied only once, at the time of object instantiation.

If functional updates must be supported — which is, obviously, the case for purely functional object calculi — early binding does not work (see, for example, [1], where early binding is called *recursive semantics*). With functional updates, each change in the object's state creates a new object. If *self* in methods is bound just once, at the time of object instantiation, it will refer to the old, incorrect object and not to the new, updated one. Therefore, *self* has to be bound each time a method is invoked. We refer to this approach as *late self* binding.

Object extension. Object extension in an object-based calculus is typically modeled by an operation that extends objects by adding new methods to them. There are two constraints on such an operation: (i) the type system must prevent addition of a method to an object which already contains a method with the same name, and (ii) since an object may be extended again after method addition, the actual host object may be larger than the object to which the method was originally added. The method body must behave correctly in any extension of the original host object, therefore, it must have a polymorphic type with respect to *self*. The fulfillment of the two constraints can be achieved, for instance, via polymorphic types built on row schemes [5] that use kinds to keep track of methods' presence.

Even more complicated is the case when object extension must be supported in a functional calculus. In the functional case, all methods modifying an object have *self* as their return type. Whenever an object is extended or has its methods redefined (overriden), the type given to *self* in all inherited methods must be updated to take into account new and/or redefined methods. Therefore, the type system should include the notion of *MyType* (a.k.a. *SelfType*) so that the inherited methods can be specialized properly. Support for *MyType* generally leads to more complicated type systems, in which forms of recursive types are required. This can be accomplished by using row variables combined with recursive types [27, 26, 28], match-bound type variables [18, 4], or by means of special forms of second-order quantifiers such as the *Self* quantifier of [1].

Tradeoffs. Our goal is to achieve a reasonable tradeoff between expressivity and simplicity. We do not support functional updates because we believe that imperative updates combined with early *self* binding provide such a tradeoff.

Without functional updates, we can use early binding of *self*. Early binding eliminates the main need for recursive object types. There is also no need for polymorphic object types in our calculus since inheritance is modeled entirely at the class level and there are no object extension operations. This choice allows us to have a simple type system and a straightforward form of structural subtyping, in contrast with the calculi that support *MyType* specialization [28, 18].

There are at least two possible drawbacks to our approach. Although methods that *return* a modified *self* can be modeled in our calculus as imperative methods that modify the object and return nothing, methods that accept a *MyType argument* cannot be simulated in our system without support for *MyType*. We therefore have no support for binary methods of the form described in [15]. Also, the type system of our calculus does not directly support *implementation types* (*i.e.*, types that include information about the class from which the object was instantiated and not just the object's interface). We believe that a form of implementation types can be provided by extending our type system with existential types.

2.3 Design of the Core Calculus

The two main concepts in object-oriented programming are *objects* and *classes*. In our calculus, objects are records of methods. Methods are represented as functions with a binding for *self* (the host record) and *field* (the private field). Since records, functions, and λ-binding are standard, we need not introduce new operational semantics or type rules for objects. Instead, we introduce new constructs and rules for mixins and classes only. The new constructs are: *class values* (representing complete classes obtained as a result of mixin application), *mixin expressions* (containing definitions of methods, fields, and constructors), and *instantiation expressions* (representing creation of objects from classes).

A class value is a tuple containing the generator function, the set of public method names, and the set of protected method names. The generator produces a function from *self* to a record of methods. When the class is instantiated, the fixed-point operator is applied to the generator's result to bind *self* in the methods' bodies, creating a full-fledged object.

Mixins — *i.e.*, classes parameterized over the superclass — are represented by mixin expressions. Inheritance is modeled by the evaluation rule that applies a mixin to a class value representing the superclass, producing a new class value. The generator of the new class takes the record of superclass methods built by the superclass generator and modifies it by adding and/or replacing methods as specified by the mixin. Only class values can be instantiated; mixins are used solely for building class hierarchies.

For simplicity, the core calculus supports only private fields and public and protected methods. Private methods can be modeled by private fields with a function type; public or protected fields can be modeled by combining private fields with accessor methods. Instead of putting encapsulation levels into object types, we express them using subtyping and binding. Protected methods are treated in the same way as public methods except that they are excluded from

the type of the object returned to the user. Private fields are not listed in the object type at all, but are instead bound in each method body. In the core calculus each class has exactly one private field, which may have a record type. Each method body takes the class's private field as a parameter.

2.4 An Example of Mixin Inheritance

Mixin inheritance can be a powerful tool for constructing class hierarchies. In this section, we give a simple example that demonstrates how a mixin can be implemented in our calculus and explain some of the uses of mixins. For readability, the example uses functions with multiple arguments even though they are not formalized explicitly in the calculus.

Mixin definition. Following is the definition of Encrypted mixin that implements encryption functionality on top of any stream class. Note that the class to which the mixin is applied may have more methods than expected by the mixin. For example, Encrypted can be applied to Socket $\diamond$ *Object* where *Object* is the root of all class hierarchies, even though Socket $\diamond$ *Object* has other methods besides *read* and *write*.

```
let File =
  mixin
    method write = ...
    method read = ...
    ...
  end in

let Socket =
  mixin
    method write = ...
    method read = ...
    method hostname = ...
    method portnumber = ...
    ...
  end in

let Encrypted =
  mixin
    redefine write = λ next. λ key. λ self. λ data. next (encrypt(data, key));
    redefine read =  λ next. λ key. λ self. λ _ .   decrypt(next (), key);
    constructor λ (key, arg). {fieldinit=key, superinit=arg};
    protect [];
  end in ...
```

Mixin expressions contain new methods (marked by the method keyword), redefined methods (redefine keyword), and constructors. The names of protected methods should be listed following the protect keyword. Instead of introducing a special field construct, every mixin contains a single private field which is λ-bound in each method body (λ key. $\cdots$).

Methods can access the host object through the *self* parameter, which is λ-bound in each method body to avoid introducing special keywords. Redefined methods can access the old method body inherited from the superclass via the *next* parameter. Constructors are simply functions returning a record of two components. The *fieldinit* value is used to initialize the private field. The *superinit* value is passed as an argument to the superclass constructor.

From the definition of Encrypted, the type system infers the constraint that must be satisfied by any class to which Encrypted is applied. The class must contain *write* and *read* methods whose types must be supertypes of those given to *write* and *read*, respectively, in the definition of Encrypted.

Mixin usage. To create an encrypted stream class, one must apply the Encrypted mixin to an existing stream class. In our calculus, the notation for applying mixin M to class C is $M \diamond C$. For example, Encrypted $\diamond$ FileStream is an encrypted file class. The power of mixins can be seen when we apply Encrypted to a family of different streams. For example, we can construct Encrypted $\diamond$ NetworkStream, which is a class that encrypts data communicated over a network. In addition to single inheritance, we can express many uses of multiple inheritance by applying more than one mixin to a class. For example, PGPSign $\diamond$ UUEncode $\diamond$ Encrypt $\diamond$ Compress $\diamond$ FileStream produces a class of files that are compressed, then encrypted, then uuencoded, then signed. In addition, mixins can be used for forms of inheritance that are not possible in most single and multiple inheritance-based systems. In the above example, the result of applying Encrypted to a stream satisfies the constraint required by Encrypted itself, therefore, we can apply Encrypted more than once: Encrypted $\diamond$ Encrypted $\diamond$ FileStream is a class of files that are encrypted twice. In our system, private fields of classes do not conflict even if they have the same name, so each application of Encrypted can have its own encryption key. Unlike most forms of multiple inheritance, it is easy to specify the *order* and *number* of times the mixins are applied.

A note on an implementation. Our calculus uses structural object types that retain no connection to the class from which the object was instantiated. Since unrelated classes may use different layouts for the method dictionary, the compiler cannot use the object's static type to determine the exact position of a method in the dictionary in order to optimize method lookup as is done in C++. Adding mixins in this environment does not impose an extra overhead.

It is possible to support efficient method lookup by introducing a separate hierarchy of mixin interfaces similar to the one analyzed by Flatt et al. [30] and requiring that the order of methods in a mixin's dictionary match that given in the interface implemented by the mixin. However, a separate interface hierarchy would make the calculus significantly more complicated.

3 Syntax of the Core Calculus

The syntax of our calculus is fundamentally class-based. There are four expressions involving classes: classval, mixin, $\diamond$ (mixin application), and new. Class-related expressions and values are treated as any other expression or value in the calculus. They can be passed as arguments, put into data structures, and so on. However, class values and object values are not intended to be written directly; instead, these expression forms are used only to define the semantics of programs. Class values can be created by mixin application, and object values can be created by class instantiation.

$$
\begin{array}{lll}
\text{Expressions:} & e ::= & const \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \mathit{fix} \mid \mathsf{ref} \mid\ ! \mid :=\\
& & \mid \{x_i = e_i\}^{i \in I} \mid e.x \mid \mathsf{H}\ h.e \mid \mathsf{new}\ e\\
& & \mid \mathsf{classval}\langle v_g, [m_i]^{i \in Meth}, [p_\ell]^{\ell \in Prot}\rangle\\
& & \mid \mathsf{mixin}\\
& & \quad \mathsf{method}\ m_j = v_{m_j};\ ^{(j \in New)}\\
& & \quad \mathsf{redefine}\ m_k = v_{m_k};\ ^{(k \in Redef)}\\
& & \quad \mathsf{protect}\ [p_\ell];\ ^{(\ell \in Prot)}\\
& & \quad \mathsf{constructor}\ v_c;\\
& & \ \mathsf{end}\\
& & \mid e_1 \diamond e_2
\end{array}
$$

$$
\begin{array}{lll}
\text{Values:} & v ::= & const \mid x \mid \lambda x.e \mid \mathit{fix} \mid \mathsf{ref} \mid\ ! \mid := \mid := v \mid \{x_i = v_i\}^{i \in I}\\
& & \mid \mathsf{classval}\langle v_g, [m_i]^{i \in Meth}, [p_\ell]^{\ell \in Prot}\rangle\\
& & \mid \mathsf{mixin}\\
& & \quad \mathsf{method}\ m_j = v_{m_j};\ ^{(j \in New)}\\
& & \quad \mathsf{redefine}\ m_k = v_{m_k};\ ^{(k \in Redef)}\\
& & \quad \mathsf{protect}\ [p_\ell];\ ^{(\ell \in Prot)}\\
& & \quad \mathsf{constructor}\ v_c;\\
& & \ \mathsf{end}
\end{array}
$$

Fig. 1. Syntax of the core calculus

Let *Var* be an enumerable set of variables (otherwise referred to as *identifiers*), and *Const* be a set of constants. Expressions E and values V (with $V \subset E$) of the core calculus are as in Fig.1, where $const \in Const$; $x, x_i, m_i, m_j \in Var$; *fix* is the fixed-point operator; ref, !, $:=$ are operators;[1] $\{x_i = e_i\}^{i \in I}$ is a record; $e.x$ is the record selection operation; h is a set of pairs $h ::= \{\langle x, v\rangle^*\}$ where $x \in Var$ and v is a value (first components of the pairs are all distinct); $[m_i]$, $[p_\ell]$ are sets of identifiers; and $I, J, K, L, Meth, Prot, New, Redef \subset \mathbb{N}$.

Our calculus takes a standard calculus of functions, records, and imperative features and adds new constructs to support classes and mixins. We chose to extend *Reference ML* [45], in which Wright and Felleisen analyze the operational soundness of a version of ML with imperative features. Our calculus does not include let expressions as primitives since we do not need polymorphism to model our objects. We do rely on the Wright-Felleisen idea of *store*, which we call *heap*, in order to evaluate imperative side effects.

The expression $\mathsf{H}\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e$ associates reference variables $x_1, \ldots x_n$ with values $v_1, \ldots, v_n$. H binds $x_1, \ldots x_n$ in $v_1, \ldots, v_n$ and in e. The set of pairs h in the expression $\mathsf{H}h.e$ represents the *heap*, where the results of evaluating imperative subexpressions of e are stored.

The intuitive meaning of the class-related expressions is as follows:

[1] Introducing ref, !, $:=$ as operators rather than standard forms such as ref$e$, !e, $:=e_1e_2$, simplifies the definition of evaluation contexts and proofs of properties. As noted in [45], this is just a syntactic convenience, as is the curried version of $:=$.

- $\mathsf{classval}\langle v_g, [m_i]^{i\in Meth}, [p_\ell]^{\ell\in Prot}\rangle$ is a *class value*, *i.e.*, the result of mixin application. It is a triple, containing one function and two sets of variables. The function v_g is the generator for the class. The $[m_i]$ set contains the names of all methods defined in the class, and the $[p_\ell]$ set contains the names of protected methods.
- mixin
  method $m_j = v_{m_j}$; $^{(j\in New)}$
  redefine $m_k = v_{m_k}$; $^{(k\in Redef)}$
  protect $[p_\ell]$; $^{(\ell\in Prot)}$
  constructor v_c;
  end
  is a *mixin*, in which $m_j = v_{m_j}$ are definitions of new methods, and $m_k = v_{m_k}$ are method redefinitions that will replace methods with the same name in any class to which the mixin is applied. Each method body $v_{m_{j,k}}$ is a function of *self*, which will be bound to the newly created object at instantiation time, and of the private *field*. In method redefinitions, v_{m_k} is also a function of *next*, which will be bound to the old, redefined method from the superclass. The v_c value in the constructor clause is a function that returns a record of two components. When evaluating a mixin application, v_c is used to build the generator as described in section 4.
- $e_1 \diamond e_2$ is an application of mixin e_1 to class value e_2. It produces a new class value. Mixin application is the basic inheritance mechanism in our calculus.
- new e uses generator v_g of the class value to which e evaluates to create a function that returns a new object, as described in section 4.

Programs and *answers* are defined as follows:

$$p ::= e \qquad \text{where } e \text{ is a closed expression}$$
$$a ::= v \mid \mathsf{H}\ h.v$$

Finally, we define the root of the class hierarchy, class *Object*, as a predefined class value:

$$Object \triangleq \mathsf{classval}\langle\ \lambda_.\lambda_.\{\},\ [\ ],\ [\ ]\ \rangle$$

The root class is necessary so that all other classes can be treated uniformly. Intuitively, *Object* is the class whose object instances are empty objects. It is the only class value that is not obtained as a result of mixin application. The calculus can then be simplified by assuming that any user-defined class that does not need a superclass is obtained by applying a mixin containing all of the class's method definitions to *Object*.

Throughout this paper, we will use let $x = e_1$ in e_2 in terms and examples as a more readable equivalent of $(\lambda x.e_2)e_1$. Also, we use unit as an abbreviation for the empty record or type $\{\}$, instead of having a new *unit* value and type. We will use the word "object" when the record in question represents an object. To avoid name capture, we apply α-conversion to binders λ and H.

4 Operational Semantics

$$
\begin{array}{rcll}
const\ v & \rightarrow & \delta(const, v)\ \text{ if } \delta(const, v) \text{ is defined} & (\delta)\\
(\lambda x.e)\ v & \rightarrow & [v/x]\ e & (\beta_v)\\
\mathit{fix}\ (\lambda x.e) & \rightarrow & [\mathit{fix}(\lambda x.e)/x]e & (\text{fix})\\
\{\ldots, x = v, \ldots\}.x & \rightarrow & v & (\text{select})\\
\mathsf{ref} v & \rightarrow & \mathsf{H}\langle x, v\rangle.x & (\text{ref})\\
\mathsf{H}\langle x, v\rangle h.R[!x] & \rightarrow & \mathsf{H}\langle x, v\rangle h.R[v] & (\text{deref})\\
\mathsf{H}\langle x, v\rangle h.R[:=xv'] & \rightarrow & \mathsf{H}\langle x, v'\rangle h.R[v'] & (\text{assign})\\
R[\mathsf{H}\ h.e] & \rightarrow & \mathsf{H}\ h.R[e], \quad R \neq [\,] & (\text{lift})\\
\mathsf{H}\ h.\mathsf{H}\ h'.e & \rightarrow & \mathsf{H}\ h\ h'.e & (\text{merge})\\
\mathsf{new\ classval}\langle g, \mathcal{M}, \mathcal{P}\rangle & \rightarrow & \lambda v.Sub_{\mathcal{M}\cup\mathcal{P}\rightarrow\mathcal{M}}(\mathit{fix}(g\ v)) & (\text{new})
\end{array}
$$

$$
\left(\begin{array}{l}
\mathsf{mixin}\\
\quad \mathsf{method}\ m_j = v_{m_j};\\
\quad \mathsf{redefine}\ m_k = v_{m_k};\\
\quad \mathsf{protect}\ [p_\ell];\\
\quad \mathsf{constructor}\ c;\\
\mathsf{end}
\end{array}\right)^{\substack{j \in New,\\ k \in Redef,\\ \ell \in Prot}} \diamond\ \mathsf{classval}\langle g, \mathcal{M}, \mathcal{P}\rangle \rightarrow \mathsf{classval}\langle Gen, [m_j] \cup \mathcal{M}, [p_\ell] \cup \mathcal{P}\rangle \qquad (\text{mixin})
$$

if $[m_j] \cap \mathcal{M} = \emptyset$, $[m_k] \subset \mathcal{M}$, and $[p_\ell] \subset [m_j] \cup \mathcal{M}$; Gen is defined below

Fig. 2. Reduction Rules

The operational semantics for our calculus extends that of *Reference* ML [45]. Reduction rules are given in Fig.2, where R are *reduction contexts* [22, 24, 37]. Expression Gen is defined below. Relation $\twoheadrightarrow$ is the reflexive, transitive, contextual closure of $\rightarrow$, with respect to *contexts* C, as defined (in a standard way) in appendix A.

Reduction contexts are necessary to provide a minimal relative linear order among the creation, dereferencing and updating of heap locations, since side effects need to be evaluated in a deterministic order. Reduction contexts R are defined as follows:

$$
\begin{array}{rl}
R ::= & [\ ] \mid R\ e \mid v\ R \mid R.x \mid \mathsf{new}\ R \mid R\ \diamond\ e \mid v\ \diamond\ R\\
& \mid \{m_1 = v_1, \ldots, m_{i-1} = v_{i-1}, m_i = R, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \le i \le n}
\end{array}
$$

To abstract from a precise set of constants, we only assume the existence of a partial function $\delta : Const \times ClosedVal \rightharpoonup ClosedVal$ that interprets the application of functional constants to closed values and yields closed values. See section 5 for the δ typability condition.

(β_v) and (*select*) rules are standard.

(*ref*), (*deref*) and (*assign*) rules evaluate imperative expressions following the linear order given by the reduction context R and acting on the heap. They are

formulated after [45]: (*ref*) generates a new heap location where the value v is stored, (*deref*) retrieves the contents of the location x, (*assign*) changes the value stored in a heap location.

(*lift*) and (*merge*) rules combine inner local heaps with outer ones whenever a dereference operator or an assignment operator cannot find the needed location in the closest local heap.

(*mixin*) rule evaluates mixin application expressions which represent inheritance in our calculus. A mixin is applied to a superclass value $\mathsf{classval}\langle g, \mathcal{M}, \mathcal{P}\rangle$. $\mathcal{M}$ is a set of all method names defined in the superclass; $\mathcal{P}$ is an annotation listing the names of protected methods in the superclass. The resulting class value is $\mathsf{classval}\langle \mathit{Gen}, [m_j] \cup \mathcal{M}, [p_\ell] \cup \mathcal{P}\rangle$ where *Gen* is the generator function defined below, $[m_j] \cup \mathcal{M}$ is the set of all method names, and $[p_\ell] \cup \mathcal{P}$ is an annotation listing protected method names. Using a class generator delays full inheritance resolution until object instantiation time when *self* becomes available.

Gen is the class generator. It takes a single argument x which is used by the constructor subexpression c to compute the initial value for the field of the new object, and returns a function from *self* to a record of methods. When the fixed-point operator is applied to the function returned by the generator, it produces a recursive record of methods representing a new object (see the (*new*) rule).

$$
\begin{array}{l}
\mathit{Gen} \stackrel{\triangle}{=} \lambda x.\lambda \mathit{self}. \\
\qquad \mathsf{let}\ t = c(x) \quad \mathsf{in} \\
\qquad \mathsf{let}\ \mathit{supergen} = g(t.\mathrm{superinit}) \quad \mathsf{in} \\
\qquad \left\{ \begin{array}{lllr}
m_j = \lambda y.v_{m_j} & & t.\mathrm{fieldinit}\ \mathit{self}\ y \\
m_k = \lambda y.v_{m_k} & (\mathit{supergen}\ \mathit{self}).m_k & t.\mathrm{fieldinit}\ \mathit{self}\ y \\
m_i = \lambda y. & (\mathit{supergen}\ \mathit{self}).m_i & y
\end{array} \right\}^{m_i \,\in\, \mathcal{M} \setminus [m_k]}
\end{array}
$$

In the mixin expression, the constructor subexpression c is a function of one argument which returns a record of two components: one is the initialization expression for the field (*fieldinit*), the other is the superclass generator's argument (*superinit*). *Gen* first calls $c(x)$ to compute the initial value of the field and the value to be passed to the superclass generator g. *Gen* then calls the superclass generator g, passing argument t.superinit, to obtain a function *supergen* from *self* to a record of superclass methods.

Finally, *Gen* builds a function from *self* that returns a record containing *all* methods — from both the mixin and the superclass. To understand how the record is created, recall that method bodies take parameters *field*, *self*, and, if it's a redefinition, *next*. Methods m_j are the *new* mixin methods: they appear for the first time in the current mixin expression. *Gen* has to bind *field* and *self* for them. Methods $m_i \in \mathcal{M} \setminus [m_k]$ are the *inherited* superclass methods: they are taken intact from the superclass's object (*supergen self*). Methods m_k are *redefined* in the mixin. Their bodies can refer to the old methods through the *next* parameter, which is bound to $(\mathit{supergen}\ \mathit{self}).m_i$ by *Gen*. They also receive a binding for *field* and *self*. For all three sorts of methods, the method bodies are wrapped inside $\lambda y. \cdots y$ to delay evaluation in our call-by-value calculus.

(*fix*) rule is standard.

(*new*) rule builds a function that can create a new object. The resulting function can be thought of as the composition of three functions: $Sub \circ fix \circ g$. Given an argument v, it will apply generator g to argument v, creating a function from *self* to a record of methods. Then the fixed-point operator *fix* (following [21]) is applied to bind *self* in method bodies and create a recursive record. Finally, we apply $Sub_{\mathcal{M}\cup\mathcal{P}\to\mathcal{M}}$, a coercion function from records to records that hides all components which are in $\mathcal{M} \cup \mathcal{P}$ but not in $\mathcal{M}$. The resulting record contains only public methods, and can be returned to the user as a fully formed object.

5 Type System

Our types are standard and the typing rules are fairly straightforward. The complexity of typing object-oriented programs in our system is limited exclusively to classes and mixins. Method selection, which is the only operation on objects in our calculus, is typed as ordinary record component selection. Since methods are typed as ordinary functions, method invocation is simply a function application.

Types are as follows:

$$\begin{aligned}\tau ::= \iota \mid \tau_1 \to \tau_2 \mid \tau \ \mathsf{ref} \mid \{x_i : \tau_i\}^{i\in I} \\ \mid \mathsf{class}\langle \tau, \{m_i{:}\tau_i\}, [p_\ell]\rangle^{i\in I, \ell\in L} \\ \mid \mathsf{mixin}\langle \tau_1, \tau_2, \{m_j{:}\tau_j\}, \{m_k{:}\tau_k\}, [p_\ell]\rangle^{j\in J, k\in K, \ell\in L}\end{aligned}$$

where ι is a constant type; $\to$ is the functional type operator; τ ref is the type of locations containing a value of type τ; $\{x_i{:}\tau_i\}^{i\in I}$ is a record type; and $I, J, K, L \subset \mathbb{N}$. In class types, $\{m_i : \tau_i\}$ is a record type and $[p_\ell]$ is a set of names, where $[p_\ell] \subseteq [m_i]$. In mixin types, $\{m_j : \tau_j\}, \{m_k : \tau_k\}$ are record types and $[p_\ell]$ is a set of names, where $[p_\ell] \subseteq ([m_j] \cup [m_k])$. Although record expressions and values are ordered so that we can fix an order of evaluation, record types are unordered. We also assume we have a function *typeof* from constant terms to types that respects the following *typability condition* [45]: for $const \in Const$ and value v, if $typeof(const) = \tau' \to \tau$ and $\emptyset \vdash v : \tau'$, then $\delta(const, v)$ is defined and $\emptyset \vdash \delta(const, v) : \tau$.

Our type system supports structural subtyping (the $<:$ relation) along with the subsumption rule (*sub*). The subtyping rules are shown in appendix B. Since subtyping on references is unsound and we wish to keep subtyping and inheritance completely separate, we have only the basic subtyping rules for function and record types. Subtyping only exists at the object level, and is not supported for class or mixin types.

Typing environments are defined as follows:

$$\Gamma ::= \varepsilon \mid \Gamma, x : \tau \mid \Gamma, \iota_1 <: \iota_2$$

where $x \in Var$, τ is a well-formed type, ι_1, ι_2 are constant types, and $x, \iota_1 \notin dom(\Gamma)$.

$$\frac{\Gamma \vdash g : \gamma \to \{m_i : \tau_i\}^{i \in All} \to \{m_i : \tau_i\}^{i \in All}}{\Gamma \vdash \mathsf{classval}\langle g, [m_i]^{i \in All}, [p_\ell]^{\ell \in Prot}\rangle : \mathsf{class}\langle \gamma, \{m_i{:}\tau_i\}^{i \in All}, [p_\ell]^{\ell \in Prot}\rangle} \quad (\textit{class val})$$

$$\frac{\Gamma \vdash e : \mathsf{class}\langle \gamma, \{m_i{:}\tau_i\}^{i \in All}, [p_\ell]^{\ell \in Prot}\rangle}{\Gamma \vdash \mathsf{new}\ e : \gamma \to \{m_j{:}\tau_j\}^{j \in All \setminus Prot}} \quad (\textit{instantiate})$$

$$\frac{\begin{array}{lr} (\mathsf{New}) \text{ For } j \in \textit{New}: & \Gamma \vdash v_{m_j} : \qquad \eta \to \sigma \to \tau^{\downarrow}_{m_j} \\ (\mathsf{Redef}) \text{ For } k \in \textit{Redef}: & \Gamma \vdash v_{m_k} : \tau^{\uparrow}_{m_k} \to \eta \to \sigma \to \tau^{\downarrow}_{m_k} \\ (\mathsf{Constr}) & \Gamma \vdash c : \gamma_d \to \{\text{fieldinit} : \eta, \text{superinit} : \gamma_b\} \end{array}}{\Gamma \vdash \begin{pmatrix} \mathsf{mixin} \\ \ \mathsf{method}\ m_j = v_{m_j}; \\ \ \mathsf{redefine}\ m_k = v_{m_k}; \\ \ \mathsf{protect}\ [p_\ell]; \\ \ \mathsf{constructor}\ c; \\ \mathsf{end} \end{pmatrix}^{\substack{j \in New \\ k \in Redef \\ \ell \in Prot}} : \begin{array}{l} \mathsf{mixin}\langle \gamma_b, \gamma_d, \{m_i : \tau^{\uparrow}_{m_i}, m_k : \tau^{\uparrow}_{m_k}\}, \\ \qquad \{m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\}, [p_\ell]\rangle \end{array}} \quad (\textit{mixin})$$

$$\text{where} \quad \begin{array}{l} \sigma = \{m_i : \tau^{\uparrow}_{m_i}, m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\} \\ [p_\ell] \ \subseteq \ [m_i] \cup [m_j] \cup [m_k] \\ m_i, \tau^{\uparrow}_{m_i}, \tau^{\uparrow}_{m_k} \text{ are inferred from method bodies} \end{array}$$

$$\frac{\begin{array}{l} \Gamma \vdash e_1 : \mathsf{mixin}\langle \gamma_b, \gamma_d, \sigma_{\text{Old}}, \sigma_{\text{New}}, P_d\rangle \\ \Gamma \vdash e_2 : \mathsf{class}\langle \gamma_c, \sigma_b, P_b\rangle \\ \Gamma \vdash \sigma_d <: \sigma_b <: \sigma_{\text{Old}} \\ \Gamma \vdash \gamma_b <: \gamma_c \end{array}}{\Gamma \vdash e_1 \diamond e_2 : \mathsf{class}\langle \gamma_d, \sigma_d, P_b \cup P_d\rangle} \quad (\textit{mixin app})$$

$$\text{where} \quad \begin{array}{ll} \sigma_b & = \{m_k : \tau_{m_k}, m_l : \tau_{m_l}, m_i : \tau_{m_i}\} \\ \sigma_{\text{Old}} & = \{m_i : \tau^{\uparrow}_{m_i}, m_k : \tau^{\uparrow}_{m_k}\} \\ \sigma_{\text{New}} & = \{m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\} \\ \sigma_d & = \{m_i : \tau_{m_i}, m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}, m_l : \tau_{m_l}\} \end{array}$$

Fig. 3. Typing Rules for Class-Related Forms

Typing judgments are as follows:

$$\Gamma \vdash \tau_1 <: \tau_2 \quad \tau_1 \text{ is a subtype of } \tau_2$$
$$\Gamma \vdash e : \tau \qquad e \text{ has type } \tau$$

The set of typing rules for class-related forms is shown in Fig.3. The remaining rules are standard and can be found in appendix B.

(*class val*) rule types class values. A class value is composed of an expression and two sets of method names. The expression g is the generator (see section 4) which produces a function that will later, at the time of new application, return a real object. The type of g can be determined by examining the type of the class value, $\mathsf{class}\langle\gamma, \{m_i{:}\tau_i\}, [p_\ell]\rangle$. Generator g takes an argument of type γ and returns a function that will return an object once the fixed-point operator is applied. The return type of g is therefore $\sigma \rightarrow \sigma$, where σ represents the type of *self*, $\{m_i : \tau_i\}$. This record type includes *all* methods, not only public methods. When the fixed-point operator is applied, $\mathit{fix}(gv)$ will have type σ when v has type γ.

(*mixin*) rule types mixin declarations. We describe it following the order of its premises. Note that mixin methods make typing assumptions about methods of the superclass to which the mixin will be applied. We refer to these types as *expected* types since the actual superclass methods may have different types. The exact relationship between the types expected by the mixin and the actual types of the superclass methods is formalized in rule (*mixin app*). We mark types that come from the *superclass* with $^{\uparrow}$ and those that will be changed or added in the *subclass* with $^{\downarrow}$.

- (New) The bodies of the new methods v_{m_j} are typed with a function type. The argument types are the type of the private field (η) and the type of *self* (σ). We do not lose generality by assuming only one field per class since η can be a tuple or record type. The return type is $\tau^{\downarrow}_{m_j}$.
- (Redef) The bodies of the redefined methods v_{m_k} are also typed with a function type. The first argument type $\tau^{\uparrow}_{m_k}$ is that of *next*, *i.e.*, the superclass method with the same name (recall that the new body can refer to the old body via *next*). The meaning of η and σ is the same as for the new methods. It is not known at the time of mixin definition to which class the mixin will be applied, so the actual type of the method replaced by m_k may be different from the expected type $\tau^{\uparrow}_{m_k}$.
- (Constr) The constructor expression c is a function that takes an argument of type γ_d and returns a record with two components. The component labelled fieldinit is the initialization expression for the private field. Clearly, it has to have the same type η as that assumed for the field when typing methods bodies. The component labeled superinit is the expression passed as the parameter to the superclass generator. Its type γ_b is inferred from the constructor definition since the superclass is not available at the time of mixin definition.

Both new and redefined methods in the mixin may call superclass methods (*i.e.*, methods that are expected to be supported by any class to which the mixin is applied). We refer to these methods as m_i. Their types $\tau^{\uparrow}_{m_i}$ are inferred from the mixin definition.

The mixin is typed with a $\mathsf{mixin}\langle\ldots\rangle$ type, which encodes the following information about the mixin:

- γ_b is the expected argument type of the superclass generator.
- γ_d is the exact argument type of the mixin generator.
- $\{m_i : \tau^{\uparrow}_{m_i}, m_k : \tau^{\uparrow}_{m_k}\}$ are the expected types of the methods that must be supported by any class to which the mixin is applied. Recall that m_i are the methods that are not redefined by the mixin but still expected to be supported by the superclass since they are called by other mixin methods, and $\tau^{\uparrow}_{m_k}$ are the types assumed for the old bodies of the methods redefined in the mixin.
- $\{m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\}$ are the exact types of mixin methods (new and redefined, respectively).
- $[p_\ell]$ is an annotation listing the names of all methods to be protected, both new and redefined.

Type information contained in the $\mathsf{mixin}\langle\ldots\rangle$ type is used when typing mixin application in rule (*mixin app*).

(*mixin app*) rule types mixin-based inheritance. In the rule definition, σ_b contains the type signatures of all methods supported by the superclass to which the mixin is applied. In particular, m_k are the superclass methods redefined by the mixin, m_i are the superclass methods called by the mixin methods but not redefined, and m_l are the superclass methods not mentioned in the mixin definition at all. Note that the superclass may have more methods than required by the mixin constraint.

Type σ_d contains the signatures of all methods supported by the subclass created as a result of mixin application. Methods $m_{i,l}$ are inherited directly from the superclass, methods m_k are redefined by the mixin, and methods m_j are the new methods added by the mixin. We are guaranteed that methods m_j are not present in the superclass by the construction of σ_b and σ_d: σ_d is defined so that it contains all the labels of σ_b plus labels m_j. Type σ_{Old} lists the (expected) types of the superclass methods assumed when typing the mixin definition. Type σ_{New} lists the exact types of the methods newly defined or redefined in the mixin.

The premises of the rule are as follows:

- $\mathsf{mixin}\langle\ldots\rangle$ and $\mathsf{class}\langle\ldots\rangle$ are the types of the mixin and the superclass, respectively.
- The $\sigma_d <: \sigma_b$ constraint requires that the types of the methods redefined by the mixin (m_k) be subtypes of the superclass methods with the same name. This ensures that all calls to the redefined methods in m_i and m_l (methods inherited intact from the superclass) are type-safe.

– The $\sigma_b <: \sigma_{\text{Old}}$ constraint requires that the actual types of the superclass methods m_i and m_k be subtypes of the expected types assumed when typing the mixin definition.
– The $\gamma_b <: \gamma_c$ constraint requires that the actual argument type of the superclass generator be a supertype of the type assumed when typing the mixin definition.

In the type of the class value created as a result of mixin application, σ_b is the argument type of the generator, and σ_d (see above) is the type of objects that will be instantiated from the class (except for the protected methods which are included in σ_d but hidden in the instantiated objects). In the resulting subclass we protect all methods that are protected either in the superclass or in the mixin.

The (*mixin app*) rule also determines how name clashes between the mixin and the superclass are handled. Suppose the superclass and the mixin contain a method with the same name m. If m is a redefined method in the mixin (*i.e.*, $m \in [m_k]$), then it will replace the method from the superclass as long as its type $\tau^{\downarrow}_{m_k}$ is a subtype of the replaced method's type τ_{m_k}. This is checked by the $\sigma_d <: \sigma_b$ premise. If m is a new method (*i.e.*, $m \in [m_j]$), then the rule's premises will fail since a method that is considered new by the mixin appears in the superclass ($m = m_j \in \sigma_b$), and the type system will signal an error.

(*instantiate*) rule types the creation of a new object. The new e term is typed as a function that takes the generator's argument and returns a fully initialized object. The object's type contains only the public methods; the protected methods are hidden.

The proof of soundness is omitted for lack of space. The complete meta-theory may be found in [8].

6 Related Work

In the literature, there exists an extensive body of work on calculi for object-oriented languages. Our calculus can be directly compared with the following class-oriented calculi:

– In the simplest of Cook's calculi [21], objects are represented by records of methods, and created by taking the fixed-point of the function representing the class (*constructor* in Cook's terminology). Inheritance is modeled by generating the subclass constructor from the superclass constructor, and *self* is bound early. However, classes are not a basic construct. The calculus relies on record concatenation operators, but typing issues associated with them are not addressed.
– The closure semantics version of the "dynamic inheritance" language analyzed by Kamin and Reddy [32] is similar to our calculus. The language is class-based, and the semantics of inheritance is similar to our generators. They also compare late and early *self* binding (*fixed-point model* and *self-application model* in their terminology). However, no type system is provided and there is no discussion of object construction or method encapsulation.

- The calculus of Wand [44] is class-based. Classes are modeled as extensible records, inheritance is record concatenation plus *self* update so that inherited methods refer to the correct object. As in our calculus, objects are records, *self* is bound early, and the *new* operation (called *constructor*) is an application of the fixed-point operator. In contrast to our calculus, the subclass must know and directly initialize the fields of the superclass. There is also no support for parameterized inheritance. Another solution, proposed in [40], is to rename the superclass fields, but this does not ensure consistent initialization.
- TOOPL [13] is a calculus of classes and objects. *MyType* specialization is used for inheritance, forcing late *self* binding (*i.e.*, *self* is bound each time a method is invoked, and not just once when the object is created). To ensure type safety when *MyType* appears in the method signature, there are standard constraints on method subtyping. A related work is PolyTOIL [18], where inheritance is completely separated from subtyping. Inheritance is based on *matching*, which is a relation between class interfaces that does not require method types to follow the standard constraints on recursive types, while object types employ standard subtyping. PolyTOIL also has imperative updating of object fields, but inheritance is still modeled with *MyType* in order to support binary methods. The drawback is the complexity of the type system. In [17], another language is presented, Loom, where only matching is used and the type system is simplified.

This paper is an attempt to built a simpler class-based calculus. The absence of *MyType* makes it weaker, but imperative updating appears sufficient to model the desirable features that are needed in practice.

Other approaches to modeling classes can be found in object-based calculi, where classes are not first-class expressions and have to be constructed from more primitive building blocks:

- Abadi and Cardelli have proposed encoding classes in a pure object system using records of pre-methods [1]. Pre-methods can be thought of as functions from *self* to method bodies or functions that are written as methods but not yet installed in any object. The difference between the result of *Gen* (see section 4 above) and a record of pre-methods is that the former is a function from *self* to a record of methods while the latter is a record of functions from *self* to methods. In the Abadi-Cardelli approach, a class is an object that contains a record of pre-methods and a constructor function used to package pre-methods into objects. The primary advantage of the record-of-pre-methods encoding is that it does not require a complicated form of objects. All that is needed is a way of forming an object from a list of component definitions. However, this approach provides no language support for classes, and imposes complicated constraints on the objects used as classes to obey to some basic requirements for class constructs (see section 2 above and [29] for a complete account).
- Another approach to modeling classes as objects is developed by Fisher [26] in a functional setting, and by Bono and Fisher [5] in an imperative setting.

Classes are modeled as encapsulated extensible objects. Inheritance is then modeled as the method addition operation on objects, which can be in one of two states [28]: a *prototype* (can be extended but not subtyped, so prototype objects are similar to classes), and a "proper" object which is subtypable but cannot be extended. A form of a bounded existential quantifier is used to (partially) abstract the class implementation when objects are in the prototype state. While the system of [5] can model a form of mixins, our calculus is simpler, more intuitive, and has encapsulation and object creation semantics closer to those used by popular programming languages.

– Pierce and Turner [39] model classes as object-generating functions. They interpret inheritance as modification of the object-generating functions used to model classes (*existential models*). This encoding is somewhat cumbersome, since it requires programmers to explicitly manipulate `get` and `put` functions which intuitively convert the hidden state of superclass objects into that of subclass objects. Hofmann and Pierce [31] introduce a refined version of $F_{<:}$ that permits only positive subtyping. With this restriction, `get` and `put` functions are both guaranteed to exist and hence may be handled in a more automatic fashion in class encodings. In our calculus, instead of encapsulation at the object type level, we use subtyping to hide protected methods and λ-binding to hide private fields.
– The Hopkins Object Group has designed a type-safe class-based object-oriented language with a rich feature set called I-LOOP [23]. Their type system is based on polymorphic recursively constrained types, for which they have a sound type inferencing algorithm. The main advantage of this approach is the extreme flexibility afforded by recursively constrained types. However, inferred types are large and difficult to read.

Bruce et al. [16] show how the main approaches to modeling objects can be seen in a unified framework. The state of the art in modeling classes is not as well established. We hope that this work might be a step in this direction.

To the best of our knowledge, there are not many formal settings in which mixin-based inheritance is analyzed.

– Flatt et al. implement mixins in the MZSCHEME language [25] and formalize an extension of a subset of JAVA with mixins in [30]. Their system supports higher-order mixin composition, a hierarchy of named interface types, and resolution of accidental name collisions. The collision resolution system allows old and new method definitions to coexist. The two are distinguished using the "view" of an object, which is carried with the object at run-time and altered at each subsumption step. As a result, method lookup is sensitive to the object's history of subsumptions. In contrast to the system of [30], our calculus is not based on any particular language. Our mixins are created and manipulated as run-time values as opposed to static top-level declarations. Mixin constraints prevent objects from having incompatible methods with the same name, so method lookup is straightforward and does not depend on the object's subsumption history. Proper object initialization is guaranteed.

– BETA [36] replaces classes, procedures, functions, and types by a single abstraction mechanism called the *pattern*. Objects are created from the patterns, and in addition to traditional objects as found in conventional object-oriented languages, objects in BETA may also represent function activations, exception occurrences, or concurrent processes. Patterns may be used as *superpatterns* to other patterns in a manner similar to conventional inheritance. Since patterns are a general concept, inheritance is available also for procedures, functions, exceptions, coroutines, and processes. *Virtual* patterns are similar to generic templates or parameterized classes with the additional benefit that the parameter may be restricted without actually instantiating the template (this is similar to computing the mixin constraint without actually applying the mixin to a class). Mixin inheritance is a partial case of the very general pattern inheritance mechanism developed in BETA.
– OCAML [34] supports a very limited form of parameterized inheritance by combining a module abstraction mechanism with classes that can inherit across module boundaries. Because the exact module containing the superclass may not be known when the subclass is defined, the same subclass can be used with multiple superclass definitions. However, methods not mentioned in the superclass type become inaccessible. In the example of section 2.4, this would mean that all methods that are present in the Socket $\diamond$ *Object* class besides *read* and *write* are forgotten once Encrypted mixin is applied to it.
– Ancona and Zucca [2] study a rigorous semantics foundations for mixins independently from the notions of classes and objects, starting from an algebraic setting for module composition. It may be possible to apply their techniques to the study of the algebraic semantics of our calculus.

7 Conclusions and Future Work

The main strengths of our calculus are its simplicity and its power in modeling mixin inheritance. Both the operational semantics and the type system are structured to combine new rules for *class*-based features (mixins, classes, and instantiation) with standard rules for *object*-based features (represented by records, functions, and assignable locations). We also preserve such properties as encapsulation (private fields, protected methods) and modularity (minimized dependencies of a subclass on superclasses, modular object creation, automatic propagation of changes in the superclass to all subclasses). All of these are desirable features for a formalism used to model classes [29]. Our mixin construct provides a formal model for a flexible inheritance mechanism, capable of expressing single inheritance, most uses of multiple inheritance, and also new uses of inheritance such as applying the same mixin more than once.

Some of the design choices may appear debatable, *e.g.*, the decision not to support *super* in the calculus. While a redefined method can refer to the old method body via *next*, other methods have no way of calling it. This decision was motivated mainly by our desire to support an efficient implementation, and,

in fact, the calculus can be easily extended to support *super* by keeping a reference to the entire superclass object (*supergen self*) instead of selecting the component being redefined (see section 4). Also debatable is the decision to support imperative instead of functional object updates. This choice was motivated by our desire for simplicity and the relative complexity of supporting functional update (*e.g.*, the need for *MyType*).

We believe that our calculus can be considered a step towards a better understanding of class-based languages, both because it shows how support for modular programming techniques can be included in a sound calculus without compromising its simplicity, and because it can serve as a starting point for more foundational studies such as denotational semantics for the class and mixin constructs. Topics for future research include developing an efficient implementation of the core calculus and extending it to a full language; studying an extension of the core calculus with ML polymorphism in order to combine classes and objects with the full power of ML type inference; combining existential types with our simple object types to provide a form of implementation types; and expanding our rules for mixins to account for higher-order mixins.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] D. Ancona and E. Zucca. An algebraic approach to mixins and modularity. In *Proc. Algebraic and Logic Programming (ALP)*, pages 179–193. LNCS 1139, Springer-Verlag, 1996.

[3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[4] V. Bono and M. Bugliesi. Matching for the lambda calculus of objects. *Theoretical Computer Science*, 1998. To appear.

[5] V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *Proc. ECOOP '98*, pages 462–497. LNCS 1445, Springer-Verlag, 1998. Preliminary version appeared in FOOL 5 proceedings.

[6] V. Bono and L. Liquori. A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects. In *Proc. CSL '94*, pages 16–30. LNCS 933, Springer-Verlag, 1995.

[7] V. Bono, A. Patel, V. Shmatikov, and J. C. Mitchell. A core calculus of classes and objects. In *Proc. 15th Conference on the Mathematical Foundations of Programming Semantics (MFPS '99)*, 1999. To appear.

[8] V. Bono, A. Patel, V. Shmatikov, and J. C. Mitchell. A core calculus of object, classes, and mixins. Technical Report, The University of Birmingham and Stanford University, 1999. Forthcoming.

[9] N. Boyen, C. Lucas, and P. Steyaert. Generalized mixin-based inheritance to support multiple inheritance. Technical Report vub-prog-tr-94-12, Vrije Universiteit Brussel, 1994.

[10] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[11] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.

[12] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages (ICCL '92)*, pages 282–290, 1992.
[13] K. B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc. POPL '93*, pages 285–298, 1993.
[14] K. B. Bruce. A paradigmatic object-oriented language: design, static typing and semantics. *J. Functional Programming*, 4(2):127–206, 1994.
[15] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.
[16] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. In *Proc. TACS '97*, pages 415–438. LNCS 1281, Springer-Verlag, 1997.
[17] K. B. Bruce, L. Petersen, and A. Finch. Subtyping is not a good "match" for object-oriented languages. In *Proc. ECOOP '97*, pages 104–127. LNCS 1241, Springer-Verlag, 1997.
[18] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proc. ECOOP '95*, pages 26–51. LNCS 952, Springer-Verlag, 1995.
[19] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.
[20] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proc. POPL '90*, pages 125–135, 1990.
[21] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.
[22] E. Crank and M. Felleisen. Parameter-passing and the lambda calculus. In *Proc. POPL '91*, pages 233–244, 1991.
[23] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proc. OOPSLA '95*, pages 169–184, 1995.
[24] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
[25] R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. ICFP '98*, pages 94–104, 1998.
[26] K. Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1996.
[27] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda-calculus of objects and method specialization. *Nordic J. of Computing*, 1(1):3–37, 1994. Preliminary version appeared in Proc. *LICS '93*, pp. 26–38.
[28] K. Fisher and J. C. Mitchell. A delegation-based object calculus with subtyping. In *Proc. 10th International Conference on Fundamentals of Computation Theory (FCT '95)*, pages 42–61. LNCS 965, Springer-Verlag, 1995.
[29] K. Fisher and J. C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–26, 1998. Preliminary version appeared in Marktoberdorf '97 proceedings.
[30] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.
[31] M. Hofmann and B. C. Pierce. Positive subtyping. *Information and Computation*, 126(1):11–33, 1996. Preliminary version appeared in Proc. *POPL '95*.
[32] S. Kamin and U. Reddy. Two semantic models of object-oriented languages. In C. Gunther and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.
[33] S. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, 1989.

[34] X. Leroy, D. Rémy, J. Vouillon, and D. Doligez. The Objective Caml system, documentation and user's guide. `http://caml.inria.fr/ocaml/htmlman/`, 1999.

[35] M. Van Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple iinheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

[36] O. Lehrmann Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Language*. Addison-Wesley, 1993.

[37] I. Mason and C. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proc. ICALP '89*, pages 574–588. LNCS 372, Springer-Verlag, 1989.

[38] D. Moon. Object-oriented programming with Flavors. In *Proc. OOPSLA '86*, pages 1–8, 1986.

[39] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *J. Functional Programming*, 4(2):207–248, 1994. Preliminary version appeared in Proc. *POPL '93* under the title *Object-Oriented Programming Without Recursive Types*.

[40] U. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[41] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. ECOOP '98*, pages 550–570, 1998.

[42] B. Stroustrup. *The C++ Programming Language (3rd ed.)*. Addison-Wesley, 1997.

[43] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. OOPSLA '96*, pages 359–369, 1996.

[44] M. Wand. Type inference for objects with instance variables and inheritance. In C. Gunther and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[45] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

A Definition of Contexts

The definition of contexts is standard but lengthy due to the number of subexpressions in the mixin expression:

$$
\begin{aligned}
C ::= {} & [\,] \mid C\ e \mid e\ C \mid \lambda x.C \mid C.x \mid C \diamond e \mid e \diamond C \\
& \mid \{m_1 = e_1, \ldots, m_{i-1} = e_{i-1}, m_i = C, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \le i \le n} \\
& \mid \mathsf{H}\ h.C \mid \mathsf{H}\langle x, C\rangle h.e \mid \mathsf{new}\ C \mid \mathsf{classval}\langle C, \mathcal{M}, \mathcal{P}\rangle
\end{aligned}
$$

$$
\begin{array}{lll}
\mid \mathsf{mixin}^{\substack{j \in New \\ k \in Redef \\ \ell \in Prot}} & \mid \mathsf{mixin}^{\substack{j \in New \setminus [i] \\ k \in Redef \\ \ell \in Prot}} & \mid \mathsf{mixin}^{\substack{j \in New \\ k \in Redef \setminus [i] \\ \ell \in Prot}} \\
\quad \mathsf{method}\ m_j = v_{m_j}; & \quad \mathsf{method}\ m_j = v_{m_j}; & \quad \mathsf{method}\ m_j = v_{m_j}; \\
\quad \mathsf{redefine}\ m_k = v_{m_k}; & \quad \mathsf{method}\ m_i = C; & \quad \mathsf{redefine}\ m_k = v_{m_k}; \\
\quad \mathsf{protect}\ [p_\ell]; & \quad \mathsf{redefine}\ m_k = v_{m_k}; & \quad \mathsf{redefine}\ m_i = C; \\
\quad \mathsf{constructor}\ C; & \quad \mathsf{protect}\ [p_\ell]; & \quad \mathsf{protect}\ [p_\ell]; \\
\ \ \mathsf{end} & \quad \mathsf{constructor}\ v_c; & \quad \mathsf{constructor}\ v_c; \\
 & \ \ \mathsf{end} & \ \ \mathsf{end}
\end{array}
$$

B Type Rules

The type rules for class-related forms were presented in section 5. The remaining type rules are presented here.

B.1 Subtyping Rules

The subtyping rules are standard. Objects support both depth and width subtyping.

$$\frac{}{\Gamma, \iota_1 <: \iota_2 \vdash \iota_1 <: \iota_2} \ (<:\ proj) \qquad \frac{}{\Gamma \vdash \tau <: \tau} \ (refl)$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3} \ (trans) \qquad \frac{\Gamma \vdash \tau' <: \tau \quad \Gamma \vdash \sigma <: \sigma'}{\Gamma \vdash \tau \to \sigma <: \tau' \to \sigma'} \ (arrow)$$

$$\frac{\Gamma \vdash \tau_i <: \sigma_i \quad i \in I \quad I \subseteq J}{\Gamma \vdash \{m_i : \tau_i\}^{i \in I} <: \{m_j : \sigma_j\}^{j \in J}} \ (<:\ record)$$

B.2 Type Rules for Expressions

The type rules for expressions other than class-related forms are simple, except for heaps, which have to be typed globally.

$$\frac{typeof(const) = \tau}{\Gamma \vdash const : \tau} \ (const) \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau} \ (proj) \qquad \frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \to \sigma} \ (\lambda)$$

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \sigma} \ (app) \qquad \frac{}{\Gamma \vdash fix : (\sigma \to \sigma) \to \sigma} \ (fix)$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau <: \sigma}{\Gamma \vdash e : \sigma} \ (sub) \qquad \frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{x_i = e_i\}^{i \in I} : \{x_i : \tau_i\}} \ (record)$$

$$\frac{\Gamma \vdash e : \{x : \sigma\}}{\Gamma \vdash e.x : \sigma} \ (lookup) \qquad \frac{}{\Gamma \vdash \mathsf{ref} : \tau \to \tau\ \mathsf{ref}} \ (ref) \qquad \frac{}{\Gamma \vdash\ ! : \tau\ \mathsf{ref} \to \tau} \ (!)$$

$$\frac{}{\Gamma \vdash := \ : \tau\ \mathsf{ref} \to \tau \to \tau} \ (:=)$$

$$\frac{\Gamma' = \Gamma, x_1 : \tau_1\ \mathsf{ref}, \ldots, x_n : \tau_n\ \mathsf{ref} \quad \Gamma' \vdash v_i : \tau_i \quad \Gamma' \vdash e : \tau}{\Gamma \vdash H\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e : \tau} \ (heap)$$

Propagating Class and Method Combination

Erik Ernst

DEVISE, Department of Computer Science
University of Aarhus, Denmark
eernst@daimi.au.dk

Abstract. This paper presents a mixin based class and method combination mechanism with block structure propagation. Traditionally, mixins can be composed to form new classes, possibly merging the implementations of methods (as in CLOS). In our approach, a class or method combination operation may cause any number of implicit combinations. For example, it is possible to specify separate aspects of a family of classes, and then combine several aspects into a full-fledged class family. The combination expressions would explicitly combine whole-family aspects, and by propagation implicitly combine the aspects for each member of the class family, and again by propagation implicitly compose each method from its aspects. As opposed to CLOS, this is type-checked statically; and as opposed to other systems for advanced class combination/merging/weaving, it is integrated directly in the language, ensuring a clear semantics and a seamless interaction with the type system. Moreover, the basic mechanism used in the combination, linearization, is formalized and generalized compared to previous presentions.

1 Introduction

In recent years the management of concerns involving multiple classes has been a very active area of research. Subject orientation [10], aspect oriented programming [13], and object collaborations [20] are all examples of such efforts. This paper presents a language integrated approach to the achievement of similar goals. A seamless integration into a statically typed general purpose programming language opens the possibility for type checking at the level of multiclass constructs, separate type-checking and compilation, and avoidance of the "impedance" problems associated with the use of several different mind-sets, languages, and tools.

The approach described in this paper is based on the use of general block structure (i.e., multi-level nesting, like inner classes in Java) to enable a natural expression of groups of mutually dependent classes, along with a very flexible inheritance and class/method combination mechanism which interacts with the block structure to support propagation of class combinations. The word *propagation* refers to the following feature: An expression, like `a&b` which denotes the combination of the classes or methods `a` and `b`, can imply a number of implicit class or method combinations. Since the explicit combination operation

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 67–91, 1999.
© Springer-Verlag Berlin Heidelberg 1999

causes the implicit ones, it is natural to think of the combination process as starting with an explicit operation and propagating to implicit ones. The benefit is similar to that of all other abstraction mechanisms—a complex but regular result is achieved from a simple expression, ensuring readability, consistency, maintainability, and so on.

The contributions of this work are the design and implementation of the propagating class and method combination mechanism and its static analysis, and a clarification and generalization of the formal foundations of inheritance graph linearization. Linearization is the low-level mechanism on which the combination mechanism builds; it does not support propagation itself, but linearization together with block structure and virtual (class and method) attributes supports it. Virtual methods are well-known; virtual classes [18, 24, 11] are attributes which may vary with the enclosing class, e.g., if `NodeType` is a virtual class attribute of the class `Graph`, then `Graph.NodeType` could be the class `Node` and `ColoredGraph.NodeType` could be the class `ColoredNode`. Virtual classes are similar to, but not equivalent with, type parameters of parameterized classes (such as template classes in C++ or generic classes in Eiffel). See [5] for examples, but also note [25].

The result of implementing this is the language `gbeta`, which is a generalization of the language BETA. BETA already offers the combination of general block structure, virtual classes, and a kind of method combination—the `INNER` mechanism. `INNER` is similar to resending a message to '`super`' in Smalltalk in that it calls the implementation of a method in another class, but `INNER` calls to the subclass (if any) where `super` calls to the superclass. Consequently, all methods in BETA (and `gbeta`) are invoked at the most *general* level, and `INNER` determines where to *insert* the next more specialized method implementation.

`gbeta` generalizes BETA in several ways in order to support the propagation mechanism. The subclass relation between classes in BETA is based on name equivalence; in `gbeta` it is based on a coarse grained, mixin based structural equivalence which has the BETA relation as a subset. BETA has single inheritance, but both for classes and methods [15]; in `gbeta` classes and methods can be combined, again with BETA semantics as a special case. In `gbeta` it is also possible to inherit from a virtual class, and to do several other things not supported in BETA, but these features are not essential for the topic of this paper.

The approach may seem tied to a particular language, but that is only because so few object-oriented languages offer general block structure and virtual classes. With the introduction of inner classes in Java and the possible addition of virtual types [24] or a similar mechanism, the same techniques would apply here.

Since CLOS [12] programmers introduced "mixin" classes as a particular programming style in context of linearization based multiple inheritance, a related but separate concept of *mixins* and mixin-based inheritance has emerged [2, 3, 23, 8]. Inheritance allows for the creation of one new class, based on zero or more superclasses and a specification of an increment (often with syntax like `{..}` containing a list of declarations). Mixins liberate the increment such that

it can be applied to different superclasses. The semantic denotation of an increment may be a function from classes to classes [8], a class-like entity which can be composed with others using special mixin combination operators [2, 3], or even a method which enhances the structure of the enclosing object [23]. In any case, the application of a mixin to an actual superclass resembles inheritance. This motivates the alternative term for mixins: *abstract subclasses*.

In a statically typed language the not-yet-known superclass of a mixin must be characterized somehow before the usage of inherited attributes in the mixin can be type-checked. In [8]—which deals with a subset of Java, enhanced with mixin support—this is achieved by requiring that mixins specify an *inheritance interface*. This is an interface which is assumed about the formal superclass during checking of the mixin, and required of the actual superclass at mixin application. In gbeta, the inheritance interface is the statically known prefix (superclass or supermethod) of the mixin. The mixin is type checked on the assumption that the prefix is available, and the run-time semantics ensure that the actual prefix always is the statically known one or a descendant of it.

In summary, the essense of typed mixins is the support for standalone, class- and method-like entities which can be applied as abstract subclasses/methods to several different actual superclasses/methods, such that the interaction between the two is precisely specified. This description matches the mechanism in gbeta, hence the use of the word 'mixin'. However, since mixins in gbeta are managed and composed not individually but in lists and using linearization, CLOS might be a closer reference point than *Jigsaw* [3], Agora [23] or MixedJava [8].

The remainder of this paper is organized as follows: Section 2 describes the usage of the propagating combination mechanism and argues for its usefulness via a number of examples. Section 3 gives a precise characterization of the linearization algorithm and the propagation mechanism. The current implementation is briefly described in Sect. 4. Related work is covered in Sect. 5, future work in Sect. 6, and Sect. 7 concludes.

2 Usage

This section gives a survey of significant usages of the propagating combination mechanism, thus illustrating its semantics and motivating its usefulness.

The concrete syntax used throughout is a modification of gbeta syntax (which is Beta syntax enhanced with a few new constructs); the differences are that (1) certain special characters which indicate kinds of declarations have been replaced with keywords, and (2) the "->" which unifies the syntax for assignment, method argument transfer, and function return has been transformed to a more mainstream notation. Specifically, assignment is designated with ":=" with the source on the right hand side and the destination on the left hand side; formal arguments to methods are declared outside the method body, and actual arguments are given in parentheses after the method name. This makes the ex-

amples more verbose but hopefully improves the readability for those who are not accustomed to the BETA style of syntax.[1]

Note that method signatures (argument types and names, and return types) are inherited (not repeated) in BETA/gbeta.

2.1 Class Combination, Propagating to Methods

The first example illustrates the use of propagation in only one level; this special case works similarly to CLOS method combination using 'before' and 'after' methods, illustrating the combination mechanism by showing how it achieves a recognizable goal.

Consider an abstract class Stack which specifies a stack data structure, along with a subclass StackImpl which contributes an implementation of the stack using a list (whose type constraint on contained objects, Element, is specified to be the same as the constraint given for the enclosing StackImpl):

```
class Stack: 1(#
  virtual class Element: Object;
  virtual method init: 2(# do INNER #);
  virtual method push: 3(elm: Element)(# do INNER #);
  virtual method pop: 4(# do INNER #): Element
#);

class StackImpl: Stack 5(#
  extended method init: 6(# do storage.init #);
  extended method push: 7(# do storage.insert(elm) #);
  extended method pop: 8(# return storage.deleteFirst #);
  object storage: list
    9(# extended class Element: this(StackImpl).Element #)
#)
```

For brevity, this example uses minimal error handling, e.g., pop on an empty StackImpl fails because deleteFirst fails.

In gbeta (and BETA) a virtual method cannot be overridden, but it can be *extended*. For example, init in StackImpl is 2(# do INNER #) extended with 6(# do storage.init #). The keyword do marks the beginning of the statements of the method. The small superscript numbers (2 and 6 in this case) are not part of the syntax; they are added to the syntax here in order to identify declaration blocks (including argument list and return type, if present). As Fig. 6 in Sect. 3.2 will show, a mixin in gbeta is a pair of a declaration block and an environment, but for simplicity we will ignore the environment at this point. The notation for a mixin is m_i where i is the identification number. This means that init in StackImpl can be concisely specified as the mixin list $[m_2, m_6]$.

As mentioned in Sect. 1, INNER is a call to the next more specific contribution to a method, so the effect of init in StackImpl is just to invoke storage.init.

[1] The examples in original gbeta syntax are available at http://www.daimi.au.dk/~eernst/gbeta/examples/PropComb/

The method body do INNER is the BETA/gbeta idiom for a deferred implementation, since it simply calls the subclass extensions of the method.

Returning to the concrete example, the StackImpl class can be used directly, but it does not protect itself from shared access in a multi-threaded context. To add concurrency control we write another subclass of Stack:

```
class StackConc: Stack 10(#
  extended method init: 11(# do mutex.init; INNER #);
  extended method push: protect;
  extended method pop: protect;
  method protect: 12(# do mutex.P; INNER; mutex.V #);
  object mutex: Semaphore;
#)
```

In StackConc, the virtual methods push and pop are extended with protect. As explained below, this equips push and pop with concurrency control.

We now have two aspects, StackConc and StackImpl. They can be combined using the combination operator '&':

```
class ThreadSafeStack:  StackConc & StackImpl;
```

The resulting class, ThreadSafeStack, is actually a thread safe stack, because the combination of the two classes propagates to the methods such that each method contains contributions from both StackConc and StackImpl.

Figure 1 shows the four classes. Each box is a mixin, so each class is a list of mixins (all classes and methods are). Moreover, the combination operator '&' is defined in terms of mixin lists, as described in Sect. 3.1. The subclass relation is the superlist relation, so we have a subclass ordering as specified below the mixin lists in the figure.

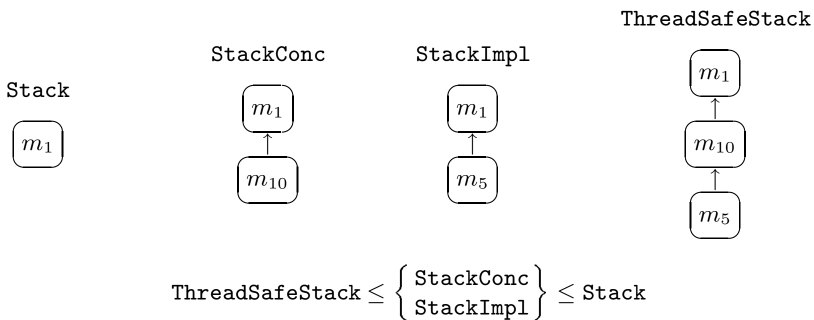


Fig. 1. The Stack classes, and their subclass relations

In Fig. 2 the contributions to push in ThreadSafeStack are listed. The 'combined result' in Fig. 2 is similar to push in that it has the same signature and will

execute the same statements in the same order, so it presents an overview of the semantics of push. As desired, the implementation (in m_7) is enclosed between the acquisition and the relinquishment of the semaphore (the P and V operations in m_{12}). Consequently, at most one thread can execute the m_7 part of push (or similarly for pop) on a ThreadSafeStack at the same time, i.e., storage has been put under concurrency control.

```
From m1:  3(elm: Element)(# do INNER #)
From m10: 12(# do mutex.P; INNER; mutex.V #)
From m5:  7(# do storage.insert(elm) #)
Combined result:
  (elm: Element)(#
  do mutex.P; storage.insert(elm); mutex.V
  #)
```

Fig. 2. The contributions to push in ThreadSafeStack

For the explanation of how push was determined to be $[m_3, m_{12}, m_7]$, we must consider the enclosing class, ThreadSafeStack. Note that it is not necessary, or even relevant, to know if or how the enclosing entity was constructed by combining other entities.

A virtual method or class is computed by combining the contributions to it, from the most general mixin of the enclosing class or method, and towards more specific ones. In this example the enclosing class is ThreadSafeStack, or $[m_1, m_{10}, m_5]$, so push is computed as follows:

$$[m_3]\&[m_{12}]\&[m_7] = [m_3, m_{12}, m_7]$$

The contributions to a virtual need not be singleton lists in general, but for the rather common case that they are singletons, we can simply collect them in order from the enclosing entity.

Note that we could have combined the concurrency control with another implementation, and the implementation with zero or more auxiliary aspects such as concurrency control. In general, we can build classes from aspects such that the implementation of methods contains contributions from any or all of those aspects.

In CLOS, a similar result could have been obtained (not in a type safe manner, though) by putting statements before INNER into a 'before' method and statements after INNER into an 'after' method, and letting ThreadSafeStack inherit from StackConc and StackImpl, in that order. This illustrates that propagation in one level from classes to enclosed methods amounts to a similar mechanism as CLOS standard method combination. The relation to Beta method combination is obvious (both Beta and gbeta use INNER for this), but there is much more freedom to build mixin lists in gbeta. In fact, the list of mixins in

any class or method in BETA is fully determined when the most specific mixin is known. In other words, mixins in BETA are *not* liberated from the superclass—so they are not really mixins, but just ordinary class increments. That is the most fundamental difference between BETA and gbeta.

2.2 Combination of Methods, Propagating to Classes

In this section we consider an example where the combination operator '&' is applied to a group of classes used in a method, and by propagation combines aspects of each of the members of the group. Assume that we have defined some auxiliary classes supporting payments and delivery of things:

```
class Person: [13](# object name: String #);
class Payer: Person(# method pay: (amnt:Integer)(#..#):Integer#);
class Paid: Person[14](# method accept: (amnt:Integer)(# .. #)#);
class Receiver: Person(# method receive: (t:Thing)(# .. #)#);
class Deliverer: Person[15](# method deliver: (# .. #):Thing #);
```

A Person has a name; a Payer can pay an amount of money, and a Paid person can accept amounts of money. Moreover, a Receiver can receive a Thing and a Deliverer can deliver a Thing.

Collaborations for these classes arise naturally. An example could be the activity "to pay" (the argument list of pay is inherited from collaboration):

```
method collaboration: (fst:First, snd:Second) (#
   virtual class First: Person;
   virtual class Second: Person;
do INNER
#);
method pay: collaboration(#
   extended class First: Payer;
   extended class Second: Paid;
   virtual method price: (# do INNER #): Integer;
do snd.accept(fst.pay(price));
   INNER
#)
```

In collaboration it may be confusing that the types of the arguments are the virtual classes First and Second declared inside collaboration, but that is actually the case. The problem only arises here because of the adjustments of the syntax to a more mainstream style; in BETA/gbeta the method arguments are declared inside the block, so the usage of other attributes inside the block is quite natural.

The collaboration method introduces two *roles*, played by fst and snd, and specified by First and Second. The pay method enhances the collaboration method by extending the role classes First and Second, and by adding one statement to the behavior in which snd accepts the payment from fst, as specified by price.

We can create a similar activity for the transfer of possession of some item, where the `snd` role player delivers a `Thing` which is then `received` by the `fst` role player:

```
method deliver: collaboration(#
   extended class First: Receiver;
   extended class Second: Deliverer;
do fst.receive(snd.deliver);
   INNER
#)
```

With these activities in place we can create a combination method which supports the combination of the activities—both transferring an amount of money and transferring an entity in exchange for the money:

```
(# method doTrade: pay & deliver;
   object diamond: Thing;
   object walrus: Paid & Receiver & Deliverer;
   object lucy: Payer & Receiver;
do
   walrus.receive(diamond);
   doTrade(lucy,walrus)
#)
```

In this piece of code we create the above mentioned combined method `doTrade`, thus by propagation combining the nested virtual classes `First` and `Second` from the contributions in `pay` and `deliver`. For example, `Second` in `doTrade` is `Paid & Deliverer`, i.e.:

$$[m_{13}, m_{14}]\&[m_{13}, m_{15}] = [m_{13}, m_{14}, m_{15}]$$

This ensures that an object which is to play the `snd` role has both an `accept` and a `deliver` method.

The behavior of `pay` and `deliver` is combined, due to the `INNER` mechanism, such that both the transfer of money and the transfer of things will occur. The object `diamond` is created, so we have something to transfer. Two role player objects, `walrus` and `lucy`, are created with the necessary mixins. Since the `walrus` must first `receive` the `diamond` in order to be able to `deliver` it to `lucy`, there is both a `Deliverer` and a `Receiver` aspect of `walrus`; `lucy` could have been a `Deliverer`, too, but she probably won't.

2.3 Growing a Family of Classes

The last example seems to be almost compulsory in papers about advanced languages and type systems recently [5, 24], but in this case we emphasize the possibility of distributing the implementation over several levels of specialization, in order to deal with various concerns as "soon" as possible—that is, at the most general level where the necessary information is available. This is an example

of combining classes which contain other classes which again contain methods, propagating the combination operations as usual.

Figure 3 specifies a class `ObserverDesignPattern` which can be used to support the Observer design pattern [9]. It contains two nested mutually recursive classes `Subject` and `Observer`. An instance of `Observer` may `attach` to an instance of `Subject`. Once inserted into the `Set` of `observers` for that `Subject` it will be a target for notifications. The `scan` method on `Set` is the standard device in BETA/`gbeta` for iteration over all elements in a collection.

`scan` contains a declaration of a reference `current`; it executes `INNER` once for each element in the collection, with `current` referring to the current element. Hence, `notify` invokes `update` on each member of `observers` with `this(Subject)` (the enclosing instance of `Subject`) as an argument. The observer may then update its own state according to the changes in the given subject.

```
class ObserverDesignPattern: (#
   virtual class Subject: (#
      method attach: (o:Observer)(# do observers.insert(o) #);
      method detach: (o:Observer)(# do observers.delete(o) #);
      method notify: observers.scan
         (# do current.update(this(Subject)) #);
      object observers: Set
         (# extended class Element: Observer #)
   #);
   virtual class Observer: (#
      virtual method update: (s:Subject)(# do INNER #)
   #)
#)
```

Fig. 3. Support for the Observer design pattern

Each (significant) change in the state of the `Subject` should be followed by an invocation of `notify` (it is a programmer responsibility to remember to invoke `notify` at the right places).

To use this we need a couple of auxiliary classes, for instance a `TextBuffer` to be observed by a `Window` which could be an instance of the subclass `ColorIcon`:

```
class TextBuffer: (#
   object name: String;
   virtual method setFileName: (n:String)(# do name := n #);
   virtual method getFileName: (# do return name #): String;
#);
class Window: (# method refresh: (# .. #); .. #);
class ColorIcon: Window(#
   method setIconTitle: (s:String)(# .. #);
#)
```

Now we can create a subclass of the `ObserverDesignPattern` which extends the virtual classes and thereby lets `Windows` observe a `TextBuffer`:

```
class WindowAndTextODP: ObserverDesignPattern(#
   extended class Subject: TextBuffer(#
      (* ensure that 'notify' is called after changes *)
      extended method setFileName: (# do INNER; notify #)
   #);
   extended class Observer: Window(#
      extended method update: (#~do~getState(s);refresh~#);
      virtual method getState: (s:Subject)(# do INNER #)
   #)
#)
```

We could now use the class `WindowAndTextODP` as a class family aspect, combining it with some other class family aspects that contribute something else to `Subject` and `Observer`; this would be an example of a two-level propagation: From combined class families to member classes to methods of each member class. But for brevity we will use `WindowAndTextODP` directly in the following.

At the level of `WindowAndTextODP` we can do part of the notification work: When an `Observer` learns that the `Subject` has changed (i.e., when `notify` invokes `update` with that observer as `current`) then we can get the state and `refresh` the `Window`. We do not yet know *how* to get the state, but that's a deferred virtual method so we can put it in later. Finally we can create an instance of the design pattern, `myODP`, and populate it with a subject `myBuffer` and an observer `myIcon`:

```
object myODP: WindowAndTextODP;
object myBuffer: myODP.Subject;

object myIcon: ColorIcon & myODP.Observer (#
   extended method getState:
      (# do setIconTitle(s.getFileName)#)
#)
```

The class of `myIcon` has two super-classes, `ColorIcon` and `myODP.Observer`. The first is a standard GUI support class, and the second contributes the design pattern related aspect, and the block provides the implementation of `getState`. This implementation depends on the following information from the static analysis:

- `s` is (a descendent of) a `TextBuffer` because `myODP` is a `WindowAndTextODP` which declares `extended class Subject: TextBuffer(#..#)`, so it has a `getFileName` method which takes no arguments and delivers a `String`
- `myIcon` is a `ColorIcon`, so it has a `setIconTitle` method which takes a `String` argument

This could not be type checked if `s` in the body of `getState` only had the the type declared in the original `ObserverDesignPattern`. However, in both Beta

and gbeta, a virtual class/method attribute *is* recognized by the type system as denoting a subclass/submethod when looked up in context of a more specialized enclosing class or method.

Compared to BETA, this example uses the support for extending virtual classes with *unrelated* classes—in BETA a virtual class must always be extended with a descendant of the value of that virtual class in the superclass (BETA has single inheritance). This means that in BETA we cannot use classes like `Window` and `TextBuffer` which have been written independently of the design pattern classes, only classes created especially for use in `ObserverDesignPattern` can be used.

With the approach in [5] and all other approaches with virtual *types* (or interfaces) as opposed to virtual *classes*, there is no support for creating instances of virtuals, or for putting state or implementation into virtuals, such as the `observers` and `notify` in `Subject`. As it is known from Java, this property can lead to duplication of code or manual delegation, to compensate for the fact that state and implementation cannot be inherited from interfaces.

By means of a mechanism called code *weaving*, AspectJ [21] is capable of adding statements to methods, and more. As the Subject/Observer example[2] shows, this approach can do similar things as gbeta, in some respects even more flexibly. There are of course many differences in the details, but one profound difference is that the types of fields and methods cannot be modified by aspects in AspectJ; for instance, there is a method `getData` in the example which has a return type of `Object`, and an dynamic cast is needed when that method is used. The gbeta example above is slightly different, but a method `getData` could easily be defined, returning a `Subject`, and that method would actually have a return type which is affected by the aspects being used. This means that no dynamic casts are required in gbeta, avoiding the potential run-time type error. Another example of this phenomenon was the `doTrade` method, where the types of the formal arguments were constructed by propagated combination. In short, AspectJ handles aspects of implementation, but not aspects of types.

3 Linearization and Propagation

In the previous section the usage of the complete language was in focus, with combination and propagation intertwined, and by example. This section presents the basic mechanisms separately and in more detail. The basic, propagationless mechanism behind class and method combination is a linearization algorithm, and propagation emerges with the semantics of virtuals. The linearization is presented and formally defined in Sect. 3.1, and the propagation mechanism is presented in Sect. 3.2 via a small functional language whose semantics is the core of gbeta.

[2] Available from [21], via ‘AspectJ Primer’

3.1 Linearization

The class and method combination mechanism in gbeta builds on a simple graph linearization algorithm which is presented in this section. The linearization is characterized as conceptually wholesome (previously [1] it has been described as a 'hybrid'); it is specified precisely what the linearization is; the linearization is generalized to handle cases which would otherwise be rejected; and some results about its properties are proved.

A graph linearization is an algorithm which constructs a list from a given directed graph, such that the list is a topological sorting of the nodes. Obviously a cyclic graph does not allow this, hence some graphs cannot be linearized. Since there are many possibilities for typical graphs, some systematic choices must be made in order to arrive at a well-defined result. Existing linearizations [6, 7, 1] are described in terms of such systematic choices of "what node to take next", and this makes it hard to understand their outcome, and to reason about their properties.

Luckily, the linearization which is used in gbeta, 'C3' [1], can be characterized in a much more declarative way, and it can even be generalized in a way that makes it a proper operation on a suitable set M:

$$\forall x, y \in M.\ x \,\&\, y \in M$$

The name C3 reflects three consistencies exhibited by this linearization, namely consistency with the local precedence order,[3] consistency with the extended precedence graph,[4] and monotonicity.[5] The other known linearizations (including the ones used in LOOPS, CLOS, and Dylan) do not have all three consistencies. The remaining problem (to be solved below) is that *none* of the linearizations can handle all inheritance hierarchies, some are rejected as inconsistent.

We will present the concrete algorithm first, and then proceed with the declarative characterization and the generalization. The concrete algorithm takes two (argument) lists and merges them into one (result) list by repeatedly choosing one element from the argument lists and transferring it to the result list, until all elements have been transferred. Figure 4 shows the rule for choosing and transferring one element. The linearization is then defined in terms of sequences of such steps:

$$A \,\&\, B = R \quad \text{iff} \quad (A, B, []) \;\rhd^*\; ([], [], R)$$

The process requires and preserves the property that the elements in each of the lists are distinct. As an example, we can demonstrate that $[a, b, c] \,\&\, [x, b] = [a, x, b, c]$ by the following process:

$$\begin{aligned}([a,b,c],[x,b],[]) &\;\rhd\; ([a,b],[x,b],[c]) \;\rhd\; ([a],[x],[b,c]) \\ &\;\rhd\; ([a],[],[x,b,c]) \;\rhd\; ([],[],[a,x,b,c])\end{aligned}$$

[3] The programmer-chosen ordering of direct superclasses.

[4] Which additionally orders classes according to the local precedence order from the most general common subclass.

[5] Avoidance of the phenomenon that an inherited feature is looked up in a class that none of the direct superclasses would have chosen.

$$([a_n \dots a_1],[b_m \dots b_1],[r_k \dots r_1]) \;\rhd\; ([a_n \dots a_1],[b_m \dots b_2],[b_1, r_k \dots r_1]), \quad \text{if } b_1 \notin \{a_1 \dots a_n\}$$

$$([a_n \dots a_1],[b_m \dots b_1],[r_k \dots r_1]) \;\rhd\; ([a_n \dots a_2],[b_m \dots b_1],[a_1, r_k \dots r_1]), \quad \text{if } b_1 \in \{a_1 \dots a_n\} \wedge a_1 \notin \{b_1 \dots b_m\}$$

$$([a_n \dots a_1],[b_m \dots b_1],[r_k \dots r_1]) \;\rhd\; ([a_n \dots a_2],[b_m \dots b_2],[a_1, r_k \dots r_1]), \quad \text{if } a_1 = b_1$$

Fig. 4. Defining ‘ $\rhd$ ’: How to take one step in a C3 linearization

The process may fail, as with $([a, b], [b, a], [])$ where no step can be taken and the configuration is not on the final form $([][], R)$. This is a problem because it means that some classes or methods cannot be combined; gbeta generates a ‘bad merge’ error message and rejects the program. As we shall see below, this problem can be overcome. It is, however, still future work to implement the generalized version of C3 in gbeta.

To reach a declarative characterization we must make a shift in mindset and terminology. If we regard the edges in a given acyclic oriented graph as a relation and take the reflexive and transitive closure of that, we get a partial order. Similarly, a list determines a total order. Hence, a linearization is a construction of a total order by adding elements to a partial order. Note that ‘$m_i < m_j$’ in this context means ‘m_i is closer to the rightmost end of a list of mixins than m_j’. This ordering is an entirely different concept than subclass ordering, which is concerned with the comparison of mixin lists as a whole.

C3 actually constructs a total order from a number of given total orders, namely the linearizations of the superclass hierarchies. The C3 principle can now be given for two order relations:

> **The C3 principle:**
> The linearization of two orders A and B, $A \,\&\, B$, is
> the union of A and B together with
> all non-contradictory edges from B to A

In other words, $A \,\&\, B$ just adds a default rule to A and B, namely that elements from B by default are smaller than elements from A. This is formalized straightforwardly below, but first we will have to establish a few facts.

Total preorders. We need to consider total preorders:

Definition 1. *A* total preorder *is a relation which is reflexive, transitive, and total. A* total order *is a total preorder which is also anti-symmetric.*

It is easy to prove that:

Lemma 1. *Assume $\preceq$ is a total preorder. The relation $\sim$ defined by $a \sim b \Leftrightarrow a \preceq b \wedge b \preceq a$ is an equivalence relation, and the relation $\leq$ on equivalence classes defined by $a^{\sim} \leq b^{\sim}$ iff $a \preceq b$ is well-defined and a total order.*[6]

Conversely, given an equivalence relation $\sim$ and a total order on the equivalence classes $\leq$, then the relation $\preceq$ defined by $a \preceq b \Leftrightarrow a^{\sim} \leq b^{\sim}$ is a total preorder.

In other words, a total preorder corresponds to a list of equivalence classes of elements, rather than a list of individual elements.

This is the desired generalization: to construct a *list of groups* of mixins, each group consisting of mixins considered equally specific.

In such a setting, clashing names are not always disambiguated. This might at first seem to be a step backwards; it is in fact an improvement. When the ordinary C3 linearization would succeed, the generalization delivers the same result (all groups have size one). When the hierarchy would be rejected by ordinary C3, the resulting non-trivial groups from generalized C3 would in many cases work quite well. For example, as long as a name is only declared in one of the mixins in a given group, there will be no clashes on that name. In fact, a number of inheritance hierarchies would be *better* described by making certain mixins equally specific, since the commitment to one order causes unnecessary restrictions on future usage.

Formalization of C3. We need a couple of tools before C3 can be formalized:

Definition 2. *When R is a relation, its* domain *is* $\mathrm{dom}(R) \triangleq \{y|(\exists z.\ (y,z) \in R) \vee (\exists x.\ (x,y) \in R)\}$, *its* inversion *is* $\overline{R} \triangleq \{(y,x)|(x,y) \in R\}$, *its* one-step transitive closure *is* $R^{+1} \triangleq R \cup \{(x,z)|\exists y.(x,y),(y,z) \in R\}$, *and its* transitive closure *is* $R^{*} \triangleq \bigcup_{i \in \omega} R_i$, *where* $R_0 \triangleq R$, $\forall i \in \omega.\ R_{i+1} \triangleq R_i^{+1}$.

The following lemma is immediate from the definitions:

Lemma 2. *Let R and S be relations. Then R^* is transitive. The domain is additive:* $\mathrm{dom}(R \cup S) = \mathrm{dom}(R) \cup \mathrm{dom}(S)$. *The domain is preserved by transitive closure and inversion:* $\mathrm{dom}(R^*) = \mathrm{dom}(\overline{R}) = \mathrm{dom}(R)$. *Reflexivity is preserved by transitive closure, inversion, and union: if* $\forall x \in \mathrm{dom}(R).\ x \preceq_R x$ *then* $\forall x \in \mathrm{dom}(S).\ x \preceq_S x$, $S \in \{R^*, \overline{R}\}$, *and if* $\forall x \in \mathrm{dom}(T).\ x \preceq_T x$, $T \in \{R, S\}$ *then* $\forall x \in \mathrm{dom}(R \cup S).\ x \preceq_{R \cup S} x$.

---

[6] $a^{\sim}$ denotes the equivalence class wrt. $\sim$ containing a

The formalization of C3 is:

Definition 3 (C3 Linearization). *Let* R_1 *and* R_2 *be relations. The* C3 linearization *of* R_1 *and* R_2 *is* $R_1 \,\&\, R_2 \;\triangleq\; R \cup (\mathrm{dom}(R_2) \times \mathrm{dom}(R_1) \setminus \overline{R})$*, where* $R \triangleq (R_1 \cup R_2)^*$.

Intuitively, the linearization combines the two given relations R_1 and R_2 into $(R_1 \cup R_2)$ which is then "repaired" to be a transitive relation R by taking the transitive closure. R is complemented with everything from $\mathrm{dom}(R_2)\times\mathrm{dom}(R_1)$ which does not contradict R. In other words, R_2 elements are smaller than R_1 elements, unless something is known to the contrary.

Now we can state the closure property that makes total preorders interesting:

Proposition 1. *Assume* R_1 *and* R_2 *are total preorders. Then* $R_1 \,\&\, R_2$ *is a total preorder.*

Proof. Let $R \triangleq (R_1 \cup R_2)^*$ as in the definition, and let $S \triangleq \mathrm{dom}(R_2)\times\mathrm{dom}(R_1)\setminus\overline{R}$. Observe that R and $\overline{R}$ are reflexive because union, transitive closure, and inversion preserve reflexivity. Moreover, since $\mathrm{dom}(S) \subseteq \mathrm{dom}(R_1) \cup \mathrm{dom}(R_2) = \mathrm{dom}(R)$, also $R_1 \,\&\, R_2$ is reflexive—"S does not touch any new elements compared to R."

For transitivity, note that $R_1 \,\&\, R_2 = R \cup S$, and $(x, y) \in S \Rightarrow (y, x) \notin R$. Assume $(x, y), (y, z) \in R_1 \,\&\, R_2$. We show that $(x, z) \in R_1 \,\&\, R_2$:

- If $(x, y), (y, z) \in R$, then $(x, z) \in R \subseteq R_1 \,\&\, R_2$ because R is transitive.
- If $(x, y) \in R$ and $(y, z) \in S$ then $(z, y) \notin R$ by definition of S. If $(z, x) \in R$ then by transitivity of R we get $(z, y) \in R$, contradiction, hence $(z, x) \notin R$. Observe that $y \in \mathrm{dom}(R_2)$ and $z \in \mathrm{dom}(R_1)$ because $(y, z) \in S$. If $x \in \mathrm{dom}(R_2)$ then $(x, z) \in \mathrm{dom}(R_2)\times\mathrm{dom}(R_1)\setminus\overline{R} = S \subseteq R_1 \,\&\, R_2$. Otherwise $x \in \mathrm{dom}(R_1)$, but then $(z, x) \in \mathrm{dom}(R_1)^2$, and by totality of R_1, $(x, z) \in R_1 \vee (z, x) \in R_1$. Since $(z, x) \in R_1$ contradicts $(z, x) \notin R$ we must have $(x, z) \in R_1 \subseteq R_1 \,\&\, R_2$.
- The case $(x, y) \in S$ and $(y, z) \in R$ is similar.
- If $(x, y), (y, z) \in S$ then $x \in \mathrm{dom}(R_2)$, $y \in \mathrm{dom}(R_1) \cap \mathrm{dom}(R_2)$, and $z \in \mathrm{dom}(R_1)$. Moreover $(y, x), (z, y) \notin R$, by definition of S. Then $(x, y) \in R_2$ by totality of R_2, and $(y, z) \in R_1$ by totality of R_1, hence $(x, y), (y, z) \in R$ and by transitivity of R finally $(x, z) \in R \subseteq R_1 \,\&\, R_2$.

For totality of $R_1 \,\&\, R_2$ we choose arbitrary $x, y \in \mathrm{dom}(R_1 \,\&\, R_2)$, and show that either $(x, y) \in R_1 \,\&\, R_2$ or $(y, x) \in R_1 \,\&\, R_2$:

- If $x, y \in \mathrm{dom}(R_1)$ then $(x, y) \in R_1 \vee (y, x) \in R_1$ by totality of R_1.
- If $x \in \mathrm{dom}(R_1)$ and $y \in \mathrm{dom}(R_2)$ then either $(y, x) \in R$ or $(x, y) \in S$.
- The remaining two cases are similar.

This proves that $R_1 \,\&\, R_2$ is reflexive, transitive, and total, i.e. it is a total preorder. □

The ordinary C3 fails precisely when the generalized C3 produces a total preorder which is not a total order. A total preorder is a total order if and only if there are no cycles, so we need to consider them:

Definition 4. *Let* R *be a relation. A sequence of distinct elements* $d_1 \ldots d_n \in \mathrm{dom}(R)$*,* $n \geq 2$*, is a* cycle *in* R *iff* $(\forall i \in 1 \ldots n{-}1.\ (d_i, d_{i+1}) \in R) \wedge (d_n, d_1) \in R$*.* R *is* acyclic *iff there are no cycles in* R*.*

Lemma 3. *Let R be a reflexive, acyclic relation. Then R^* is reflexive and acyclic.*

Proof. Given a reflexive, acyclic relation R. With $R_0 \triangleq R$, $\forall i \in \omega.\ R_{i+1} \triangleq R_i^{+1}$ we have $R^* = \bigcup_{i\in\omega} R_i$. Assume that R^* has a cycle and let $k \in \omega$ be the least number such that R_k has a cycle, say $d_1 \dots d_n$; then $k > 0$ because R is acyclic. Since

$$\{(d_i, d_{i+1}) | i \in 1 \dots n-1\} \cup \{(d_n, d_1)\} \subseteq R_k$$

and $R_k = R_{k-1}^{+1}$ we can choose $c_1 \dots c_n \in \mathrm{dom}(R)$ such that

$$\begin{array}{l} \forall i \in \{1 \dots n-1\}.\ (\{(d_i, c_i), (c_i, d_{i+1})\} \subseteq R_{k-1}) \vee ((d_i, d_{i+1}) \in R_{k-1})) \\ \wedge \\ (\{(d_n, c_n), (c_n, d_1)\} \subseteq R_{k-1}) \vee ((d_n, d_1) \in R_{k-1}) \end{array}$$

which provides us with a cycle in R_{k-1}, contradicting the minimality of k. □

We can now state and prove the "non-pre" equivalent of proposition 1:

Proposition 2. *Assume R_1 and R_2 are total orders and $R_1 \cup R_2$ does not have cycles. Then $R_1 \,\&\, R_2$ is a total order.*

Proof. Since R_1 and R_2 are total preorders we get reflexivity, transitivity, and totality directly from proposition 1. Only anti-symmetri remains to be proved. Assume that $(x, y), (y, x) \in R_1 \,\&\, R_2$; we must prove that $x = y$: Let $R \triangleq (R_1 \cup R_2)^*$ and $S \triangleq \mathrm{dom}(R_2) \times \mathrm{dom}(R_1) \setminus \overline{R}$ such that $R_1 \,\&\, R_2 = R \cup S$ and $(x, y) \in S \Rightarrow (y, x) \notin R$.

- If $(x, y), (y, x) \in R$ then $x = y$ since R is acyclic, by lemma 3.
- Both $(x, y) \in R \wedge (y, x) \in S$ and $(y, x) \in R \wedge (x, y) \in S$ are impossible by definition of S.
- Similarly, if $(x, y), (y, x) \in S$ then $(y, x), (x, y) \notin R$. This is a contradiction since $x, y \in \mathrm{dom}(R_1) \cap \mathrm{dom}(R_2)$ and in particular by totality of R_1, $(x, y) \in R_1 \vee (y, x) \in R_1 \subseteq R$.

□

We have seen that C3 can be formalized in a rather obvious manner and proved that the formalization has the nice stability property of proposition 1 and the incomplete stability property of proposition 2. It seems to be worthwhile to try to develop the strict linearization of various languages into the more wholesome total preorder model, going from class precedence lists to class *group* precedence lists. As mentioned, this has not yet been implemented in gbeta; the main problem is that the existence of unordered mixins in methods would conflict with the usage of `INNER` to determine the combined behavior.

It is not hard to see that the algorithmic version of C3 actually implements the formalization presented here—the algorithm each time selects the least remaining element according to our formalization of C3.

As an aside it is interesting to note that gbeta actually used a quite different algorithm for linearization during a period of more than a year. Only after the above formalization was created did it become clear that the C3 algoritm (which was simpler and therefore attractive) solved the exact same problem, because both algorithms clearly implement the formal characterization of the linearization. Algorithms *are* generally harder to compare and reason about than declarative formalizations like Def. 3.

3.2 The Propagation Mechanism

This section presents the propagation mechanism in gbeta indirectly, by describing an untyped functional core of gbeta, **gb**. This core expresses the essence of the semantics of object creation and attribute lookup in gbeta, including the semantics of virtuals and the combination mechanism. There is syntax for specifying a program, a rule for building a class from such a program, a rule for creating an object as an instance of a given class, and a rule for looking up a name in a given object.

The abstract syntax for **gb** programs is given in Fig. 5. It includes blocks (corresponding to `(# .. #)` blocks in gbeta), descriptors (blocks with indication of a superclass), and specifications (the right hand side of declarations). The l denotes labels, i.e., a predefined set of identifiers. The only label with a predefined semantics is '`object`' which is the class with no mixins. The syntax includes only one kind of attribute declarations, corresponding to virtual declarations in gbeta. So all attributes are virtual classes (or virtual methods—there is no semantic difference).

$b = (\# \; l_i : s_i \;^{i \in I} \; \#)$	(block)
$d = l \; b$	(descriptor)
$s = l \mid d$	(specification)
l	(label)

Fig. 5. The abstract syntax of **gb**

There is no statement syntax, but the rules for creating instances and looking up names can be applied repeatedly, so objects and classes at any level of nesting in the program can be created.

The semantic entities are shown in Fig.6. They include the syntax as Block, Descriptor, and Spec. The central concept of mixin is represented by Mixin which is a block in an environment. A class is simply a list of mixins.[7] An environment, Env, is not only the enclosing object but the list of all enclosing objects, ending in the outermost object, which contains everything in the program execution. An Object is a set of attributes, and an Attribute is a pair of a label and its value. The value of an attribute is a list of specifications, each in its own environment.

Since the result of looking up a label in **gb** is always a Class, it would have been natural to use the definition Attribute $=$ Label $*$ Class, but that definition

[7] Note that the notation used for mixin lists elsewhere in this paper has the most specific mixin *rightmost*, since that yields the most natural notation for '`&`' expressions. In this section we put the most specific mixin at the *left* end (reverting the lists), because that is necessary for the standard notation, '$h :: t$', which names the head, h, and the tail, t, of a list by pattern matching (as in SML and other functional languages).

$$\begin{aligned}\mathsf{Block} &= (\mathsf{Label} * \mathsf{Spec})\ \mathsf{set}\\ \mathsf{Descriptor} &= \mathsf{Label} * \mathsf{Block}\\ \mathsf{Spec} &= \mathsf{Label} \mid \mathsf{Descriptor}\\ \mathsf{Class} &= \mathsf{Mixin\ list}\\ \mathsf{Mixin} &= \mathsf{Env} * \mathsf{Block}\end{aligned}$$

$$\begin{aligned}\mathsf{Env} &= \mathsf{Object\ list}\\ \mathsf{Object} &= \mathsf{Attribute\ set}\\ \mathsf{Attribute} &= \mathsf{Label} * \mathsf{EnvSpec\ list}\\ \mathsf{EnvSpec} &= \mathsf{Env} * \mathsf{Spec}\end{aligned}$$

Fig. 6. Semantic Entities

conflicts with the dynamic semantics for objects which contain self references. The definition of Attribute in Fig. 6 is one way to handle recursive objects, namely by evaluating specifications lazily.

From Program to Class to Object. A **gb** program is a block. For a given program b we construct the initial class $[([\,], b)]$, which contains one mixin which places b in the empty environment. This class can then be instantiated like any other class, and that initiates the **gb** 'execution'—which is a chain of evaluations of $\textsc{New}(\cdot)$ and $\textsc{Lookup}(\cdot,\cdot)$.

Any given class can be instantiated using the function $\textsc{New}(\cdot)$ which takes a class and yields an object. It is defined in terms of the auxiliary functions $\textsc{Labels}(\cdot)$ and $\textsc{Val}(\cdot,\cdot)$. See Fig. 7.

$$\begin{aligned}
\textsc{New}(C : \mathsf{Class}) &= \{\ (l, \textsc{Val}(l, C)) \ \mid\ l \in \textsc{Labels}(C)\}\\
\forall j \in I.\ \ \textsc{Val}(l_j, (\#\ l_i : s_i\ {}^{i\in I}\ \#)) &= s_j\\
\textsc{Val}(l, (e, b) : \mathsf{Mixin}) &= \begin{cases} [(e, \textsc{Val}(l, b))] & \text{if } l \in \textsc{Labels}(b)\\ [\,], & \text{otherwise}\end{cases}\\
\textsc{Val}(l, [\,] : \mathsf{Class}) &= [\,]\\
\textsc{Val}(l, (h :: t) : \mathsf{Class}) &= \textsc{Val}(l, h) +\!\!+ \textsc{Val}(l, t)\\
\textsc{Labels}((\#\ l_i : s_i\ {}^{i\in I}\ \#)) &= \{l_i \mid i \in I\}\\
\textsc{Labels}((e, b) : \mathsf{Mixin}) &= \textsc{Labels}(b)\\
\textsc{Labels}([\,] : \mathsf{Class}) &= \emptyset\\
\textsc{Labels}((h :: t) : \mathsf{Class}) &= \textsc{Labels}(h) \cup \textsc{Labels}(t)
\end{aligned}$$

Fig. 7. Creation of objects ($+\!\!+$ concatenates lists)

Figure 8 presents the semantics attribute lookup. Given an object O and a label l, $\textsc{Lookup}(O, l)$ delivers the result of looking up l in O. It yields a class if l is defined in O, and raises an error otherwise. To lookup l in O we search the labels of O using $\mathrm{L}_{\mathrm{object}}(O, \cdot, l)$. If we find l then we have an EnvSpec list, ess, which is then looked up in O using $\mathrm{L}_{\mathrm{envspecs}}(O, ess)$. Note that ess is the result

of collecting all contributions to a given attribute—**gb** has virtual attributes, only.

The next step is crucial. The use of $\mathrm{C3}(\cdot,\cdot)$ in the definition of $\mathrm{L}_{\mathrm{envspecs}}(\cdot,\cdot)$ constructs the virtual by C3 linearizing all the contributions. A similar core language for BETA would not linearize at this point; it would *replace* the definition in the less specific enclosing class with the definition in the more specific one. Moreover, the static analysis ensures that this always replaces the virtual class with a descendant. Since $A \,\&\, B \leq X$ for $X \in \{A, B\}$ and $A \,\&\, B = B \,\&\, A = B$ whenever $B \leq A$, the BETA semantics comes out as a special case of the gbeta semantics. Finally, $\mathrm{L}_{\mathrm{env}}(\cdot,\cdot)$ is used to look up labels in the given environment e, enhanced with the current object to $O :: e$; this (very late) enhancement of the environment to include the current object is actually the essence of the lazy evaluation that makes it possible to handle recursion. This ends the brief presentation of **gb**.

$$
\begin{aligned}
\textsc{Lookup}(O : \mathsf{Object}, l : \mathsf{Label}) &= \mathrm{L}_{\mathrm{object}}(O, O, l)\\
\mathrm{L}_{\mathrm{object}}(O, [\,] : \mathsf{Object}, l) &= \text{raise Undefined}\\
\mathrm{L}_{\mathrm{object}}(O, ((l', \mathit{ess}) :: t) : \mathsf{Object}, l) &= \begin{cases} \mathrm{L}_{\mathrm{envspecs}}(O, \mathit{ess}), & \text{if } l = l'\\ \mathrm{L}_{\mathrm{object}}(O, t, l), & \text{otherwise}\end{cases}\\
\mathrm{L}_{\mathrm{envspecs}}(O, [\,] : \mathsf{EnvSpec\ list}) &= [\,]\\
\mathrm{L}_{\mathrm{envspecs}}(O, (h :: t) : \mathsf{EnvSpec\ list}) &= \mathrm{C3}(\mathrm{L}_{\mathrm{envspec}}(O, h), \mathrm{L}_{\mathrm{envspecs}}(O, t))\\
\mathrm{L}_{\mathrm{envspec}}(O, (e : \mathsf{Env}, l : \mathsf{Label})) &= \mathrm{L}_{\mathrm{env}}(O :: e, l)\\
\mathrm{L}_{\mathrm{envspec}}(O, (e : \mathsf{Env}, (l, b) : \mathsf{Descriptor})) &= (O :: e, b) :: (\mathrm{L}_{\mathrm{env}}(O :: e, l))\\
\mathrm{L}_{\mathrm{env}}([\,] : \mathsf{Env}, l) &= \begin{cases} [\,], & \text{if } l = \text{"}\mathtt{object}\text{"}\\ \text{raise Undefined}, & \text{otherwise}\end{cases}\\
\mathrm{L}_{\mathrm{env}}((h :: t) : \mathsf{Env}, l) &= \begin{cases} \textsc{Lookup}(h, l), & \text{if } l \in \textsc{Labels}(h)\\ \mathrm{L}_{\mathrm{env}}(t, l), & \text{otherwise}\end{cases}
\end{aligned}
$$

Fig. 8. Looking up a label in an object

The Relation to gbeta. The core language **gb** described in the previous section is of course very different from gbeta. It is purely functional, so the **gb** objects (in environments) are replaced with store locations ("pointers") in gbeta. In **gb**, names are matched according to their spelling. Since gbeta uses static name-binding, the identification of names in **gb** is much more inclusive (**gb** considers two declarations related in many cases where gbeta considers them unrelated). To obtain the effect of static name binding in **gb** we would need to rename identifiers in a given program, but since the static analysis of gbeta determines exactly what names are equivalent, it is certainly a tractable problem to choose new names such that only the gbeta-equivalent names are spelled identically.

In **gb**, the immanent recursion of objects is handled using lazy evaluation of attributes. In gbeta, the exact mixins contributing to a given declaration are determined at compile-time,[8] and cycles (e.g., a class which indirectly inherits from itself) are detected using a graph coloring algorithm: Whenever the type of a declaration depends on itself, the program is rejected with a 'cyclic dependency' error message. The run-time context is represented *relative* to a current object in the gbeta static analysis, since the actual objects are of course not available before run-time.

However, **gb** accurately reflects the semantics of looking up names in gbeta, starting with declarations in the currently selected mixin (e.g., the method being executed) and continuing through all enclosing objects until the outermost "universe" object is reached. Similarly, the semantics of virtuals is the same in **gb** and gbeta (apart from the name binding issue which was mentioned above). Each attribute includes the full context (potentially many objects) in **gb**, but this has been reduced to one pointer shared by several attributes in gbeta. The semantics comes out clearer and simpler with the complete environment attached to each attribute. Note that **gb** does not need to include the explicit linearization operator '`&`' since the semantics of that operator can be obtained using a couple of auxiliary classes and virtuals. This is because virtual class contributions are linearized with C3, just like '`&`' expressions.

4 Implementation

Our implementation of gbeta has been available on the Internet (with source code under GPL) since August 1997.[9] It is implemented in BETA. We wrote about 70 KLOC specifically to implement gbeta. Since gbeta generalizes the semantics of BETA at such a fundamental level the implementation essentially had to start from scratch, but of course the techniques used in the implementation of BETA, e.g. in the type checker, have been an important starting point for the more general approach in gbeta.

However, the language and its implementation are still under development, and in particular there are many possibilities for improving the performance. The primary goal so far has been to develop the language design, and to provide an implementation which would allow hands-on experience with the consequences of design decisions. The practical experience has been an invaluable source of feed-back, both when small test programs revealed new ways to use a given construct or when they demonstrated problems with design decisions, but also when a particular design proved hard to type check and consequently turned out to be be ill defined.

As mentioned above, the source code is freely available. So far the project has been more cathedral than bazaar [22], but collaboration on the further development of gbeta is indeed welcome!

[8] Unless we use the support for dynamic inheritance, not covered in this paper

[9] See `http://www.daimi.au.dk/~eernst/gbeta/`

5 Related Work and Discussion

The basis on which this work has been built is the Scandinavian tradition of object-orientation, in particular the BETA community [19, 14, 4]. In relation to this basis, gbeta represents a generalization which is intended to further develop rather than revolutionize. However, as mentioned in Sect. 4, the fundamental mechanisms of gbeta—the new semantics of virtuals and the support for class combination—are so different that a complete reimplementation was required.

To our knowledge there are no existing systems which support a mechanism like our propagation.

CLOS [12] was an important source of inspiration, and gbeta does not have an equivalent of the CLOS meta-object protocol; it is probably necessary for static type checking to lay down a firm foundation for the language and not let *everything* be user-definable. Thus, in addition to the propagation mechanism, gbeta offers static type checking and the more general combination with "&" which applies both to classes and methods; standard CLOS method combination only applies to generic functions, and only as in the special case of one-level propagation from classes to enclosed methods.

Several efforts aim at supporting advanced combination of separate entities (subjects, aspects, ...), in order to allow for a better separation of concerns which otherwise appear to be scattered globally. We believe that such efforts and gbeta basically attack the same problem from two sides, and generally a seamless language integration is a desirable end-point for any of these approaches.

In subject oriented programming [10] each subject is a separate "universe" consisting of fragments of classes in the system. This makes it possible for one (complete) class to participate in several different subjects (universes) with different interfaces in each subject, hence allowing designers to concentrate on one perspective at a time and later combine the subjects to complete systems. This allows for large-scale combination strategies. However, the interaction between the different subjects in any given class must be resolved explicitly, e.g., whether a given name denotes a shared entity in two subjects or two separate entities. In gbeta, the propagating combination of large systems (groups of groups of families of classes etc.) allows for implicit resolution of those matters in a modularized fashion: E.g., two entities are shared iff they are statically recognized as being the same, because they are declared in the same declaration of a shared superclass.

AspectJ [13, 21] supports combination of aspects in Java which amounts to the *weaving* of method implementations and class members from separate *aspects*. Otherwise globally scattered but logically connected pieces of code can then be expressed in separate (aspect) modules and combined in flexible ways. Certain features in AspectJ have no parallel in gbeta, such as selecting the methods in which to put a given aspectual piece of code by an expression which may contain wildcards ("add this statement to the beginning of all methods matching the expression `Point.*(*)`"). AspectJ is currently implemented as a textual pre-processor which generates Java-code. Even though some type checking can be carried out on pre-weaved source code there is always a *transition* which must

be dealt with, in type-checking and error reporting, and even in compilation and optimization. We believe that full language integration is a natural goal to strive for. The challenge is to reach this goal without sacrificing the flexibility; `gbeta` demonstrates one possible approach, entirely language integrated and with strict type analysis.

In [20] the notion of Adaptive Plug-and-Play Components (APPCs) is introduced. An APPC is a construct which aims to capture collaboration between classes at a level of granularity between traditional classes (too small for reuse) and modules (too large, inflexible). An *interface class graph* is used to specify the bindings between an APPC and a concrete class graph, such that a given APPC applies to different contexts regardless of such details as choice of names of classes and methods. Moreover, results from the work on Adaptive Programming [17, 16] are applied, ensuring that attributes can be looked up across a varying path of part objects (e.g., both `anObj.target` and `anObj.xyz.target` can be accessed from a given `anObj` using the same specification in an APPC). In `gbeta` there is no parallel to the adaptive programming path abstraction. It would probably be possible to add such a mechanism to `gbeta`, but that would be orthogonal to the rest of the language—a separate concern. However, the binding of a virtual class to a concrete class in `gbeta` supports a similar detachment of the collaboration (entity with nested virtual classes) from concrete choice of names as the class interface graph does with APPCs.

The other line of related work is about mixins. They were introduced as a particular usage of multiple inheritance in CLOS and later formalized and developed as an independent notion which can subsume the concepts of inheritance and class. The ground-breaking article [2] presents the notion of mixins and mixin combination as a generalization of several known inheritance models, clearly influencing this work. Linearizing multiple inheritance is described as a mechanism which supports mixin style inheritance, but adds too much complexity. In `gbeta`, linearization was introduced in order to have a mechanism for large-scale mixin combination, and for implicit propagation of mixin combination. The problems usually associated with linearization are greatly reduced in `gbeta`: firstly, the linearization in use has the three consistency properties mentioned in section 3.1. Secondly, the static name binding enables programmers to inspect at compile-time exactly what declaration any given name application refers to.

In [3] a small and beautifully formalized set of operations on modules (mixins) is defined, supporting many different notions of inheritance and class combination, although with quite low-level operations. Since there is no particular description of a coherent high-level system of usage of those primitives, this is a work which mostly serves to solidify the foundation of more high-level approaches. The notion of mixins is different in [23] where each mixin (method) is nested in a class; by executing a mixin method the object changes class and, e.g., obtains some new instance variables. In this approach each class must foresee every possible extension, and every generally applicable class must be a member of the top class "`object`". This does not seem very manageable in a large-scale

development project, but a very good module system might help. Finally, [8] presents an extension of a subset of Java with mixins, called MixedJava. Like in our approach, the interaction between a mixin and the actual superclass is known at the declaration of the mixin (using the *inheritance interface* in MixedJava and the declared superclass in gbeta), such that static type-checking of the mixin implementation can be carried out once, independently of applications. In MixedJava an interface is the only option for specifying the assumable properties of the actual superclass, whereas gbeta offers the more flexible option of using anything from a completely abstract class to a fully concrete class as the formal superclass. Of course, none of these systems have a mechanism which is similar to the propagation support offered in gbeta.

6 Future Work

The language is stabilizing, though multi-methods might still be added, so an obvious target for future work is to clarify whether (and how much) the chosen language design adversely affects the performance of compiled code, or of compilation. The current state of affairs just demonstrates that it is not trivial to reach an implementation with good performance. The fact that attributes cannot have fixed offsets (as they can in Beta) implies that simple data member lookup will be slightly more expensive in gbeta.

Another interesting topic is the linearization algorithm. As mentioned in Sect. 3.1, the linearization generalizes from lists of individual mixins to lists of *groups* of mixins, and this provides the nice closure property that *all* class combinations are well-formed. However, is does not blend well with method combination, and some solution to that problem must be found first.

Finally, several partial specifications of the formal semantics of gbeta and its type system have been created, and those efforts should be combined into a full formalization of the language.

7 Conclusion

The language gbeta was presented, with special focus on the support for propagation of class and method combination, according to static dependencies between virtuals and their context. Such propagation enables the specification of, e.g., mutually recursive families of classes as seen from different perspectives. These class family aspects can then be combined with simple top-level expressions, implicitly leading to the combination of the aspects of members of the family and by propagation also the individual methods of those members. The integration of the propagation mechanism into a statically typed language ensures seamless high-level support for the constructs in the type-system and elsewhere. We believe that language integration must be a goal for all similar efforts to support the clean separation of currently globally scattered concerns.

Acknowledgements

The design and implementation of gbeta has taken years, and lots of discussions at the University of Århus have been part of the process. In particular, Mads Torgersen and Kresten Krab Thorup and also Henry Michael Lassen and Søren Brandt gave valuable input. Recently a stay at the University of Washington has given lots of food for thought, especially from Vassily Litvinov and Craig Chambers. Dave Ungar from Sun Microsystems Laboratories has also given valuable advice. The anonymous reviewers gave well-informed and detailed advice, greatly improving the paper. Finally, the whole project would be unthinkable without the BETA tradition and community at large which includes many more people at Århus University and nearby, first of all my advisor Ole Lehrmann Madsen.

References

[1] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, Andrew L. M. Shalit, and P. Tucker Withington. A monotonic superclass linearization for Dylan. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 69–82, New York, October6–10 1996. ACM Press.

[2] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, pages 303–311, October 1990. Published as Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices, volume 25, number 10.

[3] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, April 1992. IEEE Computer Society.

[4] Søren Brandt and Jørgen Lindskov Knudsen. Generalising the BETA type system. In P. Cointe, editor, *Proceedings ECOOP '96*, LNCS 1098, pages 421–448, Linz, Austria, July 1996. Springer-Verlag.

[5] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings ECOOP '98*, LNCS 1445, pages 523–549, Brussels, July 1998. Springer-Verlag.

[6] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings OOPSLA'92*, volume 27, no 10, pages 16–24, Vancouver, USA, October 1992. ACM SIGPLAN Notices.

[7] R. Ducournau, M. Habib, and M.L. Mugnier M. Huchard. *Proposal for a Monotonic Multiple Inheritance Linearization*, volume 29, no 10, pages 164–175. ACM SIGPLAN Notices, Oregon, USA, October 1994.

[8] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 January 1998.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[10] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In Andreas Paepcke, editor, *OOPSLA 1993 Conference Proceedings*, volume 28/10 of *ACM SIGPLAN Notices*, pages 411–428. ACM Press, October 1993.

[11] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In *6th Workshop on Foundations of Object-Oriented Languages (FOOL)*, at `http://www.cs.williams.edu/~kim/FOOL/sched6.html`, January 1999.
[12] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, USA, 1989.
[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer.
[14] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson. *Object-Oriented Environments – The Mjølner Approach*. Prentice Hall, Hertfordshire, GB, 1993.
[15] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Classification of actions, or inheritance also for methods. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP'87*, LNCS 276, pages 98–107, Paris, France, June 15-17 1987. Springer-Verlag.
[16] K. J. Lieberherr, I. Silva-Lepe, and C. Xaio. Adaptive object-oriented programming using graph-based customizations. *Communications of the ACM*, 37(5):94–101, May 1994.
[17] Karl J. Lieberherr and Cun Xiao. Minimizing dependency on class structures with adaptive programs. *Lecture Notes in Computer Science*, 742:424–441, 1993.
[18] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 397–406, October 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10.
[19] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.
[20] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. *ACM SIGPLAN Notices*, 33(10):97–116, October 1998.
[21] The AspectJ project group. Aspectj home page. `http://www.parc.xerox.com/spl/projects/aop/aspectj`, March 1999.
[22] Eric S. Raymond. *The Cathedral and the Bazaar.* at `http://www.tuxedo.org/~esr/writings/cathedral-bazaar/`, 1998.
[23] Patrick Steyaert, Wim Codenie, Theo D'Hondt, Koen De Hondt, Carine Lucas, and Marc Van Limberghen. Nested Mixin-Methods in Agora. In Oscar M. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 197–219. Springer-Verlag, 1993.
[24] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP '97*, LNCS 1241, pages 444–471, Jyväskylä, June 1997. Springer-Verlag.
[25] Mads Torgersen. Virtual types *are* statically safe. In *5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, at `http://pauillac.inria.fr/~remy/fool/program.html`, January 1998.

A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks

Sylvia Dieckmann and Urs Hölzle

Department of Computer Science
University of California
Santa Barbara, CA 93106
{sylvie, urs}@cs.ucsb.edu

Abstract. We present an analysis of the memory usage for six of the Java programs in the SPECjvm98 benchmark suite. Most of the programs are real-world applications with high demands on the memory system. For each program, we measured as much low level data as possible, including age and size distribution, type distribution, and the overhead of object alignment. Among other things, we found that non-pointer data usually represents more than 50% of the allocated space for instance objects, that Java objects tend to live longer than objects in Smalltalk or ML, and that they are fairly small.

1 Introduction

Java has brought garbage collection to the mainstream, being the first truly popular language in the C/C++ tradition that requires garbage collection (GC). Since Java differs in many respects from other languages requiring GC, such as Smalltalk, ML, or Lisp, the GC behavior of Java programs may well differ from that of programs written in other languages.

To understand the GC performance of a system, one must study the allocation behavior of the targeted applications. Every GC implementation leaves room for a waggonload of knobs and levers which impact performance, but tuning is difficult since the right settings depend on the characteristics of the executed program, which again depends greatly on language features and implementation style. To better identify and optimize garbage collectors, implementors need detailed empirical information about the allocation behavior of applications. For that reason, many published studies analyze the allocation behavior in the context of several languages, including Smalltalk, SML/NJ, Lisp, C, and C++ [Ung86, SM94, HMN97, HH+98, Zor89, ZG92, DDZ94]. However, no in-depth analysis of Java programs has been published to date, in part because of the lack of a standardized benchmark suite.

In this paper, we present the first in-depth analysis of the memory usage of realistic Java programs. Our study is based on the programs of the SPECjvm98 benchmark suite recently released by the System Performance Evaluation Corporation [SPEC98]. Most programs are real-world applications with high demands on the memory system.

For each program, we measure as much low-level data as practical. To test the generational hypothesis, we measure age distributions. To allow implementors to judge the impact of segregating objects by type or size, we analyze the heap composition and identify object groups (e.g., reference-free instance objects) which might benefit from a special treatment. To determine the impact of 8-byte object alignment, we simulate its

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 92-115, 1999.
© Springer-Verlag Berlin Heidelberg 1999

effects. Since every system is different and there are so many possible GC variants, we generally refrain from making recommendations based on the data. Instead, our objective is to provide the GC community with detailed data that allows researchers to predict the impact of many GC implementation decisions for Java applications.

The remainder of this paper is organized as follows. Section 2 discusses related work and previous studies of allocation behavior, and section 3 describes the benchmarks. Section 4 presents our experimental setup. In section 5 we discuss the results including our observations on object lifetimes, reference density, heap composition, and others. Section 6 compares our numbers to those reported for other languages where possible, and section 7 summarizes our results.

2 Related Work

When implementing a language with garbage collection, it is essential to understand the expected allocation behavior. Since allocation patterns not only depend on the executed applications but also more generally on language characteristics, researchers have studied this question independently for several programming languages. Most of the earlier papers focus on lifetime and survival rates in order to estimate the overhead for generational GC. Later, after the basic characteristics of GC were understood, researchers became more interested in segregation approaches, special allocation strategies and others.

Ungar, for example, who first implemented Generation Scavenging [Ung84] in Berkeley Smalltalk (BS), analyzed the dependencies between survival rate and nursery size for Smalltalk-80 [Wil92]. Baker suggested theoretical models to explain allocation behavior [Bak94, Bak93]. Hayes analyzed survival rates for long-lived objects in Cedar, a Modula-like language, and found that objects that survive a certain age tend to die in clusters [Hay91].

Zorn presented statistical numbers on eight large Lisp applications analyzed with an object-level runtime system simulator [Zor89]. Unlike in most other studies (including ours), Zorn used memory reference counts as a metric for object lifetimes. More recently, Zorn et al. have studied large C/C++ programs, often in the context of lifetime prediction and memory allocation [BZ93, ZS98, GJS96, ZG92, ZG94].

Stefanovic and Moss analyze the allocation behavior of SML/NJ [SM94]. Gonçalves discusses object age distribution in his study on cache performance [Gon95]. We discuss these studies further in section 6.

Nettles et al. developed Oscar [MHN97], a language-independent GC testbed that can be used to analyze object allocation behavior. Unlike our simulator, Oscar does not trace heap activity but records frequent heap snapshots. Hicks et al. used Oscar to analyze both SML/NJ and Java applications [HMN97, HH+98] but mostly focus on execution time. Except for those studies, Java has not yet been the focus of an in-depth allocation behavior study. To the best of our knowledge, no other analysis of the SPECjvm98 benchmark suite has been published yet.

3 Benchmarks

All of our measurements are based on six programs from the SPECjvm98 benchmark suite [SPEC98], released in August 1998 by the System Performance Evaluation Corporation (SPEC). SPEC is a nonprofit organization of hardware vendors whose objective is to establish a standardized set of vendor-neutral, relevant, application-oriented benchmarks applicable to the newest generation of high-performance computers. Other popular SPEC benchmarks measure various system aspects such as CPU, NFS, and Web server performance.

SPECjvm98 is shipped as a set of Java class files and is intended to measure the efficiency of Java Virtual Machine (JVM) implementations, i.e., the combination of JIT compiler, runtime system, OS, and hardware platform. The individual programs were chosen by the SPEC member companies based on several criteria including high byte-code content, flat execution profile (no tiny loops), repeatability, heap usage and allocation rate, and either I-cache or D-cache misses on the reference platform. Most of the tests represent real applications and use both integer and floating-point computation, library calls, and some I/O; however, AWT (window), networking, and graphics are not covered in this suite. As a result, the SPEC benchmarks all execute little native code in the Java System Classes, and no program contains application-specific native code. All programs except one (mtrt) are single-threaded.

SPECjvm98 consists of eight different programs (see Table 1); seven are used for computing the performance score, one (check) validates the correctness of the VM. This test does not contribute to the result but must be executed correctly in order to obtain a valid score; we omit it from all our numbers. We also exclude mpegaudio since it barely allocates any data.

For all benchmarks, SPEC provides three different inputs referred to as "problem size 100, 10, and 1". Although the input names may suggest so, SPECjvm98 does not scale linearly (i.e., input size 10 does not run in 1/10th the space or 1/10th the time). Only the largest input may be used to publish benchmark results; all our runs use this input unless mentioned otherwise. Some of the programs (compress, jack, javac) iterate multiple times over the same input, which explains the repetitive shape of some graphs (for an example, see Figure 3).

In order to produce valid results, SPEC requires the user to run all applications through a harness which sets up the environment and times the experiment. In this study we too use the harness to start our programs and therefore add a constant but small amount of mostly long-lived data to all our results.

3.1 Program Descriptions

We now describe the six analyzed programs in more detail.

compress implements file compression and uncompression. It performs five iterations over a set of five tar files, each of them between 0.9 Mbytes and 3 Mbytes large. Each file is read in, compressed, the result is written to memory, then read again, uncompressed, and finally the new file size is checked.

In every cycle, compress allocates two large byte arrays for input and output; other than that, it allocates very few heap objects. As a result, the live profile of compress

Program	Description	class file size (Kbytes)[a]	max. live heap (Mbytes)	total allocation (Mbytes)[b]	time (sec)[c]	heap loads * 10^6	% of loads for references	heap stores * 10^6	% of stores for references
check	test JDK and Java features	5.8	n/a			n/a			
compress	Utility to compress/uncompress large files based on Lempel-Ziv method; several passes over same input	17.4	6.7	105	1175	2,423	49	592	0
db	Small data management program; performs DB functions on memory resident database	9.9	7.2	231	505	712	56	42	70
jack	Parser generator with lexical analysis, early version of what is now JavaCC; several passes over same input	129.4	1.2	61	455	627	49	289	46
javac	The JDK 1.0.2 Java compiler compiling 225,000 lines of code	548.3	6.5	111	425	331	42	92	21
jess	Java expert system shell; based on NASA's CLIPS expert system	387.2	1.1	161	380	341	57	30	26
mpegaudio	MPEG-3 audio stream decoder	117.4	insignif.		1100	n/a			
mtrt	Dual-threaded raytracer	56.5	3.0	147	460	372	54	54	6

Table 1. SPECjvm98 programs

[a] Total size of all class files that where delivered as part of the actual application. These numbers do not include the harness code or JVM system classes.

[b] These numbers are based on our simulation and exclude space consumed by alignment, handle space, extra header words etc.; see section 4.3 for a details. The SPECjvm98 documentation reports significantly higher numbers, but we have verified that real JVM implementations (e.g., Sun's JDK 1.2) closely correlate with the numbers given here.

[c] Run time on the SPEC reference machine, a 133MHz IBM PowerPC 604 running an interpreter.

looks somewhat odd—rather than a continuous curve it shows several vertical bars only, because we plot large object allocations as single data points followed by a gap whose length corresponds to the age of the object. Since compress has no long sequence of small allocation requests, no continuous curve can form. (Section 5.1 discusses our treatment of large objects in more detail.)

Although we believe that compress is not a typical representative of an object-oriented application we did not drop it from our suite since we assume that Java applications in the style of compress do exist. However, we sometimes exclude it from summarizing statements made in this paper where it doesn't conform with the general trend.

db is the only program in this study that is not derived from a real-world application. It simulates a simple database management system with a file of persistent records and a list of transactions as inputs. The task is to first build up the database by parsing the records file and then to apply the transactions to this set. Accordingly, db has a very distinct live heap profile: the heap size grows linearly during the building of the database but stays at a fixed level for the entire time of the execution of the transactions.

jack is a commercial application (a parser generator) and is therefore shipped without source code. From our traces, we know that jack performs 16 iterations of building up a live heap structure and collapsing it again. According to the SPEC documentation, it repeatedly generates a parser from the same input. Because no data survives between iterations, jack can run in a fairly small live heap.

javac is the JDK 1.0.2 Java compiler iterating four times over several thousand lines of Java code; the source code of jess serves as input for javac.

jess is an expert system which reads a list of facts about several word games from an input file and attempts to solve the riddles. Although the run is computationally intensive and allocates a lot of memory, jess does not store temporary results for a prolonged time. Thus, the live heap stays at a relatively constant level for the entire run of the application, even though new objects are allocated continuously.

mtrt raytraces a picture by dividing the input data in sections and starting a new working thread for every section. It is the only multithreaded application in our suite. Unfortunately, even the largest problem size currently does not create more than two work threads. In addition, the threads do not yield voluntarily which means that our tracer, which is based on a JDK with cooperative thread model, executes them sequentially [Hol98]. We therefore hesitate to consider mtrt a truly multithreaded program.

4 Experimental Setup

Or experimental setup consists of two independent phases (Figure 1). First, an instrumented version of Sun's JDK 1.1.5 VM produces a trace file while executing the benchmark application. In the second phase, a simulator (written in Java) reads this trace and simulates allocation, pointer assignments, and garbage collections while computing the statistics shown in the rest of the paper. The next two sections describe these two components.

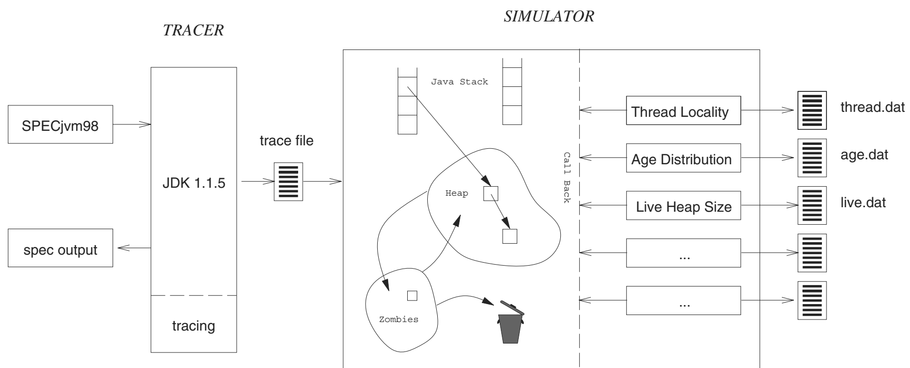


Figure 1. Experimental Setup

4.1 Tracer

We modified the Sun JDK1.1.5 VM to log all runtime information of interest. The recorded events include object creation, updates to the heap, stack, and operand stack,

and method invocation. To reduce the trace size, we do not record updates to non-reference variables. Despite this restriction, and even with a reasonably space-efficient encoding of the trace events and subsequent compression with GNU gzip, trace lengths range up to 1.5 gigabytes. Together, the traces for all benchmarks comprise more than 4.5 gigabytes of data.

On object creation, the tracer records the class, bytes allocated (for arrays), and object ID. Since the JDK1.1.5 can relocate heap objects, we use *handle* addresses rather than real addresses for identification. The JDK VM assigns a handle to every heap object to support heap compaction. This handle is nothing more than a forwarding pointer; all object references use the handle rather than referring to the object directly. If the VM now decides to relocate an object, it updates the forwarding pointer (as opposed to all object references). Consequently, a handle address never changes during the lifetime of its object, which makes it an ideal object identifier.

Together with the recorded heap stores, the object creation data is sufficient to reconstruct heap structures and the complete object allocation history from the trace. The trace also contains method invocation/return events as well as the operands of any bytecode that affects pointers in the stack or operand stack; these trace records allow the simulator to reconstruct pointers from the stack.

We currently ignore pointer stores from C code, since the JDK 1.1.5 VM contains too many places that directly manipulate Java objects without going through the JNI interface. In order to determine the impact of this simplification, and to compensate for it in the simulator, the trace file augments some events with the expected value of the result, even though the simulator should be able to deduce it from the current state. For example, pointer loads record not only the address of the location but also the value being loaded. For the same reason, the trace also records the handles of objects freed by the JDK VM, although we are not interested in (or dependent on) the JVM's GC algorithm. Fortunately, omitting stores from native code has a negligible impact on our results; we will discuss the exact impact in detail in the next section.

4.2 Simulator

The second component of our experimental setup consists of a heap simulator written in Java. The simulator reads and interprets the trace file to reconstruct heap and stack activity, and performs a garbage collection at fixed intervals to determine object lifetimes. For every Java object allocated by the application, the simulator creates a SimObject instance to hold the pointer fields of the application object, to capture a variety of GC-relevant information, and to provide additional space for temporary counters and markers.

While simulating the heap activity of the application, the simulator gathers a variety of statistics. All of our experiments are coded as independent units, each of which creates its own, sometimes partially redundant, output data. Most experiments are event-driven, registering their interest in certain events (e.g., new_object, free_object) so that the simulator calls them back whenever these events occur.

Since most of our experiments distinguish between live and dead objects, the simulator performs exact garbage collections on the simulated heap. Currently, we force a full collection after every 50 Kbytes* of allocation; we believe that a finer resolution is

unlikely to improve the result but would slow down the simulation significantly. (A typical simulation currently takes about four days on a 300MHz UltraSPARC workstation.)

When the simulator determines that an object is unreachable, it notifies the registered statistics objects and moves the SimObject to a "zombie" region. Because the trace does not contain references from the JVM's C code, an object might appear dead in the simulated heap but actually is still alive in the real application. Once the trace reports that the JDK GC has indeed freed the zombie object, the simulator discards it. However, if the object is still live, the simulator will discover that the next time it is referenced. In that case, the simulator resurrects the object and notifies the statistics objects to correct their data. As discussed in the previous section, the trace data contains some redundant information to detect these premature deaths and to recover from them.

	total objects allocated	objects resurrected	% resurrected	corrected references
compress	6,607	142	2.15	3
jess	7,924,698	226	0.00	3
db	3,211,569	146	0.00	3
javac	6,099,430	21,336	0.35	78,203
mtrt	6,587,012	106	0.00	3
jack	6,863,757	16,278	0.24	3

Table 2. Error due to simulation

Table 2 shows that the simulator often resurrects fewer than 250 objects. Only in javac and jack does it retrieve a larger amount of zombies, but even there those objects make up for less than 0.4% of all allocated objects. In addition, a single object might get resurrected multiple times, so that the actual error would be even smaller than shown. Also, any error will show up only for a limited segment of the simulation, that is, from the point where the simulator falsely collects an object until it gets resurrected (detected error) or the JDK1.1.5 frees the original.

We believe these numbers are a strong indication that the inaccuracies introduced by ignoring native code are minor and do not influence our results. Because none of the applications in SPECjvm98 adds its own native code, only native code in the standard Java system classes could potentially cause a discrepancy.

To verify the correctness of the simulator, we compared the simulation results to numbers reported by the tracing VM. For example, we ran the tracer with -verbosegc and compared the result to our simulator data in order to verify total allocation size. We also added extra counters to monitor the overhead of 8-byte alignment, the amount of memory still live at the end of the applications, and many others. All numbers obtained by the tracer agreed closely with those of the simulator. When adding experiments to the simulator, we usually implemented two algorithms and made sure the outcome was identical. For example, we would update the live heap counter dynamically on every

* All graphs in this paper are based on 50 Kbyte intervals. During our analysis we also used 10 Kbyte intervals for better accuracy. However, we found no significant differences between 10 Kbytes and 50 Kbytes.

death event. In addition, we would periodically scan a snapshot of the live heap and determine the size. Finally, we cross-checked all our experiments for consistency. The sum of all ‘live bytes of type X’ numbers in one experiment, for instance, must correspond to “currently live bytes’ measured in another experiment.

To make sure that our tracer did not miss important events, we kept a log of all warnings issued by the simulator due to resurrection, recovery points and other state inconsistencies.

4.3 Object Model

To keep this study as general as possible, we only consider aspects that are likely to occur universally in Java implementations. That is, we abstract away effects that are implementation dependent, such as fragmentation, special object headers, etc. The resulting object model (see Figure 2) is simple yet close to that of a realistic implementation.

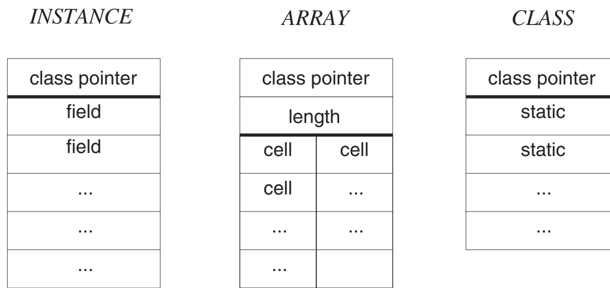


Figure 2. The simulator’s object model.

Regular objects contain a class pointer and an arbitrary number of fields; we assume that there are no additional header words. All fields are 32-bit aligned and the simulator does not attempt to pack smaller fields. For example, a Point object with integer fields x and y consumes 12 bytes of heap space. Similarly, arrays contain a class pointer, followed by a 32-bit length field and enough space to host *cell_size x length* array elements, where the cell size (1, 2, 4, or 8 bytes) depends on the array element type. For example, an integer array of size 1, a char array of size 2, and a byte array of size 4 all consume 12 bytes. Since virtually all processors either require 32-bit aligned addresses or impose a significant penalty on nonaligned addresses, we assume that all objects are aligned to 32 bits, so any fractional words (e.g., in a byte array of size 1) are padded up to the next word boundary. We make no further alignment assumptions (such as 64-bit alignment for long or double) since the space impact of such alignments depends heavily on allocator choices (e.g., unified heaps vs. size-segregated allocation).

Similarly, we do not reserve space for an element type pointer in reference arrays even though this type is required for array store checks. Some VMs (e.g., Sun’s JDK 1.1.5 VM) add this extra word to all reference arrays. But a VM could also store the element type in the array class object, eliminating the per-array overhead. Since reference arrays are usually fairly large, we don’t expect this issue to make a difference one way or the other.

Finally, we assume class objects to have a class pointer and enough space to accommodate the static fields, but we ignore all metadata such as method blocks, field blocks, constant pools, etc. Even though the class metadata might add up to several Kbytes of data, it is up to the JVM implementor to decide where and how it should be stored. For example, the JDK 1.1.5 allocates a 100 bytes struct on the heap for every class object which points to the remaining metadata in the C heap. In any case, the total allocation for all six SPECjmv98 programs is so high that the overhead added by class objects should have no influence on any of our results.

5 Experimental Results

5.1 Heap Size and Object Lifetimes

Figure 3* shows the amount of live data for each application; for better readability the graph on the right shows an enlarged subsection of the same data. As is customary in GC-related work, we measure time (x-axis) in terms of Mbytes allocated. This metric is popular because the number of bytes allocated correlates directly with the amount of work that allocator and GC have to invest.

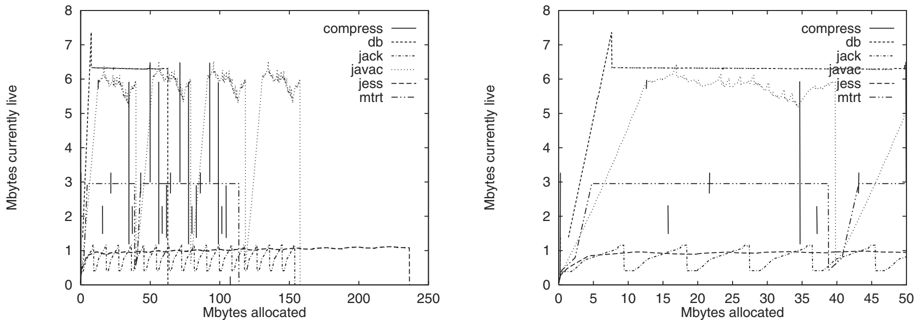


Figure 3. Minimal size of live heap (total and enlarged)

The graphs show the amount of bytes that are currently live over the total amount of space allocated up to that point. For example, after 10 Mbytes of allocation, the live heap of db is slightly larger than 6 Mbytes. The right graph shows an enlarged portion of the left graph.

The peak value of each curve in Figure 3 indicates the minimum heap size required to run the program to completion. The data points of all graphs are spaced 50 Kbytes apart since the simulator performs a garbage collection after every 50 Kbytes of allocation. jack, for example, generates a very distinct live heap curve which oscillates sixteen times between 0.4 and 1 Mbyte, with peaks spaced at 9 Mbytes, because jack allocates approximately 9 Mbytes in every of its sixteen iterations. Most of this data dies soon but a little more than 1 Mbyte stays alive until the end of one cycle, at which point it collapses into a remainder of 0.4 Mbytes.

If an allocated object is larger than 50 Kbytes, the graphs will show a gap whose length corresponds to the size of the object just allocated. This effect is especially apparent in compress which allocates mostly large arrays. Because compress allocates

* Color versions of all graphs are available at http://www.cs.ucsb.edu/oocsb/papers/.

virtually no other data, these arrays cause the curve to take the shape of a series of vertical bars where the height of each bar and the distance from the one to the right represents the size of the allocated object. (Refer to section 3.1 for a description of compress.)

Although all but one of the SPECjvm98 programs allocate more than 100 Mbytes of memory, they can all run in a fairly small heap (less than 8 Mbytes) if runtime performance is not an issue. Also, the maximum live heap size is established at a relatively early point in time, which may be important in the presence of automatic heap expansion; none of the applications would require a heap expansion after the first quarter of execution. jack and javac both repeatedly build up large live structures that collapse instantaneously when the phase terminates. db, jess and mtrt, on the other hand, maintain a constant amount of live data until the very end of the application, either because the same data structure is kept alive or, as in jess, because dying objects are continuously replaced with new allocations.

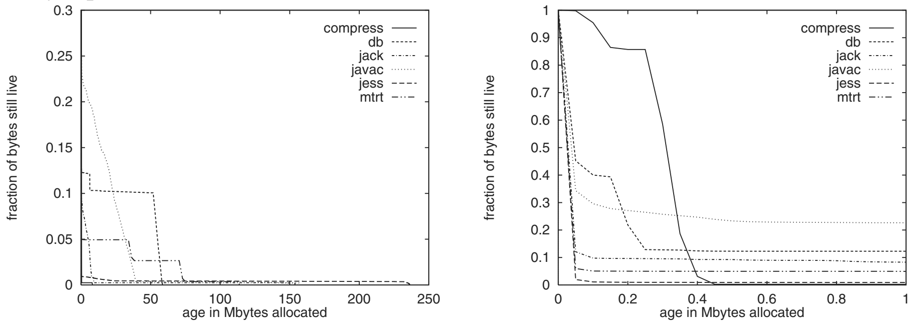


Figure 4. Age distribution in total allocated (total and enlarged)

The graphs show the fraction of bytes that are still live at time X after their creation; time is measured in allocated bytes. All curves must start at 1 since objects are always live immediately after their creation. For example, roughly 27% of all allocated bytes survive for 200 Kbytes in javac. Note that the two graphs are stretched in different dimensions; for better readability we clipped the y-axis of the left graph and the x-axis of the right graph.

Figure 4 shows the object age distributions. As in the previous figure, the x axis is measured in Mbytes of allocation, but for age distribution we treat large allocation requests differently by clipping them at 50 Kbytes. In other words, if a one-megabyte array becomes garbage before the allocation of the next object, its age is 50 Kbytes, not 1 Mbyte. We believe that clipping makes sense because the main reason the GC community is interested in age distribution is to estimate the success of generational GC and to determine the optimal configuration. Without clipping, large objects would otherwise automatically look long-lived even when, in fact, they die shortly after allocation. Although the allocation of a large object uses up heap space, it would not trigger an equal amount of GC work, thus justifying an age definition that presents it as "young".*

In the SPECjvm98 suite, clipping only affects compress, which allocates around 105 Mbytes but whose age distribution appears to end at 8 Mbytes. Essentially, after allocating a large byte array compress performs long allocation-free computations after which the array becomes garbage. Thus, even though the array is live for many seconds of real time, it appears very short-lived to the garbage collector.

The age distributions of the SPECjvm98 applications confirm the weak generational hypothesis [Hay91] that most objects die young, although the effect is not as pronounced as for other programming languages. For example, Stefanovic and Moss report that only 2-8% of all allocated bytes survive for more than 100 Kbytes in four SML/NJ applications [SM94], and Ungar measured that only 6.6% of Smalltalk objects survive 140 Kbytes of allocation [Ung86]. In contrast, 1%-40% of the SPECjvm98 objects are still live after 100 Kbytes (with jess on the lower and db on the upper end, ignoring compress). Even after one megabyte of allocation, 21% of all allocated bytes are still live in javac, 12% in db, and 8% in jack.

This data implies that Java, too, can benefit from generational GC, which allows to collect younger objects more aggressively. However, the nursery of a Java heap should be substantially larger than for functional languages such as ML or Smalltalk.

The age distribution for older objects is very application-dependent; most applications show clusters of objects dying at roughly the same age, leading to sudden steep drops in the age distribution graph. Only jack and javac show relatively smooth age distributions. This observation is consistent with Hayes' study of the behavior of old objects [Hay91], and suggests that multi-generation collectors would not necessarily work for Java. Instead, it may be worth investigating techniques such as Hayes' key object opportunism.

5.2 Instance Objects vs. Arrays

We now turn to analyses of heap composition, starting with the distinction between instance objects and arrays. A garbage collector may want to treat arrays specially, particularly if they are reference-free (e.g., strings). The left graph in Figure 5 shows that all applications initially allocate mostly arrays. Part of these objects can be attributed to JVM initialization; the VM allocates the first 160 Kbytes—mostly byte arrays—before the user application is even started. The SPEC benchmark harness and the actual benchmark setup also appear to perform a higher ratio of array allocations. However, after 10-20 Mbytes of allocation most applications stabilize with an array to instance object ratio of around 1:1. Only mtrt allocates more than twice as many instance objects than arrays over the entire run.

The right graph of Figure 5 shows arrays as a fraction of live objects only. Here, the trends are less clear, and we cannot easily generalize them. However, although the fraction of arrays in the live heap varies significantly from 25% to close to 100%, every application appears to maintain a fairly constant ratio over the entire execution. Most applications do show several extreme spikes, but when correlated with Figure 3 it

---

* We are aware that our decision to clip large objects in this manner is disputable. We looked into alternative approaches such as allocating all large objects in a special area but found each technique arbitrary in some sense and rather unsatisfying for our simulation.
However, most applications do not allocate many objects over 50 Kbytes, and the question of how to treat these cases never arises. Only for compress, treating large objects in one way or the other affects our results. But compress is not a normal application in the sense that it allocates mostly very large arrays, and it is unclear what an age curve for such an application means in the first place. Thus, we believe that no matter how an age curve for compress is computed, one always has to be aware of the unusual allocation behavior of this application.

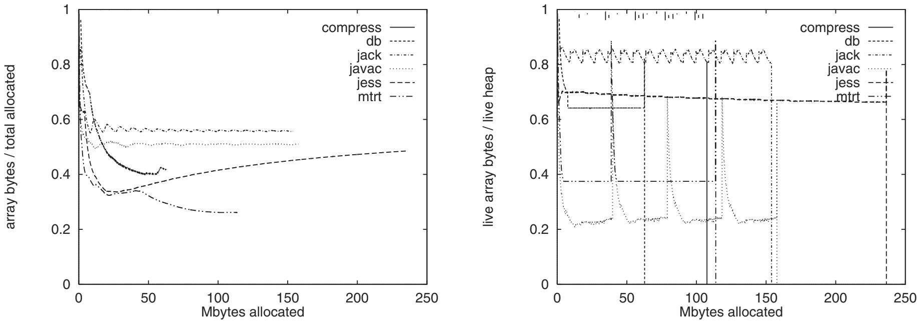


Figure 5. Fraction of array bytes

The graphs show the amount of array bytes (right graph) and live array bytes (left) as a fraction of total bytes allocated (left) and the live heap size (right); time is measured in allocated bytes. For example, after 100 Mbytes of allocation, 40% of jess' heap is used for arrays but close to 70% of all live objects are arrays.

becomes clear that these spikes occur when most instance objects die at the end of an input data set. Apparently, the arrays are live for the entire execution time, thus causing the variations to appear disproportionately large when the live heap contracts sharply.

5.3 Reference Density

We now investigate how much of the allocated space contains references rather than primitive types (e.g., bytes or integers). Again, we distinguish objects by instance objects vs. arrays, since only instance objects can mix reference and non-reference fields. Although our object model includes a class pointer in every object, we do not count it as a reference but rather as a non-reference in this section. The class pointer may well be special-cased in a GC implementation since it is immutable unless class objects are copied. If garbage collection of class objects is switched off altogether (e.g., JDK1.1.5 with -noclassgc), it does not even have to be scanned.

Figure 6 shows that all applications allocate a high percentage of non-reference fields during startup. But even later, most programs allocate less than half of their space for references. Only jess and db allocate a higher fraction of references; the latter ends with a total of 66% allocated for references. Three of the programs, jack, javac, and jess, generally have a balanced live heap with between 45-50% of non-reference bytes. Again, the periodic heap contractions in these programs let these applications appear more irregular than they are. The other three programs also maintain a fairly constant reference density in their live heap with between 65% and 98% of the live space dedicated to non-references.

This data helps estimate the number of pointers a collector must scan and update. However, some non-reference fields are easier to skip than others; in particular, non-reference arrays are faster to skip than non-reference fields scattered throughout instance objects. Thus, we now refine the data presented in Figure 6 by separating out arrays (Figure 7) and instance objects (Figure 8).*

* Unlike in Figure 6, the complements in these graphs do not show *(reference arrays)* and *(reference fields in instances)* but rather *(reference arrays + all instances)* and *(reference fields in instances + all arrays)*, respectively.

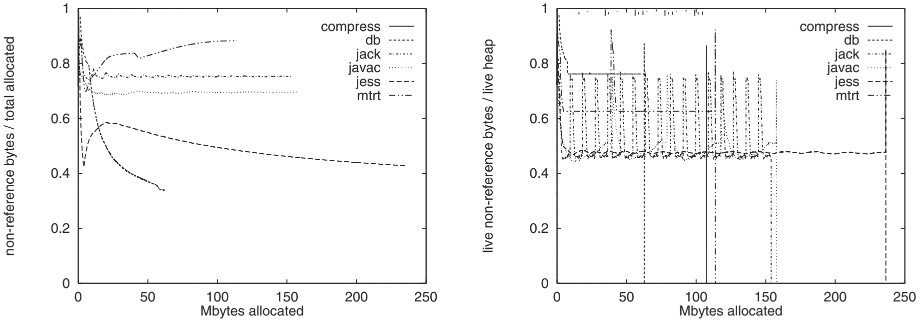


Figure 6. Fraction of non-reference bytes

The graphs show the amount of non-reference bytes (left graph) and live non-reference bytes (right) as a fraction of total bytes allocated (left) and live heap size (right); time is measured in allocated bytes. In jess, for example, after 50 Mbytes of allocation, 55% are allocated for non-reference fields, but 47% of the live objects at this point are non-reference fields.

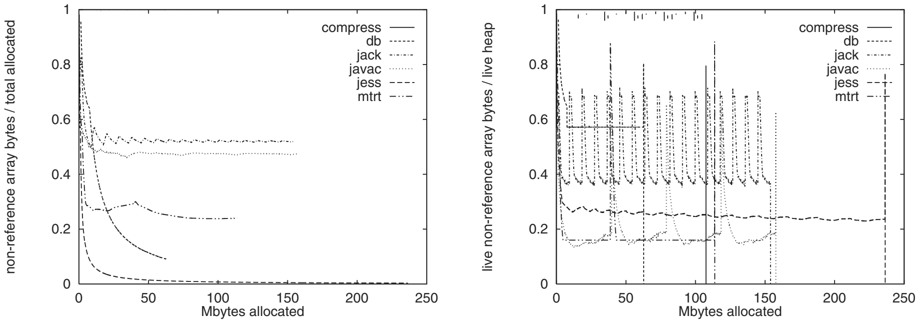


Figure 7. Fraction of non-reference bytes in arrays

The graphs show the amount of non-reference array bytes (left graph) and live non-reference array bytes (right) as a fraction of total bytes allocated resp. live heap; time is measured in allocated bytes. In jess, for example, after 50 Mbytes of allocation, only 1.5% of the space is allocated for non-reference arrays, but over 25% of the live objects at this point are non-reference arrays.

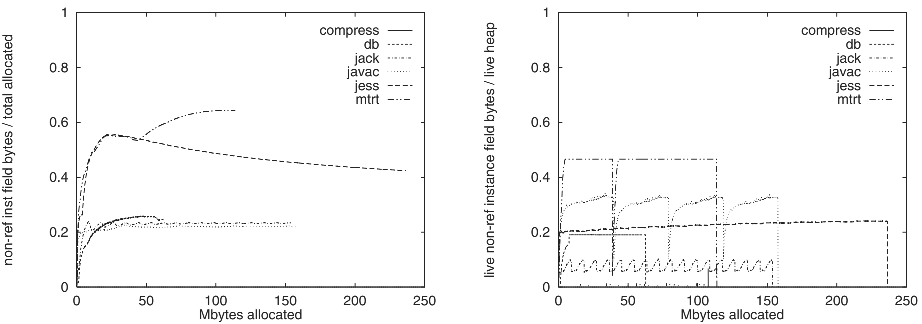


Figure 8. Fraction of non-reference bytes in instances

The graphs show the amount of non-reference instance bytes (left) and live non-reference instance bytes (right) as a fraction of total bytes allocated resp. live heap; time is measured in allocated bytes. In jess, for example, after 50 Mbytes of allocation, over 50% of the space is allocated for non-reference instance field bytes, but only 20% of the live bytes at this point are non-reference instance field bytes.

The behavior for arrays varies widely among programs, with some allocating non-reference arrays almost exclusively (compress), others allocating hardly any non-reference arrays (jess), and the rest in-between. Non-reference fields from instance objects show a somewhat more uniform pattern, with three applications around 25%, jess and mtrt considerably higher, and compress at zero.

Since we were surprised to see such a high fraction of non-reference fields in instances, we investigated the possibility of segregating instances that contain no references other than the class pointer. However, Figure 9 suggests that segregation is unlikely to work well in the general case; only mtrt creates a significant amount of reference-free instance objects.

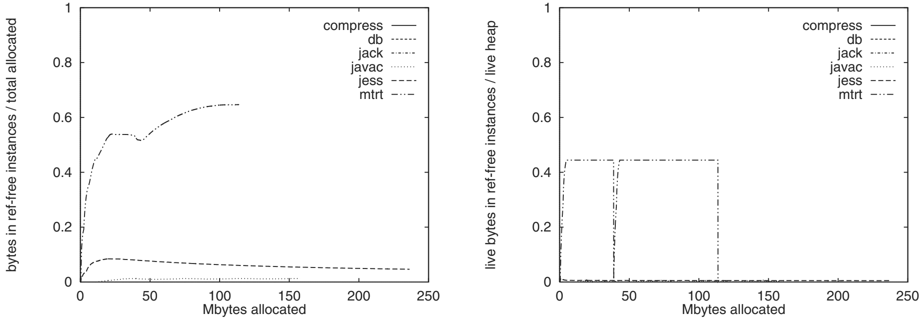


Figure 9. Fraction of bytes from reference-free instances

The graphs show the amount of bytes consumed by reference-free (expect for the class pointer) instance objects (left) and reference-free live instance objects (right) as a fraction of total bytes allocated resp. live heap; time is measured in allocated bytes. In mtrt, for example, after 100 Mbytes of allocation, 65% of the space is allocated for reference-free instance objects; 45% of the live bytes at this point are from reference-free instance objects.

5.4 Heap Composition

Since the previous section revealed that arrays and even non-reference arrays are fairly frequent, we now show what kinds of arrays occur most frequently.* None of the SPECjvm98 programs allocates a significant number of arrays of type boolean, double, float, int, or short. In Figure 10 as well as in all following graphs, these arrays are summarized as "other arrays." All programs allocate less than 2% for those arrays except for mtrt (up to 20%).

Other object kinds (instance objects, byte, char, and reference arrays) are all fairly common throughout the entire suite, although individual programs may not use certain types of arrays. Only instance objects are represented with at least 45% in all but compress. Often, instance objects and one other array type consume 85-99% of all allocated space. However, the dominant kinds depend entirely on the application: db and jess contain a high fraction of reference arrays, jack and javac both allocate around 40% for char arrays, and mtrt allocates instance objects almost exclusively. For some programs, the distribution stabilizes after some time (jack, jess) but not in others (db,

* Note that Java does not have true multidimensional arrays; for example, Java represents a two-dimensional int array as a one-dimensional *reference* array whose elements are int arrays (the rows of the two-dimensional array).

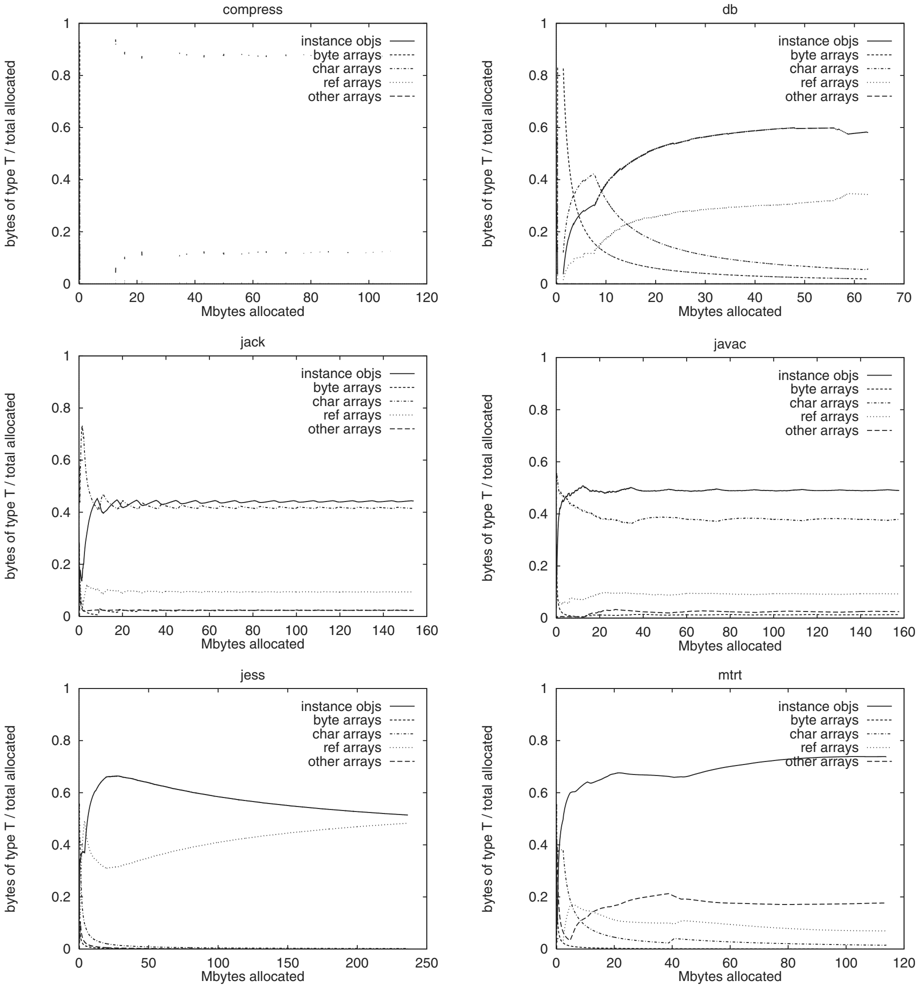


Figure 10. Heap composition for total allocated

For every application the graphs show the fraction of the total allocated space that is dedicated to a certain object type; time is measured in allocated bytes. For example, after 100 Mbytes of allocation, the heap for javac consists of 48% instance objects, 40% char arrays, 9% reference arrays, 1% byte arrays, and 2% other array types.

mtrt). An informal study of other applications confirmed this trend: many applications allocate the bulk of their space for very few kinds, with one or two of them dominating the heap. These results suggest that it may be worthwhile to segregate non-reference arrays.

The picture changes when taking only live objects into consideration (Figure 11). As seen previously, the "live only" graphs contain some noise caused by the heap contractions at the end of each repetition. Ignoring these spikes, the live heaps show a relatively even distribution that often differs markedly from the numbers presented in Figure 10. For example, db allocates 58% of its space for instance objects, 34% for reference arrays, and only 6% for char arrays, but appears to keep most of those char

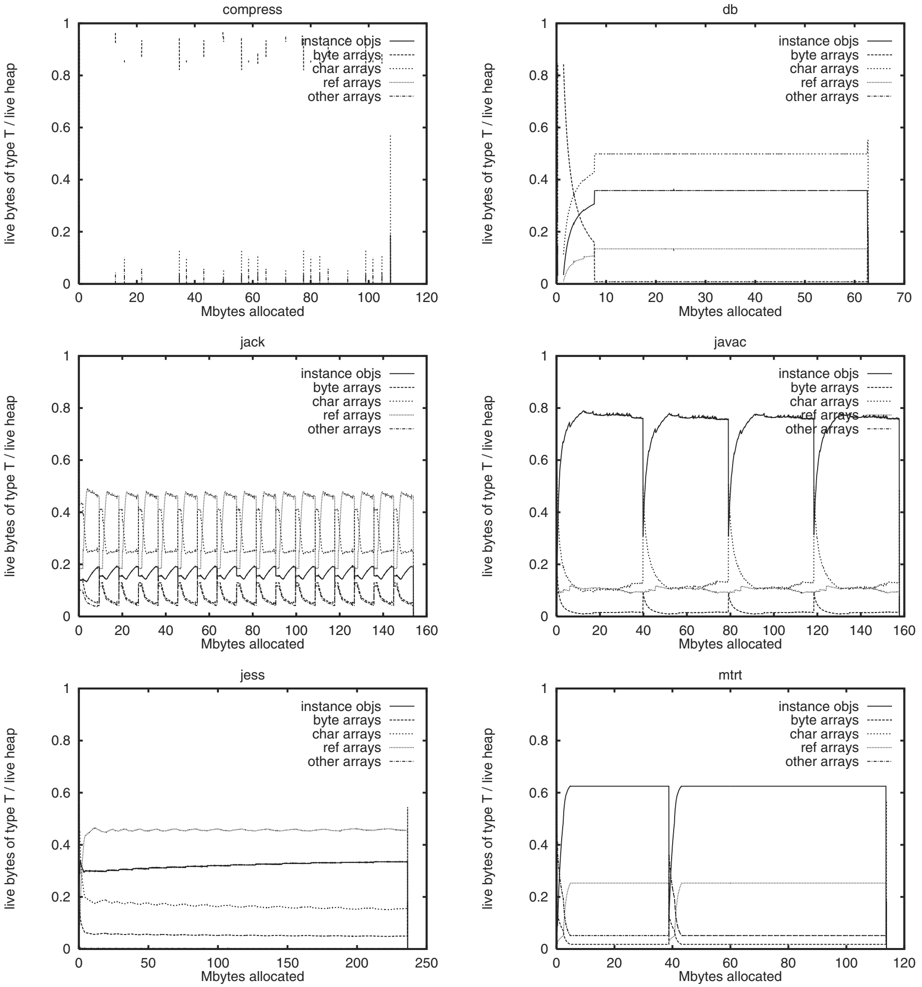


Figure 11. Heap composition for live objects only

For every application the graphs show the fraction of the current live heap that is dedicated to a certain object type; time is measured in allocated bytes. For example, after 100 Mbytes of allocation, the live portion of the heap for jess consists of 46% ref arrays, 32% instance arrays, 16% char arrays, 5% byte arrays, and 1% other array types.

arrays alive. Thus, its live heap contains about 50% char arrays but only 13% of reference arrays. In general, char arrays are more common in the live heap than their percentage of total allocation might suggest.

The differences between the composition of the live heap versus the total allocated suggest that basic types have different age distributions. For example, one would intuitively expect char arrays, which often represent strings, to be rather small and either very short-lived (for temporary results) or permanent. In the next few graphs we investigate age distribution and average object size separated by object kind.

Figure 12 and Figure 13 both show the same numbers (age distribution), but the former organizes the data per application whereas the latter presents it by object kind

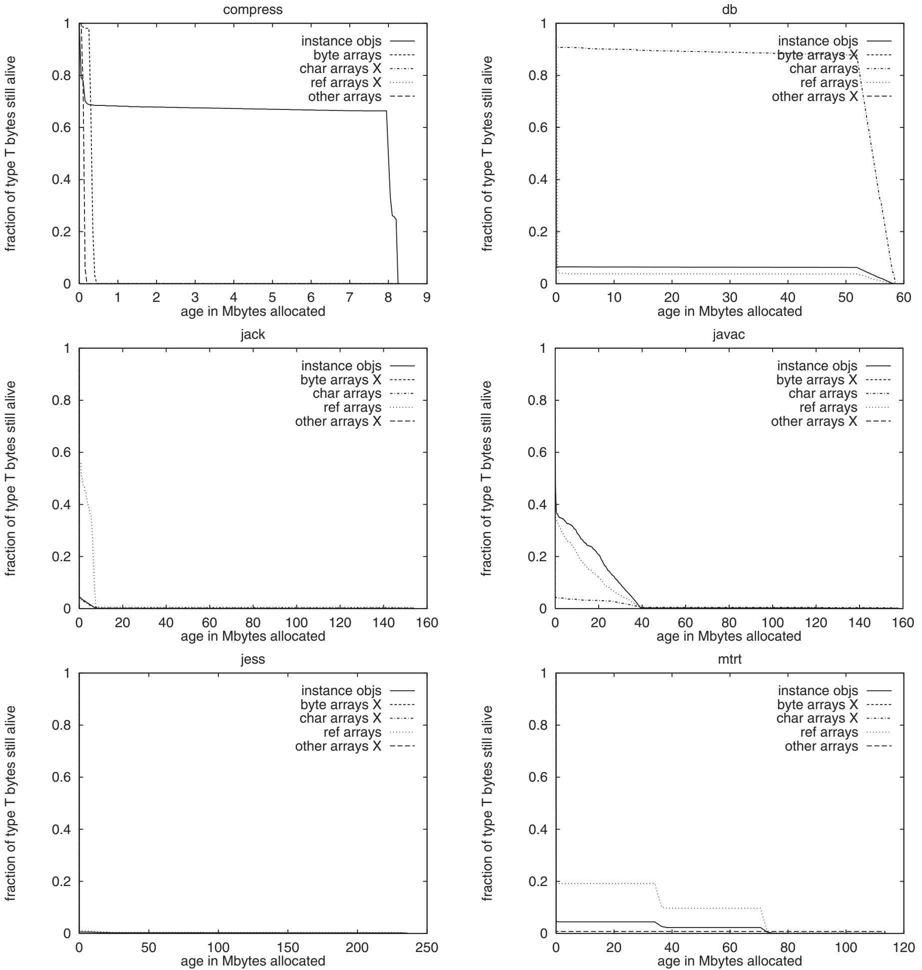


Figure 12.a Age distribution (total)

For every application the graphs show the fraction of bytes of a certain type that are still live at time X after their creation; time is measured in allocated bytes. For example, 22% of all instance object bytes, 13% of all reference array bytes, and 3% of all char array bytes survive the first 20 Mbytes of allocation in jess.

For better readability we prune all object types that represent less than 5% of the total allocation and mark their labels with the letter X.

for the three most frequent kinds. To make the graphs easier to read, we eliminated any line that would represent less than 5% of total allocation; in that case, the line's label is marked with the letter X. However, is still useful to keep in mind that not all object kinds are equally important, and to correlate these graphs with the actual heap composition shown in Figure 10.

As discussed in section 5.1, a large fraction of objects dies very young. In jess, for example, most objects don't even survive the first 50 Kbytes of allocation which at this resolution makes the upper graph in Figure 12 appear blank. Often, most longer-living data is of one type. db keeps mostly char arrays, jack retains reference arrays, and mtrt

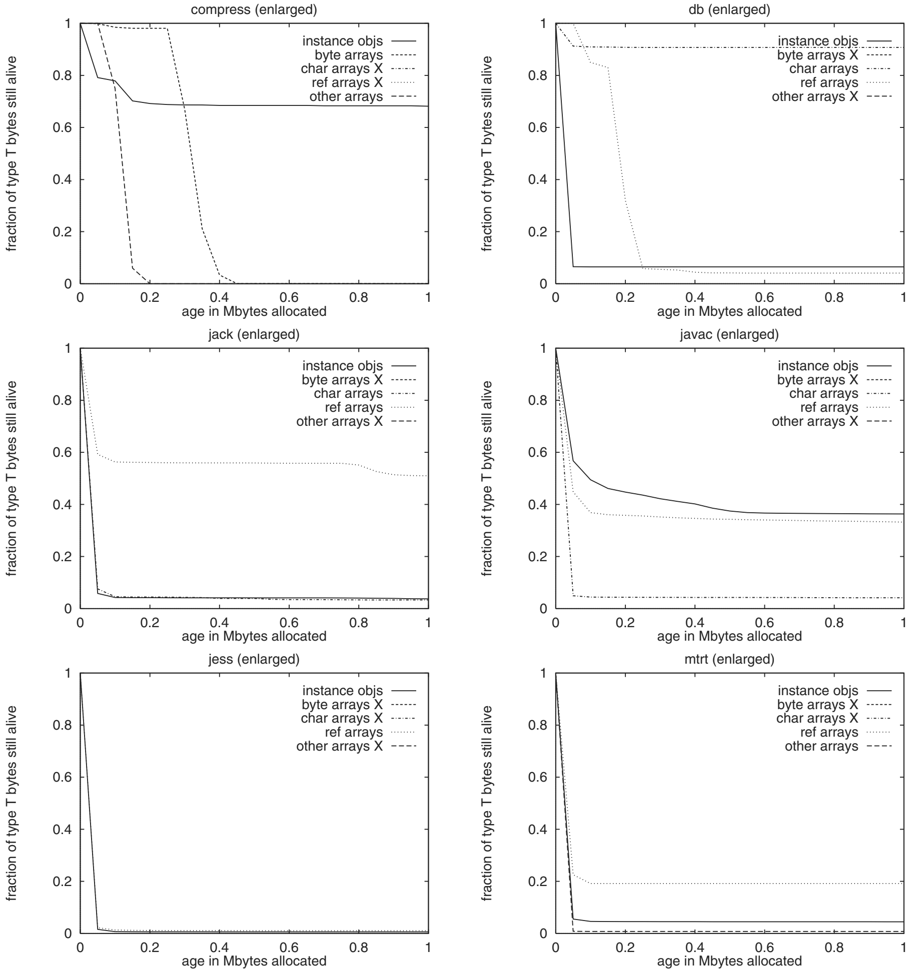


Figure 12.b Age distribution (enlarged)

For every application the graphs show the fraction of bytes of a certain type that are still live at time X after their creation; time is measured in allocated bytes. For example, 22% of all instance object bytes, 13% of all reference array bytes, and 3% of all char array bytes survive the first 20 Mbytes of allocation in jess.

For better readability we prune all object types that represent less than 5% of the total allocation and mark their labels with the letter X.

uses some longer living arrays of various types. Again, the age distribution graphs show a steep drop; only jess shows a smooth age distribution (especially for instance objects and reference arrays). It might be worthwhile to use an adaptive strategy which recognizes those long-lived types and collects them less aggressively.

Even compress seems to keep only instance objects for a prolonged time. However, in this case the numbers might be slightly misleading due to the fact that we clip large object allocations to 50 Kbytes when measuring the age. compress uses several large byte arrays for arithmetic computation but it does not allocate more space during the lifetime of each of those arrays. Consequently, the byte arrays seem to die young.

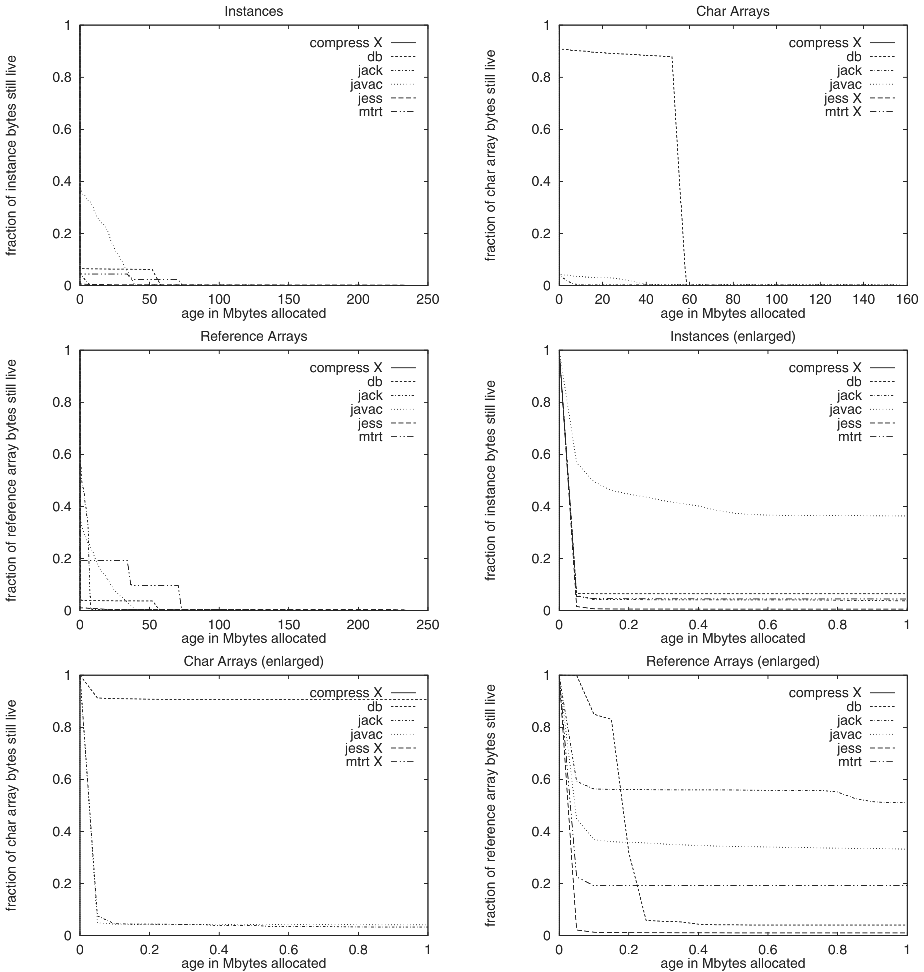


Figure 13. Age distribution for selected types (second row shows enlarged graphs)

For every type the graphs show the fraction of bytes in any application that are still live at time X after their creation. For example, 90% of all char object bytes in db, 3% in javac, and 0.5% in jack survive the first 20 Mbytes of allocation.

For better readability we prune all object types that represent less than 5% of the total allocation and mark their labels with the letter X.

Figure 13 shows that reference arrays tend to survive longer than instance objects and especially char arrays. They also have a smoother age distribution than the other two types. Character arrays, on the other hand, either die very young (as in jack and javac) or to stay alive for most of the execution time (as in db). This observation suggests that segregating all arrays into type-specific heaps and applying different collection strategies to them might pay off. However, our benchmark suite may be too small to draw reliable general conclusions about type-specific age characteristics.

5.5 Object Size

Table 3 lists the average object sizes, computed separately for each of the five frequent object kinds. Again, these normalized numbers can be misleading in cases where they are based on a small sample size. Thus, for each type the table includes the percentage of total allocation per entry, and entries based on less than 5% of total allocation are shaded.

	instance objs			byte arrays			char arrays			ref arrays			other arrays			sum		
	mean	median	% of total	mean	median	% of total	mean	median	% of total	mean	median	% of total	mean	median	% of total	mean	median	% of total
compress	17	16	0	238 Kb	24	87	115	46	0	948	412	0	170 Kb	1.9 Kb	13	11.4 Kb	20	100
db	12	12	58	1.2 Kb	9	2	27	22	6	1.3 Kb	48	34	227	170	0	19	12	100
jack	17	16	45	10	9	2	26	10	41	92	48	10	29	24	2	22	16	100
javac	19	16	48	460	264	1	42	40	40	41	12	1	140	120	2	27	20	100
jess	23	24	51	74	9	0	64	40	0	44	48	48	385	20	0	30	24	100
mtrt	16	16	74	151	9	0	64	72	2	32	32	7	20	20	18	17	16	100

Table 3. Average object size in bytes[a]

[a] A cell is shaded if this type comprises less than 5% of the total allocated heap space

Instance objects clearly are the smallest objects on the heap. Depending on the application, their average size varies between 16 and 23 bytes. But char arrays are also fairly small, with an average size between 26 and 42 bytes. All other array types are less predictable; depending on the application they can range between a few bytes and several Kbytes. Given the small average object size, it appears imperative for implementors to keep the number of header words to a minimum; for example, even a single extra header word per object will increase mtrt's heap by about 20%.

5.6 Object Alignment

Since all program heaps are dominated by relatively small objects, object alignment can add a significant space overhead. All numbers in this paper are based on 32-bit aligned object sizes as explained in section 4.3. However, real JVM implementations may often align objects to other boundaries to simplify memory management or to satisfy hardware alignment requirements. For example, the Sun's JDK 1.1.5 VM aligns all objects to 8-byte boundaries. This extra alignment increases the heap size by up to 19% (see Table 4). The programs that suffer most from alignment, jack and mtrt, both have a high percentage of small objects. The reverse implication (programs with many small objects show high overhead) does not necessarily hold since the size of frequently-allocated objects may be a multiple of eight.

	total allocated (Mbytes)	8-byte aligned (Mbytes)	increase (Mbytes)	% of total allocated
compress	105	105	0.01	0.0%
db	61	73	11.57	18.9%
jack	147	165	18.62	12.7%
javac	161	171	10.25	6.4%
jess	231	240	9.27	4.0%
mtrt	111	115	4.26	3.8%

Table 4. Heap size increase due to 8-byte alignment

6 Comparison With Other Studies

In general, it is difficult to compare empirical GC studies because each study uses different assumptions (e.g., nursery size, large object area), different techniques (simulation, code instrumentation etc.), different metrics, different benchmarks, and different languages. Nevertheless, we attempt to make some comparisons below.

Barrett and Zorn report lifetime quantiles for five allocation intensive C programs [BZ93]. They find that 75% of all objects survive less than 849-32,000 bytes, and that 91%-100% of all objects are "short-lived", i.e, die within 32 Kbytes of allocation. Stefanovic and Moss report a similarly high object mortality for SML/NJ [SM94]; with the smallest nursery size setting of 32 Kbytes, the three ML applications presented have survival rates of approximately 2%, 7%, and 9%. In addition, the survival rate drops dramatically if the nursery size is increased to 250 Kbytes. The four Lisp programs measured in [Sha88] have a slightly lower mortality: between 5% and 25% survive the first 32 Kbytes, and approximately 2% to 8% are still live after 1 Mbyte. In comparison, Java objects appear to be more tenacious, with up to 99% of objects surviving 32 Kbytes of allocation and more than 10% still live after 200 Kbytes.

The eleven benchmarks in the study of C/C++ programs by Detlefs et al. [DDZ94] have average object sizes between 15 and 249 bytes, with six programs over 30 bytes. In comparison, only one of our six benchmarks (compress) exceeds an average object size of 30 bytes. Although Wilson et al [WJ+95] do not measure the average object size, they distinguish popular object sizes when presenting the memory profile for five C programs. These programs, too, have larger object sizes than our Java examples with objects of several kilobytes being fairly common.

A few studies report numbers on type composition, such as Shaw for Lisp [Sha88] and Gonçalves for ML [Gon95]. Stefanovic and Moss analyze nursery survival rate depending on object kind in ML [SM94]. Since all three studies use different categories for types (e.g., Cons, Vect/String, Float, Object, Ratio, Symbol, and Fundef in Lisp), their numbers are difficult to compare to ours. However, it appears that the four Lisp programs analyzed in [Sha88] allocate less space for arrays than typical Java applications (2%-48%). This effect seems even more pronounced in ML; none of the 10 benchmarks studied in [Gon95] uses more than 1% for arrays. In addition, three of them use over 30% of their space for reference-free objects (real).

7 Conclusions and Future Work

We have analyzed the memory allocation behavior of six large Java programs from the SPECjvm98 benchmark suite. All applications allocate significant amounts of memory (between 60 Mbytes and 230 Mbytes) but can run with a minimal heap size of less than 8 Mbytes. Our observations confirm the weak generational hypothesis that young objects are most likely to die, although the result is less pronounced than in other languages—after one megabyte of allocation, up to 21% of all Java objects are still live.

Regarding heap composition, we found that both arrays and instance objects contain a high fraction of non-reference fields. However, only one application (mtrt) allocates a considerable number of instance objects containing no references except for the class pointer. When analyzing the heap for object types, we found that one kind of object (e.g., instance objects, or byte arrays) often dominates allocation, but none dominates all applications.

Eight-byte alignment, a technique used in many real JVM implementations, can increase the allocation rate by up to 19%, with a median of 5%. Extra header words (on top of the class pointer) are even costlier: given average object sizes around 20 bytes, a single extra word would increase the allocation rate by about 20%.

We are currently working on additional experiments for an extended version of this paper. Among others, the new data will illuminate the influence of strings on the heap composition and the average density of incoming references (how many references point to an object?) We will also analyze different alignment and packing strategies and match the results with detailed histograms showing the distribution of object sizes. Furthermore we will investigate the possibility of a thread-local heap by studying thread locality.

We hope that this information too will prove useful to JVM implementors when optimizing Java garbage collection and memory allocation.

Acknowledgments

We thank Steffen Grarup, Mark Reinhold, and the members of the OOCSB group at UCSB for valuable comments on earlier versions of this paper. This work was funded in part by Sun Microsystems, the State of California MICRO program, and by the National Science Foundation under CAREER grant CCR96-24458.

References

[Bak93] H. Baker. 'Infant mortality' and generational garbage collection. In *ACM Sigplan Notices*, 28(4), April 1993, pages 55-57.

[Bak94] H. Baker. The thermodynamics of garbage collection. In *ACM Sigplan Notices*, 29(4), April 1994, pages 58-63.

[BZ93] D. Barrett and B. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation,* ACM SIGPLAN Notices 28(6), Albuquerque, NM, June 1993, ACM Press, pages 187-196.

[CW86] P. Caudill and A. Wirfs-Brock. A third-generation Smalltalk-80 implementation. In *OOPSLA'86 ACM Conference on Object-Oriented Systems, Languages and Applications,* ACM SIGPLAN Notices 21(11), Portland, OR, October 1986, ACM Press, pages 119-130.

[DDZ94] D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. In *Software Practice and Experience*, 24(6), 1994.

[GA95] R. Giladi and N. Ahituv. SPEC as a performance evaluation measure. In *Computer*, 28(8), August 1995. p.33-42.

[Gon95] M. Gonçalves. *Cache Performance of Programs with Intensive Heap Allocation and Generational Garbage Collection.* Ph.D. thesis, Princeton University, May 1995. Technical Report CS-TR-492-95.

[GJS96] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. Addison-Wesley, 1996.

[Hay91] B, Hayes. Using key object opportunism to collect old objects. In *Proceedings of OOPSLA'91 Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices 26(11), Phoenix, Arizona, October 1991. ACM Press, pages 33-46.

[Hay93] B. Hayes. *Key Objects in Garbage Collection.* Ph.D thesis, Stanford University, March 1993.

[HH+98] M. Hicks, L. Hornof, J. Moore, and S. Nettles. A study of Large Object Spaces. In *Proceedings of the First International Symposium on Memory Management,* Vancouver, October 1998, ACM Press, pages 138-145.

[HMN7] M. Hicks, J. Moore, and S. Nettles. The measured cost of copying garbage collection mechanisms. In *Proceedings of International Conference on Functional Programming,* Amsterdam, June 1997.

[Hol98] A. Holub. Programming Java threads in the real world, part 1. In *JavaWorld*, Sept. 1998. http://www.javaworld.com/javaworld/jw-09-1998/jw-09-threads.html

[JL96] R. Jones and R.l Lins. *Garbage Collection.* John Wiley & Sons, 1996.

[LH83] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. In *Communications of the ACM*, 26(6):419-29,1983.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[MHN97] J. Moore, M. Hicks, and S. Nettles. Oscar: A GC testbed. Position paper for *OOPSLA'97 Workshop on Garbage Collection and Memory Management.* Atlanta, GA, October 97.

[Sha88] R. Shaw. *Empirical Analysis of a Lisp System.* Ph.D thesis, Stanford University, 1988. Technical Report CSL-TR-88-351.

[SPEC98] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation, Release 1.0.* August 1998. Online version at http://www.spec.org/osg/jvm98/jvm98/doc/index.html

[SM94] D. Stefanovic and J. E. Moss. Characterization of object behavior in Standard ML of New Jersey. In *Conference Record of the 1994 ACM Symposium on Lisp and Functional Programming*. ACM Press, June 1994.

[Ung84] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM Software Eng. Notes/SIGPLAN Notices Software Engineering Symposium on Practical Software Development Environments*, ACM SIGPLAN Notices 19(5), Pittsburgh, PA, April 1984.

[Ung86] D. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* ACM Distinguished Dissertations. MIT Press, Cambridge, MA, 1987. Ph.D. Dissertation, University of California at Berkeley, February 1986.

[Wil92] P. Wilson. Uniprocessor garbage collection. In *Proceedings of International Workshop on Memory Management (IWMM'92),* Lecture Notes in Computer Science 637, Springer Verlag, 1992.

[WJ+95] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management*, Lecture Notes in Computer Science 986, Kinross, Scotland, September 1995. Springer-Verlag.

[Zor89] B. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. Ph.D. thesis, University of California at Berkeley, March 1989

[ZG92] B. Zorn and D. Grunwald. Empirical measurements of six allocation-intensive C programs. *ACM SIGPLAN Notices*, 27(12), pages 71-80, December 1992.

[ZG94] B. Zorn and D. Grunwald. Evaluating models of memory allocation. In *ACM Transactions on Modeling and Computer Simulation*, 4(1), 1994.

[ZS98] B. Zorn and M. Seidl. Segregating heap objects by reference behavior and lifetime. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998.

Visualizing Reference Patterns for Solving Memory Leaks in Java

Wim De Pauw and Gary Sevitsky

IBM T. J. Watson Research Center, P.O. Box 704
Yorktown Heights, NY 10598 USA
{wim, sevitsky}@watson.ibm.com

Abstract. Many Java programmers believe they do not have to worry about memory management because of automatic garbage collection. In fact, many Java programs run out of memory unexpectedly after performing a number of operations. A memory leak in Java is caused when an object that is no longer needed cannot be reclaimed because another object is still referring to it. Memory leaks can be difficult to solve, since the complexity of most programs prevents us from manually verifying the validity of every reference.
In this paper we show a new methodology for finding the causes of memory leaks. We have identified a basic memory leak scenario which fits many important cases. In this scenario, we allow the programmer to identify a period of time in which temporary objects are expected to be created and released. Using this information we are able to identify objects that persist beyond this period and the references which are holding on to them. Scaling this methodology to real-world systems brings additional challenges. We propose a novel combination of visual syntax and reference pattern extraction to manage this additional complexity. We also describe how these techniques can be applied to a wider class of memory problems, including the exploration of large data structures. These techniques have been implemented and have been proven successful on large projects.

1 Introduction

Complexity in software systems is still growing significantly. Modern languages like Java take away some of the burden of memory management by offering automatic garbage collection [1,2]. This feature can be a double-edged sword, however. Programmers may get the false impression that they don't have to worry about memory at all when using a garbage-collected language. In fact, a very common problem in Java is to inadvertently maintain a reference to a temporary object long after it is needed, preventing it from being reclaimed, and in effect causing a "memory leak". Thus, even in Java, programmers need a way to identify leaking objects, and to tackle the more difficult task of discovering who is holding on to these objects and why. In this paper we propose a new methodology to uncover these memory leaks and, more important, to identify their causes.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 116–134, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Our framework for solving memory leaks begins with the observation that memory leaks often occur according to a simple scenario, which we illustrate below. In this scenario, there is a self-contained operation (for example, the display of a temporary dialog window) in which temporary objects are created. By the end of the operation (in this example when the dialog window is closed) we expect all of these temporary objects to be released. Only some, however, are actually released. Many programs with memory leaks fit this pattern in one way or another. Later in the paper we discuss more complex variations of this scenario.

Figure 1 illustrates this scenario in more detail. A program reaches a stable state; its object population is shown in the lower area in the figure. Then we perform a temporary operation, for example opening a window. The new objects allocated for this operation are shown in the upper area. When the user closes the window, the program will typically set the reference between the old objects (lower) and the new objects (upper) to null. Ideally, the next garbage collection will then clean up all the new objects which were associated with the window.

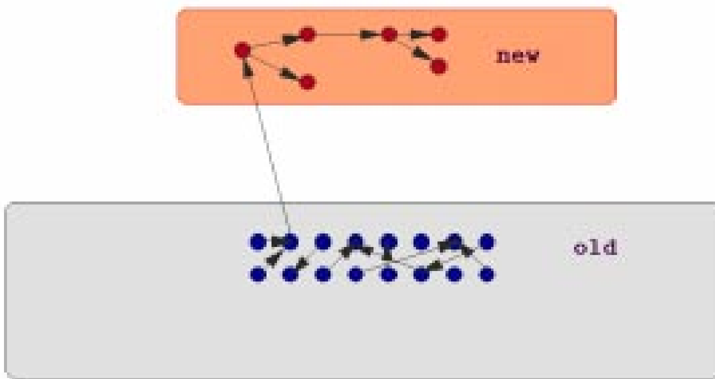


Fig. 1. Ideal scenario: reference from old (lower) to new (upper) objects is removed

What happens very often, however, is that during such a temporary operation, other old objects may get to know about some of the new objects. This situation is shown in Figure 2. A typical case is a registry (in the lower area of the figure) that acquires a reference to a new object (as shown by the solid red arrow on the right). The programmer may not be aware of this hidden reference, and may fail to set this reference to null at the end of the operation. As a result the garbage collector will not reclaim some objects that were meant to be collected after the temporary operation has finished.

In section 2 of the paper we describe our approach for identifying and solving memory leaks which fit this basic scenario. The key idea is to have the

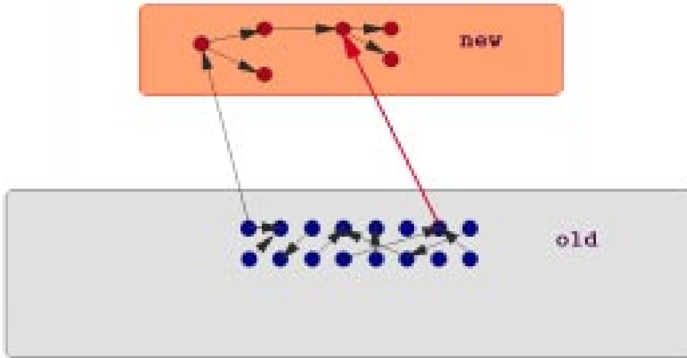


Fig. 2. Memory leak: vestigial reference (right arrow) prevents garbage collection of some of the new objects

programmer mark a period of time corresponding to the expected lifetime of some temporary objects (for example, the lifetime of the window in the above example). Using this information we are able to identify objects that persist beyond this period and the references which are holding on to them.

Scaling this methodology to real-world systems brings additional challenges. Even when we have identified leaking objects and references to them, some further exploration will usually be required by the programmer in order to fully understand the results. For example, the programmer may want to dig deeper and understand why an object was created in the first place, or distinguish true leaks from artifacts like cached objects which are retained intentionally. Searching through individual object references is not practical in anything but the simplest program, given the large number of objects and the richness of their interconnections in most programs. In section 3 we propose a novel combination of a new visual syntax and reference pattern extraction to address these needs.

In section 4 we explain the practical use of these techniques to solve memory leaks that fit the basic scenario we have described above. In section 5 we discuss how these techniques can be applied to a wider variety of memory problems.

All of the work described here has been done within the context of Jinsight [3], a research tool for visualizing and exploring many different aspects of a Java program's run-time behavior. To use Jinsight, the programmer first runs the target program using an instrumented Java virtual machine, which can produce traces of various types of information about the program's execution. The Jinsight visualizer can then be used to explore the program's behavior from various angles.

In section 6 we explain our implementation, the integration of the memory leak finder within Jinsight, and some practical results. In section 7 we discuss related work. We conclude with possible future directions.

2 Basic Scheme for Finding and Solving a Memory Leak

Java provides the programmer with automatic garbage collection. The garbage collector will periodically free all the objects that can no longer be reached from the running program. Problems arise when an object that is more permanent still refers to a temporary object after the temporary object is no longer needed (as shown in Figure 2).

Our approach to memory leaks is based on the observation that many memory leaks occur during well-defined operations which are supposed to release all of their temporary objects upon completion. If we let the programmer tell us the boundaries in time of such an operation, we can use this information to greatly simplify the discovery and diagnosis of memory leaks.

We have two goals: to identify leaking objects, and then to find out who is erroneously referring to these objects, thereby causing the leak. Given the user-supplied boundaries of a “critical” operation, we can differentiate between “old” objects which existed before the operation, and “new” objects which were created during it (see Figure 3 below). To identify the leaking objects, we find the new objects that were created during this critical operation but cannot be reclaimed at the end. To identify the cause of the leak, we find the more permanent objects which are referring to them. These more permanent objects include the old objects which existed before the critical operation, as well as class objects, which are the holding place for static data members in Java. We also note other types of references, such as native code and local variables on the stack which could be referring to the leaking, new objects.

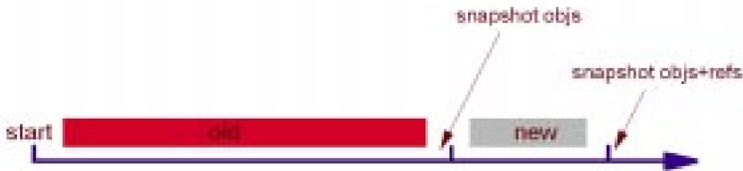


Fig. 3. Timeline of program execution: old objects are created before, and new objects during a critical operation

Using the Jinsight instrumented Java virtual machine, the user takes two snapshots of the program’s object population: one just before, and one after the critical operation. Each snapshot will have an inventory of all the objects in the heap at that point in time. Each of these objects has a unique identifier. When taking a snapshot, we only include objects that are not garbage collectable; we exclude objects that could be reclaimed if the garbage collector were to run at this time. To accomplish this, we traverse the heap using the Java virtual machine’s internal garbage collection routines.

The objects that appear new in the second snapshot when compared to the first snapshot are the ones that could not be collected after the critical operation completed - the leaking objects.

The next step is to find out who is pointing to these objects. In order to do this, we capture some additional information during the second snapshot. In addition to recording the inventory of the objects present in the heap, we also capture all the references between these objects, as well as references to these objects from variables on the stack and native code. We can then find out if an old object, class object, Java local variable, or native code reference is pointing to a new object.

3 Managing Complexity

The approach described so far works well for simple programs. Many programs, however, have data spaces that are large and densely interconnected. Although this basic approach goes a long way toward narrowing the problem, the results are often still too large and complex to explore manually, for a number of reasons.

Very often a program with a memory leak will have thousands of objects unexpectedly remaining active because of a chain-like effect, where an object that is inadvertently retained will in turn hold on to more objects, preventing them from being reclaimed, and so on. The real cause may be buried deep in an immense pile of uncollected objects. In addition, not only the size but also the complexity of the information may prevent the programmer from solving the problem. Ultimately the programmer's goal is to understand why a reference was created and not severed at the right time, and this will often require understanding the context in which these objects and references exist.

We propose a novel methodology, combining a new pattern extraction technique and visual syntax, to allow the user to work with this high level of complexity. These techniques will enable the user to find and solve memory leaks which follow the basic scenario we have described, and also to solve more general types of memory problems, which we describe in section 5.

3.1 Reference Patterns

Fortunately, most programs with large data spaces also have much repetition in their data structures. We take advantage of this fact and extract reference patterns, which are a concise way to represent the interconnections among large numbers of objects. Reference patterns allow us to work with the essential structure of a data space in a simplified, aggregated form. Reference patterns are analogous to execution patterns [4], which make repetitive execution sequences more understandable by eliminating redundancy and bringing out their inherent structure.

The goal of the reference pattern extraction scheme we present here is to produce a concise representation which highlights how groups of objects refer to one another. We begin with a user-defined starting set of objects. We would like

to understand the patterns of reference from other objects directly or indirectly to these objects. The result of the pattern extraction process is a forest of trees, which we extract from the directed graph of object references leading in some way to the starting set. We describe each tree below:

- The root of each tree represents all objects of a single class in the starting set.
- Each node of a tree represents all objects of a single class which refer to at least one of the objects represented by the parent of this node.

In other words, we are grouping objects by a combination of class and what actual objects they refer to. The details of the pattern extraction algorithm are given in the appendix.

The figures below illustrate an example of reference pattern extraction. Figure 4 shows the actual structure of the data, where $a_1, \ldots, a_4$ are instances of class A, $b_1, \ldots, b_4$ are instances of class B, etc. In this example, the user would like to see the pattern of references to objects of class A, and specifies these objects as the starting set.

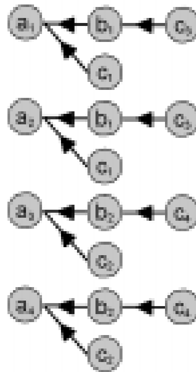


Fig. 4. Actual objects and references

Figure 5 shows how the structure of the data would be summarized. Since the starting set in this case has all objects of the same class, the result of the pattern extraction is a single tree. Note also that c_1 and c_2 are grouped together, but segregated from c_3 and c_4, since these two groups of objects have differing patterns of references even though they are of the same class.

To apply this to the basic memory leak scenario, the user would typically specify the starting set to be all the new objects which cannot be reclaimed, and then extract the pattern of references to these objects, in order to understand

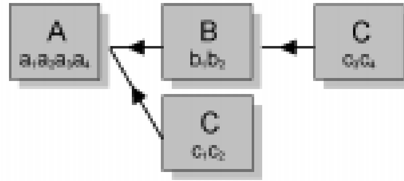


Fig. 5. Result of pattern extraction

who is pointing to them. There are also other choices of starting set that are useful for working with this and other memory scenarios; we discuss these further in sections 4 and 5.

Rather than looking at patterns of references to objects of a given starting set, it is also possible to reverse the sense of the pattern extraction, to bring out the structure of references emanating from a starting set. Each node would then contain all the objects of a given class that are referred to by any object in the parent node. Applications of this type of reference pattern extraction are discussed in section 5.

3.2 Visual Syntax

By extracting patterns of reference relationships among objects, we are shifting our focus from individual objects to groups of objects, allowing the user to work with a greater degree of complexity. Now we present a new visual syntax that makes this information easy to understand and to explore interactively. This presentation of the information, beyond helping the user understand patterns of object references, has additional features specifically for solving memory leaks of the type we have been describing. Figure 6 shows an example illustrating the basic elements of this syntax.

In this example, our starting set is the set of new objects that were created but not collected during the period of the critical operation. We apply the reference pattern extraction algorithm to this set, so that we can understand who is referring to these objects. The gray area on the left contains the starting set of objects, grouped by class. Each icon in the gray area is the root of a reference pattern tree, which expands to the right.

Each icon in the view represents a group of objects of the same class, with the same pattern of references. In other words, each is a node in the reference pattern tree. The color denotes the class of objects in the group. Twin squares represent a group with two or more objects. A diamond represents a class object, which contains the static members of a class. A diamond with a square behind it shows a group that includes the class object for that class. Flying over an icon with the cursor will show more detailed information about that group of objects, in the status line below.

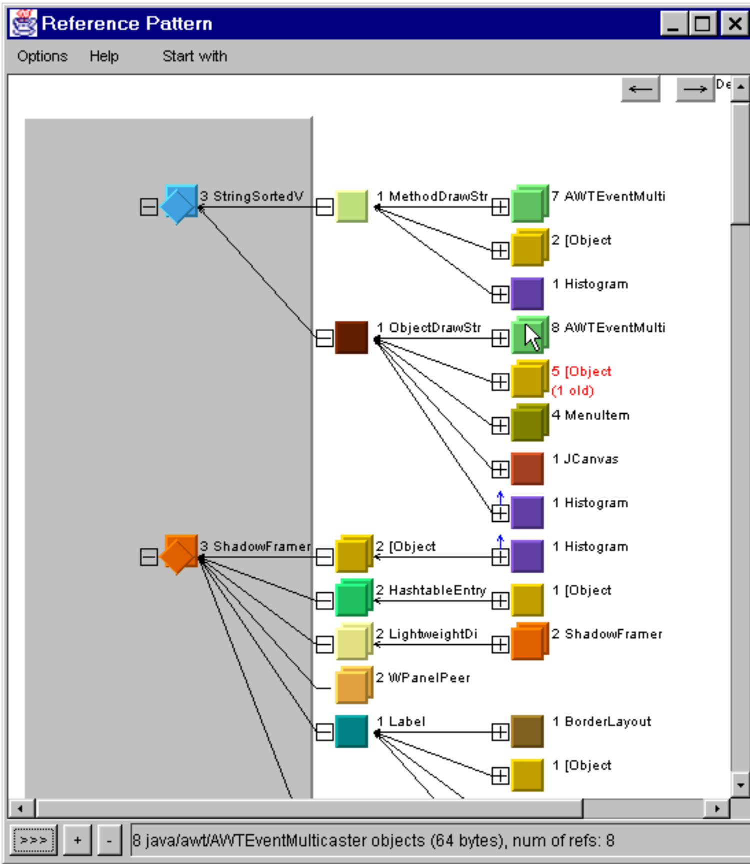


Fig. 6. Visualization of reference patterns

The arrows between the nodes are references. They also indicate the direction of reference between groups of objects, in this example from right to left. The direction can be also reversed to show the pattern of references from (rather than to) the starting set.

By default, each tree is shown expanded to an abbreviated depth, which may be set by the user. Every node which is marked with "+" may be expanded individually by clicking there.

While the aggregation of objects into groups is the key to uncovering patterns, sometimes this same grouping can hide important differences among individual objects. For this reason we allow the user to "ungroup" a node to work with objects individually. The view will then show a separate icon for each of the objects in the group. To the right of each of these icons will be the pattern of references to that object only. Figure 7 shows the result of ungrouping the set of five *[Object* arrays (the notation means array of *Object*) from Figure 6.

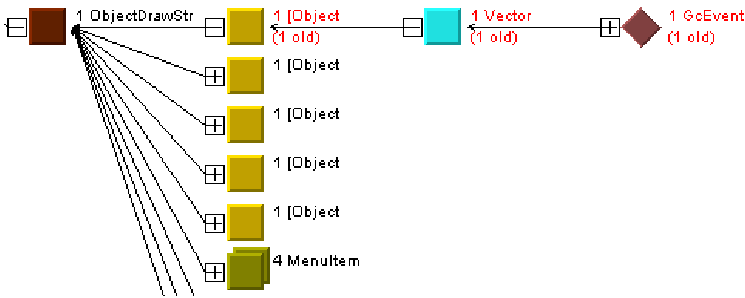


Fig. 7. Result of ungrouping the set of five *[Object* arrays

References between objects form a directed graph. Presenting a graph in a clear and uncluttered form can be a challenge when there are a large number of nodes and interconnections. Therefore, we have chosen to lay out the reference pattern graph as a set of trees. We also believe this format will lend itself naturally to the ways users will explore the data. We expect the user will often begin exploring from a group of objects of interest, and then proceed to progressively expand in a given direction, to see, for example, the objects referring to them. We describe some of these scenarios in more detail in later sections.

Because the view presents the graph as a forest of trees, the same node can appear more than once. An example is shown in Figure 6, where a single *Histogram* object, in purple, appears three times in the view. The way we indicate that an identical node (representing all the same objects) has appeared earlier in the view is with a small blue arrow to the left of the icon, pointing upward. Clicking on the arrow will expand it to point to the first occurrence of this object or set of objects, as shown in Figure 8.

Each node is labeled with the class and the number of objects it represents. The color of the label distinguishes old from new objects. A red label indicates that at least one object in the group is an old object, created before the critical operation began. A node with a black label has only new objects, created during the critical operation. These cues are intended to make it easier to find the offending reference causing a memory leak. In this example, one object labeled in red of type *[Object* is still pointing to an object labeled in black, preventing that object from being collected. Another aid to finding these references quickly is an option which reduces the view to only those nodes leading to old or class objects within a given depth.

4 Solving Memory Leaks in Practice in the Basic Memory Leak Scenario

So far we have described a basic scheme for solving a memory leak by comparing memory before and after a user-identified critical operation, and we

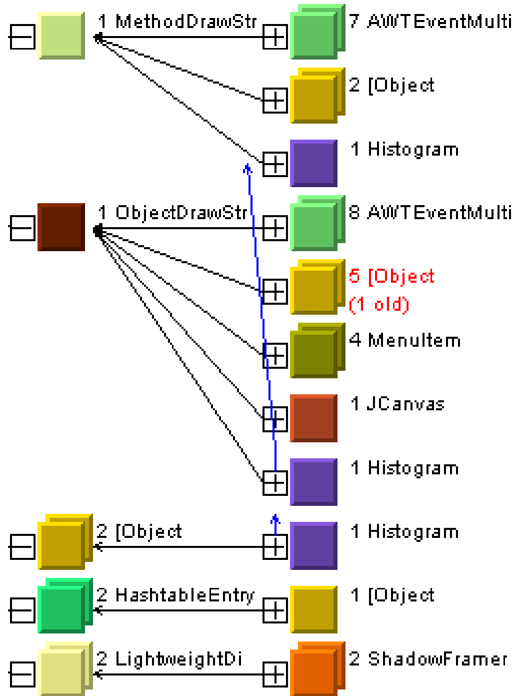


Fig. 8. Clicking on the short blue arrow points to an earlier occurrence of this node

have presented new techniques for exploring the results of this step given their complexity. In this section we describe our methodology, including some additional techniques to find and solve memory leaks in practice. We still restrict ourselves in this section to our basic memory leak scenario, where the user is able to identify a critical operation which allocates memory that is unintentionally retained afterward.

The user runs his or her program under Jinsight's instrumented Java virtual machine. Just before the critical operation, the user issues a command to take a snapshot of objects. After the critical operation has completed, the user issues another command to take a snapshot of objects and references. The user then loads the traces into the Jinsight visualizer, and opens the Reference Pattern View, the view we described in the previous section.

The user then selects a starting set of objects from a number of choices. The starting set will typically be all the new objects created during the critical operation that are still in existence afterward. This starting set will then appear in the gray area on the left, showing the objects that are leaking. The user then looks on the right side of the visualization for objects with a red label (these

are the old objects) or for diamonds (class objects). These are more permanent objects which are referring, directly or indirectly, to one of the new, temporary objects.

The example in Figure 7 shows an old *[Object* (an array) referring to a new *ObjectDrawStr* object, which is no longer needed and should have been reclaimed. After expanding the references we see that *[Object* is referred to by a *Vector*, which is part of a *GcEvent*. The source code for this example revealed that *GcEvent*'s *Vector* (which has an array inside) is a registry of subscribers to that event type. The *ObjectDrawStr* object had at one time been a subscriber to *GcEvent*, but in this case the programmer forgot to have the *ObjectDrawStr* cancel its subscription to that event.

The user can also look for nodes marked with a small colored circle, as shown in Figure 9. These are objects held in memory by a source other than a Java object, such as a local variable on the stack or a native method.

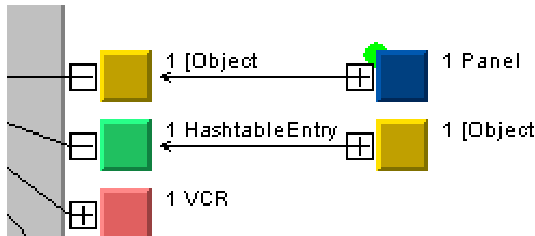


Fig. 9. *Panel* object (with green circle) is pinned by a variable on a Java stack

The tool also allows the user to focus on the new objects of a single class. For example, to find out who is pointing to all the new, leaking *StringSortedV* objects, the user can select just these objects as the starting set, as shown in Figure 10.

Typically there will be a large number of interrelated objects leaking together. In most cases, fixing one spurious reference will have a domino effect; freeing one object will usually allow the garbage collector to free a chain of objects. Therefore we have found it is a good idea to fix one problem at a time, and rerun the program after each fix before performing any further analysis.

With this in mind, it is a good idea to focus first on problems with application-level objects, since these objects will often prevent lower-level library objects from being reclaimed. Jinsight allows the user to start with the set of new objects, where JDK and array objects have been excluded.

So far our approach has been to start with the objects that are leaking and then see who is pointing to them. A complementary approach is to start with the sources that could anchor objects in memory, and see which objects they are holding on to. It may be useful, for example, to look at all objects that are directly or indirectly referred to by static data members. To do this, our tool allows the user to explore reference patterns starting from the class objects. In

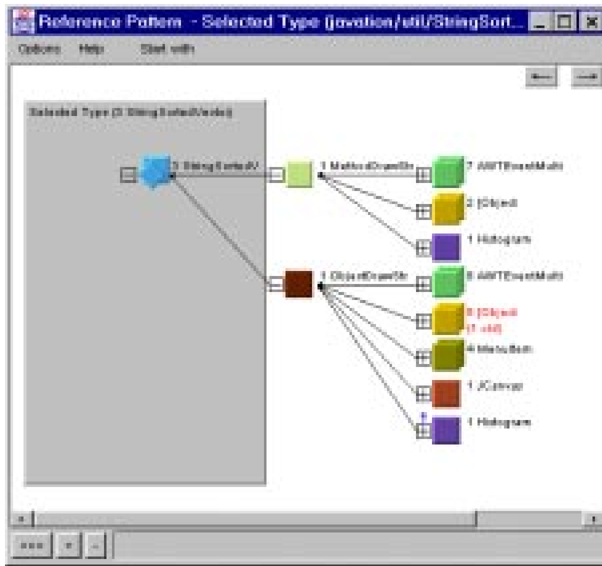


Fig. 10. All the new, leaking *StringSortedV* objects, and the objects holding on to them

this case, the user sets the direction of pattern extraction to look for references from, rather than to, the starting set of class objects. Similarly, the user can explore the pattern of references from local variables and from native code, as shown in Figure 11.

When a spurious reference has been found, the user may want to fix the code by setting this reference to null. If it is not immediately apparent where the appropriate place in the code is, the user may need to explore further to better understand the context of the object. One way to do this is to continue exploring with the reference pattern visualization, expanding references to or from this object to see what other objects it is connected to.

Another way to understand the wider context is to use some of Jinsight's other capabilities. The Jinsight visualization environment can be used to reveal thread behavior, execution patterns, object population, performance bottlenecks, reference patterns and memory leaks. It has a number of views, each allowing the user to explore the run-time behavior of a program from a different angle. The user can navigate from one view to another and correlate data across views. The instrumented Java virtual machine has a number of user-settable options for tracing a program: it can trace method enters and exits, object creation and collection, or take snapshots of object population and object references.

These different features of Jinsight can be used separately or in combination. For example, to find memory leaks, as we have seen so far, we only need to take two population snapshots - no further tracing is required. However, if we would like to understand the execution sequence which is causing the leak, the user can

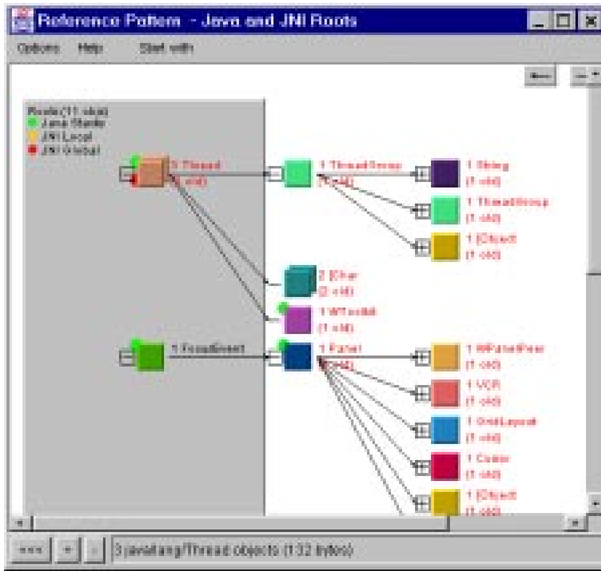


Fig. 11. All objects pinned by Java local variables and JNI roots

rerun the program with full tracing turned on during the appropriate operation, in addition to taking the memory snapshots. This extra information will allow the user to find out the execution history for the objects of interest.

5 Applications Beyond the Basic Memory Leak Scenario

5.1 Finding and Solving Other Types of Memory Leaks

We have premised our work so far on the observation that the expected lifetime of objects will often match the duration of an operation which the user can identify. While we found this scenario occurs often in practice, we also know that there are many situations which do not fit this mold, where memory leaks occur. Even so, the techniques presented so far, with some variation, can still be useful to solve these more complex cases.

One common variation is where a critical operation is identified in which only some of the objects it creates are meant to disappear at the end, while other objects are intended to be more permanent. To solve a memory leak in this scenario we can still apply our methodology to reveal the new objects remaining after the critical operation. However, these new objects will now be a mix of correctly allocated objects and objects that should have been released. Our experience with real cases has been that the reference pattern visualization provides an easy way to sift through these results and determine which objects are legitimately persisting and which are memory leaks. Once the leaking objects are identified, the methodology is the same as in the basic scenario for discovering the offending references to these objects.

Another very common scenario is a repetitive operation which may be causing a memory leak. For example, a server is handling requests from clients, and runs out of memory after a while. In this case, we can examine the incremental effect of one or more of these requests. In order to do that, we can take a snapshot before and after a few cycles of the repetitive operation. We can then see which objects were created during the interval that could not be reclaimed, and who is holding on to them. Since we are dealing with repetitive operations, we expect that fixing the source code for one cycle will apply to all cycles.

Note also that we may have a combination of a repetitive operation and the previous scenario, where some of the objects are meant to persist beyond each cycle. Again, exploring the pattern of references visually helps differentiate the leaking objects from the others.

5.2 Exploring Data Structures

Besides solving memory leaks, there are many other reasons why a programmer may need to understand the data structures of a program: for debugging, performance analysis, or understanding the way an existing program works in order to maintain it. The pattern extraction and visualization techniques we have introduced can be useful in these cases also, enabling the exploration of large, complex data structures in general.

We give an example of how visualization of reference patterns can be used to understand the structure of a hash table. A programmer may want to look at the physical structure of a hash table to determine if a hash function is producing a balanced distribution. Figure 12 shows a *Hashtable* of *JObjects* (values), keyed by *String*. This particular hash table is composed of about 1500 objects.

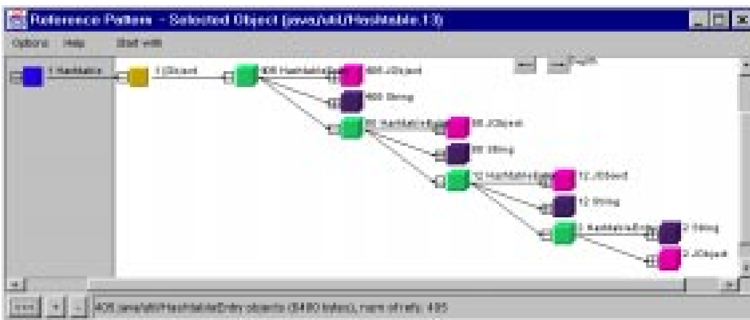


Fig. 12. Reference pattern of a hash table

To begin, the user selects the *Hashtable* object as the starting set. The user then sets the direction of arrows to point from left to right, to see the pattern of references from the *Hashtable*. We see the *Hashtable* object is shown on the left, and it is referring to an array object (*[Object*). This array is the main table of

linked lists in the hash table. Each linked list represents a hash bucket, and is composed of *HashtableEntry* objects, shown in green. Each *HashtableEntry* has a (key, value) pair, in this case (*String*, *JObject*). From the visualization we can see that 405 *HashtableEntry* elements are at the head of a linked list, 80 elements are at the second position of their list, 12 are at the third position and 2 are at the fourth position. We can quickly see that the distribution of this hash table is reasonable, since most of the entries are at the head of their list. The user may want to explore some individual objects further by ungrouping a particular node.

6 Implementation and Experience

The visualization of reference patterns is implemented as part of the Jinsight research tool, available for free from IBM alphaWorks [3]. Jinsight traces the target program using an instrumented version of the Sun JDK Java virtual machine. The user can specify various tracing options to control the type of information recorded. For the analysis of memory problems, we recommend taking snapshots of objects and references, although it is also possible in Jinsight to gather some of this information with a more detailed trace of the execution sequence. The advantage of a snapshot over full tracing is that the amount of trace information collected will usually be significantly smaller. In addition, there will be no perturbation of the running program except while the snapshots are taken, and this usually takes less than a second. This is crucial for analyzing programs that are running for long periods of time.

If needed, the user may turn on full tracing in parts of the program in addition to taking the memory snapshots. This extra information can give the user more insight into the execution history of the objects of interest. The user could find out which objects created these objects, for example, or what sequences of methods were invoked on them.

The Jinsight visualizer is built in 100% pure Java. Figure 13 shows the basic architecture of Jinsight. A transceiver reads events from a trace file or socket and populates Jinsight's dictionaries and models. The Jinsight dictionaries store information about basic object-oriented entities, such as objects, methods, classes, and threads. The models contain more abstract information about the execution of a program, such as the history of each thread's stack over the course of execution, or the references between objects. Modeling the information formally allows us to flexibly combine the information and present it from many angles in the various views. Each view highlights different aspects of the target program behavior, such as the pattern of messages between objects, the program's time and resource bottlenecks, or, as we have described here, the structure of references between objects.

We have used the memory leak tool successfully on many large projects, and it has scaled well to the demands of real-world applications. In an early field test Jinsight was used on a large commercial Java product, where a team of ten developers had been struggling to diagnose memory leaks which would have

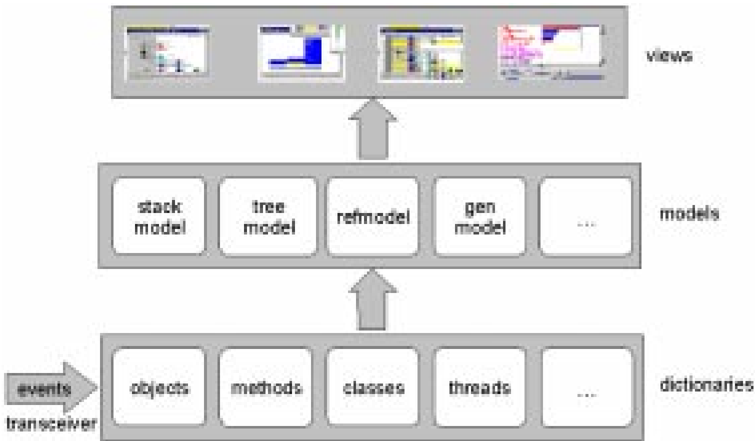


Fig. 13. Architecture of the Jinsight visualizer

delayed the delivery of the product. Using Jinsight, the team was able to increase the rate of memory leak discovery and diagnosis by two orders of magnitude, and was able to ship the product on time.

Table 1 illustrates some statistics from three projects where Jinsight was used to solve memory leaks. The first one shown was a word processor, and the second an industrial-strength web server. The third was an application of Jinsight to itself to diagnose memory leaks. All of these numbers were obtained running the target program and Jinsight on a 200 MHz Pentium Pro PC with 128 MB of RAM.

Program	Total number of objects	Number of leaking objects	Time to read trace	Time to calculate & show leaks	Memory needed by Jinsight	Trace length
Word processor	25 K	3300	15 s	2 s	17 MB	1.2 MB
Web server	20 K	35	15 s	$<$ 1 s	13 MB	0.9 MB
Jinsight	110 K	800	90 s	40 s	40 MB	3.5 MB

Table 1: Three applications where Jinsight was used to diagnose memory leaks

7 Related Work

A significant amount of work has been done on tools that perform static checking of source code. Some of these use annotations [5] to provide ways of expressing assumptions about memory allocation, initialization and sharing, or assumptions about interfaces, variables and types. In [5] the tool calculates the constraints to satisfy these assumptions at compile-time and flags the corresponding places as possible bugs. In [6] B. Blanchet describes an Escape Analysis, to determine if

the lifetime of data exceeds its static scope. This technique can be used for stack allocation. This work takes into account polymorphism. Static analysis tries to cover all possible executions of a program. In order to achieve this, however, it must be very conservative. Therefore it may be very difficult in practice to predict the actual behavior of a program. Dynamic analysis examines exactly this behavior. Although dynamic analysis depends on a test set, and it will not catch all possible problems, it is very often a more efficient aid in solving memory problems.

The Heap Analysis Tool [7] from Sun is an example of a dynamic analysis tool. It allows the programmer to create snapshots of the object population and references, and to browse through the results. However, the amount of information can still be overwhelming for anything other than a simple case, since the browsing interface is very limited. The inability to see the high-level structure of objects and references, and to explore the result set interactively is a hindrance in practice when working with memory problems. Vendors KL Group (JProbe [8]) and Intuitive Systems (Optimizeit [9]) also provide memory heap browsers as part of their performance analysis tools. Although these tools have interactive browsing interfaces, neither provides a way to see overall patterns when there is a large numbers of objects, so finding the cause of a memory leak can still be a slow process.

8 Conclusion

In this paper we have presented a new methodology for solving memory leak problems in Java. The combination of the following three novel features enables us to tackle memory problems in very complex applications:

- We allow the user to identify a critical operation in which temporary objects are expected to be created and released.
- We exploit the fact that there is usually much repetition in the object structures of a complex system, and we extract patterns from the object reference space. Instead of individual objects, the user can now work with groups of objects with similar reference patterns.
- In order to present the results from this synthesized, more abstract space of reference patterns, we introduce a new visual syntax. This visualization points the user immediately to the causes of a memory leak. In addition, it allows the user to navigate and explore the results, in order to understand the dynamic context in which the leak occurred.

Although our initial and motivating objective was to solve a restricted, though still important, class of memory problems, we showed how these techniques were extended and applied to a wider variety of memory problems, such as solving more complex memory leak scenarios, and examining large data structures.

We have tested and integrated the tool into our visualization system, Jinsight. It has been successfully used by many users to solve memory problems in large, real-world systems.

In the future we would like to expand the range of memory problems which we are able to address. An area we are currently pursuing is to allow the user to specify complex query criteria in the visualization of reference patterns. One application of this will be to help the user distinguish temporary from permanent objects in much more complex memory leak scenarios. Another use will be to allow the user much greater flexibility in the exploration of complex data structures in general.

Another area of future work is the ability to perform what-if studies on large data structures. By simulating the Java garbage collection algorithm within our visualization system, we will enable the user to experiment by observing the effect that severing a reference would have on the reclamation of objects.

Our techniques could be applied to other object-oriented languages with garbage collection and run-time type information, such as Smalltalk and Eiffel. The basic functionality that we would need is the ability at run time to scan the heap for all live objects and to report all references between these objects.

Acknowledgments

We would like to thank our colleagues Olivier Gruber, Erik Jensen, Ravi Konuru, John Vlissides and Mark Wegman for their participation in the practical work realizing Jinsight and for many valuable technical discussions. We are also grateful to all the people using Jinsight who have given us feedback and suggestions on our strategy for solving memory leaks.

References

1. Wilson, P.: Uniprocessor garbage collection techniques. Proceedings of the International Workshop on Memory Management, St. Malo, France, September 1992. Lecture Notes in Computer Science, Vol. 637. Springer-Verlag (1992) 1-42
2. Jones, R., Lins, R.: Garbage Collection. John Wiley and Sons (1996)
3. De Pauw, W., Jensen, E., Konuru, R., Gruber, O., Sevitsky, G. and Vlissides, J.: Jinsight, A Visual Tool for Optimizing and Understanding Java Programs. IBM Corporation, Research Division. Information and tool available at http://www.alphaWorks.ibm.com/formula/jinsight (1998)
4. De Pauw, W., Lorenz, D., Vlissides, J., and Wegman, M.: Execution patterns in object-oriented visualization. In Proceedings of the Fourth Conference on Object-oriented Technologies and Systems (COOTS), Santa Fe, New Mexico (1998) 219-234
5. Evans, D.: Static detection of dynamic memory errors. In Programming Language Design and Implementation, May 1996. ACM SIGPLAN Notices, 31(5) (1996) 44-53
6. Blanchet, B.: Escape analysis: correctness, proof, implementation and experimental results. ACM SIGPLAN-SIGACT, Proceedings of the 25th Annual Symposium on Principles of Programming Languages, San Diego, CA, January 1998. (1998) 25-37
7. Foote, W.: Heap Analysis Tool. Sun Microsystems, Inc. Described in http://developer.java.sun.com/developer/earlyAccess/hat (1998)
8. JProbe. KL Group, Inc. Described in http://www.klg.com/jprobe (1999)
9. Optimizeit. Intuitive Systems, Inc. Described in http://www.optimizeit.com (1999)

Appendix. Algorithm for the Extraction of Reference Patterns from an Object Space

We build a forest of trees with nodes representing object sets, using the following algorithm:

1. The user first indicates a starting set A of objects. The choice of this starting set will be motivated by the specific problem at hand. For example, the user might ask to start the exploration of the object reference space from the "new" objects.
2. Partition the starting set A into subsets, A_1, A_2, ..., A_n, of objects grouped by class. Each of these subsets will become the root of a reference pattern tree.
3. To each of these starting subsets A_1, A_2, ..., A_n, apply the recursive operation (step 4):
4. For a subset of objects, S_i (containing objects of the same class), which is a node in a tree:
   (a) Create a new set, R_i, of all the objects referring to any object in S_i.
   (b) Partition this set R_i into subsets of objects, grouped by class: R_{i1}, R_{i2}, ..., R_{im}.
5. These subsets R_{i1}, R_{i2}, ..., R_{im}. will become the child nodes of the node representing S_i.
6. Apply the recursive operation (step 4) to every child node in the tree, until reaching a minimum tree depth, set by the user, or as needed when a user interactively expands a node of the tree in the visualization.

To reverse the direction of references in this algorithm, replace "referring to" in step 4(a) by "referred to by".

Dynamic Query-Based Debugging

Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh*

Abstract. Program errors are hard to find because of the cause-effect gap between the time when an error occurs and the time when the error becomes apparent to the programmer. Although debugging techniques such as conditional and data breakpoints help to find error causes in simple cases, they fail to effectively bridge the cause-effect gap in many situations. Dynamic query-based debuggers offer programmers an effective tool that provides instant error alert by continuously checking inter-object relationships while the debugged program is running. To speed up dynamic query evaluation, our debugger (implemented in portable Java) uses a combination of program instrumentation, load-time code generation, query optimization, and incremental reevaluation. Experiments and a query cost model show that selection queries are efficient in most cases, while more costly join queries are practical when query evaluations are infrequent or query domains are small.

1 Introduction

Many program errors are hard to find because of a cause-effect gap between the time when the error occurs and the time when it becomes apparent to the programmer by terminating the program or by producing incorrect results [Eis97]. The situation is further complicated in modern object-oriented systems which use large class libraries and create complicated pointer-linked data structures. If one of these references is incorrect and violates an abstract relationship between objects, the resulting error may remain undiscovered until much later in the program's execution.

For example, consider the javac Java compiler, a part of Sun's JDK distribution. During a compilation, this compiler builds an abstract syntax tree (AST) of the compiled program. Assume that this AST is corrupted by an operation that assigns the same expression node to the field right of two different parent nodes (Figure 1). The parent

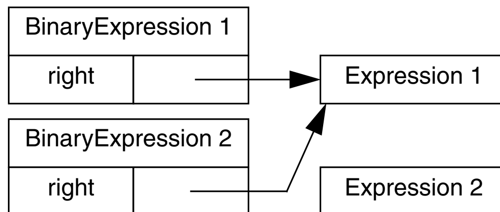


Figure 1. Error in javac AST

nodes may be instances of any subclass of BinaryExpression; for example, the parent may be an AssignAddExpression object or a DivideExpression object, while the child could be an IdentifierExpression. The compiler traverses the AST many times, performing type checks and inlining transformations. During these traversals, the child expression will receive contradictory information from its two parents. These contradictions may eventually become apparent as the compiler indicates errors in correct Java

* Dept. of Computer Science, Univ. of California, Santa Barbara, CA 93106, {raimisl,urs,ambuj}@cs.ucsb.edu
http://www.cs.ucsb.edu/{~raimisl,~urs,~ambuj}

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 135-160, 1999.
© Springer-Verlag Berlin Heidelberg 1999

programs or when it generates incorrect code. But even after discovering the existence of the error, the programmer still has to determine which part of the program originally caused the problem. How can we help programmers to find such errors as soon as they occur?

The programmer could try to use data breakpoints [WLG93], i.e., breakpoints that stop the program when the value of a particular field changes. However, data breakpoints (even if conditional) do not help to debug this error because they are specific to a particular instance. With hundreds or even thousands of BinaryExpression instances, and in the presence of asynchronous events and garbage collection, the effectiveness of data breakpoints is greatly diminished. In addition, it is hard to express the above error as a simple boolean expression. The error occurs only if the expression is shared by another parent node—a relationship difficult to observe from the other parent or from the child itself. In other words, by looking just at the field right of some BinaryExpression object we cannot determine whether this object and its new field value are erroneous.

A programmer could also try to use another conventional tool, conditional breakpoints [Kes90]. Conditional breakpoints check a condition at a particular program location and stop the program if this condition is true. Conditional breakpoints fail to find our bug for the same reason: the condition cannot easily reference objects which are not reachable from the scope containing the breakpoint. Yet we must find exactly such an object—the BinaryExpression containing a duplicate reference to the child Expression object. To accomplish this task, the programmer could write custom testing code for use by conditional breakpoints. For example, the javac compiler could keep a list of all BinaryExpression objects and include methods that iterate over the list and check the correctness of the AST. However, writing such code is tedious, and the testing code may be used only once, so the effort of writing it is not easily recaptured. Finally, even with the test code at hand, the programmer still has to find all assignments to the field right and place a breakpoint there; in javac, there are dozens of such statements. In summary, the tool (conditional breakpoints) provides minimal support and the programmer ends up doing all the work "by hand".

A more effective way to check an inter-object constraint would be to combine conditional breakpoints with a query-based debugger [LHS97]. Similar to an SQL database query tool, a query-based debugger (QBD) finds all object tuples satisfying a given boolean constraint expression. For example, the query

```
BinaryExpression* e1, e2.    e1.right == e2.right && e1 != e2
```

would find the objects involved in the above javac error. The breakpoints would then carry the condition that the above query return a non-empty result. Unfortunately, even well-optimized QBD executions would be inefficient for this task. With hundreds or thousands of BinaryExpression objects, each query becomes quite expensive to evaluate, and since the query is reevaluated every time a conditional breakpoint is reached, the program being debugged may slow down by several orders of magnitude. (We will substantiate this claim in section 4.3.1.)

We propose a new solution, *dynamic* query-based debugging, which can overcome these problems. In addition to implementing the regular QBD query model, a dynamic query-based debugger continually updates the results of queries as the program runs, and can stop the program as soon as the query result changes. To provide this function-

ality, the debugger finds all places where the debugged program changes a field that could affect the result of the query and uses sophisticated algorithms to incrementally reevaluate the query. Therefore, a dynamic query-based debugger finds the javac AST bug as soon as the faulty assignment occurs, and it does so with minimal programmer effort and low program execution overhead.

We have implemented such a dynamic query-based debugger for Java. Our prototype is portable (written in 100% pure Java), and surprisingly efficient. Experiments with large programs from the SPECjvm98 suite [SPEC98] show that selection queries are very efficient for most programs, with a slowdown of less than a factor of two in most experiments. Through measurements, we determined that 95% of all fields in the SPECjvm98 applications are assigned less than 100,000 times per second. Using these numbers and individual evaluation times, our performance model predicts that selection queries will have less than 43% overhead for 95% of all fields in the SPECjvm98 applications. More complicated join queries are less efficient but still practical for small query domains or programs with infrequent queried field updates.

2 Query Model and Examples

Dynamic query-based debugging uses the query model proposed in QBD [LHS97]. The query syntax is as follows:

```
<Query>               ::== <DomainDeclaration> { ; <DomainDeclaration> } .
                           <ConditionalExpression>
<DomainDeclaration>   ::== <ClassName> [*] <DomainVariableName>
                           { <DomainVariableName> }
```

The query has two parts: one or more DomainDeclarations that declare variables of class ClassName, and a ConditionalExpression. The first part is called the *domain part* and the second the *constraint part*. Consider another javac query:

```
FieldExpression fe; FieldDefinition fd.
fe.id == fd.name && fe.type == fd.type && fe.field != fd
```

The first part of the query defines the *search domain* of the query, using universal quantification. The domain part of the above example should be read as “for all FieldExpressions fe and all FieldDefinitions fd...”. FieldExpression is a class name and its domain contains all instances of the class. If a “*” symbol in a domain declaration follows the class name (as in the javac query discussed in the introduction), the domain includes all objects of subclasses of the domain class, otherwise the domain contains only objects of the indicated class itself.

The second part of the query specifies the constraint expression to be evaluated for each tuple of the search domain. Constraints are arbitrary Java conditional expressions as defined in the Java specification §15.24 [GJS96] with certain syntactic restrictions. We disallow variable increments which have no semantic meaning in a query. We currently also disallow array accesses but plan to implement them in the future. Constraints can contain method invocations; we assume that these methods are side effect free.

Semantically, the expression will be evaluated for each tuple in the Cartesian product of the query’s individual domains, and the query result will include all tuples for which the expression evaluates to true (similarly to an SQL select query). Conceptually, the dynamic debugger reevaluates a query after the execution of every bytecode, ensuring that no result changes are unnoticed. The debugger stops the program whenever the

result changes. In reality, the debugger reevaluates the query as infrequently as possible without violating these semantics. In addition, the debugger will reevaluate only the part of the query that changed since the last evaluation. We describe the incremental reevaluation technique in detail in section 3.4.1.

We refer to queries with a single domain variable as *selection queries*; following common database terminology, we call the rest of the queries *join queries* because they involve a join (Cartesian product) of two or more domain variables. Join queries with equality constraints only (e.g., p1.x == p2.x) are *hash joins* because they can be evaluated more efficiently using a hash table [LHS97].

2.1 Examples

We now discuss examples of queries that illustrate the need for dynamic query debuggers.

2.1.1 Javac Compiler

What are examples of inter-object constraint violations that may be difficult to trace back to their origins? We have already discussed one possible error in the javac Java compiler in the introduction. Another error that could occur in javac involves the relationship between FieldExpression and FieldDefinition objects. Consider a situation where a FieldExpression object no longer refers to the FieldDefinition object that it should reference. Due to an error, the program may create two FieldDefinition objects such that the FieldExpression object refers to one of them, while other program objects reference the other FieldDefinition object (Figure 2). In other words, javac maintains a

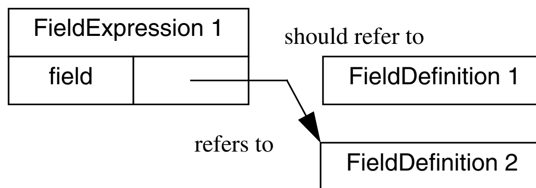


Figure 2. Another error in javac AST

constraint that a FieldExpression object that shares the type and the identifier name with a FieldDefinition object must reference the latter through the field field. We can detect a violation of this constraint using the following query:

```
FieldExpression fe; FieldDefinition fd.
fe.id == fd.name && fe.type == fd.type && fe.field != fd
```

This complicated constraint can be specified and checked with a simple dynamic query, but it would be difficult to verify using conditional breakpoints.

2.1.2 Ideal Gas Tank Example

Another program we examined is an applet simulating a tank with ideal gas molecules. Though this applet is a simple simulation of gas molecules moving in the tank and colliding with the tank walls and each other, it has some interesting inter-object constraints. First, all molecules have to remain within the tank, a constraint that can be specified by a simple selection query:

Molecule* m. m.x < 0 || m.x > X_RANGE || m.y < 0 || m.y > Y_RANGE

Another constraint requires that molecules not occupy the same position as other molecules, and the following query checks this constraint:

Molecule* m1 m2. m1.x == m2.x && m1.y == m2.y && m1 != m2

This constraint is interesting because its violation is a transient failure. Transient failures disappear after some period of time, so even though the program behaves differently than the programmer expected, queries will not be able to detect failures if they are asked too late. The molecule collision error is such a transient failure—it will disappear as the molecules continue to move. However, the applet will behave erroneously: for example, molecules that should have collided with each other will pass through each other. Dynamic queries are necessary to find transient failures, as a delayed query reevaluation may fail to detect the error entirely.

3 Implementation

We have implemented a Java dynamic query-based debugger in pure Java. Java contains a number of features that simplified the implementation. We used the ability to write custom class loaders [LB98] to perform load-time code instrumentation. Java's bytecode class files proved simple to instrument. The debugger creates custom query evaluation code by using load-time code generation. The debugger can be ported to other languages (e.g. Smalltalk) that have an intermediate level format similar to bytecodes.

3.1 General Structure of the System

Figure 3 shows a data-flow diagram of the dynamic query-based debugger. To debug a program, the user runs a standard Java virtual machine with a custom class loader. The custom class loader loads the user program and instruments the bytecodes loaded, by adding debugger invocations for each domain object creation and relevant field assignment. The class loader also generates and compiles custom debugger code. After loading, the Java virtual machine executes the instrumented user program. Whenever the program reaches instrumentation points, it invokes the custom debugger code, which calls other debugger runtime libraries to reevaluate the query and to generate query results. The debugger currently does not handle multithreaded code.

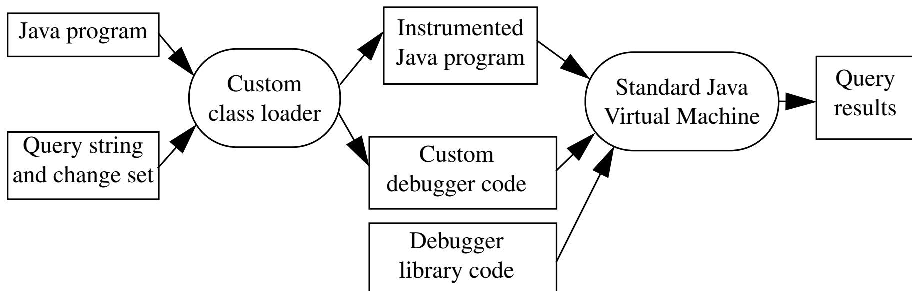


Figure 3. Data-flow diagram of dynamic query-based debugger

The rest of this section discusses the most important parts of the debugger in more detail: how the debugger instruments a Java program, what parts it instruments, and how it evaluates a query.

3.2 Java Program Instrumentation

To enable a dynamic query for a program, the user specifies a query string. The debugger then instruments class files to invoke the debugger after all events that may change the result of the query. The debugger finds assignments to the fields referenced in the query change set (section 3.3) and inserts debugger invocations after each one of them. The system also inserts debugger invocations after each call to a constructor of a domain object.

Figure 4 shows an example of the instrumentation process for a Java method. To instrument class files, the loader transforms them in memory into a malleable format using modified class file handling tools borrowed from the BCA class library [KH98]. Then the loader finds all putfield bytecodes that assign to the fields of interest—like field x in Figure 4—and replaces these putfield bytecodes with invokestatic bytecodes invoking debugger code. The system also inserts such debugger invocations after each call to a constructor of a domain object. When the debugger replaces a putfield bytecode with an invokestatic call, it also inserts the reference to the custom debug method of the DebuggingCode class into the constant pool of the instrumented class. The custom method takes two arguments: the object that the putfield would have updated—a Molecule object in the example—and the newValue value to be assigned to the object field. These objects are already on the stack before execution of the putfield, so they will be correctly passed as arguments to the debug method, and the debugger does no stack manipulation of the instrumented method. Since the original putfield has been replaced by the invokestatic bytecode, the custom debug method performs the assignment originally executed by the putfield. The debugger determines the name of the assigned field and the correct types of objects and values from the class file's constant pool. After instrumentation, the class loader transforms the code back into the class file format and passes the image to the default defineClass method.

The class loader instruments assignments and object constructors that influence the query result. The next section describes how the debugger determines which assignments and constructors to instrument.

3.3 Change Monitoring

The dynamic query debugger updates the query result every time the debugged program performs an operation that may affect the query result. Thus, the program being debugged has to invoke the debugger after every event that could change the query result. The query result may change because some object assigns a new value to one of its fields or because a new object is constructed. However, not all field assignments and object creations affect the query. We call the set of constructors and object field assignments affecting the results of a query the query's *change set*. Though we can use all assignments and all constructors as a conservative change set for any query, we are interested in a minimal change set for efficient query evaluation. Such a change set contains only constructors of domain objects and assignments to domain object fields referenced in a query.

Consider the Molecule query:

```
Molecule* m1 m2.    m1.x == m2.x && m1.y == m2.y && m1 != m2
```

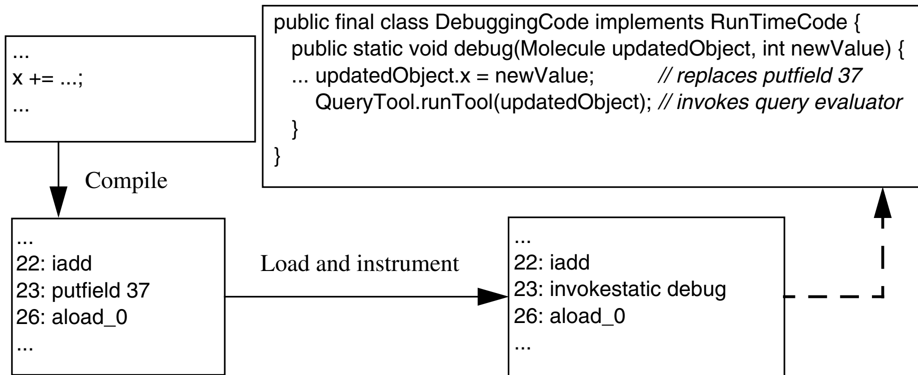


Figure 4. Java program instrumentation

The change set of this query consists of the constructors of the Molecule class and its subclasses as well as assignments to Molecule fields x and y. Assignments to other molecule fields such as color do not belong to the change set.

The change set of a query tells the class loader what assignments and constructors it should instrument. The debugger tracks all domain objects by maintaining domain object collections. Every time a domain object is created, the program invokes the debugger which places the new domain object into its domain collection. The debugger uses the domain collection in query evaluations to iterate through all domain objects. To maintain query correctness and to facilitate garbage collection, the debugger should allow the garbage collector to delete dead objects from domain collections. While such behavior can be implemented using weak pointers, we have not done so yet.

The change set of a query becomes complicated if constraints contain a chain of references. Consider a query for the SPECjvm98 ray tracing program:

```
IntersectPt ip.     ip.Intersection.z < 0
```

The Intersection field is a Point object, and the query result depends on its z value. The query result may change if the z value changes, or if a new value is assigned to the Intersection field. Furthermore, the Point object referenced by the Intersection field may be shared among multiple domain objects. In this case, a change in one Point object can affect multiple domain objects. A chain of references also occurs when a domain instance method invokes methods on objects referenced in its fields, and these methods in turn depend on the fields of the receiver. Tracking which objects accessed through a chain of field references influence which domain objects becomes a complicated task; for example, to do it efficiently, nested objects need to point back to the domain objects that reference them. To simplify the prototype implementation, we support only the explicit chains of references in the query, and we do not handle methods that access chains of references. Our debugger rewrites the query by splitting the chain into single-level accesses and by adding additional domains and constraints. For example, the ray tracing query above is rewritten as:

```
IntersectPt ip; Point* __Intersection.
ip.Intersection == __Intersection && __Intersection.z < 0
```

Chain reference splitting adds overhead by introducing additional joins into the query but it also allows users to ask more complex queries. The overhead can be an order of

magnitude when a selection query is rewritten as a join query. We do not handle native methods, because their debugging is outside the scope of a Java debugger.

To summarize, we use the change set of the query to instrument the Java program. The instrumented program calls the debugger after every event that could change the result of the query, and the debugger reevaluates the query during each call.

3.4 Overview of Query Execution

In this section we describe what happens after an instrumented event occurs in the debugged program. Whenever the program invokes the debugger, it passes the object involved in the event. If the event is a field assignment, the program also passes the new value to be assigned to the field. Figure 5 shows the control flow of the query execution. First, the debugger checks whether the changed object is a domain object. Consider a query that finds Id objects with a negative type code:

Id x. x.type < 0

Here, Id is a subclass of the Expression class, and the type field is defined in Expression. Thus, the program may invoke the debugger when the type field inherited from the Expression class is assigned in an object of another Expression subclass. For example, the program invokes the debugger after assigning the type field in an ArithmeticExpression object. This object shares the type field with the domain class objects, but it does not belong to the query domain, so the debugger immediately returns to the execution of the user program without reevaluating the query.

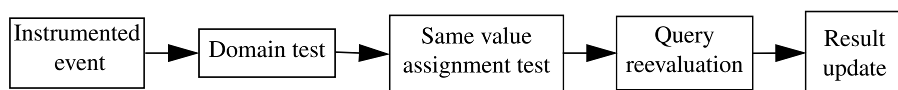


Figure 5. Control flow of query execution

If the object passes the domain test, the debugger checks whether the value being assigned to the object field is equal to the value previously held by the field. For example, some molecules do not move in the ideal gas simulation, yet their coordinates are updated at each simulation step. Such assignments do not change the result of the query and can be ignored by the debugger*. The debugger does not perform this test if the invoking event is an object creation.

After these two tests, the debugger starts reevaluating the query. Our previous work on non-incremental query-based debuggers [LHS97] contained a query evaluation algorithm similar to the evaluation of a relational database join coupled with a selection. The dynamic query-based debugger improves upon the previous algorithm by using incremental reevaluation as discussed below.

3.4.1 Incremental Reevaluation

When the program invokes the debugger, it passes the changed object to the debugger. From the properties of our change sets, we know that this object is the only object that changed since the last query evaluation. Consequently, a full reevaluation of the query for all domain objects is unnecessary. We use incremental reevaluation techniques

* This test is just one example of tests that quickly verify whether the query result changed due to the assignment. We are currently investigating more sophisticated tests that detect more query-invariant assignments.

developed for updates of materialized views in databases [BC79, BLT86] to speed up query execution. Consider a query, a join of three domains A * B * C, e.g.,

```
A a; B b; C c.    a.x == b.y && b.z < c.w
```

The "*" symbol denotes a Cartesian product with some selection constraint; the "+" symbol below denotes set union. If an object of domain B changes, the new result of the query is

$$A * (B + \Delta B) * C = (A * B * C) + (A * \Delta B * C)$$

The first part of the result is the result of the previous query evaluation. The debugger stores this result—usually empty for assertion queries—and does not need to reevaluate it. The second part of the result contains only the changed object (ΔB) of domain B combined with objects of the other domains. The debugger evaluates the changed part in the same way as it would evaluate the whole query. Figure 6 shows an incremental evaluation of changes in the query result. The execution starts with the changed object ΔB passed from the user program. Because this is the only object for which the debugger evaluates the first constraint, the intermediate result is likely to be empty. In general, the size of intermediate results is much smaller in the incremental evaluation, speeding up the query evaluation. If intermediate results are not empty, the debugger continues the evaluation in the usual manner and produces an incremental result (A * ΔB * C). The system then merges the result with the previous result to form the complete query result.

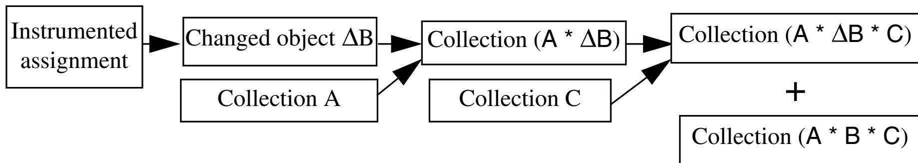


Figure 6. Incremental query evaluation

The query evaluation is further optimized by finding efficient join orders and by using hash joins as described in [LHS97]. Because sizes of domains change during program runtime and we cannot efficiently determine the selectivities of constraints, we use simple heuristics for join ordering: execute selections first, equality joins next, and inequality constraints last.

3.4.2 Custom Code Generation for Selection Queries

Constraints of selection queries are usually very simple and can be evaluated very fast. Instead of performing the general query execution algorithm described in section 3.4.1, which goes through numerous general steps and calls a number of methods, the debugger can evaluate just the few tests necessary to check the selection constraints. Because these tests depend on the query asked, the code for their evaluation has to be generated at program load time. During the loading of the user program, the debugger generates a Java class with a debug method. We show such a method in Figure 7 for the query

```
Molecule1 m.    m.x > 350
```

The first three statements of the method contain the code common for both unoptimized and optimized versions. This code performs the domain test and the same value assignment test described in section 3.4. The optimized code that follows evaluates the selection constraint on the changed object and calls the debugger runtime only if the

```
public final class DebuggingCode implements RunTimeCode {
    public static void debug (Molecule updatedObject, int newValue) {
      // Code common for both general and optimized versions
      if (! (updatedObject instanceof Molecule1))
          { updatedObject.x = newValue; return; }
      if (updatedObject.x == newValue)  return;
      updatedObject.x = newValue;
      // Instead of calling general query evaluation method,
      // evaluate constraint here
      if (updatedObject.x > 350) QueryTool.outputResult(updatedObject);
    }
}
```

Figure 7. Selection evaluation using custom code

query has a non-empty result. The debugger uses the debug method as an entry point that the user program calls when it reaches instrumentation points. With custom code generated, the debug method contains all code needed to evaluate a selection, so the reevaluation costs only one static method call. Furthermore, the debug method—a member of a final class—may even be inlined into the instrumentation points by a JIT compiler. We could also inline the bytecodes into the instrumented method.

4 Experimental Results

Ideally, a test of the efficiency of a dynamic query-based debugger would use real debugging queries asked by programmers using the tool for their daily work. Though we tried to predict what queries programmers will use, each debugging situation is unique and requires different queries. To perform a realistic test of the query-based debugger without writing hundreds of possible queries, we selected a number of queries that in complexity and overhead cover the range of queries asked in debugging situations. The selected queries contain selection queries with low and high cost constraints. The test also includes hash-join and nested-join queries with different domain sizes. The queries check programs that range from small applets to large applications and (for stress-tests) microbenchmarks. These applications invoke the debugger with frequencies ranging from low to very high, where a query has to be evaluated at every iteration of a tight loop. Consequently, the experimental results obtained for the test set should indicate the range of performance to be expected in real debugging situations.

For our tests we used an otherwise idle Sun Ultra 2/2300 machine (with two 300 MHz UltraSPARC II processors) running Solaris 2.6 and Solaris Java 1.2 with JIT compiler (Solaris VM (build Solaris_JDK_1.2_01, native threads, sunwjit)) [Sun99]. Execution times are elapsed times and were measured with millisecond accuracy using the System.currentTimeMillis() method.

4.1 Benchmark Queries

To test the dynamic query-based debugger, we selected a number of structurally different queries (Table 1) for a number of different programs (Table 2):

- Queries 1 and 13 check a small ideal gas tank simulation applet that spends most of the time calculating molecule positions and assigns object fields very infre-

Table 1. Benchmark queries

Query	Slowdown	Invocation frequency (events / s)
1. Molecule1 z. z.x > 350	1.02	15,000
2. Id x. x.type < 0	1.11	16,000
3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	1.25	169,000
4. spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	1.18	1,900,000
5. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	1.27	
6. spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	1.37	
7. spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	5.83	
8. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	1.18	933,000
9. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	1.10	196,000
10. spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	1.83	
11. spec.benchmarks._205_raytrace.Point p. p.x == 1	1.23	787,000
12. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	1.98	2,300,000
13. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	2.13	54,000
14. Lexer l; Token t. l.token == t && t.type == 27 (120,000x600 hash join)	3.43	25,000
15. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	229	350,000
16. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1x1 hash join)	157	1,500,000
17. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count (1x1 join)	77	2,600,000
18. Test5 z. z.x < 0	6.4	42,000,000
19. TestHash5 th; TestHash1 th1. th.i == th1.i (1x20 hash join)	228	40,000,000
20. TestHash5 th; TestHash1 th1. th.i < th1.i (1x20 join)	930	

quently. It has 100 molecules divided among Molecule1, Molecule2 and Molecule3 classes. The application performs 8,000 simulation steps.

- Queries 2 and 14 check the Decaf Java subset compiler, a medium size program developed for a compiler course at UCSB. The Token domain contains up to 120,000 objects.
- Query 3 checks the Jess expert system, program from the SPECjvm98 suite [SPEC98].
- Queries 4–10, and 16–17 check the compress program from the SPECjvm98 suite. Our queries reference frequently updated fields of compress.
- Queries 11–12 and 15 check the ray tracing program from the SPECjvm98 suite. The Point domain contains up to 85,000 objects; the IntersectPt domain has up to 8,000 objects.

- Queries 18–20 check artificial microbenchmarks. These microbenchmarks stress test debugger performance by executing tight loops that continuously update object fields.

Table 2. Application sizes and execution times

Application	Size (Kbytes)	Execution time (s)
1. Compress	17.4	50
2. Jess	387.2	22
3. Ray tracer	55.7	17
4. Decaf	55	15
5. Ideal gas tank	14.3	57

Structurally, queries can be divided into the following classes:

- Queries 1–12 and 18 are simple one-constraint selection queries with a wide range of constraint complexities. For example, query 4 has a very simple low-cost constraint that compares an object field to an integer. The more costly constraint in query 5 invokes a method to retrieve an object field. Another costly alternative constraint (query 6) invokes a comparison method that takes a value as a parameter. Finally, the most costly constraint in query 7 performs expensive mathematical operations before performing a comparison. Queries 8 and 9 have very similar constraints, but differ 4.8 times in debugger invocation frequency. In this paper, by "debugger invocation frequency" we mean the frequency of events in the original program that would trigger a debugger invocation, i.e., the invocation frequency for a debugger with no overhead. Query 12 compares the parameter of the method to the distance of a point to the origin. This query combines costly mathematical operations with increased debugger invocation frequency, because its result depends on all three coordinates of Point objects.
- Queries 13–17 and 19–20 are join queries. Queries 13–16 and 19 can be evaluated using hash joins. The evaluation of queries 17 and 20 has to use nested-loop joins. For join queries, the slowdown depends both on the debugger invocation frequency and sizes of the domains. Queries 13–14 have low invocation frequencies; queries 15–17, 19–20 have high invocation frequencies. Queries 14 and 15 have large domains.

In the next section, we discuss the performance of these queries. Section 4.3 then discusses the efficiency benefits of incremental evaluation, custom selection code, and unnecessary assignment detection.

4.2 Execution Time

Figure 8 shows the program execution slowdown for application programs when queries are enabled. The slowdown is the ratio of the running time with the query active to the running time without any queries. For example, the slowdown of query 3 indicates that the Jess expert system ran 25% slower when the query was enabled.

Overall the results are encouraging. All selection queries except query 7 have overheads of less than a factor of 2. The median slowdown is 1.24. We expect overheads of common practical selection queries to be in the same range as our experimental queries; the performance model discussed in section 5 supports this prediction.

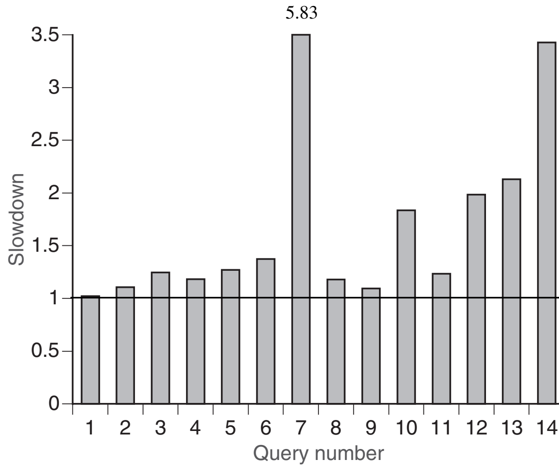


Figure 8. Program slowdown (queries 15—20 not shown)

The slowdown is the ratio of the running time with the query active to the running time without any queries. For example, the slowdown of query 3 indicates that the Jess expert system ran 25% slower when the query was enabled.

Join queries have overheads ranging from 2.13 to 229 for applications. Hash queries (which can be used for equality joins) are efficient for queries 13–14, and other joins are practical for query 13 in which the domains contain only 33 objects each. Queries 15–17 have large overheads because of frequent invocations (e.g., 2.6 million times per second for query 16) and large domains. Join query performance is acceptable if join domains are small, and the program invokes the debugger infrequently. For large domains and frequently invoked queries, the overhead is significant.

Microbenchmark stress-test queries 18–20 show the limits of the dynamic query-based debugger. The benchmark updates a single field in a loop 40 million times per second. When queries depend on this field, the program slowdown is significant. Selection query 18 has a slowdown factor of 6.4, the hash-join evaluation has a slowdown of 228 times, and the slower nested-loop join that checks twenty object combinations in each evaluation has a slowdown of 930 times.

Though the microbenchmark results indicate that in the worst case the debugger can incur a large slowdown, these programs represent a hypothetical case. Such frequent field updates are possible only with a single assignment in a loop. Adding a few additional operations inside the loop drops the field update frequency to 3 million times per second which is more in line with the highest update frequencies in real programs. For such update frequencies, the slowdown is much lower as indicated by query 4. We discuss the likelihood of high update frequencies in section 5.

Figure 9 shows the components of the overhead:

- *Loading time*, the difference between the time it takes to load and instrument classes using a custom class loader, and the time it takes to load a program during normal execution.
- *Garbage collection time*, the difference between the time spent for garbage collection in the queried program and the GC time in the original program.

- *First evaluation time,* the time it takes to evaluate the query for the first time. For join queries, the first query is the most expensive, because it sets up data structures needed for future query reevaluations. We separate this time from the rest of the query evaluation time, because it is a fixed overhead incurred only once.
- *Evaluation time*, the time spent evaluating the query. This component does not include the first evaluation time. The first evaluation time and the evaluation time together compose the *total evaluation time*.

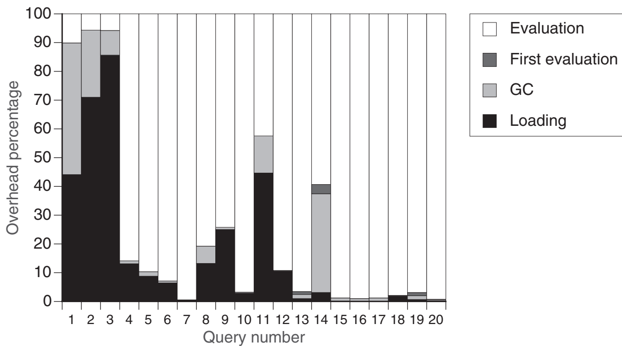


Figure 9. Breakdown of query overhead as a percentage of total overhead

For example, 3% of query 14 overhead is spent on instrumentation, 34% on garbage collection, 3% in the first evaluation, and 60% in subsequent reevaluations.

Figure 9 shows the components of the overhead. For example, 3% of the overhead of query 14 is spent on instrumentation, and 34% on garbage collection. The total evaluation time is 63% of the overhead, with 3% spent in the first evaluation, and 60% spent in subsequent reevaluations. On average, the largest part of the overhead is the evaluation time (75.5%), while loading takes only 17% and garbage collection has a negligible overhead (less than 7%) in most cases*. The loading overhead becomes a significant factor when the loaded class hierarchy is large, as in query 3 on the Jess system. The loading overhead also takes a larger proportion of time when query reevaluations are infrequent or fast as in queries 1, 2, 9, and 11. Garbage collection was not a significant factor except in query 14 which creates 120,000 token objects, and in query 1 which has such a small absolute overhead that even a slight increase in GC and loading time becomes a large part of the overhead. Since the evaluation component dominates the overhead, especially in high-overhead, long-running queries, evaluation optimizations are very important for good performance. We discuss some optimizations already reflected in this graph in the next section.

4.3 Optimizations

To evaluate the benefit of optimizations implemented in the dynamic query-based debugger, we performed a number of experiments by turning off selected optimizations.

4.3.1 Incremental Reevaluation

The dynamic query debugger benefits considerably from the incremental evaluation of queries. We disabled incremental query evaluation and reran all queries. Table 3 shows

* Experiments were run with 128M heap, a factor that decreased the GC overhead.

Table 3. Overhead of non-incremental evaluation

Query	Slowdown versus non-instrumented	Slowdown versus optimized
1. Molecule1 z. z.x > 350	1.19	1.16
2. Id x. x.type < 0	613	554
3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	7135	5,725
4. spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	475	402
5. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	474	373
6. spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	587	428
7. spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	513	88
8. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	275	233
9. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	37	33.8
10. spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	40	21.8
11. spec.benchmarks._205_raytrace.Point p. p.x == 1	10,500	8,496
12. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	17,800	8,972
13. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	21.96	10.3
14. Lexer l; Token t. l.token == t && t.type == 27 (120,000x600 hash join)	1,973	576
15. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	12,400	54
16. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1x1 hash join)	1,708	11
17. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count (1x1 join)	697	9
18. Test5 z. z.x < 0	5,213	821
19. TestHash5 th; TestHash1 th1. th.i == th1.i (1x20 hash join)	1,491	6.6
20. TestHash5 th; TestHash1 th1. th.i < th1.i (1x20 join)	5,602	6.02

the results of this experiment. The first column of numbers in the table shows the ratio of non-incremental query running time to the running time of the original program. The second column shows the ratio of non-incremental query running time to the running time of fully optimized incremental query evaluation. For example, query 2 had a factor of 613 overhead and ran for 2.5 hours. In contrast, the same query ran 554 times faster using the incremental reevaluation, had only 11% overhead and finished in 16.4 seconds. Query 1 was the only query that the non-incremental debugger could evaluate in a reasonable time. The overheads of all other queries were enormous; some programs would have run for more than a day. (For queries 3–12 and 14–17, we stopped query reevaluation after the first 100,000 evaluations and estimated the total overhead.) Despite the large overall overhead, the individual non-incremental query evaluations are reasonably fast. For example, even for large join queries 14 and 15, a single query evaluation only took about 50 ms.

The join queries on compress have an overhead of only 9–11 compared to the incremental optimized version. These joins did not benefit much from incremental evaluation and its optimizations because the domains of these joins contain only a single object.

Overall, the experiments with non-incremental evaluation of queries show that incremental evaluation is imperative, greatly reducing the overhead and making a much larger class of dynamic queries practical for debugging.

4.3.2 Custom Generated Selection Code

To estimate the benefit of generating custom code as discussed in section 3.4.2, we ran all selection queries with the optimization disabled. The results of the experiment are shown in Table 4. The first column of numbers shows the slowdown of the unoptimized version compared to the original program. The second column indicates the slowdown of the unoptimized version compared to the optimized version. For example, query 4 ran 68.5 times slower than the original program and 58 times slower than the optimized query.

Table 4. Benefit of custom selection code (selection queries only)

Query	Slowdown versus non-instrumented	Slowdown versus optimized
1. Molecule1 z. z.x > 350	1.05	1.03
2. Id x. x.type < 0	1.46	1.34
3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	11.70	9.26
4. spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	68.5	58
5. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	64	51
6. spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	65	47
7. spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	69.6	12
8. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	43.6	37
9. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	10.5	9.6
10. spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	11	6
11. spec.benchmarks._205_raytrace.Point p. p.x == 1	21	15
12. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	61	31
13. Test5 z. z.x < 0	1,952	307

The ideal gas tank applet and Decaf compiler queries did not benefit from this optimization, because these programs reevaluate the query infrequently, and the optimization benefit is masked by variations in start-up overhead. All other queries show significant speedups with the optimization enabled. The benefit of the optimization increases with the frequency of debugger invocations; overall, custom generated selection code produces a median speedup of 15.

4.3.3 Same Value Assignment Test

Before evaluating a query after a field assignment, the debugger checks whether the value being assigned to the object field is equal to the value previously held by the field.

Such assignments do not change the result of the query and can be ignored by the debugger.

Table 5 shows that the number of unnecessary assignments differs highly depending on the programs or fields. While some programs and fields do not have them at all, others have from 7% to 95% of such assignments. Only the ideal gas tank simulation, the Jess expert system, and the ray tracing application have unnecessary assignments to the queried fields.

Table 5. Unnecessary assignment test optimization (excluding queries with no unnecessary assignments)

Query	Slowdown versus optimized	% unnecessary assignments
1. Molecule1 z. z.x > 350	0.99	95%
2. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	0.997	7%
3. spec.benchmarks._205_raytrace.Point p. p.x == 1	0.988	15%
4. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	1.16	40%
5. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	1.61	54%
6. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	1.02	15%

To check the efficiency of the same-value test, we disabled it while leaving all other optimizations enabled. The results show that the test does not make much of a difference in query evaluation for most queries. For selections that can be evaluated fast, the cost of the same-value test is similar to the cost of the full selection evaluation. Only when the selection constraint is costly (as in query 4), does the same-value test reduce the overhead. For joins, the cost reduction is significant for the ideal gas tank query that contains 54% unnecessary assignments. For other joins, the percentage of unnecessary assignments is too low to make a difference.

To summarize, the test whether an assignment changes a value of a field costs only one extra comparison per debugger invocation. It does not change the overhead for most programs, but saves time when the number of unnecessary assignments is large or the query expression is expensive.

5 Performance Model

To better predict debugger performance for a wide class of queries, we constructed a query performance model. The slowdown depends on the frequency of debugger invocations and on the individual query reevaluation time. This relationship can be expressed as follows:

$$T = T_{original} (1 + T_{nochange} * F_{nochange} + T_{evaluate} * F_{evaluate})$$

This formula relates the total execution time of the program being debugged T and the execution time of the original program $T_{original}$ using frequencies of field assignments in the program and individual reevaluation times. The model divides field assignments into two classes:

Table 6. Frequencies and individual evaluation times

Query	$F_{evaluate}$ (assignments per second)	$T_{evaluate}$ (µs)
1. Molecule1 z. z.x > 350	N/A	N/A
2. Id x. x.type < 0	16,000	3.73
3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	169,000	3
4. spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	1,900,000	0.140
5. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0		0.208
6. spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)		0.286
7. spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)		3.7
8. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	933,000	0.193
9. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	196,000	0.488
10. spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)		4.26
11. spec.benchmarks._205_raytrace.Point p. p.x == 1	787,000	0.486
12. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	2,300,000	0.461
13. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	N/A	N/A
14. Lexer l; Token t. l.token == t && t.type == 27 (120,000x600 hash join)	25,000	56.8
15. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	350,000	546
16. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1x1 hash join)	1,500,000	60
17. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count (1x1 join)	2,600,000	51
18. Test5 z. z.x < 0	42,000,000	0.131
19. TestHash5 th; TestHash1 th1. th.i == th1.i (1x20 hash join)	40,000,000	5.7
20. TestHash5 th; TestHash1 th1. th.i < th1.i (1x20 join)		23

- Assignments that do not change the value of a field. These assignments do not change the result of the query. The debugger has to perform only two comparisons in this case—a domain test and the value equality test, so it spends a fixed amount of time ($T_{nochange}$) in such invocations independent of the query. We calculated $T_{nochange}$ by running a query on a program that repeatedly assigned the same value to the queried field; for the machine/JVM combination we used, $T_{nochange}$ = 66 ns.
- Assignments that lead to the reevaluation of a query. The time to reevaluate a query $T_{evaluate}$ for such an assignment depends on the query structure and on the cost of the query constraint expression. For each query, we calculate $T_{evaluate}$ by dividing the additional time it takes to run a program with a query into the number of debugger invocations. This calculation gives an exact result for programs that have no unnecessary assignments ($F_{nochange} = 0$). For example, for query 18 $T_{evaluate}$ is 131ns. $T_{evaluate}$ for query 4 is 140 ns, which is close to the time to evaluate a similar query in a microbenchmark. When constraints are more costly,

$T_{evaluate}$ increases; for example, for the highest cost selection query (query 10) it is 4.26 µs. It is even higher for join queries where it depends on the size of domains in joins; for example, for query 16 it is 60 µs, and for query 15 which has large domains, it is 546 µs.

Using the values of reevaluation times and the frequency of assignments to the fields of the change set, we can estimate the debugging overhead. First, we determine the typical field assignment frequency.

5.1 Debugger Invocation Frequency

Debugger invocation frequency is an important factor in the slowdown of programs during debugging. The program invokes the debugger after object creation and after field assignments. For most queries, the field assignment component dominates the debugger invocation frequency. To find the range of field assignment frequencies in programs, we examined the microbenchmarks and the SPECjvm98 application suite. We instrumented the applications to record every assignment to a field. Table 7 shows results of these measurements.

Table 7. Maximum field assignment frequencies

Application	Maximum frequency (field assignments per second)	Original program execution time (s)
1. Compress	1,900,000	50.4
2. Jess	169,000	22.45
3. Db	254	75
4. Javac	217,000	38
5. Mpegaudio	495,000	57.4
6. Jack	27,000	27
7. Ray tracer	787,000	17
8. Decaf	56,000	15
9. Ideal gas tank	23,150	57
10. Microbenchmark	40,000,000	2.4

The maximum field assignment frequency in microbenchmarks is 40 million assignments per second, but that would be difficult to reach in an application because the microbenchmarks contain a single assignment inside a loop. The compress program has the highest field assignment frequency in the SPECjvm98 application suite, 1.9 million assignments per second. Other SPEC applications, as well as the Decaf compiler and the ideal gas tank applet, have much lower maximum field assignment frequencies.

Figure 10 shows the frequency distribution of field assignments in the SPECjvm98 applications. The left graph indicates how many fields have an assignment frequency in the range indicated on the x axis. For example, only four fields are assigned between one million and two million times per second. The right graph shows the cumulative percentage of fields that have assignment frequencies lower than indicated on the x axis; 95% of all fields have fewer than 100,000 assignments per second.

To predict the overhead of a typical selection query, we can now calculate the overhead as a function of invocation frequency. Figure 11 uses the minimum (130 ns) and

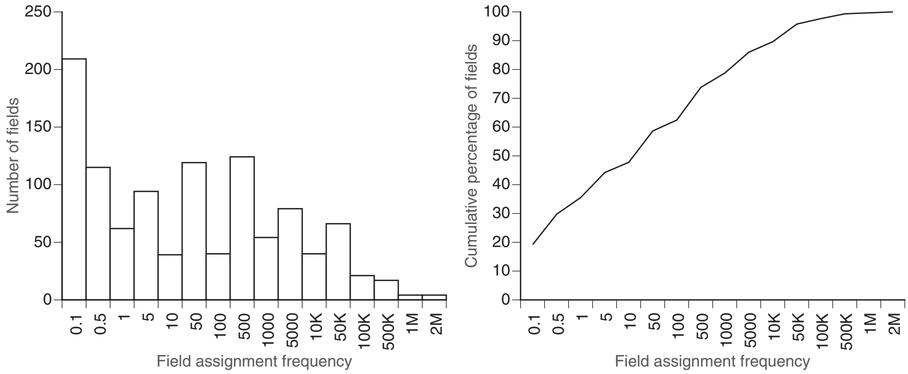


Figure 10. Field assignment frequency in SPECjvm98

maximum (4.26 µs) values of $T_{evaluate}$ from Table 6 to plot the estimated selection query overhead for a range of invocation frequencies. For example, a selection query on a field updated 500,000 times per second would have an overhead of 6.5% if its reevaluation time was 130 ns. If the reevaluation time was 4.26 µs, the overhead will be a factor of 3.13. The graph reveals that selection queries on fields assigned less than 100,000 times a second—95% of fields—have a predicted overhead of less than 43% even for the most costly selection constraint. For less costly selections, the query overhead is acceptable for all fields.

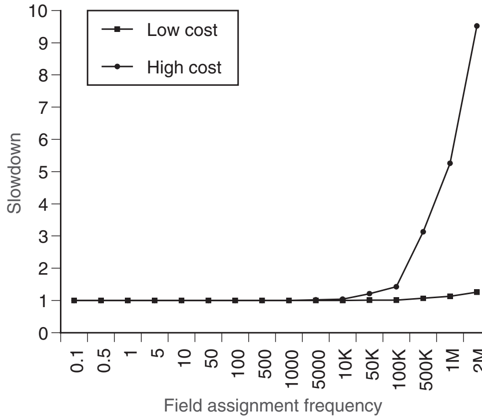


Figure 11. Predicted slowdown

The graph shows the predicted overhead as a function of update frequency. For example, the predicted overhead of a low-cost selection query on a field updated 500,000 times per second is 6.5%; the predicted overhead of a high-cost query with the same frequency is a factor of 3.13.

In the current model, the evaluation time $T_{evaluate}$ models all sources of query overhead. This time includes the actual reevaluation time as well as the additional garbage collection time, the class instrumentation cost, and the first evaluation cost. It would be more exact to model each of these overheads separately. However, for long running programs the evaluation time dominates the total cost, so the values of $T_{evaluate}$ are likely to fall in the range we have covered.

In summary, the performance model predicts that most selection queries will have less than 43% overhead. The model can be used as a framework for concrete overhead predictions and future model refinements.

6 Queries with Changing Results

So far we discussed using dynamic queries for debugging, where the program stops as soon as the query returns a non-empty result. However, programmers can also use queries to monitor program behavior. For example, in the ideal gas tank simulation, users may want to monitor all molecule near-collisions with a query:

```
Molecule* m1 m2.    m1.closeTo(m2) && m1 != m2
```

Programmers may use this information to check the frequency of near-collisions, to find out if near-collisions are handled in a special way by the program, or to check the correspondence of program objects with the visual display of the simulation. In this case, the debugger should not stop after the result becomes non-empty, but instead should continue executing the program and updating the query result as it changes. Such monitoring, perhaps coupled with visualization of the changing result, can help users understand abstract object relationships in large programs written by other people. How can a debugger support continuous updating of query results while the program executes?

Table 8. Benchmark queries with non-empty results

Query	Slowdown
1. Molecule1 z. z.x < 200	1.05
2. Id x. x.type == 0	1.23
3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == 0	1.3
4. spec.benchmarks._201_compress.Compressor z. z.OutCnt == 0	1.19
5. spec.benchmarks._201_compress.Compressor z. z.out_count == 0	1.09
6. Molecule1 z; Molecule2 z1. z.x < z1.x && z.y < z1.y (33x33 join)	1.47
7. Lexer l; Token t. l.token == t && t.type == 0 (120,000x600 hash join)	4.09
8. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. (p.z == ip.t) && (p.z > 100) (85,000x8,000 hash join)	212.4
9. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.out_count (1x1 hash join)	9.07
10. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < z.InCnt (1x1 join)	127
11. Test5 z. z.x % 2 == 0	45

The dynamic query-based debugger described above needs only a few changes to support monitoring queries. The basic scheme and the implementation of the dynamic query-based debugger discussed in section 3 remain the same. The only new component of the debugger is a module that maintains the current query result. As discussed in section 3.4.1, the debugger reevaluates only the changed part of the query. Consequently, the result handling module must store the query result from the previous evaluation and then merge it with the new partial result. To achieve that, after query execution

the debugger deletes all tuples from the previous result that contain the changed domain object and inserts the new tuples generated by the incremental reevaluation.

Experiments with queries similar to the ones in Table 1 show that adding the query result update functionality does not significantly change the query evaluation overhead (Table 8). The only exception is the microbenchmark selection query 11 which updates the query result during each reevaluation. Consequently, the overhead of the selection increases from 6.4 times to 45 times, although part of this increase can be attributed to the more costly selection constraint. However, such frequent result updates are unlikely for most monitoring queries: programmers can only absorb infrequent result changes, so, if results change rapidly, the display will be unintelligible unless it is artificially slowed down or used off-line.

To summarize, monitoring queries are useful for understanding and visualizing program behavior. With slight modifications our debugger supports monitoring queries. Unless the result changes very rapidly, the additional overhead of monitoring query execution is insignificant when compared to similar debugging queries.

7 Related Work

We are unaware of other work that directly corresponds to dynamic query-based debugging. The query-based debugging model and its non-dynamic implementation are presented in a previous paper [LHS97].

Extensions of object-oriented languages with rules as in R++ [LMP97] provide a framework that allows users to execute code when a given condition is true. However, R++ rules can only reference objects reachable from the root object, so R++ would not help to find the javac error we discussed. Due to restrictions on objects in the rule, R++ also does not handle join queries.

Sefika et al. [SSC96] implemented a system allowing limited, unoptimized selection queries about high-level objects in the Choices operating system. The system dynamically shows program state and run-time statistics at various levels of abstraction. Unlike our dynamic query-based debugger, the tool uses instrumentation specific to the application (Choices).

While no one has investigated the query-based debugging specifically, various researchers have proposed a variety of enhancements to conventional debugging [And95, Cop94, DHKV93, GH93, GWM89, KRR94, Laf97, LM94, LN97, WG94]. The debuggers most closely related to dynamic query-based debugging visualize object relationships—usually references or an object call graph. Duel [GH93] displays data structures by using user script code. HotWire [LM94] allows users to specify custom object visualizations in constraint language. Look! [And95], Object Visualizer [DHKV93], PV [KRR94], and Program Explorer [LN97] provide numerous graphical and statistical run-time views with class-dependent filtering but do not allow general queries. Our debugger can gather statistical data through queries with non-empty results ("How many lists of size greater than 500 exist in the program?") but does not display animated statistical views.

Visualizing debuggers gather information by either instrumenting the source code [DHKV93, LM94] or by using program traces [KRR94, LN97]. A port of our debugger to C++ would have to use one of these techniques. Laffra [Laf97] discusses visual

debugging in Java using source code instrumentation or JVM changes. We opted for the third method—class file instrumentation at load time. Consens et al. [CHM94, CMR92] use the Hy$^+$ visualization system to find errors using post-mortem event traces. De Pauw et al. [DLVW98] and Walker et al. [WM+98] use program event traces to visualize program execution patterns and event-based object relationships, such as method invocations and object creation. This work is complementary to ours because it focuses on querying and visualizing run-time events while we query object relationships.

Dynamic query-based debugging extends work on data breakpoints [WLG93]—breakpoints that stop a program whenever an object field is assigned a certain value. Pre-/postconditions and class invariants as provided in Eiffel [Mey88] can be thought of as language-supported dynamic queries that are checked at the beginning or end of methods. Unlike dynamic queries, they are not continuously checked, they cannot access objects unreachable by references from the checked class, nor can they invoke arbitrary methods. Dynamic queries could be used to implement class assertions for languages that do not provide them. The current implementation of dynamic queries cannot use the "old" value of a variable, as can be done in postconditions. We view the two mechanisms as complementary, with queries being more suitable for program exploration as well as specific debugging problems.

Software visualization systems such as BALSA [Bro88], Zeus [Bro91], TANGO/XTANGO/POLKA [Sta90], Pavane [Rom92], and others [HKWJ95, Mos97, RC93] offer high-level views of algorithms and associated data structures. Software visualization systems aim to explain or illustrate the algorithm, so their view creation process emphasizes vivid representation. Hart et al. [HKR97] use Pavane for query-based visualization of distributed programs. However, their system only displays selected attributes of different processes and does not allow more complicated queries.

Dynamic queries are related to incremental join result recalculation in databases [BC79, BLT86]. We use the basic insights of this work to implement the incremental query evaluation scheme. Coping with inter-object constraints in the extended ODMG model [BG98] may require methods similar to dynamic query-based debugging.

Slicing [Wei81, Tip95] determines the program statements that affect a certain program point. It could be modified to determine the change sets of queries.

8 Conclusions

The cause-effect gap between the time when a program error occurs and the time when it becomes apparent to the programmer makes many program errors hard to find. The situation is further complicated by the increasing use of large class libraries and complicated pointer-linked data structures in modern object-oriented systems. A misdirected reference that violates an abstract relationship between objects may remain undiscovered until much later in the program's execution. Conventional debugging methods offer only limited help in finding such errors. Data breakpoints and conditional breakpoints cannot check constraints that use objects unreachable from the statement containing the breakpoint.

We have described a dynamic query-based debugger that allows programmers to ask queries about the program state and updates query results whenever the program changes an object relevant to the query, helping programmers to discover object

relationship failures as soon as they happen. This system combines the following novel features:

- An extension of query-based debugging to include dynamic queries. Not only does the debugger check object relationships, but it determines exactly when these relationships fail while the program is running. This technique closes the cause-effect gap between the error's occurrence and its discovery.
- Implementation of monitoring queries. The debugger helps users to watch the changes in object configurations through the program's lifetime. This functionality can be used to better understand program behavior.

The implementation of the query based debugger has good performance. Selection queries are efficient with less than a factor of two slowdown for most queries measured. We also measured field assignment frequencies in the SPECjvm98 suite, and showed that 95% of all fields in these applications are assigned less than 100,000 times per second. Using these numbers and individual evaluation time estimates, our debugger performance model predicts that selection queries will have less than 43% overhead for 95% of all fields in the SPECjvm98 applications. Join queries are practical when domain sizes are small and queried field changes are infrequent.

Good performance is achieved through a combination of two optimizations:

- Incremental query evaluation decreases query evaluation overhead by a median factor of 160, greatly expanding the class of dynamic queries that are practical for everyday debugging.
- Custom code generation for selection queries produces a median speedup of 15, further improving efficiency for commonly occurring selection queries.

We believe that dynamic query-based debugging adds another powerful tool to the programmer's tool chest for tackling the complex task of debugging. Our implementation of the dynamic query-based debugger demonstrates that dynamic queries can be expressed simply and evaluated efficiently. We hope that future mainstream debuggers will integrate a similar functionality, simplifying the difficult task of debugging and facilitating the development of more robust object-oriented systems.

9 Acknowledgments

We thank the anonymous reviewers and Amer Diwan, Karel Driesen, Sylvie Dieckmann, Andrew Duncan, and Jeff Bogda for valuable comments on earlier versions of this paper. This work was funded in part by Sun Microsystems, the State of California MICRO program, and by the National Science Foundation under CAREER grant CCR96–24458 and grants CCR92–21657 and CCR95–05807.

10 References

[And95] Anderson E., Dynamic Visualization of Object Programs Written in C++, *Objective Software Technology Ltd.,* http://www.objectivesoft.com/, 1995.

[BC79] Buneman, O.P.; Clemons E.K., Efficiently Monitoring Relational Databases. *ACM Transactions on Database Systems,* 4(3), pp. 368-382, September 1979.

[BG98] Bertino, E., Guerrini, G., Extending the ODMG Object Model with Composite Objects, *Proceedings of OOPSLA'98*, pp. 259-270, Vancouver, October 1998. Published as *SIGPLAN Notices* 33(10), October 1998.

[BLT86] Blakeley, J.A.; Larson P.-A.; Tompa F. Wm.; Efficiently Updating Materialized Views. *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 61-71, Washington, D.C., USA, May 1986. Published as *SIGMOD Record* 15(2), June 1986.

[Bro88] Brown, M.H., Exploring Algorithms Using Balsa-II, *IEEE Computer* 21(5), pp. 14-36, May 1988.

[Bro91] Brown, M.H., Zeus: A System for Algorithm Animation and Multi-View Editing, *Proceedings of IEEE Workshop Visual Languages*, pp. 4-9, IEEE CS Press, Los Alamitos, CA., 1991.

[CHM94] Consens, M. P., Hasan M.Z., Mendelzon A.O., Debugging Distributed Programs by Visualizing and Querying Event Traces, *Applications of Databases, First International Conference*, ADB-94, Vadstena, Sweden, June 21-23, 1994, Proceedings in Lecture Notes in Computer Science, Vol. 819, Springer, 1994.

[CMR92] Consens, M.; Mendelzon, A.; Ryman, A., Visualizing and Querying Software Structures, *International Conference on Software Engineering*, Melbourne, Australia, May 11-15, 1992, ACM Press, IEEE Computer Science, p. 138-156, 1992.

[Cop94] Coplien, J.O., Supporting Truly Object-Oriented Debugging of C++ Programs., In: Proceedings of the 1994 USENIX C++ Conference, Cambridge, MA, USA, 11-14 April 1994. pp. 99-108, Berkley, CA, USA: USENIX Assoc, 1994.

[DHKV93] De Pauw, W.; Helm, R.; Kimelman, D.; Vlissides, J. Visualizing the Behavior of Object-Oriented Systems. In *Proceedings of the 8th Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1993*, Washington, DC, USA, 26 Sept.-1 Oct. 1993. SIGPLAN Notices, Oct. 1993, vol.28, (no.10):326-37.

[DLVW98]de Pauw, W.; Lorenz, D.; Vlissides, J.; Wegman, M. Execution Patterns in Object-Oriented Visualization. *Proceedings of the Fourth USENIX Conference on Object-Oriented Technologies and Systems*, Sante Fe, NM, USA, 27-30 April 1998, USENIX Association, 1998. pp. 219-34.

[Eis97] Eisenstadt, M., My Hairiest Bug War Stories, *Communications of the ACM*, Vol. 40., No. 4, pp. 30–38, April 1997.

[GH93] Golan, M.; Hanson, D.R. Duel-A Very High-Level Debugging Language. In: USENIX Association. *Proceedings of the Winter 1993 USENIX Conference*. San Diego, CA, 25-29 Jan. 1993. Berkley, CA, USA: USENIX Assoc, 1993. p. 107-17.

[GJS96] Gosling, J., Joy, B., Steele, G., *The Java Language Specification*, Addison-Wesley 1996.

[GWM89] Gamma E., Weinand A., Marty R., Integration of a Programming Environment into ET++ - a Case Study, *Proceedings ECOOP'89* (Nottingham, UK, July 10-14), pp. 283-297, S. Cook, ed. Cambridge University Press, Cambridge, 1989.

[HKR97] Hart D., Kraemer E., Roman G.-C., Interactive Visual Exploration of Distributed Computations. *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, pp.121-127, April 1997.

[HKWJ95]Hao, M.C.; Karp, A.H.; Waheed, A.; Jazayeri, M., VIZIR: An Integrated Environment for Distributed Program Visualization. *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS '95*, pp.288–92, Durham, NC, USA, January 1995.

[Kes90] Kessler, P., Fast Breakpoints: Design and Implementation. *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation* 1990, Published as *SIGPLAN Notices* 25(6), pp. 78–84, ACM Press, June 1990.

[KH98] Keller, R., Hölzle, U.; Binary Component Adaptation, *Proceedings ECOOP'98*, Springer Verlag Lecture Notes on Computer Science, Brussels, Belgium, July 1998.

[KRR94] Kimelman D., Rosenburg B., Roth T., Strata-Various: Multi-Layer Visualization of Dynamics in Software System Behavior, *Proceedings of Visualization'94*, pp. 172-178, IEEE 1994.

[Laf97] Laffra C., *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips*, pp. 229-252, Prentice Hall 1997.

[LB98] Liang, S., Bracha, G.; Dynamic Class Loading in the Java^{TM} Virtual Machine, *Proceedings of OOPSLA'98*, pp. 36-44, Vancouver, October 1998. Published as *SIGPLAN Notices* 33(10), October 1998.

[LHS97] Lencevicius, R.; Hölzle, U.; Singh, A.K., Query-Based Debugging of Object-Oriented Programs, *Proceedings of OOPSLA'97*, pp. 304-317, Atlanta, GA, October 1997. Published as *SIGPLAN Notices* 32(10), October 1997.

[LM94] Laffra C., Malhotra A., HotWire: A Visual Debugger for C++, *Proceedings of the USENIX C++ Conference*, pp. 109-122, Usenix Association 1994.

[LMP97] Litman D.; Mishra A.; Patel-Schneider P.F., Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules, *Proceedings of OOPSLA'97*, pp. 77-92, Atlanta, GA, October 1997. Published as *SIGPLAN Notices* 32(10), October 1997.

[LN97] Lange, D.B., Nakamura Y. Object-Oriented Program Tracing and Visualization, *IEEE Computer*, vol. 30, no. 5, pp. 63–70, May 1997.

[Mey88] Meyer B., *Object-Oriented Software Construction*, pp. 111 - 163, Prentice-Hall, 1988.

[Mos97] Mössenböck, H., Films as Graphical Comments in the Source Code of Programs. *Proceedings of the International Conference on Technology of Object Oriented Systems and Languages, TOOLS-23*, pp. 89-98, Santa Barbara, CA, USA, July-August 1997.

[RC93] Roman G.-C., Cox K.C., A Taxonomy of Program Visualization Systems, *IEEE Computer* 26(12), pp. 11-24, December 1993.

[Rom92] Roman, G.-C. et al., Pavane: A System for Declarative Visualization of Concurrent Computations, *Journal of Visual Languages and Computing*, Vol. 3, No. 2, pp. 161-193, June 1992.

[SPEC98] Standard Performance Evaluation Corporation, SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98/, 1998.

[SSC96] Sefika M., Sane A., Campbell R.H., Architecture-Oriented Visualization, In *Proceedings of OOPSLA'96*, pp. 389-405, San Jose, CA, October 1996. Published as SIGPLAN Notices 31(10), October 1996.

[Sun99] Java^{TM} 2 SDK Production Release, http://www.sun.com/solaris/, 1999.

[Sta90] Stasko, J., TANGO: A Framework and System for Algorithm Animation, *IEEE Computer* 23(9), pp. 27-39.

[Tip95] Tip, F., A Survey of Program Slicing Techniques. *Journal of Programming Languages*, vol.3, (no.3) pp. 121-89, Sept. 1995.

[Wei81] Weiser, M., Program Slicing. In: *5th International Conference on Software Engineering*, San Diego, CA, USA, 9-12 March 1981. New York, NY, USA, pp. 439-49, IEEE, 1981.

[WLG93] Wahbe R., Lucco S., Graham S.L., Practical Data Breakpoints: Design and Implementation. *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation* 1993, Albuquerque, June 1993. ACM Press 1993.

[WG94] Weinand, A.; Gamma, E. ET++-a portable, homogenous class library and application framework. In: *Computer Science Research at UBILAB, Strategy and Projects. Proceedings of the UBILAB Conference '94*, Zurich, Switzerland, 1994. pp. 66-92. Edited by: Bischofberger, W.R.; Frei, H.-P. Konstanz, Switzerland: Universitätsverlag Konstanz, 1994.

[WM+98] Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J., Visualizing Dynamic Software System Information through High-level Models, *Proceedings of OOPSLA'98*, pp. 271-283, Vancouver, October 1998. Published as *SIGPLAN Notices* 33(10), October 1998.

Foundations for Virtual Types

Atsushi Igarashi and Benjamin C. Pierce

Department of Computer & Information Science
University of Pennsylvania
200 South 33rd St.
Philadelphia, PA 19104, USA
{igarasha,bcpierce}@saul.cis.upenn.edu

Abstract. *Virtual types* have been proposed as a notation for generic programming in object-oriented languages—an alternative to the more familiar mechanism of *parametric classes*. The tradeoffs between the two mechanisms are a matter of current debate: for many examples, both appear to offer convenient (indeed almost interchangeable) solutions; in other situations, one or the other seems to be more satisfactory. However, it has proved difficult to draw rigorous comparisons between the two approaches, partly because current proposals for virtual types vary considerably in their details, and partly because the proposals themselves are described rather informally, usually in the complicating context of full-scale language designs.

Work on the foundations of object-oriented languages has already established a clear connection between parametric classes and the polymorphic functions found in familiar typed lambda-calculi. Our aim here is to explore a similar connection between virtual types and dependent records. We present, by means of examples, a straightforward model of objects with embedded type fields in a typed lambda-calculus with subtyping, type operators, fixed points, dependent functions, and dependent records with both "bounded" and "manifest" type fields (this combination of features can be viewed as a measure of the inherent complexity of virtual types). Using this model, we then discuss some of the major differences between previous proposals and show why some can be checked statically while others require run-time checks. We also investigate how the partial "duality" of virtual types and parametric classes can be understood in terms of translations between universal and (dependent) existential types.

1 Introduction

Language support for generic programming plays an important role in the development of reusable libraries. In object-oriented languages, two different approaches to genericity have been considered. The more familiar one—based closely on the classical *parametric polymorphism* of functional languages such as ML and Haskell—can be found, for example, in the *template* mechanism of C++ [32] and the *parametric classes* in a number of proposed extensions to

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 161–185, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Java [26, 25, 2, 3, 12, etc.]. An alternative approach, commonly called *virtual types* (or *virtual classes*), allows classes and objects to contain types as members, along with the usual fields and methods.[1] Virtual types were originally developed in Beta [23] and have recently been proposed for Java [33].

The static typing of virtual types is not yet clearly understood. Indeed, early proposals were statically unsafe, requiring extra runtime checks; more recent work has produced several proposals for type-safe variants [35, 5]. These proposals vary substantially in their details, and have generally been presented in rather informal terms—and in the complicating context of full-scale language designs—making them difficult to evaluate and compare.

Our goal in this paper is to establish a rigorous setting in which to understand and discuss the basic mechanisms of virtual types. Following a long line of past work on foundations for object-oriented programming (see [4] for history and citations), we model objects and classes with virtual types as a particular style of programming in a fairly standard typed lambda-calculus. On this basis, we examine (1) the type-theoretic features that seem to be required for modeling virtual types, (2) the similarities and differences between existing proposals, and (3) the type-theoretic intuitions behind the much-discussed "overlap" between virtual types and parametric classes in practice.

The rest of the paper is organized as follows. Section 2 reviews the idea of virtual types by means of a standard example, the animal/cow class hierarchy of Shang [31]. Section 3 sketches the main features of the typed lambda-calculus that forms the setting for our model. (The calculus is defined in full in Appendix A, for expert readers.) Section 4 develops the encoding of the Animal/Cow example in detail. Section 5 discusses the relation between virtual types and parametric classes as mechanisms for generic programming. Section 6 reviews previous work on virtual types in the light of our model. Section 7 sketches some directions for future work.

Our presentation is self-contained, but somewhat technical at times. Familiarity with past work on modeling objects in typed lambda-calculi (e.g., [29], [19], [4], or Chapter 18 of [1]) will help the reader interested in following in detail. Another useful source of background is Harper and Lillibridge [18, 21] and Leroy's [20] papers on modeling module systems using dependent records with "manifest" bindings.

2 Virtual Types

We begin by reviewing the notion of virtual types through an example. This example, used throughout the paper, is a variant of the animal/cow example of Shang [31]. (Our notation is Java-like, but does not exactly correspond to any of the existing proposals for virtual types in Java.)

We begin by defining a generic class of animals, along with its interface.

[1] Referring to this approach with the phrase "virtual types" is somewhat confusing, since—as we will see—these type members may or may not be "virtual" in the sense of *virtual* or *abstract methods*. But the terminology is standard.

```
interface AnimalI {                  virtual class Animal
  type FoodType <: Food;                     implements AnimalI {
  void eat (FoodType f);               virtual type FoodType <: Food;
  void eatALot (FoodType f); }         virtual void eat (FoodType f);
                                       void eatALot (FoodType f) {
                                         eat(f); eat(f); }}
```

Every animal has methods `eat` and `eatALot`, both accepting some food as an argument. The body of the `eat` method, which is specific to particular kinds of animals, is omitted; the `virtual` marker defers the responsibility of providing an implementation to subclasses. (We use the C++ keyword `virtual` in preference to Java's `abstract` to avoid terminological confusion: locutions like "abstract type" already have a well-established meaning.) The calls to `eat` from the body of the `eatALot` method will call whatever body is provided by the subclass.

Similarly, the class `Animal` defers specifying exactly what kind of food a given kind of animal likes to eat. The virtual member `FoodType` acts as placeholder for this type, allowing it to be mentioned in the types of `eat` and `eatALot`, just as the declaration of `eat` provides a placeholder for its eventual implementation, allowing it to be referred to from the body of `eatALot`. Classes with virtual members (either types or methods) cannot be instantiated, since they are incomplete: they can only be subclassed.

The interface `AnimalI` specifies that every animal object has three members: a type `FoodType` and methods `eat` and `eatALot`. The `FoodType` member of every animal is known to be some kind of `Food` (`FoodType<:Food`), but, since different animals eat different kinds of food, the exact identity of this type is not visible. It follows immediately that it is not possible to feed an animal without knowing what kind of animal it is: if `a` is an object of type `AnimalI`, then `a`'s `eat` method requires an argument of type `a.FoodType`; but there is no way to obtain a value of this type (except, perhaps, by building a nutrient-free empty value using `new`).

Specific kinds of animals are modeled by classes inheriting from `Animal`. For example, here is a `Cow` class and its interface:

```
interface CowI                       class Cow extends Animal
         extends AnimalI {                   implements CowI {
  type FoodType ↔ Grass; }             final type FoodType ↔ Grass;
                                       void eat (FoodType f) { ... }}
```

In `Cow`, the virtual method `eat` is given a concrete implementation (shown as "`...`"). Similarly, the virtual type member `FoodType` is given a concrete value, `Grass`. The annotation `final` on the `FoodType` member means that it cannot be redefined by subclasses: every subclass of `Cow` is guaranteed to have `Grass` as its `FoodType`. The interface `CowI` reflects the fact that `FoodType` is final: in effect, it tells the world that every cow eats food whose type is *equal* to `Grass`. Thus, given an object `a` of type `CowI`, we may validly obtain some grass from any source and pass it to the `eat` or `eatALot` methods.

Virtual types are also useful in more standard examples of generic programming. For example, a generic `Bag` class can be defined with a virtual type

ElementType. Then classes NatBag, StringBag, etc. can be defined by inheriting from Bag and giving ElementType a final binding to Nat or String. Other examples of generic programming with virtual types can be found in [23, 33].

3 Summary of Type System

It is well understood [29, 6, etc.] how parametric classes—classes abstracted on type parameters—can be understood as polymorphic functions in a typed lambda-calculus. By analogy, objects with type members should clearly be modeled as some kind of records with type fields. Fortunately, such records have been studied extensively in the type-theory literature (e.g. [9]). Indeed, even the constraints on type members appearing in the interfaces AnimalI (FoodType<:Food) and CowI (FoodType↔Grass) correspond to well-known constructions in the typed lambda-calculi used by Harper and Lillibridge [18, 21] and Leroy [20] to model module systems. Records with type fields constrained by <: are a generalization of *partially abstract types* [11]; records with type fields constrained by ↔ correspond to *translucent* or *manifest* sums.

The typed lambda-calculus sketched in this section is based directly on these intuitions. In essence, it can be described as System $F^{\omega}_{\leq}$ (the omega-order polymorphic lambda-calculus with subtyping [8, 10, 27, 14]) plus dependent records with both "bounded" [11] and "manifest" [18, 21, 20] type fields, plus dependent functions.[2] We begin by briefly reviewing the features of System $F^{\omega}_{\leq}$ (Sections 3.1 and 3.2); we then concentrate on explaining records with type fields (Section 3.3) and dependent functions (Section 3.4), which are less familiar. Appendix A gives a more formal summary of the whole system.

3.1 Functions, Polymorphism, and Parameterized Types

The core of the system is Girard's System F^{ω} [17]. This calculus can be viewed as a simple functional programming language with three distinct forms of abstraction: (1) *ordinary functions* (i.e., terms abstracted over terms); (2) *polymorphic functions* (i.e., terms abstracted over types); and (3) *parametric types* (i.e., types abstracted over types). We write all three forms with similar concrete syntax. For example,

```
plustwo = λ[x:Nat] succ(succ(x));
```

is an ordinary function that adds two to its argument. Similarly,

```
id = λ[X:*] λ[x:X] x;
```

is the polymorphic identity function, and

[2] The system can also be described as an extension of Coquand and Huet's *Calculus of Constructions* [15] with subtyping and dependent records [13]. Experts will note that we use an unstratified presentation of dependent records; this renders the system inconsistent as a logic, but the fix operator does that anyway.

```
double = λ[X:*] λ[f:X→X] λ[x:X] f(f(x));
```

is a polymorphic function that accepts a type `X`, a function `f` (of type `X→X`), and an argument `x` (of type `X`), and applies `f` twice to `x`. (The annotation `X:*` indicates that `X` is a type parameter.) Thus,

```
plusfour = double Nat plustwo;
```

is a fancy way of writing the function that adds four to its (numeric) argument.

Parametric types are written in a similar style. For example,

```
Pair = λ[A:*] λ[B:*] {|fst:A, snd:B|};
```

is a convenient abbreviation for the parametric type of pairs, and

```
PairNatNat = Pair Nat Nat;
```

is the concrete type of pairs of numbers. The usual (polymorphic) operations on pairs can be defined as follows:

```
fst = λ[A:*] λ[B:*] λ[p: Pair A B] p.fst;
snd = λ[A:*] λ[B:*] λ[p: Pair A B] p.snd;
pair = λ[A:*] λ[B:*] λ[a:A] λ[b:B] ({fst=a, snd=b} :: Pair A B);
```

The types of these operations are:

```
fst : ∀[A:*] ∀[B:*] Pair A B → A
snd : ∀[A:*] ∀[B:*] Pair A B → B
pair : ∀[A:*] ∀[B:*] A → B → Pair A B
```

(In the following, we will often display defined terms together with their types.) Note that the definition of `pair` uses an explicit coercion (`:: Pair A B`) to control how its type is printed by the typechecker. Leaving it off results in a definition with exactly the same behavior

```
pair = λ[A:*] λ[B:*] λ[a:A] λ[b:B] {fst=a, snd=b};

pair : ∀[A:*] ∀[B:*] A → B → {|fst:A, snd:B|}
```

(since we have defined `Pair A B` to be interchangeable with `{|fst:A,snd:B|}`), but less intuitive for the reader.

To ensure their well-formedness, types and type operators are assigned *kinds*, `K`, which have the form `*` or `K→K`. Type expressions of kind `*` (pronounced "type") are ordinary types; type expressions of kind `*→*` are functions from types to types; etc.

It is sometimes useful to write *higher-order* type operators—that is, type operators whose arguments are type operators. For example,

```
BothBool = λ[F:*→*→*] F Bool Bool;
```

is higher-order type operator that, when applied to any operator `O`, yields the type `O Bool Bool`. Thus:

```
mypair = pair Bool Bool true false :: BothBool Pair;
```

A more natural example of higher-order type operators will be seen later in the `Object` type constructor: its argument `I` is itself an operator abstracted over the "self type" `Rep`.

For constructing objects, we shall also need a fixed-point constructor. If `t` is a function from `T` to `T`, then `fix T t` is its fixed point. (Writing `T` explicitly simplifies the typechecking of `fix` in the presence of dependent types.)

$$\frac{\Gamma \vdash \texttt{t} : \texttt{T}{\rightarrow}\texttt{T}}{\Gamma \vdash \texttt{fix T t} : \texttt{T}}$$

For example, here is how `fix` is used to construct a factorial function:

```
fact = fix (Nat→Nat) λ[f:Nat→Nat] λ[n:Nat]
         if eq n 0 then 1 else times n (f (pred n)) :: Nat → Nat;
```

3.2 Subtyping

Next, we add the familiar notion of subtyping. For example, subtyping of function types is contravariant on the left and covariant on the right. The subtype relation has a maximal element, called `Top`. Constraining a type variable to be a subtype of `Top` is actually no constraint at all, so we can recover unbounded quantification from bounded, writing `λ[X<:Top]t` in place of `λ[X:*]t`. (We will continue to write `λ[X:*]t` in what follows, for readability.)

Subtyping is extended *pointwise* to type operators: `λ[X:K]S` is a subtype of `λ[X:K]T` if `S` is a subtype of `T` under all legal substitutions for `X`.

$$\frac{\Gamma, \texttt{X:K} \vdash \texttt{S} <: \texttt{T}}{\Gamma \vdash \lambda\texttt{[X:K]S} <: \lambda\texttt{[X:K]T}}$$

For example, `λ[T:*] Top→T` is a subtype of `λ[T:*] Nat→T` since `Nat` is a subtype of `Top`.

3.3 Records with Type Fields

To support records with type fields, a bit of machinery is required. First, we must deal with the fact that later fields in a record may refer to earlier fields by name—e.g., the type of the `eat` field must refer to the `FoodType` field. (Thus, in particular, the order of fields is significant in dependent records.) Second, we must be able to deal with record-projection expressions like `a.FoodType` appearing in the types of values (e.g., `a.eat`). The second requirement in particular goes somewhat beyond what can be expressed using ordinary existential types, taking us into the realm of *dependent records*.

In general, a dependent record has the form $\{\beta_i{}^{i\in 1\cdots n}\}$, where each β_i is a *field* of one of two forms: either a term field $\texttt{x}_i\texttt{=}\texttt{t}_i$ or a type field $\texttt{X}_i\texttt{=}\texttt{T}_i$. The name $\texttt{x}_i$ or $\texttt{X}_i$ is not only used to project a record from outside but also is a

binder whose scope is the rest of the fields in the record.[3] For example, in the record value `r = {X=Nat,x=λ[y:X]y+1}`, `X` in the second field is bound by the first occurrence of `X`.

A record type has the form $⦃\mathtt{B}_i{}^{i\in 1\cdots n}⦄$, where $\mathtt{B}_i$ is a *binding* of one of three forms: a *term binding* `x:T`, a *bounded type binding* `X<:T`, or a *manifest type binding* `X↔T`. (In examples, we will also use type bindings of the form `X:*` as an abbreviation for `X<:Top`.) For example, the record `r` above has type `⦃X↔Nat,x:X→X⦄`. A less informative type also possessed by `r` is `⦃X<:Top,x:X→X⦄`, which hides the representation of `X` and corresponds to the usual existential type `∃X.X→X`. In order to remind us of a connection to existential types, we sometimes write ∃ before a field name in records or record types, like `⦃∃X<:Top,x:X→X⦄`, although ∃ itself doesn't have a significant meaning. Formally, the typing rule for record introduction is:

$$\frac{\Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{j-1} \vdash \beta_j : \mathtt{B}_j \;{}^{j\in 1\cdots n} \qquad \Gamma \vdash ⦃\mathtt{B}_i{}^{i\in 1\cdots n}⦄ : *}{\Gamma \vdash \{\beta_i{}^{i\in 1\cdots n}\} : ⦃\mathtt{B}_i{}^{i\in 1\cdots n}⦄}$$

Each field definition β_i must satisfy the corresponding binding $\mathtt{B}_i$ under a context augmented with the information of the preceding fields $(\Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{i-1})$. Term fields $\mathtt{x}_i\mathtt{=t}_i$ satisfy bindings of the form $\mathtt{x}_i\mathtt{:T}_i$; type fields $\mathtt{X}_i\mathtt{=T}_i$ satisfy manifest type bindings $\mathtt{X}_i\leftrightarrow\mathtt{T}_i$. (Note that we cannot directly derive a record type with a bounded type binding using the rule above . For example, the type given to `r` above is `⦃∃X↔Nat,x:X→X⦄`. If we want to hide the identity of `X` and give `r` the abstract type `⦃∃X:*,x:X→X⦄`, we must use the usual subsumption rule plus the record subtyping rules discussed below.)

The rule for record projections is basically the same as the standard record elimination rule: if a field `x` of `t` has binding `x:T`, then `t.l` has type `T`. If `T` depends on other fields—that is, if the name $\mathtt{X}_i$ (or $\mathtt{x}_i$) occurs free in `T`—then the corresponding record projection $\mathtt{t.X}_i$ (or $\mathtt{t.x}_i$, resp.) should be substituted for $\mathtt{X}_i$ (or $\mathtt{x}_i$, resp.) to prevent the field name from escaping its scope.[4]

$$\frac{\Gamma \vdash \mathtt{t} : ⦃\mathtt{B}_i{}^{i\in 1\cdots n}⦄ \qquad \mathtt{B}_j\mathtt{=x:T}}{\Gamma \vdash \mathtt{t.l}_j : \{BV(\mathtt{B}_i) \mapsto \mathtt{t.l}_i{}^{i\in 1\cdots j-1}\}\mathtt{T}}$$

[3] Strictly speaking, these two mechanisms should be kept separate. In the full typing rules in Appendix A, each field is given two names: an *external* name, which can be used for projections, and an *internal* name, which binds the subsequent occurrences in the record. The simplified syntax presented in the body of the paper corresponds to the special case where the external and internal names are identical.)

[4] Experts will note that we give a somewhat simpler version of this rule than Harper and Lillibridge [18, 21] or Leroy's [20] formulations. The reason we can do this is that we are not—at this stage—considering computational effects such as references or exceptions. If any "effectful" constructs are added to the system, our projection rule needs to be refined to ensure soundness. This can be done in different ways, but the basic intuition is that a dependent projection $\mathtt{t.l}_j$ should be allowed only if the expression `t` is *pure*. Similar comments apply to application rule below.

We write $BV(\mathtt{B})$ for the bound variable of the binding B; that is, $BV(\mathtt{x:T}) = \mathtt{x}$, $BV(\mathtt{X}{\leftrightarrow}\mathtt{T}) = \mathtt{X}$, and $BV(\mathtt{X{<:}T}) = \mathtt{X}$. We also write $\{\mathtt{X} \mapsto \mathtt{T}\}$ for capture-avoiding substitution of T for X.

The subtyping rule for record types is:

$$\frac{\vdash \Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{n+k}\ \mathtt{ok} \qquad \vdash \Gamma, \mathtt{B}'_1, \ldots, \mathtt{B}'_n\ \mathtt{ok} \qquad \Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{j-1} \vdash \mathtt{B}_j <: \mathtt{B}'_j\ {}^{j\in 1\cdots n}}{\Gamma \vdash \{\!|\, \mathtt{B}_i{}^{i\in 1\cdots n+k} \,|\!\} <: \{\!|\, \mathtt{B}'_i{}^{i\in 1\cdots n} \,|\!\}}$$

As usual for ordinary (non-dependent) records, "width subtyping" is allowed: extra fields (the n+1-st to n+k-th fields) can be dropped. Also, corresponding bindings $\mathtt{B}_i$ and $\mathtt{B}'_i$ are compared using a *sub-binding* relation. When both are term bindings—i.e., $\mathtt{B}_i$ and $\mathtt{B}'_i$ are of the form x:S and x:T—S should be a subtype of T: this captures ordinary "depth subtyping." For type bindings, we have (X↔T) <: (X<:S) <: (X<:U) if T <: S <: U; the first clause ((X↔T) <: (X<:S)) allows the exact identity of a type field to be replaced with an upper bound; the second ((X<:S) <: (X<:U)), corresponding to subtyping of bounded existential types, allows us to loosen the bound of X. For example, we can derive {|∃X↔Nat,x:X→X|} <: {|∃X:*,x:X→X|}. (As usual, this rule leads to an undecidable subtyping relation [28, 21].)

3.4 Dependent Functions

For the encoding of classes, we will need to be able to give quite precise types to functions, showing the dependency of the *type* of the result on the *value* of the argument.

In outline, the intuition is this. Suppose we write a function

```
cl = λ[self:{|∃T:*, x:T, f:T→T|}]
         {T=self.T, x=self.f(self.x), f=self.f};
```

whose argument is a record containing a type, a value (of that type), and a function (on that type), and whose result is a record with a similar shape, but where the value field is calculated by applying the argument's function field to the argument's value field. The type of this function

```
cl : Π[self: {|∃T:*,x:T,f:T→T|}] {|∃T↔self.T, x:T, f:T→T|}
```

expresses the fact that the T field of the result is identical to the T field of the argument. Next, suppose we create a record containing these three items

```
r1 = {T=Nat, x=3, f=plusfour} :: {|∃T:*, x:T, f:T→T|};

r1 : {|∃T:*, x:T, f:T→T|}
```

and use the function cl to obtain another record of the same shape:

```
r2 = cl r1;

r2 : {|∃T↔r1.T, x:T, f:T→T|}
```

Notice that, because of the dependent typing of `cl`, the type of `r2` exposes the fact that it was built from `r1`—in particular, that their type components are equal. Hence, it is legal to project the function field from `r2` and apply it to the value field from `r1`:

```
i = r2.f r1.x;

i : r2.T
```

In the absence of dependent functions, the best type we could have given to `cl` would be:

```
cl : {|∃T:*,x:T,f:T→T|} → {|∃T:*, x:T, f:T→T|}
```

If we build `r2` from `r1` using this less refined type for `cl`,

```
r2 = cl r1;

r2 : {|∃T:*, x:T, f:T→T|}
```

we obtain no information about the relation between `r1`'s `T` field and `r2`'s, and the application `r2.f r1.x` is not allowed.

In general, a function λ`[x:S]t` has type Π`[x:S]T`, where `x` is allowed to appear in `T`. (When `x` does *not* appear in `T`, we write Π`[x:S]T` as `S→T`, recovering the usual notation for function types as a special case of dependent function types.) The rules for function abstraction and application are generalized accordingly:

$$\frac{\vdash \Gamma, \texttt{x:S ok} \qquad \Gamma, \texttt{x:S} \vdash \texttt{t} : \texttt{T}}{\Gamma \vdash \lambda\texttt{[x:S]t} : \Pi\texttt{[x:S]T}} \qquad\qquad \frac{\Gamma \vdash \texttt{t} : \Pi\texttt{[x:S]T} \qquad \Gamma \vdash \texttt{s} : \texttt{S}}{\Gamma \vdash \texttt{t s} : \{\texttt{x} \mapsto \texttt{s}\}\texttt{T}}$$

4 Encoding Virtual Types

With the formalities of our typed lambda-calculus now in hand, we can proceed to the technical heart of the paper: a straightforward encoding of the animal example from Section 2 in terms of records with type fields. For the sake of concreteness, we extend the familiar *existential encoding* of objects [29, 19].

To avoid introducing additional complexities in the type theory, we give an encoding of *purely functional* objects; for example, we assume that an animal's `eat` method returns a new, satiated animal rather than side-effecting the internals of the receiving animal.

4.1 Interfaces

To get warmed up, let's begin with an example that does *not* involve virtual types: one-dimensional point objects with methods `get` to retrieve a current coordinate, `set` to move to a new coordinate, and `bump` to move a little from the present position.

In the simple existential encoding, the *interface* of an object is represented as a type operator of the form λ[Rep:*]{|m_i:$T_i$$^{i\in 1\cdots n}$|}, where the bound variable Rep stands for the hidden type of the object's internal state, and where each T_i is type of the corresponding method m_i. Each method takes the internal state of the object as an explicit argument and, if appropriate, returns a new internal state as its result. For example, the interface PointI of point objects is represented as

```
PointI = λ[Rep:*] {|get:Rep→Nat, set:Rep→Nat→Rep, bump:Rep→Rep|};

PointI : * → *
```

Interfaces for objects with virtual types may include not only methods but also type fields, which declare the bounds of the virtual types. The interface AnimalI is represented as

```
AnimalI=λ[Rep:*]{|∃FT<:Food, eat:Rep→FT→Rep, eatALot:Rep→FT→Rep|};
```

The binding FT<:Food is a direct transliteration of the constraint on FT in Section 2. Similarly, the interface CowI is represented as

```
CowI = λ[Rep:*]{|∃FT↔Grass, eat:Rep→FT→Rep, eatALot:Rep→FT→Rep|};
```

where the binding of FT is now manifest. Note that CowI is a subtype of AnimalI; this will later allow Cow objects to be regarded as animals.

4.2 Objects

Intuitively, an object with interface I comprises some hidden internal state, some methods (described by I) that can manipulate that state, and some mechanism for hiding the type of the state from outside view. In the simple existential encoding, an existential quantifier is used to achieve this hiding (it can also be done with recursive types), so the type of our point objects is:

```
Point = {|∃Rep:*, state: Rep,
          meth: {|get:Rep→Nat, set:Rep→Nat→Rep, bump:Rep→Rep|}|};
```

More generally, the type of objects with interface I is a record type including a representation type Rep, a method vector field containing a record of type I Rep, and a state field of type Rep. We can capture this structure uniformly by defining a (higher-order) type operator Object that takes I as a parameter:

```
Object = λ[I:*→*]{|∃Rep:*, meth:I Rep, state:Rep|};

Object : (*→*) → *
```

The type Point is now expressed concisely as:

```
Point = Object PointI;
```

A point object—i.e., an element of type Point—can be constructed "from scratch" as follows (we will see how to create points from classes in Section 4.3):

```
PointR = {|x:Nat|};
point = {∃Rep=PointR,
         meth= fix (PointI Rep) λ[self:PointI Rep]
                {get= λ[s:Rep]s.x, set= λ[s:Rep]λ[n:Nat]{x=n},
                 bump= λ[s:Rep] self.set s (plus 1 (self.get s))},
         state= {x=0}} :: Point;
```

PointR is the concrete representation type of the internal state. The method get just returns the x field of state, while set returns a new state with the x field set to its second argument, n. The method bump is defined in terms of the other methods get and set. In order to access other methods, the record of methods is abstracted on a parameter self of type PointI Rep; the fixed-point operator is used to "tie the knot," making self refer to the record itself.

Invocation of the get method of a Point object requires simply extracting the get field of the object's methods and applying it to the state field:

```
x = point.meth.get point.state :: Nat;
```

More generally, we can write

```
get = λ[p:Point] p.meth.get p.state :: Point → Nat;
```

for the function that "sends the get message" to an arbitrary point object p.

To send the set and bump messages to point objects, we need to do a little more work: the implementations of these methods return updated copies of just the internal representation, which must then be repackaged with the original methods into complete objects:

```
bump = λ[p:Point]{∃Rep=p.Rep, meth= p.meth,
                  state= p.meth.bump p.state} :: Point → Point;
```

The construction of a Cow object is similar. The only significant difference is that the record of methods includes a type field FT, which should be given a concrete definition of food for a cow. Furthermore, methods taking arguments of FT can do grass-specific operation (such as enoughGrass) to the argument. Choosing the simple representation

```
CowR = {|hungry:Bool|};
```

for the internal state of cows, we can define an element of the type Object CowI as follows:

```
cow = {∃Rep=CowR,
       meth= fix (CowI Rep) λ[self:CowI Rep]
              {∃FT=Grass,
               eat=λ[s:Rep]λ[f:FT]
                   if enoughGrass f then {hungry=false} else s,
               eatALot=λ[s:Rep]λ[f:FT](self.eat (self.eat s f) f)},
       state= {hungry=true}} :: Object CowI;
```

Like the bump method of point objects, the eatALot method of cows is defined by invoking the eat method via the self parameter.

Since we know FT is equal to Grass (by the definition of CowI), we can feed grass to our cow:

```
feed = λ[c:Object CowI] λ[g:Grass]
         {∃Rep=c.Rep, meth=c.meth, state=c.meth.eatALot c.state g}
    :: Object CowI → Grass → Object CowI;
satisfiedCow = feed cow grass :: Object CowI;
```

4.3 Classes

So far, virtual types have presented no special difficulties: the encodings of points and cows have been essentially identical. For encoding classes, however, the virtual types lead to some extra complications.

A *class* is a data structure providing implementations for a collection of methods and abstracted on a self-parameter. Concretely, a class whose instances are objects with interface I is represented as a function taking self as an argument and returning a record of methods of type I R, where R is the representation type of the state. For example, a class of point objects can be defined as follows:

```
pointClass = λ[self: PointI PointR]
                {get=λ[s:PointR]s.x, set=λ[s:PointR]λ[n:Nat]{x=n},
                 bump=λ[s:PointR]self.set s (plus 1 (self.get s))}
           :: PointI PointR→PointI PointR;
```

To build a point object from the point class, we choose some particular representation (some element of type PointR) and calculate its record of methods by taking the fixed point of the class:

```
point = {∃Rep=PointR, meth=fix (PointI Rep) pointClass, state={x=0}}
     :: Object PointI;
```

The fact that the methods of pointClass are abstracted on self allows us to define new subclasses of pointClass that *inherit* some of its behavior. For example, here is a class of colored point objects:

```
CPointI = λ[Rep:*] {|get:Rep→Nat,  set:Rep→Nat→Rep,
                    bump:Rep→Rep, color:Rep→Color|};
cpointClass = λ[self: CPointI PointR]
               let super = pointClass self in
                {get=super.get, set=super.set, bump=super.bump,
                 color=λ[s:PointR] red}
            :: CPointI PointR → CPointI PointR;

cpoint = {∃Rep=PointR, meth=fix (CPointI Rep) cpointClass,
          state={x=0}} :: Object CPointI;
```

The superclass's method suite super is obtained by application of pointClass to (cpointClass's) self. Note that, for brevity, we choose the same representation

type for both pointClass and cpointClass; it is easy to generalize this so that cpointClass can add new instance variables (such as a color field), but the extra mechanism would make the examples harder to read.

When virtual types are involved, we need to be a little more precise about the typing of classes. Here, for example, is the definition of a generic animalClass. (Again, for brevity we use the same representation type (AnimalR) for both animalClass and cowClass.)

```
AnimalR = {|hungry:Bool|};
animalClass = λ[self:AnimalI AnimalR]
                {∃FT=self.FT, eat=self.eat,
                 eatALot=λ[s:AnimalR]λ[f:FT]self.eat (self.eat s f) f}
            :: Π[self: AnimalI AnimalR]
                 {|∃FT↔self.FT, eat: AnimalR→FT→AnimalR,
                               eatALot: AnimalR→FT→AnimalR|};
```

This definition involves a few subtle points. First, since the type FT and the method eat are virtual, their concrete definitions cannot be provided. Instead of concrete definitions, the corresponding fields of self are used. Second, type of animalClass is not AnimalI AnimalR→AnimalI AnimalR, but a dependent function type (a more refined subtype of AnimalI AnimalR→AnimalI AnimalR). This typing is essential when we derive cowClass from animalClass, as we will see below.

In the definition of cowClass, the FT and eat fields are filled with their concrete definitions and the eatALot method is inherited from animalClass. Since cowClass's self is passed to animalClass, self.eat in method eatALot refer to the eat method of cowClass (not the virtual eat method of animalClass.) Now, since FT is not derived from self, the type of cowClass is just a (non-dependent) function type.

```
cowClass = λ[self:CowI CowR]
            let super = animalClass self in
             {∃FT=Grass,
              eat=λ[s:CowR]λ[f:FT]
                  if enoughGrass f then {hungry=false} else s,
              eatALot=super.eatALot}
          :: CowI CowR → CowI CowR;
```

The dependent function type of animalClass is critical for cowClass to be well-typed: if animalClass had only type AnimalI AnimalR→AnimalI AnimalR, cowClass would be ill-typed since super.eatALot has type CowR→FT→CowR where FT <: Food, which is *not* a subtype of CowR→Grass→CowR. Thanks to the dependent function type of cowClass, the projection super.eatALot has type CowR→self.FT→CowR, which is exactly equal to CowR→Grass→CowR.

Finally, a cow object can be created by instantiating cowClass in the usual way:

```
cow = {∃Rep=CowR, meth=fix (CowI Rep) cowClass,
                  state={hungry=true}} :: Object CowI;
```

5 Generic Programming with Virtual Types

The "overlap" between virtual types and parametric classes as alternative mechanisms for achieving similar kinds of genericity has been remarked by several authors [5, 34, etc.]. To build a generic Bag class, for example, one can proceed in two ways. On one hand, we can make the type of the bag's elements a (virtual) field of the Bag class and obtain concrete instances by *subclassing* the generic Bag class, overriding the member type field with the actual member type. On the other hand, we can make the element type a parameter to the class definition, essentially making the class into a polymorphic function, and obtain concrete instances by *instantiating* this polymorphic function with the actual member type. In this section, we first compare these two styles by means of a fully worked example, then comment on the general case. The overlap between the styles can be viewed, in terms of our encoding, as a corollary of the inter-definability of universal and existential polymorphism in the presence of dependent records.

Generic programming was one of the first applications of virtual types. The typical pattern proceeds in two steps: (1) a generic class with a virtual type is defined, with generic implementations of its operations in terms of the virtual type; (2) this class is then specialized, overriding the virtual type to some concrete instance. For example, suppose we want to program with homogeneous collections (bags) of objects of some type T. We start by building a generic Bag class with a virtual type E (which stands for type of elements) and implementations of the bag methods (put, get, etc.). Since the representation type of state of bags is parameterized by E, the interface of bags takes a type operator Rep of kind *→*, and the type of the state is actually represented as Rep E.

```
BagI = λ[Rep:*→*] {|∃E:*, put:(Rep E)→E→(Rep E), get:(Rep E)→E|};
```

Choosing lists of elements as our internal representation,

```
BagR = λ[E:*] {|elts:List(E)|};
```

we can define a generic bag class as follows:

```
bagClass = λ[self:BagI BagR]
              {∃E=self.E,
               put=λ[s:BagR E]λ[e:E]({elts= cons E e s.elts} ::BagR E),
               get=λ[s:BagR E] car E s.elts}
           :: Π[self:BagI BagR]
              {|∃E↔self.E, put:BagR E→E→BagR E, get:BagR E→E|};
```

The next step is to make a subclass with a concrete definition for the element type. The class natBagClass is defined by giving the concrete value Nat to the virtual type E and by inheriting all methods from bagClass.

```
NatBagI = λ[Rep:*→*]{|∃E↔Nat,
                      put:(Rep E)→E→(Rep E), get:(Rep E)→E|};
natBagClass = λ[self:NatBagI BagR]
                let super = bagClass self in
                 {∃E=Nat, put= super.put, get= super.get}
             :: NatBagI BagR → NatBagI BagR;
```

The interfaces and classes here are fairly similar to the examples we saw in Section 4 (modulo the fact that the representation type here is a type operator); the construction of bag *objects*, however, requires a little explanation.

```
NatBag = {|∃Rep:*→*, ∃meth:NatBagI Rep, state:Rep meth.E|};
natBag = {∃Rep=BagR, meth=fix (NatBagI Rep) natBagClass,
          state= {elts= (nil Nat)}} :: NatBag;
```

The first observation is that the hidden state type is now a type operator. (Intuitively, we "see" that the representation of the object may involve the virtual type field `E`, but that is all we are allowed to know about the representation.) The second is that the order of the `state` field and the `meth` field is essential, since the type of the state depends both on `Rep` and on the `E` component of the `meth`. The code for invoking operations on bag objects is adjusted accordingly:

```
sendget = λ[b:NatBag] b.meth.get b.state :: NatBag→Nat;
sendput = λ[b:NatBag] λ[e:Nat]
            {∃Rep=b.Rep, meth= b.meth,
             state= b.meth.put b.state e} :: NatBag→Nat→NatBag;
```

By contrast, let's look at how bags can be modeled in terms of parametric classes. Instead of the element type being a *member* of the bag class, it will be a *parameter* to the class. Similarly, the interface `BagI` is parameterized by `E`:

```
BagI = λ[E:*] λ[Rep:*] {|put:Rep→E→Rep, get:Rep→E|};
bagClass = λ[E:*] λ[self:BagI E (BagR E)]
            {put= λ[s:BagR E] λ[e:E] {elts= cons E e s.elts},
             get= λ[s:BagR E] car E s.elts}
         :: ∀[E:*]BagI E (BagR E)→BagI E (BagR E);
```

Note that `bagClass` has a polymorphic function type. (Also, note that `Rep` has kind `*` now, not `*→*`, since it is being supplied from the outside and there is no need to apply it to anything in this definition.)

The concrete instance `natBagClass` is now defined by *instantiating* `bagClass` with the type parameter `Nat`.

```
NatBagI = λ[Rep:*]{|put:Rep→Nat→Rep, get:Rep→Nat|};
natBagClass = bagClass Nat;
```

A bag object is defined by instantiating the class in the usual way. (Here there are no subtle dependencies between the type of the `meth` and `state` fields.)

```
NatBag = {|∃Rep:*, meth:NatBagI Rep, state:Rep|};
natBag = {∃Rep=BagR Nat,
          meth=fix (NatBagI Rep) natBagClass,
          state= {elts= (nil Nat)}} :: NatBag;

sendget = λ[b:NatBag] b.meth.get b.state :: NatBag→Nat;
sendput = λ[b:NatBag] λ[e:Nat]
            {∃Rep=b.Rep, meth= b.meth,
             state= b.meth.put b.state e} :: NatBag→Nat→NatBag;
```

These examples illustrate the basic difference between virtual types and parametric classes as mechanisms for generic programming. A parametric class is instantiated by type application, taking the element type directly as an argument. With virtual types, on the other hand, type parameterization is realized by a dependent function whose argument has a type field in it. Since the `get` field depends on `self.E`, it will have type $\texttt{(List Nat)}\to\texttt{Nat}$ when the `E` field of the supplied `self` record has been set to `Nat`.

This correspondence can be viewed as an instance of a more general observation: polymorphic functions can be encoded in terms of dependent functions on dependent records. A polymorphic abstraction $\lambda[X{<:}S]t$ of type $\forall[X{<:}S]T$ can be represented as the dependent function $\lambda[x{:}\{\!|\exists X{<:}S|\!\}](t\{X \mapsto x.X\})$ of type $\Pi[x{:}\{\!|\exists X{<:}S|\!\}](T\{X \mapsto x.X\})$; it takes an argument $\{\!|\exists X{\leftrightarrow}U|\!\}$ where U is some subtype of S and behaves as a term of type $T\{X \mapsto \{\!|\exists X{\leftrightarrow}U|\!\}.X\}$, which is equal to the type $T\{X \mapsto U\}$ of the corresponding polymorphic application $(\lambda[X{<:}S]t)\ U$. The table below summarizes this encoding:

	$\forall$	$\exists + \Pi$
type	$\forall[X{<:}S]\ T$	$\Pi[x{:}\{\!|\exists X{<:}S|\!\}]\ (T\{X \mapsto x.X\})$
abstraction	$\lambda[X{<:}S]\ t$	$\lambda[x{:}\{\!|\exists X{<:}S|\!\}]\ (t\{X \mapsto x.X\})$
application	$t\ T$	$t\ \{\exists X{=}T\}$

6 Comparisons

Virtual types (called *virtual classes* in the original proposal) were first introduced in Beta [24] by Madsen and Møller-Pedersen [23] as a mechanism to achieve genericity in object-oriented languages. Later, Thorup [33] introduced virtual types as an extension for Java. In all of this work, virtual types in classes are in fact *not* actually virtual in our sense: the interface of animal objects, according to their view, would better be modeled by

```
AnimalI=λ[Rep:*]{|∃FT↔Food, eat:Rep→FT→Rep, eatALot:Rep→FT→Rep|};
```

where `FT` is declared equal to `Food`. However, they also allow type fields to be specialized, so that

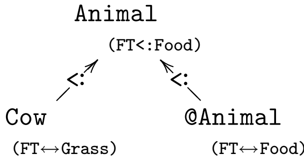
```
CowI = λ[Rep:*]{|∃FT↔Grass, eat:Rep→FT→Rep, eatALot:Rep→FT→Rep|};
```

as before. Finally, they want to regard cows as animals, i.e., `CowI <: AnimalI` and `Object CowI <: Object AnimalI`. Taken together, these properties (specifically, the inclusion `CowI <: AnimalI`) yield a statically unsafe type system: we can take a cow, regard it as an animal, and feed it some meat (which has type `Meat`, a subtype of `Food`, and hence an acceptable argument to an `Animal`'s `eat` method).

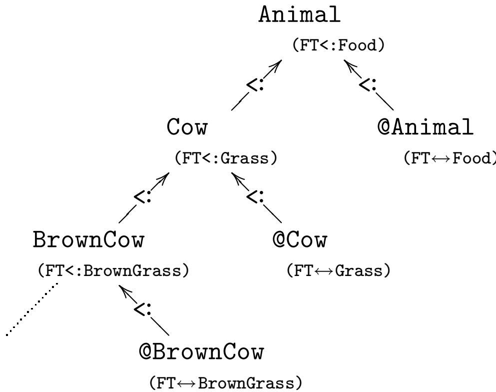
Various approaches have been suggested to remedy this unsoundness. In Beta and in Thorup's proposed Java extension, run-time checks are added to methods like `eat` to make sure that their arguments are actually acceptable. (In [22], it is observed that these checks can be omitted in the case where a type binding is marked `final`.)

Torgersen [35] proposed a statically typesafe variant of virtual types, introducing essentially the same distinction as we have made between virtual type bindings (which may be specialized in subclasses, but which block instantiation of the classes containing them) and final ones (which allow instantiation but block further specialization in subclasses). Our model of objects with virtual types corresponds closely to his proposal.

A possible criticism of Torgersen's idea is that, in general, it may lead to duplication of the class hierarchy. For one thing, if the class `Animal` contains virtual types but no virtual methods (i.e., if `eat` is given a concrete generic implementation), then we may want to instantiate the class `Animal` itself. This requires making an explicit subclass (let's call it `@Animal`) of `Animal` in which `FT` is equal to `Food`.



Also, rather than making `Cow` a leaf of the subclass hierarchy, we may wish to allow further specialization in subclasses. In this case, we should change the constraint on `FT` to `<:Grass`, make `Cow` a virtual class, and introduce another leaf class `@Cow` in which `FT`↔`Grass`.



Fortunately, the `@` variants can be derived mechanically from the other classes, as Torgersen himself pointed out in his original paper. More recently, Bruce, Odersky and Wadler [5] have proposed another statically safe variant of virtual types, which can be viewed as making this idea explicit. (They do not present their proposal in this light, but we find this to be a helpful way of understanding what they did.) In their system, virtual types are always introduced with `<:` constraints (they write "`FT as Food`"); for each class `C`, the "exact" class `@C` is automatically provided. The `new` operator generates instance of exact classes, so that the expression `new Cow ()` yields an object of type `@Cow`, which can be regarded as a `Cow` by forgetting its "exactness," and further regarded as

an `Animal` (but not an `@Animal`) by ordinary subtyping. Note that their type system does not allow `@` types to have non-trivial subtypes, whereas `@Animal` here has many subtypes, which can be obtained by adding extra fields to `@Animal`. Their restriction becomes crucial when binary methods are involved, since an object type with binary methods will be expressed with recursive types where the recursion variable appears in contra-variant positions, which do not have any non-trivial subtypes.

Bruce, Odersky, and Wadler also pointed out that virtual types have an advantage over parametric classes in defining mutually recursive classes such as alternating lists or the Subject/Observer pattern [16]. In the Subject/Observer pattern, a group of objects (called subjects) has a reference to another group of objects (called observers) and reports their own behavior to observers, which will send back messages to subjects according to the reported behavior. Typically, a subject is realized by a class which has a virtual type bound to corresponding observers and vice versa. Then, generic subject (resp., observer) classes are extended to more specific classes, for example, window subject (resp., window observer) class by overriding virtual types with window observer (resp., window subject) and by implementing specific behavior of them. In [5], they used an extension of inner classes of Java to define mutually recursive classes, extensions (window subject/observer) had to be defined simultaneously.

Recently, Bruce and Vanderwaart [7] also used virtual types as a convenient device to define mutually recursive object types "incrementally"—just as extending an interface of Java, object types can be extended by adding specifications of new methods. Since their language can define object types separately from classes, a subject class and its corresponding observer class do not have to be defined simultaneously: virtual types will refer not to class names, but to object types. Rémy and Vouillon [30] showed programming with virtual types can be expressed in terms of parametric classes with mutually recursive types. Since their language has not only separate notion of object types but type reconstruction, programmers do not even need to write object types. As we discussed in Section 5, it is not so surprising that classes involving virtual types can be expressed in terms of parametric classes: an animal class would be just a parametric class which has a `FT` as a type parameter and there is no generic animal object types. However, they did not take into account type abstraction nature of virtual types. As for object types, our dependent record formulation seems to be essential, especially in order for cows to be animals.

7 Conclusions and Future Work

We have presented a straightforward encoding of objects with virtual types in a fairly standard (though quite powerful) type theory. In our model, objects are expressed as dependent records with manifest and/or bounded type fields; classes are modeled as dependent functions. The overlap between parametric classes and virtual types can then be viewed as a consequence of the encodability of universal

polymorphism in terms of existential polymorphism with dependent functions. We are working to extend this encoding in two main directions:

- Imperative variants of the encoding, where methods like `eat` work by side-effecting mutable instance variables.
- Recursive and mutually recursive classes involving virtual types, such as the well-known subject-observer example.

The second of these seems relatively straightforward. The first, somewhat surprisingly (and disappointingly) does not—the technicalities of the underlying type theory required to achieve soundness when imperative features are combined with dependent types become astonishingly subtle.

An obvious question is whether other type-theoretic encodings of simple objects—for example, the standard recursive-records encoding [4]—could be used instead of the existential encoding presented here. Surprisingly, we have *not* been able to extend a naive recursive-records encoding to include virtual types. Intuitively, the problem is that `Animal` in this encoding would be a recursive type whose body is a dependent record type with an `FT` field. But now every unfolding of the recursive type produces a *different* `FT` field, whose (abstract) type is incomparable with all the others.

Another interesting question is whether the type theory in which we are working here is the simplest possible for the task. All of the features described in Section 3—in particular, both dependent records and dependent functions—are used by our encoding, but it is possible that a different encoding could get by with less.

Finally, it would be worthwhile to formalize the translation from a high-level language with virtual types into low-level structures like the ones we have explored here. The interesting point is to see how much of of the type-theoretic complexity of the target language will also show up in the typing rules of the high-level language.

Acknowledgments

This work was supported by Indiana University, the University of Pennsylvania, and the National Science Foundation under grant CCR-9701826, *Principled Foundations for Programming with Objects*. Igarashi is a research fellow of the Japan Society for the Promotion of Science.

Discussions with Kim Bruce, Bob Harper, Didier Rémy, and Philip Wadler deepened our understanding of this material. Comments from the FOOL and ECOOP referees helped us improve the final presentation.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
[2] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, October 1997.
[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, October 1998.
[4] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 1998. To appear in a special issue with papers from *Theoretical Aspects of Computer Software (TACS)*, September, 1997. An earlier version appeared as an invited lecture in the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.
[5] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.
[6] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *Proceedings of* ECOOP '95, LNCS 952, pages 27–51, Aarhus, Denmark, August 1995. Springer-Verlag.
[7] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Fifteenth Confertence on the Mathematical Foundations of Programming Semantics*, April 1999.
[8] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.
[9] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.
[10] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.
[11] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
[12] Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, SIGPLAN Notices volume 33 number 10, pages 201–215, Vancouver, BC, Octover 1998. ACM.
[13] Gang Chen and Giuseppe Longo. Subtyping parametric and dependent types. In Kamareddine et al., editor, *Type Theory and Term Rewriting*, September 1996. Invited lecture.
[14] Adriana B. Compagnoni. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, September 1994. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled "Subtyping in $F^{\omega}_{\wedge}$ is decidable".
[15] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.
[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[17] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. A summary appeared in the Proceedings of the Second Scandinavian Logic Symposium (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

[18] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the Twenty-First ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 123–137, Portland, OR, January 1994.

[19] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[20] Xavier Leroy. Manifest types, modules and separate compilation. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, Portland, OR, January 1994.

[21] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1997.

[22] Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen. Strong typing of object-oriented languages revisited. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) and European Conference on Object-Oriented Programming (ECOOP)*, pages 140–150, Ottawa, ON Canada, October 1990. ACM Press, New York, NY , USA. Published as SIGPLAN Notices, volume 25, number 10.

[23] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, 1989.

[24] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[25] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parametrized types for Java. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 1997.

[26] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[27] Benjamin Pierce and Martin Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

[28] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). A preliminary version appeared in POPL '92.

[29] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[30] Dedier Rémy and Jérôme Vouillon. On the (un)reality of virtual types, November 1998. manuscript.

[31] David Shang. Are cows animals? *Object Currents 1*, 1996. http://www.sigs.com/objectcurrents/.

[32] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Longman, Reading, MA, third edition, 1997.

[33] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471. Springer-Verlag, 1997.

[34] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining the benefits of virtual types and parametrized classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, June 1999.

[35] Mads Torgersen. Vitrual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998.

A Summary of Calculus

This appendix summarizes the syntax, reduction rules, and typing rules of the type system used in this paper. All of the individual features of this calculus have been studied carefully in the literature, and their combination here is believed (but has not been proved!) to be sound. All of the examples in the paper have been checked mechanically by our implementation.

A.1 Syntax

The sets of kinds K, types T, bindings B, terms t, fields β, and contexts Γ are defined by the following grammar:

K ::=	
`*`	proper types
K→K	operator kind

T ::=	
X	type variable
Top	Top type
∀[X<:T]T	polymorphic func. type
Π[x:T]T	dependent func. type
λ[X:K]T	type operator
T T	type op. application
{| $\mathtt{l}_i$/$\mathtt{B}_i{}^{i\in 1\cdots n}$ |}	type of records
t.l	type field projection

B ::=	
X<:T	bounded type binding
X↔T	manifest type binding
x:T	term binding

t ::=	
x	variable
λ[X<:T]t	polymorphic abstraction
λ[x:T]t	abstraction
t T	polymorphic application
t t	application
{$\mathtt{l}_i$/$\beta_i{}^{i\in 1\cdots n}$}	record intro
t.l	record projection
fix T t	fixed-point operator

β ::=	
X=T	type field
x=t	term field

Γ ::=	
•	empty context
Γ, B	extended by binding
Γ, X:K	extended by type binding

A.2 Reduction

$$\texttt{(}\lambda\texttt{[x:T]s) t}\longrightarrow\{\texttt{x}\mapsto\texttt{t}\}\texttt{s} \qquad \texttt{(}\lambda\texttt{[X<:S]t) T}\longrightarrow\{\texttt{X}\mapsto\texttt{T}\}\texttt{t}$$

$$\frac{\texttt{s=\{l}_i/\beta_i{}^{i\in 1\cdots n}\texttt{\}} \qquad \beta_j = \texttt{x=t}}{\texttt{s.l}_j\longrightarrow\{BV(\beta_i)\mapsto\texttt{s.l}_i{}^{i\in 1\cdots j-1}\}\texttt{t}} \qquad \texttt{fix T s}\longrightarrow\texttt{s (fix T s)}$$

A.3 Judgments

$\vdash \Gamma\ \texttt{ok}$	Γ is a well-formed context	$\Gamma \vdash \texttt{S <: T}$	S is a subtype of T
$\Gamma \vdash \texttt{T} : \texttt{K}$	T is a type constructor of kind K	$\Gamma \vdash \texttt{B}_1 \texttt{ <: } \texttt{B}_2$	$\texttt{B}_1$ is a subbinding of $\texttt{B}_2$
$\Gamma \vdash \texttt{t} : \texttt{T}$	t is a term of type T	$\Gamma \vdash \texttt{S} \leftrightarrow \texttt{T}$	S and T are equivalent
$\Gamma \vdash \beta : \texttt{B}$	β is a field of binding B	$\Gamma \vdash \texttt{B}_1 \leftrightarrow \texttt{B}_2$	$\texttt{B}_1$ and $\texttt{B}_2$ are equivalent

Context Well-formedness

$$\vdash \bullet\ \texttt{ok} \qquad \frac{\vdash \Gamma\ \texttt{ok} \qquad \texttt{X} \notin dom(\Gamma)}{\vdash \Gamma, \texttt{X:K ok}}$$

$$\frac{\vdash \Gamma\ \texttt{ok} \qquad \texttt{X} \notin dom(\Gamma) \qquad \Gamma \vdash \texttt{T} : \texttt{K}}{\vdash \Gamma, \texttt{X<:T ok}} \qquad \frac{\vdash \Gamma\ \texttt{ok} \qquad \texttt{X} \notin dom(\Gamma) \qquad \Gamma \vdash \texttt{T} : \texttt{K}}{\vdash \Gamma, \texttt{X}\leftrightarrow\texttt{T ok}}$$

$$\frac{\vdash \Gamma\ \texttt{ok} \qquad \texttt{x} \notin dom(\Gamma) \qquad \Gamma \vdash \texttt{T} : *}{\vdash \Gamma, \texttt{x:T ok}}$$

Kinding Rules

$$\Gamma \vdash \texttt{Top} : * \qquad \frac{\texttt{X:K} \in \Gamma}{\Gamma \vdash \texttt{X} : \texttt{K}}$$

$$\frac{\texttt{X}\leftrightarrow\texttt{T} \in \Gamma \qquad \Gamma \vdash \texttt{T} : \texttt{K}}{\Gamma \vdash \texttt{X} : \texttt{K}} \qquad \frac{\texttt{X<:T} \in \Gamma \qquad \Gamma \vdash \texttt{T} : \texttt{K}}{\Gamma \vdash \texttt{X} : \texttt{K}}$$

$$\frac{\vdash \Gamma, \texttt{X<:S ok} \qquad \Gamma, \texttt{X<:S} \vdash \texttt{T} : *}{\Gamma \vdash \forall\texttt{[X<:S]T} : *} \qquad \frac{\vdash \Gamma, \texttt{x:S ok} \qquad \Gamma, \texttt{x:S} \vdash \texttt{T} : *}{\Gamma \vdash \Pi\texttt{[x:S]T} : *}$$

$$\frac{\vdash \Gamma, \texttt{X:K}_1\ \texttt{ok} \qquad \Gamma, \texttt{X:K}_1 \vdash \texttt{T} : \texttt{K}_2}{\Gamma \vdash \lambda\texttt{[X:K}_1\texttt{]T} : \texttt{K}_1\rightarrow\texttt{K}_2} \qquad \frac{\Gamma \vdash \texttt{T} : \texttt{K}_1\rightarrow\texttt{K}_2 \qquad \Gamma \vdash \texttt{U} : \texttt{K}_1}{\Gamma \vdash \texttt{T U} : \texttt{K}_2}$$

$$\frac{\vdash \Gamma, \texttt{B}_1, \ldots, \texttt{B}_n\ \texttt{ok}}{\Gamma \vdash \{\!|\texttt{l}_i/\texttt{B}_i{}^{i\in 1\cdots n}|\!\} : *} \qquad \frac{\Gamma \vdash \texttt{t} : \{\!|\texttt{l}_i/\texttt{B}_i{}^{i\in 1\cdots n}|\!\} \quad \texttt{B}_j\texttt{=X<:T} \qquad \Gamma, \texttt{B}_1, \ldots, \texttt{B}_{j-1} \vdash \texttt{T} : \texttt{K}}{\Gamma \vdash \texttt{t.l}_j : \texttt{K}}$$

Typing Rules

$$\frac{\texttt{x:T} \in \Gamma}{\Gamma \vdash \texttt{x} : \texttt{T}} \qquad \frac{\vdash \Gamma, \texttt{X<:S ok} \qquad \Gamma, \texttt{X<:S} \vdash \texttt{t} : \texttt{T}}{\Gamma \vdash \lambda\texttt{[X<:S]t} : \forall\texttt{[X<:S]T}}$$

$$\frac{\Gamma \vdash \texttt{t} : \forall\texttt{[X<:T]U} \qquad \Gamma \vdash \texttt{S <: T}}{\Gamma \vdash \texttt{t S} : \{\texttt{X}\mapsto\texttt{S}\}\texttt{U}} \qquad \frac{\vdash \Gamma, \texttt{x:S ok} \qquad \Gamma, \texttt{x:S} \vdash \texttt{t} : \texttt{T}}{\Gamma \vdash \lambda\texttt{[x:S]t} : \Pi\texttt{[x:S]T}}$$

$$\frac{\Gamma \vdash \texttt{t} : \Pi\texttt{[x:S]T} \qquad \Gamma \vdash \texttt{s} : \texttt{S}}{\Gamma \vdash \texttt{t s} : \{\texttt{x} \mapsto \texttt{s}\}\texttt{T}}$$

$$\frac{\Gamma, \texttt{B}_1, \ldots, \texttt{B}_{j-1} \vdash \beta_j : \texttt{B}_j \;^{j\in 1\cdots n} \qquad \Gamma \vdash \{\!|\, \texttt{l}_i/\texttt{B}_i{}^{i\in 1\cdots n} \,|\!\} : *}{\Gamma \vdash \{\texttt{l}_i/\beta_i{}^{i\in 1\cdots n}\} : \{\!|\, \texttt{l}_i/\texttt{B}_i{}^{i\in 1\cdots n} \,|\!\}}$$

$$\frac{\Gamma \vdash \texttt{t} : \{\!|\, \texttt{l}_i/\texttt{B}_i{}^{i\in 1\cdots n} \,|\!\} \qquad \texttt{B}_j\texttt{=x:T}}{\Gamma \vdash \texttt{t.l}_j : \{BV(\texttt{B}_i) \mapsto \texttt{t.l}_i{}^{i\in 1\cdots j-1}\}\texttt{T}}$$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{T}\to\texttt{T}}{\Gamma \vdash \texttt{fix T t} : \texttt{T}}$$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{S} \qquad \Gamma \vdash \texttt{S} <: \texttt{T}}{\Gamma \vdash \texttt{t} : \texttt{T}}$$

Typing Rules for Record Fields

$$\frac{\Gamma \vdash \texttt{T} : \texttt{K}}{\Gamma \vdash (\texttt{X=T}) : (\texttt{X}\leftrightarrow\texttt{T})}$$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{T}}{\Gamma \vdash (\texttt{x=t}) : (\texttt{x:T})}$$

Subtyping Rules

$$\frac{\Gamma \vdash \texttt{S} \leftrightarrow \texttt{T}}{\Gamma \vdash \texttt{S} <: \texttt{T}}$$

$$\frac{\Gamma \vdash \texttt{S} <: \texttt{T} \qquad \Gamma \vdash \texttt{T} <: \texttt{U}}{\Gamma \vdash \texttt{S} <: \texttt{U}}$$

$$\frac{\texttt{X<:T} \in \Gamma}{\Gamma \vdash \texttt{X} <: \texttt{T}}$$

$$\frac{\Gamma \vdash \texttt{T} : *}{\Gamma \vdash \texttt{T} <: \texttt{Top}}$$

$$\frac{\vdash \Gamma, \texttt{X<:S}_1 \;\texttt{ok} \qquad \vdash \Gamma, \texttt{X<:S}_2 \;\texttt{ok} \qquad \Gamma \vdash \texttt{S}_2 <: \texttt{S}_1 \qquad \Gamma, \texttt{X<:S}_2 \vdash \texttt{T}_1 <: \texttt{T}_2}{\Gamma \vdash \forall\texttt{[X<:S}_1\texttt{]T}_1 <: \forall\texttt{[X<:S}_2\texttt{]T}_2}$$

$$\frac{\vdash \Gamma, \texttt{x:S}_1 \;\texttt{ok} \qquad \vdash \Gamma, \texttt{x:S}_2 \;\texttt{ok} \qquad \Gamma \vdash \texttt{S}_2 <: \texttt{S}_1 \qquad \Gamma, \texttt{x:S}_2 \vdash \texttt{T}_1 <: \texttt{T}_2}{\Gamma \vdash \Pi\texttt{[x:S}_1\texttt{]T}_1 <: \Pi\texttt{[x:S}_2\texttt{]T}_2}$$

$$\frac{\vdash \Gamma, \texttt{X:K} \;\texttt{ok} \qquad \Gamma, \texttt{X:K} \vdash \texttt{S} <: \texttt{T}}{\Gamma \vdash \lambda\texttt{[X:K]S} <: \lambda\texttt{[X:K]T}}$$

$$\frac{\Gamma \vdash \texttt{S} <: \texttt{T} \qquad \Gamma \vdash \texttt{S U} : \texttt{K}}{\Gamma \vdash \texttt{S U} <: \texttt{T U}}$$

$$\frac{\vdash \Gamma, \texttt{B}_1, \ldots, \texttt{B}_{n+k} \;\texttt{ok} \qquad \vdash \Gamma, \texttt{B}'_1, \ldots, \texttt{B}'_n \;\texttt{ok} \qquad \Gamma, \texttt{B}_1, \ldots, \texttt{B}_{j-1} \vdash \texttt{B}_j <: \texttt{B}'_j \;^{j\in 1\cdots n}}{\Gamma \vdash \{\!|\, \texttt{l}_i/\texttt{B}_i{}^{i\in 1\cdots n+k} \,|\!\} <: \{\!|\, \texttt{l}_i/\texttt{B}'_i{}^{i\in 1\cdots n} \,|\!\}}$$

$$\frac{\Gamma \vdash \texttt{t} : \{\!|\, \texttt{l}_i/\texttt{B}_i{}^{i\in 1\cdots n} \,|\!\} \qquad \texttt{B}_j\texttt{=X<:T}}{\Gamma \vdash \texttt{t.l}_j <: \{BV(\texttt{B}_i) \mapsto \texttt{t.l}_i{}^{i\in 1\cdots j-1}\}\texttt{T}}$$

Sub-binding Rules

$$\frac{\Gamma \vdash \texttt{S} <: \texttt{T}}{\Gamma \vdash (\texttt{X<:S}) <: (\texttt{X<:T})}$$

$$\frac{\Gamma \vdash \texttt{S} <: \texttt{T}}{\Gamma \vdash (\texttt{X}\leftrightarrow\texttt{S}) <: (\texttt{X<:T})}$$

$$\frac{\Gamma \vdash \texttt{S} \leftrightarrow \texttt{T}}{\Gamma \vdash (\texttt{X}\leftrightarrow\texttt{S}) <: (\texttt{X}\leftrightarrow\texttt{T})}$$

$$\frac{\Gamma \vdash \texttt{S} <: \texttt{T}}{\Gamma \vdash (\texttt{x:S}) <: (\texttt{x:T})}$$

Type Equivalence Rules

$$\frac{\Gamma \vdash \texttt{T} : \texttt{K}}{\Gamma \vdash \texttt{T} \leftrightarrow \texttt{T}}$$

$$\frac{\Gamma \vdash \texttt{S} \leftrightarrow \texttt{T}}{\Gamma \vdash \texttt{T} \leftrightarrow \texttt{S}}$$

$$\frac{\Gamma \vdash \texttt{S} \leftrightarrow \texttt{T} \qquad \Gamma \vdash \texttt{T} \leftrightarrow \texttt{U}}{\Gamma \vdash \texttt{S} \leftrightarrow \texttt{U}}$$

$$\frac{\texttt{X}\leftrightarrow\texttt{T} \in \Gamma}{\Gamma \vdash \texttt{X} \leftrightarrow \texttt{T}}$$

$$\frac{\vdash \Gamma, \mathtt{X}\mathtt{<:}\mathtt{S}_1 \ \ \mathtt{ok} \qquad \Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{S}_2 \qquad \Gamma, \mathtt{X}\mathtt{<:}\mathtt{S}_1 \vdash \mathtt{T}_1 \leftrightarrow \mathtt{T}_2}{\Gamma \vdash \forall[\mathtt{X}\mathtt{<:}\mathtt{S}_1]\mathtt{T}_1 \leftrightarrow \forall[\mathtt{X}\mathtt{<:}\mathtt{S}_2]\mathtt{T}_2}$$

$$\frac{\vdash \Gamma, \mathtt{x}:\mathtt{S}_1 \ \ \mathtt{ok} \qquad \Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{S}_2 \qquad \Gamma, \mathtt{x}:\mathtt{S}_1 \vdash \mathtt{T}_1 \leftrightarrow \mathtt{T}_2}{\Gamma \vdash \Pi[\mathtt{x}:\mathtt{S}_1]\mathtt{T}_1 \leftrightarrow \Pi[\mathtt{x}:\mathtt{S}_2]\mathtt{T}_2}$$

$$\frac{\vdash \Gamma, \mathtt{X}:\mathtt{K} \ \ \mathtt{ok} \qquad \Gamma, \mathtt{X}:\mathtt{K} \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash \lambda[\mathtt{X}:\mathtt{K}]\mathtt{S} \leftrightarrow \lambda[\mathtt{X}:\mathtt{K}]\mathtt{T}}$$

$$\frac{\Gamma \vdash \mathtt{S}_1\ \mathtt{T}_1 : \mathtt{K} \qquad \Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{S}_2 \qquad \Gamma \vdash \mathtt{T}_1 \leftrightarrow \mathtt{T}_2}{\Gamma \vdash \mathtt{S}_1\ \mathtt{T}_1 \leftrightarrow \mathtt{S}_2\ \mathtt{T}_2}$$

$$\frac{\Gamma \vdash (\lambda[\mathtt{X}:\mathtt{K}_1]\mathtt{S})\ \mathtt{T} : \mathtt{K}_2}{\Gamma \vdash (\lambda[\mathtt{X}:\mathtt{K}_1]\mathtt{S})\ \mathtt{T} \leftrightarrow \{\mathtt{X} \mapsto \mathtt{T}\}\mathtt{S}}$$

$$\frac{\Gamma \vdash \mathtt{t} : \{\!|\, \mathtt{l}_i/\mathtt{B}_i{}^{i\in 1\cdots n} \,|\!\} \qquad \mathtt{B}_j\mathtt{=}\mathtt{X}\leftrightarrow\mathtt{T}}{\Gamma \vdash \mathtt{t.l}_j \leftrightarrow \{BV(\mathtt{B}_i) \mapsto \mathtt{t.l}_i{}^{i\in 1\ldots j-1}\}\mathtt{T}}$$

$$\frac{\vdash \Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_n \ \ \mathtt{ok} \qquad \vdash \Gamma, \mathtt{B}'_1, \ldots, \mathtt{B}'_n \ \ \mathtt{ok} \qquad \Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{j-1} \vdash \mathtt{B}_j \leftrightarrow \mathtt{B}'_j{}^{\ j\in 1\cdots n}}{\Gamma \vdash \{\!|\, \mathtt{l}_i/\mathtt{B}_i{}^{i\in 1\cdots n} \,|\!\} \leftrightarrow \{\!|\, \mathtt{l}_i/\mathtt{B}'_i{}^{i\in 1\cdots n} \,|\!\}}$$

Binding Equivalence Rules

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash (\mathtt{X}\mathtt{<:}\mathtt{S}) \leftrightarrow (\mathtt{X}\mathtt{<:}\mathtt{T})}$$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash (\mathtt{X}\leftrightarrow\mathtt{S}) \leftrightarrow (\mathtt{X}\leftrightarrow\mathtt{T})}$$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash (\mathtt{x}:\mathtt{S}) \leftrightarrow (\mathtt{x}:\mathtt{T})}$$

Unifying Genericity

Combining the Benefits of Virtual Types and Parameterized Classes

Kresten Krab Thorup* and Mads Torgersen**

Center for Experimental Computer Science
Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Aarhus C, Denmark
{krab,madst}@daimi.au.dk

Abstract. Generic types in programming languages are most often supported with various forms of parametric polymorphism, i.e. functions on types. Within the framework of object-oriented languages, virtual types present an alternative where specific types are derived from generic ones using inheritance rather than function application. While both mechanisms are statically safe and support basic genericity, they have very different typing properties, each of them providing for the description of useful relationships, which are not expressible with the other. In this paper we present, through the use of examples, a mechanism for describing generic classes: *structural virtual types*. This mechanism is essentially a merger of parameterized classes and virtual types and includes the benefits of both, in particular retaining mutual recursion and covariance of virtual types as well as the function-like nature of parameterized classes.

1 Introduction

The term *genericity* in our use covers the ability to describe generic classes, i.e. classes abstracting over some type information, and derive classes from these by supplying that type information. In the world of functional programming, functions are the natural means to make such abstractions and derivations. This has led to the notion of *parameterized classes*, which are essentially functions that take the missing types as parameters and return a class. Within the object-oriented world, however, inheritance presents an alternative way of deriving classes. Unlike functional derivation, with inheritance the entity derived *from* is also a class. For this to work as a means to support genericity, classes should be able to contain partially abstract types, and subclasses should be able to complete these as part of the inheritance declaration. This kind of genericity is provided by *virtual types* [MMP89,Tho97,Tor98].

Recent discussions have brought attention to the benefits of either approach, most notably [BOW98] which suggests a language containing both mechanisms.

* Currently visiting Professor Akinori Yonezawa at University of Tokyo, Japan.
** Currently visiting Luca Cardelli at Microsoft Research, Cambridge, UK.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 186–204, 1999.
© Springer-Verlag Berlin Heidelberg 1999

As reviewed therein (and elaborated upon in this paper), there are significant problems that are conveniently expressed in either model, but hard to describe with the other. Unlike the work in [BOW98], this paper presents a *unification* of the mechanisms of parameterized classes and virtual types, gaining the benefits of both, as originally suggested in [TT98].

The paper proceeds as follows. Section 2 introduces the notions of parameterized classes and virtual types, and discusses similarities and differences in what can be achieved with these two genericity mechanisms. Section 3 describes how we propose to unify them into *structural virtual types*, providing several examples to show that this mechanism not only retains the benefits of both constituents, but in fact introduces new expressive power through a richer subtype relation. Section 4 describes how an important subset of structural virtual types can be expressed with a minimal syntactic extension of parameterized classes. Section 5 discusses perspectives and related work, and Sect. 6 concludes.

2 Background

This section reviews the mechanisms of parameterized classes and virtual types, examines the extent of their common ability to express basic genericity, and highlights the advantages of each over the other in terms of expressiveness.

Some observations of this section have been previously put forth in [BOW98], which also contains a comparison of virtual types and parameterized classes.

2.1 Parameterized Classes

In his often quoted paper [Mey86] Bertrand Meyer presents parameterized classes as a programming language mechanism in Eiffel which combines the benefits of polymorphism, as in ML, and those of inheritance. Several other programming languages, including CLU [LSAS77], C++ [ES90] and Sather [Omo93] implement similar features. Most recently several such proposals have come forth for Java [OW97,MBL97,AFM97,BOSW98,CS98].

The most basic mechanism, found e.g. in C++, is so-called *unconstrained genericity*, in which a parameterized class is described using simple universal quantification:

```
Queue⟨E⟩ = {
  get(): E { ... }
  put(e: E) { ... }
}
```

With this definition, Queue is a function taking any kind of type as argument. One problem with this approach is that it prohibits local type checking of method bodies, since nothing is known about what type may appear. In context of C++, which provides a wide range of elementary types, unconstrained genericity also has some advantages. In particular, more efficient (although space-consuming) code can be generated since classes are compiled seperately for each instantiation.

Eiffel provides *constrained genericity* in which the abstraction variable E may be constrained to be a subtype (<:) of some other class type:

```
Queue⟨E <: Object⟩ = {
  get(): E { ... }
  put(e: E) { ... }
}
```

This has several advantages over unconstrained genericity, primarily that it allows local typechecking of method bodies, which can now assume that the element type E is indeed a subtype of the specified bound (in this case Object).

The discovery of several serious problems in Eiffel's type system [Coo89] led to the design of an improved mechanism, F-bounded polymorphism [CCH+89], which is the underlying mechanism in modern implementations of parameterized classes, e.g. GJ [BOSW98]. With F-bounded polymorphism, the type variable can appear in its own bound, allowing in particular recursive classes to be written in a type safe way. As an example, consider the following classes:

```
Ordered⟨S <: Object⟩ = { leq(o: S): bool }
Comparator⟨M <: Ordered⟨M⟩⟩ = {
  compare(e: M, f: M): bool { return e.leq(f); }
}
Point = Ordered⟨Point⟩{
  x, y: Int;
  leq(o: Point): bool { return (x ≤ o.x) ∧ (y ≤ o.y); }
}
```

The F-bound comes into play in the class Comparator, in which the parameter M is itself used to express the bound on a parameter. Since Point is a subclass (and thus a subtype) of Ordered⟨Point⟩, it is allowed as an argument to the Comparator parameterized class:

```
cp: Comparator⟨Point⟩;
p1: Point := new Point;
p2: Point := new Point;
if (cp.compare (p1, p2)) { ... }
```

Type checking this example (in particular the body of the compare method) would not be possible using simple type recursion and constrained genericity.

2.2 Virtual Types

Virtual types provide an alternative means for describing generic classes. The notion of virtual types originates from the *virtual patterns* [MMP89] of the BETA programming language [KMMPN83,MMPN93]. Recently [Tho97] provides a suggestion for how to add virtual types to Java. Building on this, [BOW98] suggest combining a limited version of virtual types with parameteric polymorphism to obtain the ability to express mutually recursive classes. BETA's virtual types allow some statically unsafe programs by inserting runtime type checks, but

in [Tor98] Torgersen argues that such programs can be straightforwardly rewritten to become statically safe. Therefore BETA may be strengthened to reject such statically unsafe programs without loosing expressiveness. Throughout this paper we built on this result, assuming a virtual types mechanism which is statically safe. In [IP99], Igarashi and Pierce present the first formal account of these statically safe virtual types.

With virtual types, type variables are introduced as attributes of objects rather than parameters of the class. Using virtual types, the Queue example from the previous section reads:

```
Queue = {
  E <: Object;
  get(): E { ... }
  put(e: E) { ... }
} ,
```

where E is a virtual type *constrained* by Object.

The 'virtual' in virtual types comes from analogy with virtual methods in the sense that they may be refined in context of a subclass. With virtual types this means that the bound may be narrowed to a subtype:

```
SomePointQueue = Queue{ E <: Point; } ,
```

which describes a subclass of Queue that constrains its elements to be some kind of Point. In both Queue and SomePointQueue, E is covariant – its bound is said to be *open*. Alternatively a subclass may *bind* the value of the virtual type E:

```
PointQueue = SomePointQueue{ E = Point; } .
```

Now E is no longer covariant, and cannot be altered in subclasses of PointQueue – its bound is said to be *closed*.

2.3 Elements of Basic Genericity

Both parameterized classes and virtual types are able to express much the same basic elements of genericity in a statically type safe way. The following introduces a common terminology for these elements, and shows how they are expressed with either mechanism.

Generic class A partial description of a class, from which actual classes can be derived by supplying type parameters. Embodied by parameterized classes (with $\langle \ldots \rangle$); and by classes containing open virtual types (with <:).

Derivation The mechanism used to create a class from a generic class: function application with parameterized classes, subclassing with virtual types.

Derived class The class resulting from the derivation.

(Type) Parameter A symbolic name in the generic class for types not completely described, yet used in the body of the generic class. Represented by formal parameters in $\langle \ldots \rangle$; and by virtual types.

Bound A subtype constraint on the parameters of a generic class. In both mechanisms designated by "<:".

Binding An assignment of a specific type to a parameter of a generic class. Embodied by actual parameters enclosed in ⟨...⟩, and by the target of a binding denoted by "=".

Specialization Inheritance from one generic class to another. With parameterized classes, all parameters have to be declared anew in a subclass and explicitly passed to the superclass. Virtual types are inherited.

Narrowing Further constraining the bound of a parameter as part of a specialization.

The Collection example in Fig. 1 contains all these elements in both a parameterized and a virtual version. Collection, Set and HashedSet are all generic classes, while VehicleSet and IntegerSet are derived classes. E is a class parameter. In Collection and Set the bound of E is Object, whereas in HashedSet it is Hashable. In VehicleSet the binding of E is Vehicle, while in IntegerSet it is Integer. Set is a specialization of Collection, and HashedSet of Set. The latter narrows the bound E to Hashable.

```
Collection⟨E <: Object⟩ = {
  insert(elm: E) { ... }
   ...
}

Set⟨E <: Object⟩ = Collection⟨E⟩ {
  insert(elm: E) { ... }
   ...
}

VehicleSet = Set⟨Vehicle⟩;

HashedSet⟨E <: Hashable⟩ = Set⟨E⟩ {
  insert(elm: E) { ... elm.hash() ... }
}

IntegerSet = HashedSet⟨Integer⟩;
```

```
Collection = {
  E <: Object;
  insert(elm: E) { ... }
   ...
}

Set = Collection {  // E inherited
  insert(elm: E) { ... }
   ...
}

VehicleSet = Set {E = Vehicle}

HashedSet = Set {
  E <: Hashable;
  insert(elm: E) { ... elm.hash() ... }
}

IntegerSet = HashedSet { E = Integer; }
```

Fig. 1. Collection classes with simple constrained genericity, expressed with parameterized classes (left column) and virtual types (right column).

2.4 Advantages of F-bounded Polymorphism

Structural Subtyping Parameterized classes in all their variations are based on the notion of function abstraction, meaning that generic classes are considered functions which have class types as arguments and results. In accordance with this functional nature, most implementations of parameterized classes provide

a structural equivalence between two occurrences of the same parameterization, so that e.g. Collection⟨X⟩ and Collection⟨Y⟩ denote the same class when X and Y are the same. Following a similar line of argumentation, a subtype relationship can be derived between instantiations of generic classes and their specializations. Given the Set and Collection class of Fig. 1, we get that

$$\mathsf{Set}\langle\mathsf{X}\rangle <: \mathsf{Collection}\langle\mathsf{X}\rangle\ .$$

Note that structural equivalence is a consequence of reflexivity of the subtyping relation, i.e.

$$\mathsf{Collection}\langle\mathsf{X}\rangle <: \mathsf{Collection}\langle\mathsf{X}\rangle\ .$$

Obviously, these inferred subtyping relationships allow the programmer to write code that is more generic. With virtual types, subclasses of Collection and Set which bind E to Point, would have no relation to each other, whereas with parameterized types, Set⟨P⟩ is substitutable for Collection⟨P⟩.

F-bounded Parameters Also, structural subtyping is a prerequisite for the usefulness of F-bounds in generically expressing operations on self recursive structures. Consider for instance the OrderedSet of Fig. 2, reusing the Ordered and Point classes of Sec. 2.1. In the F-bounded version the compare method is completely generic, because the F-bound ensures that the content type is something that can be compared with itself. In the version with virtual types, such self-comparability can not be expressed, and the most precise bound we can give to e is ordered. This in turn means that a compare method body containing the code a.leq(b) can not be typechecked, so the method has to be abstract in OrderedSet, and must be implemented with an identical body in all concrete subclasses, giving rise to a considerable redundance.

In fact a syntactic equivalent of the F-bound of this example *can* be expressed with virtual types:

```
OrderedSet = Set {
  E <: Ordered { S = E }
  sort() { ... compare(a,b) ... }
  compare(a: E, b: E): bool { return a.leq(b); }
}
```

Only, due to lack of structural subtyping, no subclass of Ordered declared independently of OrderedSet would qualify as a subtype of the bound of E, so even though the above class type checks, it is useless in practice.

It does demonstrate, however, that it is only the lack of structural subtyping of virtual types that prevents them from expressing F-bounds; i.e. if an adequate notion of structural subtyping was added to virtual types (as will be the case later in the paper), F-bounds come for free.

2.5 Advantages of Virtual Types

Covariant Subtyping In the PointQueue example of Sect. 2.2, the virtual type E is *bound* to Point, prohibiting further covariance in subclasses. This means that we can safely both insert and retrieve points via references to PointQueue:

```
Ordered⟨S <: Object⟩ = {
   leq(o: S): bool;
}

Point = Ordered⟨Point⟩ {
   x,y: Int;
   leq(o: Point): bool { ... }
}

OrderedSet⟨E <: Ordered⟨E⟩⟩ = Set⟨E⟩ {
   sort() { ... compare(a,b) ... }
   compare(a: E, b: E): bool {
      return a.leq(b);
   }
}

PointSet = OrderedSet⟨Point⟩;
```

```
Ordered = {
   S <: Object;
   leq(o: S): bool;
}

Point = Ordered {
   S = Point;
   x,y: Int;
   leq(o: S): bool { ... }
}

OrderedSet = Set {
   E <: Ordered;
   sort() { ... compare(a,b) ... }
   abstract compare(a: E, b: E): bool;
}

PointSet = OrderedSet {
   E = Point;
   compare(a: E, b: E): bool {
      return a.leq(b);
   }
}
```

Fig. 2. An example of class OrderedSet and PointSet expressed with F-bounded polymorphism (left column) and virtual types (right column).

```
f(c: PointQueue): Point {
   c.put(new Point);
   return c.get();
}
```

On the other hand, Queue and SomePointQueue are covariant in E. These classes can be used as types for specific queues, from which elements can safely be extracted (using get), and are known to have at least the type of the bound. Inserting elements (using put), however, is not possible: all we know is that E is *some* subtype of Object, but we don't know which:[1]

```
f(c: Queue) {
   o: Object;
   o := c.get();
   c.put(o);      // error!
}
```

Even so, classes like Queue are useful as types, because they range over all queues regardless of element type. This allows for quite generic code to be writ-

[1] In BETA, calling a method like this will in fact be permitted: it causes the compiler to insert a dynamic type check, and emit a warning. This is where BETA differs from the statically safe variant used in this paper.

ten. In Fig. 3, a drawShapes procedure is able to take any subclass of ShapeSet as argument, because it knows that the element type will be some subclass of Shape and thus have a draw method.

```
Shape = {
  abstract draw();
  ...
}

Line = Shape {
  draw() { ... }
  ...
}
```

```
ShapeSet = Set{E <: Shape}

LineSet = ShapeSet{E <: Line}

drawShapes(ss: ShapeSet) {
  forall s: ss.E in ss do s.draw();
}

drawShapes(new LineSet);
```

Fig. 3. Utilizing the covariance of collections (with virtual types).

Also, because virtual types are attributes of objects, they may be remotely accessed and used outside of the object they reside in. Thus, the erroneous code above may be type safely rewritten as:

```
f(c: Queue) {
  o: c.E;
  o := c.get();
  c.put(o);
}
```

Here c.E extracts the virtual type E from the method parameter c. When c is constant (and we assume method parameters to be), c.E will always denotes the *same* type, namely the element type of c, even though we don't know exactly what that type *is*. Thus with the type c.E extracted elements of c may be safely reinserted.

Mutually Recursive Types A special feature of virtual types is that instead of a class name, the bound may be given in the form of an "anonymous" nested class body. We refer to this mechanism as *nested virtual classes*. When narrowing or binding such a virtual type, the original bound can then be subclassed using (in our version) the keyword super in place of the supertype. An example is found in Fig. 4, which implements a framework example by Gail Murphy and David Notkin [MN93,MN96]. They use it to illustrate the inability of current static type systems to support refinement of frameworks, but virtual types seem to solve this problem.

The example illustrates an important feature of virtual types, namely the ability to express mutually recursive classes using nested virtual classes. In this example, a general model-view framework is described by grouping a Model and a View class together within an enclosing class ModelViewFW. To create a specific framework for drawings, the whole ModelViewFW class is subclassed to

```
ModelViewFW = {
  Model <: {
    registerView(v: View) {
      vs.append(v);
    }
    changed() {
      forall v: View in vs do {
        v.update(this);
      }
    }
    vs: ViewList = new ViewList;
  }
  View <: {
    abstract update(m: Model);
  }
  ViewList = List{E = View}
}
```

```
DrawFW = ModelViewFW {
  Model = super{ // Drawing
    getFigure(): Text { ... }
  }
  View = super{
    update(m: Model) {
      ... m.getFigure() ...
    }
  }
}

const draw: DrawFW := new DrawFW;
model: draw.Model := new draw.Model;
view: draw.View := new draw.View;

model.registerView (view);
model.changed ();
```

Fig. 4. Mutually recursive classes with nested virtual classes

a DrawFW which binds the nested virtual Model and View classes to a specific implementation. To make use of this, an instance of the DrawFW framework must be created, as in the global constant draw, so that the nested virtual classes can be accessed and instantiated from the outside as draw.Model and draw.View.

The qualities of this implementation are manyfold. Primarily, the recursive structure among Model and View is maintained in subclasses of ModelViewFW, without the need for rechecking of types. Secondly there is a subtype relation between the Model classes of the different frameworks, so that

$$\text{draw.Model} <: \text{DrawFW.Model} <: \text{ModelViewFW.Model}\ ,$$

where the *class.class* syntax is explicitly allowed to designate e.g. "a Model of any ModelViewFW". (Of course the same holds for the View classes). Thus the generic benefits of covariance described above, also apply to classes taking part in a mutual recursion.

A more fundamental quality of the example is that it type checks at all. The current BETA implementation [Mjø97], as well as our experimental language IDEA [TT99], allows the example in Fig. 4 to be fully compile-time type checked. In most languages, this kind of mutual recursion is not possible, but even for the special case of self recursive classes, which many languages support, this approach (where self recursion amounts to having just one nested virtual class) has advantages. When F-bounded polymorphism or built-in self-types are used for the expression of self recursion it is impossible for one instance of the recursive class in a method body to type safely pass *itself* to another participant of the recursion. However this is precisely what the changed method does in our example.

3 Unifying Parameterized Classes and Virtual Types

From the above discussion it is clear that it would be interesting to develop a type system that combines the advantages of virtual types (covariant subtyping and mutual recursion) with those of F-bounded polymorphism (structural subtyping and F-bounds). There are at least two ways to go about it: enhance parameterized types with the expressiveness of virtual types, or vice versa. Because the syntactic overhead of introducing an equivalent of nested virtual classes to the framework of parameterized classes can be expected to be considerable, we have chosen the latter approach. Thus, in this section we enhance virtual types with a structural subtyping relation, which of course is why we refer to the resulting mechanism as *structural virtual types*. In the next section we then show how the resulting covariant subtyping scheme of this mechanism can be expressed in context of parameterized classes.

In F-bounded polymorphism the structural subtyping of the type parameters is syntactically delimited from the name-based subtyping of the class body by angle brackets ⟨...⟩ in both declaration and application of the generic class. Intuitively, we want to introduce this same distinction to a language with virtual types by syntactically splitting a class body in two parts: we introduce a new kind of block delimited by square brackets [...] containing declarations, narrowings and bindings of virtual types. This precedes the class body proper (in {...} as usual), which contains instance variables and methods. The subtype relationship between class types is established using a structural subtyping rule for the virtual types, and the "usual" name-based subtyping for class bodies, relying only on explicit inheritance. Figure 5 shows some simple collection classes and sample use in this new notation.

```
Collection = [E <: Object] {
  insert(elm: E) { ...  }
   ...
}

Set = Collection { // E inherited
  insert(elm: E) { ...  }
   ...
}

HashedSet = Set [E<:Hashable] {
  insert(elm: E) { ...  e.hash() ... }
}

vc: Collection[E<:Vehicle];
vs: Set[E<:Vehicle];
vs := new HashedSet[E=Vehicle];
vc := vs;
v: Vehicle := vc.get();
```

Fig. 5. Simple collection classes with structural virtual types

The notation Collection[E<:Vehicle] denotes a subclass of collection, which narrows the bound of E to be Vehicle. Similarly, HashedSet[E=Vehicle] denotes a subclass of HashedSet which binds the bound to be exactly Vehicle. Even though these two classes are not subclasses of one another, we can infer them to be subtypes. With structural virtual types, class types are always in a subtype

relationship if they are subclasses, but if the ‘parameters’ match correctly they can be inferred to be in a subtype relationship even if they are not subclasses.

3.1 Name-Based Versus Structural Subtyping

In functional settings with structural typing, two record types are the same if their fields have the same names and types. Thus, types can be declared completely independently and still be in a subtyping relationship. In an object-oriented setting, where-clauses as in CLU [LSAS77,MBL97] and matching in $\mathcal{LOOM}$ [Bru97] use variations of structural subtyping.

As an alternative to this, many object-oriented programming languages use name-based typing (as in Java, C++ or BETA). With name-based typing, two class-types are only in a subtype relation if one is explicitly a subclass of the other. As a consequence of this, two attributes of different classes are “the same” only if they are inherited from the same declaration. Thus, with name-based typing, an attribute can be seen as having an implicit *attribute identity* which is composed of its name and the location of its original declaration.

The difference between these approaches is highlighted in the ‘cowboy’ example due to Boris Magnusson [Mag91]. Consider two unrelated classes GraphicalObject with methods draw and move, and Cowboy with methods draw, move and shoot. In a setting with structural subtyping, Cowboy would unintendently be a subtype of GraphicalObject, whereas with name-based subtyping, the methods would not match up – i.e., have the same attribute identity – because, even though they have similar names, they originate from distinct declarations.

In a language such as BETA, this useful notion of attribute identity holds for any kind of virtual attribute (virtual method or virtual type) which may be overwritten in subclasses. The downside of the name-based subtyping rule is that it only allows a class to be a subtype of another if it is also explicitly a subclass.

3.2 Structural Virtual Types

On the borderline between structural and name-based subtyping we have found that there is a very useful intermediate approach, which we will use for structural virtual types, using attribute identity as explained above to determine whether two attributes are the *same* and a structural subtyping approach on top of this to determine whether two classes are in a subtype relation. As a special case one class will always be a subtype of another if the former is declared to be a subclass of the latter. However, it is sufficient if the two have a common superclass two which the latter class adds nothing but new bounds and bindings for existing virtual types, all of which are matched by equivalent or stronger bounds and bindings in the former class. As an example, the two subclasses of Set,

$$\mathsf{Set[E <: Vehicle]} \quad \text{and} \quad \mathsf{Set[E <: Car]}\ ,$$

are in a subtype relation even though one is not a subclass of the other because

- The E parameters of both are "the same", i.e. have the same attribute identity, by both having the name E and being inherited from the same declaration – in the Collection class.
- The bound on E in the latter parameterization is a "sub-bound" of the one in the former (being a subtype of Car implies being a subtype of Vehicle by transitivity of subtyping).

The notion of *sub-bound* introduced here is based on what is statically safe to say about the relation of two different bounds on a given parameter. This gives rise to three dimensions of subtyping:

Subclassing As with regular constrained genericity, if one generic class is a subclass of another, as e.g. Set of Collection, then a subtype relationship exists between identical parameterizations of the two; i.e.

$$\text{Set[E <: Vehicle] <: Collection[E <: Vehicle]} ,$$

and

$$\text{Set[E = Vehicle] <: Collection[E = Vehicle]} .$$

Covariance As in the narrowing of virtual types, two parameterizations of the same generic class with different open bounds are in a subtype relation if the bounds are; i.e.

$$\text{Set[E <: Car] <: Set[E <: Vehicle]} .$$

Binding As in the binding of a virtual type, a generic class with a binding to a given class is a subtype of the same generic class bounded by the same class; i.e.

$$\text{Set[E = Vehicle] <: Set[E <: Vehicle]} .$$

The combination of these three dimensions by transitivity of subtyping yields the subtype graph depicted in Fig. 6. This is strictly more expressive than a language with only F-bounded polymorphism (such as GJ) since such languages cannot represent the types with open bounds; and also more expressive than a language with virtual types (such as BETA) in which only subclasses yield subtypes. As can be seen the figure is an *almost* complete cube. The two "missing" edges are not a shortage of the model, but reflect the very reason for having both bounds and bindings: The ability to choose between covariant and nonvariant subtyping.

The tiny example below uses all three dimensions at the same time: a procedure meet takes *any* kind of collection containing *some* kind of Vehicles and orders them all to drive to the same place. In the example the procedure is actually passed a Set containing *any* kind of Car:

```
meet(c: Collection[E<:Vehicle], p: Place) {
  forall v: Vehicle in c { v.driveTo(p) }
}

carset: Set[E=Car];
...
meet(carset, Aarhus)
```

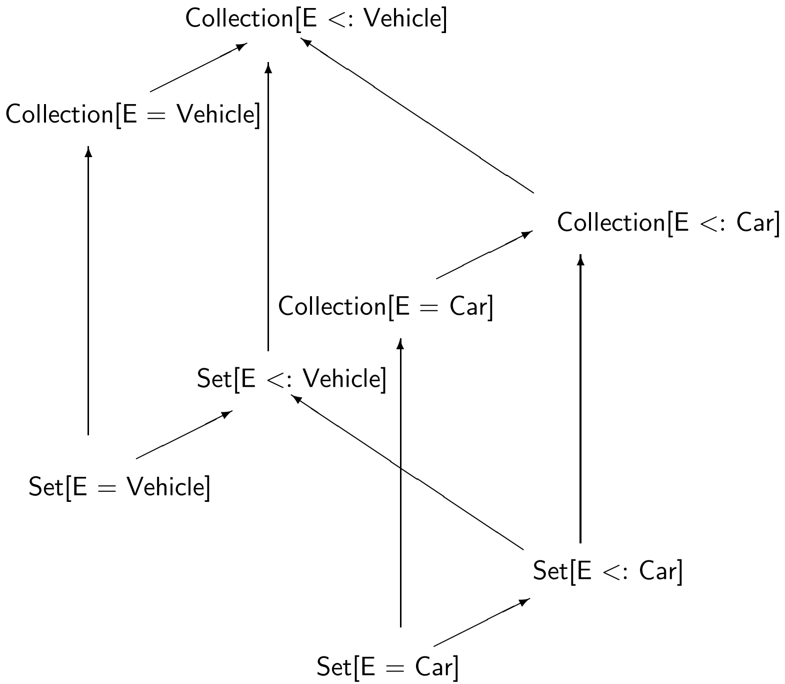


Fig. 6. The three dimensions of structural subtyping

Expressing F-bounds Structural virtual types support F-bounds, thereby subsuming languages with parameterized classes based on F-bounded polymorphism. To see this, consider the version of the previous OrderedSet example shown in Fig. 7. In particular, the class type OrderedSet[E=Point] is valid, i.e.,

$$\mathsf{E} = \mathsf{Point} \;\Rightarrow\; \mathsf{E} <: \mathsf{Ordered[S{=}E]}\ ,$$

since Point is declared as a subclass of Ordered[S=Point].

Expressing Virtual Classes Finally it is essential to notice that the ability to express mutual recursive types, as well as remote access to type attributes is not hampered by the introduction of structural subtyping of virtual types. The ModelView framework example of Sect. 2.5 utilizing both these mechanisms, can easily be rewritten in the new syntax by just replacing a few curly brackets with square brackets – see Fig. 8.

Thus, structural virtual types provide, in one common mechanism, the full expressiveness of both F-bounded polymorphism and virtual types.

```
Ordered = [S <: Object] {
  leq(o: S): bool;
}
OrderedSet = Set [E <: Ordered[S=E]] {
  sort() { ...  }
}

ps: OrderedSet[E = Point];
p1: Point := new Point;
```

```
Point = Ordered[S = Point] {
  x, y: Integer;
  leq (o: S): bool {
    return  (x ≤ o.x) ∧ (y ≤ o.y);
  }
}

ps.insert(p1);
p1 = ps.get();
```

Fig. 7. Utilizing F-bounds with structural virtual types

```
ModelViewFW = [
  Model <: {
    registerView(v: View) {
      vs.append(v);
    }
    changed() {
      forall v: View in vs do {
        v.update(this);
      }
    }
    vs: List[E=View] = new List[E=View];
  }
  View <: {
    abstract  update(m: Model);
  }
]
```

```
DrawFW = ModelViewFW [
  Model = super{ // Drawing
    getFigure(): Text { ... }
  }
  View = super{
    update(m: Model) {
      ... m.getFigure() ...
    }
  }
]

const draw: DrawFW := new DrawFW;
model: draw.Model := new draw.Model;
view: draw.View := new draw.View;

model.registerView (view);
model.changed ();
```

Fig. 8. Mutually recursive classes with nested virtual classes

3.3 Type Checking

To a large extent, the typing of structural virtual types is similar to the typing of virtual types in Beta, for which there exists a well-tested stable implementation [Mjø97]. To try things out, we have implemented a typechecker for a language, Idea [TT99], with syntax similar to that presented in this paper, embodying structural virtual types. All examples in this paper have been type checked by this implementation. In only two significant ways does Idea's typecheck differ from that of Beta:

Static type safety. The typechecker issues an error in places where Beta inserts a runtime check.

Structural subtyping. In Beta, two classes are in a subtyping relation if and only if one is a subclass of the other. This simple check is replaced my a more elaborate rule in Idea, accounting for the degree of structural subtyping introduced.

While the former is a trivial change, the latter requires more attention. Basically, it amounts to checking along the three dimensions of subtyping described earlier in this section, but is complicated by having to deal at the same time with arbitrary nesting, and remote access of types both via constant references (e.g. c.T) and other types (e.g. $T_1.T_2$). The precise description of the Idea type system and its implementation will be the subject of a future paper.

4 Covariance in Parameterized Classes

The above work inspired us to a simple way of adding covariance (but not the other advantages of virtual types) to a language with F-bounded polymorphism. In the above approach a class with an open virtual type plays a dual role as

- a generic class from which other classes can be derived
- a covariant supertype for other classes

Parameterized classes play the first role, but there is nothing to play the second, i.e., there is no way to express the type Collection[E <: Vehicle] from the meet example in the previous section.

From a syntactical point of view, however, it is very easy to add one: We simply introduce a new mode of passing an actual parameter to a parameterized class by prepending it with a "+" sign; e.g. Collection⟨+Vehicle⟩. This produces a type which is covariant in E bounded by Vehicle and hence has subtypes in all the three dimensions described above, namely Set⟨+Vehicle⟩ (subclassing), Collection⟨+Car⟩ (covariance) and Collection⟨Vehicle⟩ (binding). The meet example can then be written as:

```
meet(c: Collection⟨+Vehicle⟩, p: Place) {
  forall v: Vehicle in c { v.driveTo(p) }
}
```

```
carset: Set⟨Car⟩;
...
meet(carset, Aarhus)
```

At first glance, this is strikingly close to what is possible in NextGen [CS98], a proposal for adding parameterized classes to Java. With NextGen, declarations of parameterized classes can be annotated with a "+" sign on the *declaration* of parameters, as in:

$\mathsf{Collection}\langle \mathsf{E}^+ <: \mathsf{Object}\rangle = \{ \ldots \}$,

which has the effect that *all* instantiations of such a parameterized class are covariant in the parameter E. In practice, the result of this is that instantiations of such a class can only use the parameter as an output type. In our proposal, this limitation does not apply, since the choice between covariance and novariance is deferred until the actual instantiation of a parameterized class.

Yet, as already noted this approach is less expressive than structural virtual types (and apparently also than what can be expressed in Transframe; see Sect. 5). Its main attraction is that, syntactically, it would constitute a very minor extension to existing implementations of parameteric polymorphism. While it would probably require some work on the typechecker to add it to GJ or a similar language, it is free in the sense that it preserves both syntax and semantics of the original language.

5 Discussion and Related Work

In recent years, there has been quite a lot of research within the field of generic types for object-oriented programming, as already noted throughout the paper. In particular, the activities surrounding the "Java Genericity" discussion forum [ABB+98] has spawned a lot of research on generic classes in relation to Java. Yet, only one of the resulting papers, namely [BOW98], has aimed at combining the advantages of parametric polymorphism and virtual types. This work suggests extending Java with *both* F-bounded polymorphism *and* a downscaled variation of virtual types, essentially only including nested virtual classes. The primary aim is tto embrace the ability of virtual types to express mutually recursive classes, and their mechanism is seen as an extension of the MyType of $\mathcal{LOOM}$ [Bru97] which is used for self recursive classes. In analogy with MyType, the proposed virtual types are always covariant; rather than allowing them to be bound (preventing further covariance), the rather more restrictive mechanism of exact types is used to obtain type safe use of the virtual classes. Also, no equivalent of the subtype relation of Fig. 6 is present in their proposal, because the structural subtyping of parameterized classes and the covariance of virtual types are not actually merged, but only coexist in the same language.

The Transframe language [Sha96b], designed by David Shang, is somewhat complimentary to this work. It incorporates a lot of BETA features in the structural subtyping framework of parameterized classes, including the distinction between bounded and bound parameters. Also, like in BETA, parameterized

classes are not distinguished from ordinary classes [Sha96a]. On the other hand, Transframe does not have an equivalent of the nested virtual classes which are the core of the [BOW98] proposal. Unfortunately, research on Transframe has, to our knowledge, never been published at the major scientific conferences and journals of object-orientation, and no implementation is available. It has been very informally introduced in Shang's monthly column in the online journal *Object Currents*. Compared to structural virtual types, it seems from the sparse material that the subtyping rule of Transframe does not capture the subtype relationship between similar parameterizations of subclasses, i.e., Transframe does not infer that Set[E <: Point] is a subtype of Collection[E <: Point]. From recent personal communication with David Shang we understand, however, that this is now included, so that the subtype relation is like the one depicted in Fig. 6.

In [IP99] Pierce and Igarashi present an extension of $F^{\omega}_{<:}$ with dependent functions and dependent records with type fields having either *bounded* or *manifest type bindings*. The idea is to present a framework with a formal semantics and type system into which virtual types can be encoded. The type system is structural, and thus, it seems, would allow the encoding of the subtype relation of structural virtual types. On the other hand, mutually recursive classes are not captured, basically because a type field of a dependent record cannot "see" following fields but only previous ones. Still the paper provides a very promising approach to a formal treatment of virtual types, something which has been lacking.

6 Conclusion

The main result of this paper is a statically type safe language construct, *structural virtual types*, which unifies the mechanisms of virtual types (as in Beta) and F-bounded polymorphism (as in recent proposals for adding parameterized classes to Java). As the name suggests, structural virtual types are the result of augmenting virtual types with a structural subtype relation as found in parametric polymorphism. Through examples, the construct is shown not only to retain the benefits of both its constituent mechanisms, but also to introduce new flexibility. All examples in the paper have been succesfully checked with an experimental automatic type checker, but a precise description of the typing rules remains to be given.

In addition, the paper proposes a minimal extension to existing languages with parametric polymorphism, which allows them to capture the covariance benefits of structural virtual types.

Acknowledgements The authors would like to thank the members of the "Java Genericity" mailing list [ABB+98] for many inspiring discussions on the subject of parameterized classes versus virtual types. A special thanks goes to Peter Ørbæk for many fruitful discussions, and for comments on drafts of this paper.

References

ABB+98. Ole Agesen, Gilad Bracha, Kim Bruce, Luca Cardelli, Corky Cartwright, Erik Ernst, Kathleen Fisher, Martin Odersky, Bill Joy, Benjamin Pierce, John Rose, Guy Steele, Jr., David Stoutamire, Kresten Krab Thorup, Mads Torgersen, David Ungar, Philip Wadler, et al. Personal communication. The "Java Genericity" mailing list, 1997-1998.

AFM97. Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java programming language. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*. ACM, October 1997.

BOSW98. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In Chambers [Cha98].

BOW98. Kim Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.

Bru97. Kim Bruce. Subtyping is not a good match for object-oriented programming languages. In Mehmet Akşit and Satoshi Matsouka, editors, *European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS, Jyväskylä, Finland, 1997. AiTO, Springer Verlag.

CCH+89. Peter Canning, William Cook, Walt Hill, Walter Olthoff, and John Mitchell. F-bounded qualification for object-oriented programming. In *ACM Conference on Functional Programming and Computer Architecture*. ACM Press, 1989.

Cha98. Craig Chambers, editor. *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*, Vancouver, BC, October 1998. SIGPLAN, ACM Press.

Coo89. William Cook. A proposal for making Eiffel type-safe. In Stephen Cook, editor, *European Conference on Object-Oriented Programming (ECOOP)*, pages 57–70, Nottingham, July 1989. AiTO, Nottingham University Press.

CS98. Robert Cartwright and Guy L. Steele. Compatible genericity with runtime-types for the Java programming language. In Chambers [Cha98].

ES90. Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

IP99. Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In Rachid Guerraoui, editor, *European Conference on Object-Oriented Programming (ECOOP)*, LNCS, Lisbon, Portugal, 1999. AiTO, Springer Verlag.

Jon97. Niel D. Jones, editor. *Conf. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Paris, January 1997. ACM Press.

KMMPN83. Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Abstraction mechanisms in the Beta programming language. In *Conf. Proceedings of the 10th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Austin, TX, 1983.

LSAS77. Barbara Liskov, Alan Snyder, Russel Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8), August 1977.

Mag91. Boris Magnusson. Code reuse considered harmful. *JOOP – Journal of Object-Oriented Programming*, 4(3), November 1991.

MBL97. Andrew Myers, Joseph Bank, and Barbara Liskov. Parameterized types for Java. In Jones [Jon97].

Mey86. Bertrand Meyer. Genericity versus inheritance. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 391–405, 1986.

Mjø97. Mjølner Informatics, ApS, Aarhus. *The Mjølner System BETA Compiler Reference Manual*, 1997. MIA 90-02(1.6), www.mjolner.dk.

MMP89. Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*. SIGPLAN, ACM Press, 1989.

MMPN93. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

MN93. Gail C. Murphy and David Notkin. The interaction between static typing and frameworks. Technical Report TR-93-09-02, University of Washington, 1993.

MN96. Gail C. Murphy and David Notkin. On the use of static typing to support operations on frameworks. *Object Oriented Systems*, 3(4):197–213, December 1996.

Omo93. Steven Omohundro. The Sather programming language. *Dr. Dobb's Journal*, 18(11), October 1993.

OW97. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In Jones [Jon97].

Sha96a. David Shang. Subtypes and convertible types. *Object Currents*, 1(6), June 1996.

Sha96b. David Shang. *Transframe: The Annotated Reference*. Software Systems Research Laboratory, Motorola, Inc., Shamburg, IL, November 1996. www.transframe.com.

Tho97. Kresten Krab Thorup. Genericity in Java with virtual types. In Mehmet Akşit and Satoshi Matsouka, editors, *European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS, pages 444–471, Jyväskylä, Finland, 1997. AiTO, Springer Verlag.

Tor98. Mads Torgersen. Virtual types are statically safe. In Kim Bruce, editor, *5th Workshop on Foundations of Object-Oriented Languages*, San Diego, CA, January 1998.

TT98. Kresten Krab Thorup and Mads Torgersen. Structured virtual types. Informal session on types for Java, 5th Workshop on Foundations of Object-Oriented Languages, January 1998.

TT99. Kresten Krab Thorup and Mads Torgersen. The IDEA programming language. Technical report, Department of Computer Science, University of Aarhus, 1999. (To appear).

An Object-Oriented Effects System*

Aaron Greenhouse[1] and John Boyland[2]

[1] Carnegie Mellon University, Pittsburgh, PA 15213, aarong@cs.cmu.edu
[2] University of Wisconsin–Milwaukee, Milwaukee, WI 53201, boyland@cs.uwm.edu

Abstract. An effects systems describes how state may be accessed during the execution of some program component. This information is used to assist reasoning about a program, such as determining whether data dependencies may exist between two computations. We define an effects system for Java that preserves the abstraction facilities that make object-oriented programming languages attractive. Specifically, a subclass may extend abstract regions of mutable state inherited from the superclass. The effects system also permits an object's state to contain the state of wholly-owned subsidiary objects. In this paper, we describe a set of annotations for declaring permitted effects in method headers, and show how the actual effects in a method body can be checked against the permitted effects.

1 Introduction

The *effects* of a computation include the reading and writing of mutable state. An effects system is an adjunct to a type system and includes the ability to infer the effects of a computation, to declare the permitted effects of a computation, and to check that the inferred effects are within the set of permitted effects.

The effects system we describe here is motivated by our desire to perform semantics-preserving program manipulations on Java source code. Many of the transformations we wish to implement change the order in which computations are executed. Assuming no other computations intervene and that each computation is single-entry single-exit, it is sufficient to require that the effects of the two computations do not *interfere*: one computation does not write state that is read or written by the other. Therefore our system only tracks reads and writes of mutable state, although other effects can be interesting (e.g., the allocation effect of FX [6]). Because Java supports separate compilation and dynamic loading of code, one of the goals of the project is to carry out transformations on

* This material is based upon work supported by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0241. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government. The work of John Boyland is supported by an equipment grant from Sun Microsystems, Inc.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 205–229, 1999.
© Springer-Verlag Berlin Heidelberg 1999

incomplete programs. In earlier work [4], we proposed a system of program annotations including ones to declare the permitted effects of missing code. The effects system described in this paper uses and expands upon these annotations.

Declaring the (permitted) effects of a method necessarily constrains the implementation of the method and any method that overrides it, but we do not want to "break abstraction," by revealing the implementations details of the program's classes. One of the requirements for a useful object-oriented effects system is that it still provide the ability to hide names of private fields. It should use abstract names that map to multiple mutable locations. This paper introduces a concept of a named "region" in an object. The regions of an object provide a hierarchical covering of the notional state of the object. This idea is extended to objects which have the sole reference to other ("unique") objects; the state of such owned objects is treated as extending the state of the owning object. As a result, a complex of numerous objects, such as a hash table, can be handled as a single notional object comprising a few regions of mutable state.

Our effects system uses declared annotations on fields and methods in Java. These annotations are non-executable, and can be ignored when implementing the program, although an optimizing compiler may find the information useful. The exact representation of the annotations is not important, but for the purposes of this paper, they are shown using an extension to Java syntax. The syntax of the effects annotations is given in Section 2.

A method body can be checked against its annotation using the annotations on the methods it calls. Similar to type-checking in the context of separate compilation, if all method bodies are checked at some point, the program as a whole obeys its annotations.

In the following section of the paper, we describe the basic concept behind the effects system: an abstract region of mutable state within an object. We also describe the technique for inferring the effects of a computation and checking inferred effects against declared effects. In the following section, we show how the regions of unique objects can be mapped into their owners. The precise description of the system is given in the appendix.

2 Regions of Objects

A *region* is an encapsulation of mutable state. The read and write effects of a method are reported with respect to the regions visible to the caller. In this section, we describe the general properties of a region, and how regions are specified by the programmer. Then we describe how methods are annotated, how the effects are computed, and how the annotation on a method can be checked for correctness.

2.1 Properties of Regions

The regions of a program are a hierarchy: at the root of the hierarchy there is a single region `All` that includes the complete mutable state of a program; at

the leaves of the hierarchy are all the (mutable) fields which again comprise the entire mutable state of the program.[1] Thus we see that each field is itself a region without child regions. The hierarchy permits us to describe the state accessed by a computation with varying precision: most precise is listing the individual fields accessed, least precise is `All`.

Java fields are declared in classes and are either `static` or not. In the latter case (instance fields), the declaration actually implies multiple fields, one per object of the class in which the field was declared. Our generalization of fields, regions, are similarly declared in classes in two ways: *static* declarations add one new region; *instance* declarations add a set of disjoint regions, one for each object of the class in which the region is declared. We call the non-field regions *abstract regions*.[2]

Each region is declared to be inside a parent region, thus creating the hierarchy. The parent region specified in a static region declaration must not refer to an instance region; such a static region would in effect have multiple parents. The root region `All` is declared static; one of its child regions is an instance region `Instance` declared in the root class `Object`.

Instance regions are inherited by subclasses, and a subclass may declare its own regions, possibly within an inherited (non-field) region. Being able to extend existing regions is critical to being able to specify the effects of a method and later being able to meaningfully override the method (see Example 2).

Arrays are treated as instances of a class `Array`, which has the instance region `[]` (pronounced "element") as a subregion of `Instance`.[3] When a subscripted element of an array is accessed, it is treated as an access of region `[]`.

The same accessibility modifiers that apply to field declarations (in Java, `public`, `protected`, default (package) access, or `private`) also apply to region declarations. The root region `All`, and its descendants `Instance` and `[]`, are `public`.

2.2 Specifying Regions

New syntax is used to declare abstract regions in class definitions. The syntax `region` *region* declares abstract instance regions. Adding the keyword `static` will declare an abstract static region. The parent of a region may be specified in the declaration by appending the annotation "`in` *parent*" to the field or abstract region declaration. Region declarations without an explicit parent are assigned to `All` or `Instance` respectively if they are static or not.

[1] External state (such as file state) is beyond the scope of this paper. Suffice it that special regions under `All` can be used to model this external state.

[2] We use the term "abstract" to emphasize the implementation hiding characteristics of these regions, not in the usual sense of Java "abstract methods" which impose implementation requirements on subclasses.

[3] More precision could be obtained by distinguishing the different types of arrays, especially arrays of primitive types.

```
class Point {
  public region Position;
  private int x in Position;
  private int y in Position;
  public scale(int sc)
    reads nothing writes Position
  {
    x *= sc;
    y *= sc;
  }
}

class ColorPoint extends Point {
  public region Appearance;
  private int color in Appearance;
}
```
(a)

```
class Point3D extends Point {
  private int z in Position;
  public void scale(int sc)
    reads nothing writes Position
  {
    super.scale(sc);
    z *= sc;
  }
}
```
(b)

Fig. 1. The definitions of (a) classes `Point` and `ColorPoint`, and (b) class `Point3D`

The Java field selection syntax (*object*.*field* for instance fields and *Class*.*field* for static fields, as well as the provision for omitting `this` or the current class) is extended to all regions.

Example 1. Consider the class `Point` in Figure 1a. It declares the fields `x` and `y`, and the abstract region `Position` as their parent region. An instance *o* of class `Point` has four instance regions: *o*`.Instance`, which contains *o*`.Position`, which in turn contains *o*`.x` and *o*`.y`. The class `Point` has a method `scale`. This method is correctly annotated with `writes Position` (short for `writes this.Position`). The annotation `writes this.x, this.y` should not be used because the fields are private but the method is public. The annotation `writes Instance` would be legal, but less precise.

The class `ColorPoint` (also in Figure 1a) inherits from `Point`, declaring field `color` in a new abstract region `Appearance`, implicitly a subregion of `Instance`. If `ColorPoint` overrode `scale(int)`, the overriding method could not access `color` because `color` is not in `Position`, which is reasonable because color is not a geometric property. Given the less precise annotation on `scale(int)` mentioned above, the limitation would be lifted, because `color` is indeed in `Instance`.

We can create a second subclass of `Point`, `Point3D`, that adds the new field `z` to `Position` allowing it to be changed in the overriding of `scale(int)` as shown in Figure 1b.

2.3 Regions and Effects

As stated earlier, effects on objects are reported in terms of the regions that are affected. We distinguish between two kinds of effects: *read effects*, those that

$$
\begin{array}{rl}
\textit{annotation} \rightarrow & \texttt{reads}\ \textit{targets}\ \texttt{writes}\ \textit{targets} \\
\textit{targets} \rightarrow & \texttt{nothing} \mid \textit{targetSeq} \\
\textit{targetSeq} \rightarrow & \textit{target} \mid \textit{target},\ \textit{targetSeq} \\
\textit{target} \rightarrow & \ \ \textit{var.region} \qquad\qquad \text{region of a specific instance} \\
 & \mid \texttt{any}(\textit{class}).\textit{region}\ \text{region of any instance of class } \textit{class} \\
 & \mid \textit{class.region} \qquad\quad \text{(static) region of a class}
\end{array}
$$

var is restricted to being one of the method parameters or the receiver.

Fig. 2. The syntax of method annotations used in this paper.

may read the contents of a region; and *write effects*, those that *may* change *or read* the contents of a region. Making writing include reading is useful when overriding methods, as demonstrated in Examples 1 and 2. It also closely follows how an optional write would be modeled using static single-assignment form (SSA): as a merge of a read and a write.

Methods are labeled with their permitted effects using a notation similar to the Java `throws` clause. Figure 2 gives the syntax of method effect annotations. Again, analogous to `throws` clauses, the declared effects for a method describe everything that the method *or any of its overriding implementations* might affect. It is an error, therefore, for an overriding method to have more effects than declared for the method it overrides. The soundness of static analysis of the effects system would otherwise be compromised.

Example 2. This example shows the importance of abstract regions and the usefulness of having "write" include reading. Consider the class `Var` shown below, that encapsulates the reading and writing of a value. First we will show a set of annotations that do not permit a useful subclass, even assuming it were legal to expose names of private fields (regions) in annotations:

```
class Var {
  private int val;
  public void set( int x ) reads nothing writes this.val { val = x; }
  public int get() reads val writes nothing { return val; }
}
```

Suppose we now extended the class to remember its previous value. The annotations below are illegal because the declared effects of `UndoableVar.set()` are greater than the declared effects of `Var.set()`.

```
class UndoableVar extends Var {
  private int saved;
  public void set( int x ) reads nothing writes this.val, this.saved {
    saved = get(); super.set( x );
  }
  public void undo() reads this.saved writes this.val {
    super.set(saved);
  }
}
```

By instead placing the fields `val` and `saved` inside of a new abstract region `Value`, we are able to produce legal effect declarations:

```
class Var {
  public region Value;
  private int val in Value;
  public void set( int x ) reads nothing writes this.Value { val = x; }
  public int get() reads this.Value writes nothing { return val; }
}

class UndoableVar extends Var {
  private int saved in Value;
  public void set( int x ) reads nothing writes this.Value {
    saved = get(); super.set( x );
  }
  public void undo() reads nothing writes this.Value {
    super.set(saved);
  }
}
```

The implementation of `UndoableVar.set()` would not be legal if writing did not include the possibility of reading.

Computing Method Effects Effects are computed in a bottom-up traversal of the syntax tree. Every statement and expression has an associated set of effects. An effect is produced by any expression that reads or writes a mutable variable or mutable object field (in Java, immutability is indicated by the `final` modifier). The effects of expressions and statements include the union of the effects of all their sub-expressions. The computed effects use the most specific regions possible.

The analysis is intraprocedural; effects of method calls are obtained from the annotation on the called method, rather than from an analysis of the method's implementation. Because the method's declared effects are with respect to the formal parameters of the method, the effects of a particular method invocation are obtained by substituting the actual parameters for the formal parameters in the declared effects.

Checking Method Effects To check that a method has no more effects than permitted, each effect computed for the body of the method is checked against the effects permitted in the method's annotation. Any computed effect other than effects on local variables (which are irrelevant once the method returns) and effects on newly created objects must be *included* in at least one of the permitted effects in the annotation.[4] The hierarchical nature of regions ensures that if an effect is included in a union of other effects then it must be included in one of the individual effects. Constructor annotations are not required to include write effects on the `Instance` region of the newly created object.

[4] These exceptions can be seen as an instance of *effects masking* as defined for FX [6].

A read effect e_1 includes another read effect e_2 if e_1's target includes e_2's target. A write effect e_1 includes an effect e_2 if e_1's target includes e_2's target; e_2 may be a read or write effect. The target *var*`.`*region* includes any target for the same instance and a region included in its region. The target `any(`*class*`).`*region* includes any instance target referring to a region included in its region.[5] The target *class*`.`*region* includes any target using a region included by its region. Appendix A.3 defines effect inclusion precisely.

Example 3. Consider the body of `set` in class `UndoableVar`. In inferring the effects of the method body, we collect the following effects:

```
writes this.saved from this.saved = ...
 reads this.Value from this.get()
writes this.Value from super.set(...)
           reads x from x
```

The last effect can be ignored (it only involves local variables). The other three effects are all included in the effect `writes this.Value` in the annotation. This annotation is the same as that of the method it overrides

3 Unshared Fields

In this section we describe an additional feature of our effects system that permits the private state of an object to be further abstracted, hiding implementation details of an abstract data type from the user of the class. Ideally the user of the class `Vector` in Figure 3 should not care that it is implemented using an array, but any valid annotation of the `addElement` method based on what has been presented so far must reveal that its effects interfere with any access of an existing array. In this example, the array used to implement the vector is entirely encapsulated within the object, and so it is impossible that effects on this internal array could interfere with any other array.

Therefore, in addition to effect annotations, we now add the annotation `unshared` to the language as a modifier on field declarations. Unshared fields have the semantics of unique fields described in our earlier work [4] and formalized in a recently submitted paper [2]. Essentially an unshared field is one that has no aliases at all when it is read, that is, any object read from it is not accessible through any other variable (local, parameter or field).[6] An analysis ensures this property partly by keeping track of ("dynamic") aliases created by a read; the analysis rejects programs that may not uphold the property. We also use `unique` annotations on formal parameters and return values. The previously mentioned analysis ensures that the corresponding actual parameters are unaliased at the point of call and that a unique return value is unaliased when a method returns. For a more detailed comparison with the related terms used by other researchers, please see our related paper [2].

[5] The class is used to resolve the region but not to define inclusion.

[6] More precisely, the object *will not* be accessed through any other variable.

```
public class Vector {
  public region Elements, Size;
  private Object[] list in Elements;
  private int count in Size;

  public Vector() reads nothing writes nothing
  {
    this.list = new Object[10];
    this.count = 0;
  }

  public void addElement( Object obj )
      reads nothing writes this.Size, this.Elements, any(Array).[]
  {
    if( this.count == this.list.length ) {
      Object[] temp = new Object[this.list.length << 1];
      for( int i = 0; i < this.list.length; i++ )
        temp[i] = this.list[i];
      this.list = temp;
    }
    this.list[this.count++] = obj;
  }
  ...
}
```

Fig. 3. Dynamic array class implemented *without* unshared fields. Notice the exposure of `writes any(Array).[]` in method `addElement()`.

3.1 Using Unshared Fields

Let p refer to an object with an unshared field f that refers to a unique object. Because the object referenced by $p.f$ is *always* unaliased, it is reasonable to think of the contents of the referenced object as contents of the object p. Therefore, the instance regions of the object $p.f$ can be mapped into the regions of the object p. In this way, the existence of the object $p.f$ can be abstracted away. By default, the instance regions of the object referred to by an unshared field are mapped into the region that contains the unshared field (equivalent to mapping the region $p.f$.`Instance` into the region $p.f$). This can be changed by the programmer, however. In the examples, we extend the syntax of unshared fields to map regions of the object referenced by the unshared field into regions of the object containing the field. The mapping is enclosed in braces `{}`. (See the declaration of the field `list` in Figure 4.)

The mapping of instance regions of the unshared field (more precisely, those regions of the object under `Instance`) must preserve the region hierarchy. Consider an unshared field f of object o that maps region $f.q$ into region $o.p$. If region $f.q'$ is a descendant of $f.q$, then it must be mapped into a descendant of $o.p$.

```
public class Vector {
  public region Elements, Size;
  private unshared Object[] list in Elements {Instance in Elements};
  private int count in Size;

  public Vector() reads nothing writes nothing
  {
    this.list = new Object[10];
    this.count = 0;
  }

  public void addElement( Object obj )
      reads nothing writes this.Size, this.Elements
  {
    if( this.count == this.list.length ) {
      Object[] temp = new Object[this.list.length << 1];
      for( int i = 0; i < this.list.length; i++ )
        temp[i] = this.list[i];
      this.list = temp;
    }
    this.list[this.count++] = obj;
  }
  ...
}
```

Fig. 4. Dynamic array class implemented using unshared fields. Differences from Figure 3 are underlined. Now `writes any(Array).[]` is absent from the annotation of method `addElement`.

Example 4. Consider a class `Vector`, shown in Figure 3 that implements a dynamic array. Because nothing is known about the possible aliasing of the field `list`, it must be assumed that the array is aliased, and that any use of an element of the array referenced by `list` interferes with any other array. There are two main reasons that this is undesirable. First, by requiring an effect annotation using the region `[]`, it is revealed that the dynamic array is implemented with an array. There is no reason that the user of the class needs to know this. Second, the required effect annotation prevents the following two statements from being swapped, even when `vector1` and `vector2` are not aliased (recall that we intend to use this analysis to support semantics-preserving program manipulations).

```
vector1.addElement( obj1 );
vector2.addElement( obj2 );
```

There is no reason for the user of the class to expect that two distinct dynamic arrays should interfere with each other. The problem is that the implementation of `Vector` does not indicate that two instantiations are necessarily independent, although a simple inspection of the code would reveal that they are. By making the `list` field `unshared`, as in Figure 4, this information is provided. Now the

`Instance` region (which includes the `[]` region) of `list` is mapped into into the region `Elements` of the `Vector` object, and it is no longer necessary to include the effect `writes Array.[]` in the annotation of `addElement`.

3.2 Checking Method Effects (Reprise)

In the previous section, we mentioned in passing that effects on local variables could be omitted when checking whether a method had no more effects than permitted. Effects on newly created objects can also be omitted, and similarly effects on unique references passed in as parameters can be omitted because by virtue of passing them, the caller retains no access (similar to "linear" objects) and thus any effects are irrelevant.

Unshared fields affect how we check inferred effects against permitted effects. Specifically, the declared mappings are used to convert any effects on regions of the unshared field into effects on the regions of the owner. This process is called "elaboration" and is detailed in Appendix A.4. The irrelevant effects are then masked from the elaborated effects and the remaining effects checked against those in the method's annotation.

Example 5. Because we can use unshared fields to abstract away the implementation of a class, we can implement the same abstract data type in two different ways, but with both classes' methods having the same annotations. Consider the implementation of a dynamic array shown in Figure 5. The field `head` in the class `LinkedVector` is unshared. This captures the notion that each dynamic array maintains a unique reference to its list of elements. This is necessary, but not sufficient, to make the effect annotations on `addElement` correct. It is not sufficient because it only indicates that the *first* element of the linked list is unshared; it says nothing about any of the other elements. Because it is the intention that each dynamic array maintains a distinct linked list, the `next` field of `Node` is declared to be `unshared` as well.

Let us verify the annotation of the method `addElement()` of class `Node`: `reads nothing writes this.Next`. The effect of the condition of the `if` statement is `reads this.next`, and the effect of the `then`-branch is `reads x, writes this.next`. The effect on the parameter `x` can be ignored outside the method, and the other effects are covered by the declared effect `writes this.Next`. The `else`-branch is more interesting, having a recursive call to `addElement()`. Looking up the declared effects of `addElement()`, we find `reads nothing writes this.Next`. The actual value of the receiver must now be substituted into the method effects. The receiver in this case is `this.next`, which is an unshared field whose `Next` region is mapped into region `this.Next`. The effect of the method call is thus `writes this.Next` (instead of a write to region `Next` of the object `this.next`), which is covered by the declared effects.

Checking that the declared effects of `addElement()` of `LinkedVector` are correct (and the same as the effects of `Vector.addElement()`) is now straightforward. The creation of a new node does not produce effects noticeable outside of the method. The conditional statement reads and writes `this.head`,

```
class Node {
  public region Obj, Next;
  public Object obj in Obj;
  public unshared Node next in Next
    {Instance in Next};

  public unique Node( Object o, Node n ) reads nothing writes nothing
  {
    this.obj = o;
    this.next = n;
  }

  public void addElement( unique Node x )
      reads nothing writes this.Next
  {
    if( this.next == null ) this.next = x;
    else this.next.addElement( x );
  }
}

class LinkedVector {
  public region Elements, Size;
  private unshared Node head in Elements
    { Instance in Elements };
  private int count in Size;

  public LinkedVector() reads nothing writes nothing
  {
    this.head = null;
    this.count = 0;
  }

  public void addElement( Object obj )
      reads nothing writes this.Elements, this.Size
  {
    Node tail = new Node( obj, null );
    if( this.head == null )
      this.head = tail;
    else
      this.head.addElement( tail );
    this.count += 1;
  }
  ...
}
```

Fig. 5. A dynamic array implemented using a linked list. Note that the annotation on `addElement(Object)` is the same as in Figure 4.

and calls `addElement()` on `this.head`. The first two effects are covered by the declared effect `writes this.Elements`, while the effect of the method call is `writes this.Elements` because the method call writes the `Next` region of the unshared field `this.head`, which maps `Next` to `this.Elements`. Finally, incrementing `this.count` has the effect `writes this.count`, which is covered by the declared effect `writes this.Size`.

Example 6. Figure 6 gives a simple implementation of a dictionary using association lists with annotations that abstract away the implementation details. It demonstrates the utility of being able to map the regions of an unshared field into multiple regions of the owner. The same set of annotations could be used for a hash table using the built-in hash code method `System.identityHashCode()`.

Assuming we had a variable `t` declared with type `Table`, the annotations permit the reordering of `t.containsKey()` and `t.pair()`, but do not permit reordering of `t.containsKey()` with `t.clear()`. The annotation on `clear()` is the most precise possible.

4 Using Effect Analysis

We will now briefly describe how the effects analysis can be used when applying semantics-preserving program manipulations. A precondition of many such manipulations (typically those that cause code motion) is that there not be any data dependencies between the parts of the program affected by the manipulation. In general, the two computations interfere if one mutates state also accessed by the other. Assuming the effects system is sound (see Section 6), then if the computations interfere, there is an effect from one computation that "conflicts" with an effect from the second. Two effects conflict if at least one is a write effect and they involve targets that may "overlap," that is, may refer to the same mutable state at run time.

Two targets may overlap only if they refer to overlapping regions, and the hierarchical nature of regions ensures that regions overlap only if one is included in the other. Furthermore, an instance target (a use of an instance region of a particular object) can only overlap another instance target if the objects could be identical. The effect inference system computes effects (and targets) using the expressions in the computation in context. Thus equality cannot be compared devoid of context. This observation has led us to formalize the desired notion of equality in a new alias question $\mathit{MayEqual}(e, e')$ [3]. Steensgaard's "points-to" analysis [11] may be used as a conservative approximation. These considerations are further complicated by unshared fields. The details may be seen in the Appendix.

5 Related Work

Reynolds [10] showed how interference in Algol-like programs could be restricted using rules that prevent aliasing. His technique, while simple and general, requires access to the bodies of procedures being called in order to check whether

```
class Node {
  Object key, val;
  unshared Node next { Instance in Instance, key in key, val in val, next in next };

  unique Node(Object k, Object v, unique Node n) reads nothing writes nothing
  { key=k; val=v; next=n; }

  int count() reads next writes nothing
  { if (next == null) return 1; else return next.count()+1; }
  Object get(Object k) reads key, val, next writes nothing
  { if (k == key) return val; else if (next == null) return null;
    else return next.get(k); }
  .
  .
  .
  boolean containsKey(Object k) reads key, next writes nothing
  { return k == key || next != null && next.containsKey(k); }
  void pair() reads key, next writes val
  { val = key; if (next != null) next.pair(); else ;}
}

public class Table {
  region Table;
  region Key in Table, Value in Table, Structure in Table;

  private unshared Node list in Structure
    { Instance in Instance, key in Key, val in Val, next in Structure };

  /** Return number of (key,value) pairs */
  public int count() reads Structure writes nothing
  { if (list == null) return 0;
    else return list.count(); }

  /** Remove all entries from table. */
  public void clear() reads nothing writes Structure { list = null; }

  public Object get(Object k) reads Table writes nothing
  { if (list == null) return null;
    else return list.get(k); }

  public void put(Object k, Object v) reads nothing writes Table { ... }
  public void remove(Object k) reads nothing writes Table { ... }
  public boolean contains(Object v) reads Val,Structure writes nothing { ... }

  public boolean containsKey(Object k) reads Key,Structure writes nothing
  { return list != null && list.containsKey(k); }

  /** Pair each existing key with itself. */
  public void pair() reads Key,Structure writes Value
  { if (list != null) list.pair(); else ; }
}
```

Fig. 6. Simple dictionary class with effects annotations.

they operate on overlapping global variables. The work includes a type system with mutable records, but the records cannot have recursive type (lists and trees are not possible).

Jackson's Aspect system [7] uses a similar abstraction mechanism to our regions in order to specify the effects of routines on abstract data types. He uses specifications for each procedure in order to allow checking to proceed in a modular fashion. In his work, however, the specifications are *necessary* effects and are used to check for missing functionality.

Effects were first studied in the FX system [6], a higher-order functional language with reference cells. The burden of manual specification of effects is lifted through the use of effects inference as studied by Gifford, Jouvelot, and Talpin [8, 12]. The work was motivated by a desire to use stack allocation instead of heap allocation in mostly pure functional programs, and also to assist parallel code generation. The approach was demonstrated successfully for the former purpose in later work [1, 13]. These researchers also make use of a concept of disjoint "regions" of mutable state, but these regions are global, as opposed to within objects. In the original FX system, effects can be verified and exploited in separately developed program components.

Leino [9] defines "data groups," which are sets of instance fields in a class. Each method declares which data groups it could modify. As with regions, a subclass may add new instance fields to existing data groups, but unlike our proposal, a field may be added to more than one data group. In their system, this ability is sound because the modifies clause is only used to see if a given method will modify a given field, not to see if effects of two methods could interfere.

Numerous researchers have examined the question of unique objects, as detailed in a paper of ours [2]. Most recently, Clarke, Potter and Noble have have formalized a notion of "owner" that separates ownership from uniqueness [5]. Their notion of ownership appears useful for the kind of effects analysis we present here, although the precise mechanism remains further work.

6 Further Work

The work described in this paper is ongoing. We are continuing to extend our effects system to cover all of Java. We also intend to use the system to infer suggested annotations on methods. Ultimately, the effects system will be used to determine possible data dependencies between statements, which will require a form of alias analysis. This section sketches our ideas for further work.

An important aspect of a static effect system is for it to be *sound*, that is, it does not say two computations do not interfere through the access of some shared state when in fact they do. Proving soundness requires a run-time definition of interference which we must then show is conservatively approximated by our analysis. The run-time state of the program we are interested in includes fields of reachable objects as well as the state of local variables and temporaries in all activation frames. The effects of a computation can then be seen as reads and

writes on elements of the state. These effects can then be compared to that of the second computation. Interference occurs when an element of state is written by one computation and read or written by the other. We have begun formulating a proof of correctness using the concept of what run-time state is accounted for by a target (is "covered" by a target), proving that if a target includes another target than it covers everything covered by the other.

Java permits a limited form of multiple inheritance in the form of code-less "interfaces." In order to permit useful annotations on method headers in interfaces, we must be able to add regions to interfaces and thus we must handle multiple inheritance of instance regions. (As with static fields, static regions are not inherited and thus do not complicate matters.) Multiple inheritance of instance regions is handled by permitting a class or interface inheriting regions from an interface to map them into other regions as long as the hierarchy is preserved. Any two regions, both of which are visible in a superclass or superinterface of the classes or interfaces performing the mapping, must have the same relation ($<$, $>$, or unrelated) in the latter class or interface. This restriction ensures that reasoning about regions remains sound. Conflicting relations in superclasses or superinterfaces may forbid certain inheritance combinations.

Java has *blank final* fields which are fields that are initialized in each constructor of the class that declared them but otherwise follow the restrictions on final fields—they may not be assigned. The Java language does not forbid methods from *reading* the value of a final field, even if the method is called in the constructor before the field is initialized. This situation is aggravated by classes which override methods called by the constructor in the superclass. In order to treat blank finals the same as finals (as immutable fields) and yet determine data dependencies correctly, extra annotations are needed on methods that may be called by a constructor. In particular, these methods must be annotated with the list of blank final fields that may be read. This annotation breaks abstraction to some degree but is needed only when checking the class itself and its subclasses.

Java supports multiple threads. So far we have ignored the possibility that state could be modified by a concurrent thread. As explained in a recent Java-World article [14], thread safety for the access of a field can be ensured in three situations:

1. the access is protected by a synchronization on the owning object;
2. the field is immutable ("final");
3. the object has a "thread-safe" wrapper.

The first two cases can be easily checked through syntactic checks. The third condition is satisfied when the field is accessed through unshared or unique references.

As discussed in our earlier work [4], the task of adding annotations can be made less burdensome if it can be semi-automated. The same technique used to test whether a method stays within the permitted effects annotation can help to generate an annotation for an unannotated method. Essentially, the set of effects inferred from the method is culled by removing redundant effects, and

effects on regions inaccessible to the caller are promoted to effects on the nearest accessible parent region.

7 Conclusion

We have introduced an effects system for Java that permits an implementor to express the effects of a method in terms of abstract regions instead of in terms of mutable private fields. The effects system properly treats wholly-owned subsidiary objects as part of the owning object. We also introduce a way of checking the effects inferred from the body of a method against the effects permitted in a method's annotation. As described in this paper, this effects system can be used to check consistency of annotations. It can also be used to determine whether there may be data dependencies between two computations involving separately compiled code.

Acknowledgments

We thank our colleagues at CMU (William L. Scherlis and Edwin C. Chan) and UWM (Adam Webber) for comments on drafts of the paper and numerous conversations on the topic. We also thank the anonymous reviewers.

References

1. Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, San Diego, California, USA, *ACM SIGPLAN Notices*, 30(6):174–185, June 1995.
2. John Boyland. Deferring destruction when reading unique variables. Submitted to IWAOOS '99, March 1999.
3. John Boyland and Aaron Greenhouse. MayEqual: A new alias question. Submitted to IWAOOS '99, March 1999.
4. Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '98)*, Kyoto, Japan, April 19–25, pages 167–176. IEEE Computer Society, Los Alamitos, California, 1998.
5. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, *ACM SIGPLAN Notices*, 33(10):48–64, October 1998.
6. D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.
7. Daniel Jackson. Aspect: Detecting bugs with abstract dependencies. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

8. Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 303–310. ACM Press, New York, 1991.
9. K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, *ACM SIGPLAN Notices*, 33(10), October 1998.
10. John C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 39–46. ACM Press, New York, January 1978.
11. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, St. Petersburg, Florida, USA, January 21–24, pages 32–41. ACM Press, New York, 1996.
12. Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.
13. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the Twenty-first Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Portland, Oregon, USA, January 17–21, pages 188–201. ACM Press, New York, 1994.
14. Bill Venners. Design for thread safety: Design tips on when and how to use synchronization, immutable objects, and thread-safe wrappers. *JavaWorld*, August 1998.

A Definitions

Throughout the appendix, class names/types and variable names are represented as elements of a set of identifiers **Ide**. We assume that facilities are available for determining the type/class of an expression/variable, and that the class hierarchy is available for determining if two classes are related. We use the phrase "class c is a descendant of class c'" to mean that class c is equal to c' or c's superclass is a descendant of c'.

$$\begin{array}{rlrl} \mathit{var} \rightarrow & \mathbf{Ide} & \qquad \mathit{type} \rightarrow & \mathbf{Ide} \\ \mathit{name} \rightarrow & \mathbf{Ide} & \qquad \mathit{class} \rightarrow & \mathbf{Ide} \end{array}$$

A.1 Syntax of Programming Language

Several language features used in the text have been omitted for simplicity; the remaining features can express the omitted features. The method receiver object is never implicit. It is illegal to write to a formal parameter. Parent regions must always be specified, and there are no default mappings for unshared fields. We require instance fields/regions to be explicitly declared as such using the keyword `instance`. A preprocessing pass can convert a program from the language presented in the body of the paper into the one presented here.

$$
\begin{array}{rcl}
\mathit{program} & \rightarrow & \mathit{classDecl}^{*} \\
\mathit{classDecl} & \rightarrow & \texttt{class}\ \mathit{name}\ \texttt{extends}\ \mathit{name}\ \mathit{classBody} \\
\mathit{classBody} & \rightarrow & \{\ \mathit{classBodyDecl}^{*}\ \} \\
\mathit{classBodyDecl} & \rightarrow & \mathit{region} \mid \mathit{field} \mid \mathit{method} \mid \mathit{constructor} \\
\mathit{region} & \rightarrow & \mathit{kind}\ \texttt{region}\ \mathit{name}\ \texttt{in}\ \mathit{name}\texttt{;} \\
\mathit{field} & \rightarrow & \mathit{kind}\ \mathit{type}\ \mathit{name}\ \texttt{in}\ \mathit{name}\texttt{;} \\
 & \mid & \mathit{kind}\ \texttt{unshared}\ \mathit{type}\ \mathit{name}\ \texttt{in}\ \mathit{name}\ \{\mathit{mappings}\}\texttt{;} \\
\mathit{mappings} & \rightarrow & \mathit{mapping} \mid \mathit{mapping}\texttt{,}\ \mathit{mappings} \\
\mathit{mapping} & \rightarrow & \mathit{name}\ \texttt{in}\ \mathit{name} \\
\mathit{kind} & \rightarrow & \texttt{instance} \mid \texttt{static} \\
\mathit{method} & \rightarrow & \mathit{type}\ \mathit{name}\ \texttt{(}\ \mathit{params}\ \texttt{)}\ [\mathit{annotation}]\ \mathit{block} \\
\mathit{constructor} & \rightarrow & \mathit{name}\ \texttt{(}\ \mathit{params}\ \texttt{)}\ [\mathit{annotation}]\ \mathit{block} \\
\mathit{annotation} & \rightarrow & \texttt{reads}\ \mathit{locations}\ \texttt{writes}\ \mathit{locations} \\
\mathit{locations} & \rightarrow & \texttt{nothing} \mid \mathit{targetSeq} \\
\mathit{targetSeq} & \rightarrow & \mathit{location} \mid \mathit{location}\texttt{,}\ \mathit{targetSeq} \\
\mathit{location} & \rightarrow & \mathit{var}\texttt{.}\mathit{region} \mid \texttt{any(}\mathit{class}\texttt{).}\mathit{region} \mid \mathit{class}\texttt{.}\mathit{region} \\
\mathit{params} & \rightarrow & \epsilon \mid \mathit{paramDecls} \\
\mathit{paramDecls} & \rightarrow & \mathit{paramDecl} \mid \mathit{paramDecl}\texttt{,}\ \mathit{paramDecls} \\
\mathit{paramDecl} & \rightarrow & \mathit{type}\ \mathit{name} \\
\mathit{block} & \rightarrow & \{\ \mathit{stmt}^{*}\ \} \\
\mathit{stmt} & \rightarrow & \mathit{block} \mid \texttt{;} \mid \texttt{return;} \mid \texttt{return}\ \mathit{expr}\ \texttt{;} \mid \mathit{type}\ \mathit{name}\texttt{;} \mid \\
 & & \mathit{expr}\ \texttt{=}\ \mathit{expr}\texttt{;} \mid \mathit{stmtExpr}\texttt{;} \mid \mathit{constuctObj}\texttt{(}\mathit{args}\texttt{);} \mid \\
 & & \texttt{if(}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt} \mid \texttt{while(}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt} \\
\mathit{expr} & \rightarrow & \mathit{name} \mid \mathit{rexpr} \mid \mathit{vexpr} \\
\mathit{rexpr} & \rightarrow & \mathit{stmtExpr} \mid \mathit{allocExpr} \mid \mathit{constuctObj} \mid \mathit{expr}\ \texttt{[}\mathit{expr}\texttt{]} \mid \mathit{expr}\texttt{.}\mathit{field} \\
\mathit{vexpr} & \rightarrow & \mathit{expr} \oplus \mathit{expr} \mid \ominus \mathit{expr} \mid \texttt{true} \mid \texttt{1} \mid \texttt{null} \mid \mathit{type} \\
\mathit{args} & \rightarrow & \epsilon \mid \mathit{argsSeq} \\
\mathit{argsSeq} & \rightarrow & \mathit{expr} \mid \mathit{expr}\texttt{,}\ \mathit{argsSeq} \\
\mathit{stmtExpr} & \rightarrow & \mathit{expr}\texttt{.}\mathit{method}\texttt{(}\mathit{args}\texttt{)} \\
\mathit{allocExpr} & \rightarrow & \texttt{new}\ \mathit{type}\ (\texttt{[}\mathit{expr}\texttt{]})^{+} \mid \texttt{new}\ \mathit{class}\texttt{(}\mathit{args}\texttt{)} \\
\mathit{constuctObj} & \rightarrow & \texttt{super} \mid \texttt{this}
\end{array}
$$

A.2 Tracking locals

We need to track the mutable state references that could occur in local variables. The analysis here computes a relation between local variables and such reference expressions (*rexpr* in the preceding grammar) that each variable may be equal to. The relation also permits a local to be paired to the initial value of a formal parameter. It is not necessary that these expressions are available at the analysis point or even side-effect-free for this purpose. This analysis is very similar to standard def-use dataflow analyses. The relations will be drawn from a set **VB** for "variable bindings."

$$
\begin{aligned}
\mathbf{Bindings} &= \mathit{rexpr} + \mathbf{Ide} \\
\mathbf{VB} &= 2^{\mathbf{Ide} \times \mathbf{Bindings}}
\end{aligned}
$$

$$
\begin{aligned}
B \;:\;& \mathit{expr} \rightarrow \mathbf{VB} \rightarrow \mathbf{Bindings} \\
B\,\mathit{name}\,V =& \{b \mid (\mathit{name}, b) \in V\} \\
B\,\mathit{rexpr}\,V =& \{\mathit{rexpr}\} \\
B\,\mathit{vexpr}\,V =& \{\}
\end{aligned}
$$

$$
V \backslash \mathit{name} = V - \{(\mathit{name}, b) \mid b \in \mathbf{Bindings}\}
$$

For each statement s, we define two sets V_s^- and V_s^+ as the least fixed point solution to the following set of equations defined with regard to the syntax of each method or constructor:

$\ldots\ \mathit{name}\ \texttt{(}\ \mathit{params}\ \texttt{)}\ \ldots\ \mathit{block}$
$\quad V_{block}^- = \{(\mathit{var}, \mathit{var}) \mid \mathit{var} \in \mathit{params}\}$
$\mathit{block} \rightarrow \texttt{\{}\ \mathit{stmt}_1\ \ldots\ \mathit{stmt}_n\ \texttt{\}}$
$\quad V_{stmt_1}^-, \ldots, V_{stmt_n}^-, V_{block}^+ = V_{block}^-, V_{stmt_1}^+, \ldots, V_{stmt_n}^+$
$\mathit{stmt} \rightarrow \mathit{block}$
$\quad V_{block}^-, V_{stmt}^+ = V_{stmt}^-, V_{block}^+$
$\mathit{stmt} \rightarrow \texttt{;}$
$\quad V_{stmt}^+ = V_{stmt}^-$
$\mathit{stmt} \rightarrow \mathit{type}\ \mathit{name}\texttt{;}$
$\quad V_{stmt}^+ = V_{stmt}^- \backslash \mathit{name}$
$\mathit{stmt} \rightarrow \mathit{name}\ \texttt{=}\ \mathit{expr}\texttt{;}$
$\quad V_{stmt}^+ = (V_{stmt}^- \backslash \mathit{name}) \cup \left\{(\mathit{name}, b) \mid b \in B\,\mathit{expr}\,V_{stmt}^-\right\}$
$\mathit{stmt} \rightarrow \mathit{expr}\ \texttt{=}\ \mathit{expr}\texttt{;}$ (Otherwise)
$\quad V_{stmt}^+ = V_{stmt}^-$
$\mathit{stmt} \rightarrow \mathit{stmtExpr}\texttt{;} \mid \mathit{constructObj}\texttt{(}\mathit{args}\texttt{)}$
$\quad V_{stmt}^+ = V_{stmt}^-$
$\mathit{stmt} \rightarrow \texttt{if (}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2$
$\quad V_{stmt_1}^-, V_{stmt_2}^- = V_{stmt}^-, V_{stmt}^- \qquad V_{stmt}^+ = V_{stmt_1}^+ \cup V_{stmt_2}^+$
$\mathit{stmt} \rightarrow \texttt{while (}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt}'$
$\quad V_{stmt'}^-, V_{stmt}^+ = V_{stmt}^- \cup V_{stmt'}^+, V_{stmt}^- \cup V_{stmt'}^+$

When analyzing a target involving a local variable v, we will need the set of binding for that use. Let $B\,v$ be shorthand for $B\,v\,V_s^-$ where s is the immediately enclosing statement.

A.3 Formalization of Regions and Effects

A region is a triple (*class*, *name*, *tag*), where *class* is the class in which the region is declared, *name* is the name of the region, and *tag* indicates whether the region is static or instance. We require the name of each region to be unique.

$$
\begin{aligned}
\mathbf{Regions}_I &= \mathbf{Ide} \times \mathbf{Ide} \times \{\mathsf{instance}\} \text{ (Set of instance regions)} \\
\mathbf{Regions}_S &= \mathbf{Ide} \times \mathbf{Ide} \times \{\mathsf{static}\} \text{ (Set of static regions)} \\
\mathbf{Regions} &= \mathbf{Regions}_I \cup \mathbf{Regions}_S \;=\; \mathbf{Ide} \times \mathbf{Ide} \times \{\mathsf{static}, \mathsf{instance}\}
\end{aligned}
$$

An effect is a read of or write to one of four kinds of targets, which represent different granularities of state.

$$\begin{array}{c} \mathit{effect} \rightarrow \mathsf{read}(\mathit{target}) \mid \mathsf{write}(\mathit{target}) \\ \mathit{target} \rightarrow \mathsf{local}\ \mathbf{Ide} \mid \mathsf{instance}\ \mathit{expr} \times \mathbf{Regions}_I \mid \mathsf{anyInstance}\ \mathbf{Regions}_I \mid \mathsf{static}\ \mathbf{Regions}_S \end{array}$$

The region hierarchy is a triple $(\mathit{regions}, \mathsf{parent}_R, \mathit{unshared})$, where *regions* is the set of regions in the program, parent_R is the parent ordering of the regions; $r\,\mathsf{parent}_R\, s \Leftrightarrow s$ is the super region of r; and *unshared* is the mapping of regions of unshared fields to regions of the class. The mapping is a set of 4-tuples $(\mathit{class}, \mathit{field}, \mathit{region}_u, \mathit{region}_c)$ with the interpretation that *field* is an unshared field of instances of class *class*, and region region_u of the object referenced the field is mapped into region region_c of the object containing *field*.

$$\begin{aligned} \mathbf{Map} &= \mathbf{Ide} \times \mathbf{Ide} \times \mathbf{Regions} \times \mathbf{Regions} \\ \mathbf{Hier} &= 2^{\mathbf{Regions}} \times (\mathbf{Regions} \rightarrow \mathbf{Regions}) \times 2^{\mathbf{Map}} \\ \mathsf{All} &= (\texttt{Object}, \texttt{All}, \mathsf{static}) \\ \mathsf{Instance} &= (\texttt{Object}, \texttt{Instance}, \mathsf{instance}) \\ \mathsf{Element} &= (\texttt{Array}, \texttt{[]}, \mathsf{instance}) \\ \mathbf{H}_0 &= (\{\mathsf{All}, \mathsf{Instance}, \mathsf{Element}\}, \{(\mathsf{Instance}, \mathsf{All}), (\mathsf{Element}, \mathsf{Instance})\}, \emptyset) \end{aligned}$$

$$\begin{aligned} \mathsf{lookup} &: 2^{\mathbf{Regions}} \rightarrow \mathbf{Ide} \rightarrow \mathbf{Ide} \rightarrow \mathbf{Regions} \\ \mathsf{lookup}\ r\ c\ f &= (c', f, t) \in r \text{ s.t. class } c \text{ is a descendant of class } c' \end{aligned}$$

The region hierarchy $(\mathit{rgns}, \mathsf{parent}_R, m)$ is built from $\mathbf{H}_0$, the initial hierarchy, by analyzing the field and region declarations in the program's class definitions. It must have the following properties: no region may have more than one parent; the region `Instance` of an unshared field must be mapped into a region of the owning object; the unshared mappings must respect the tree structure; only instance regions of unshared fields may be mapped; and regions of a static unshared field may only be mapped into static regions.

$$\begin{array}{c} \forall (c, r, p) \in \mathit{rgns}.\, \forall (c, r, p') \in \mathit{rgns}.\ p = p' \\ (\exists (c, f, r_u, r_c) \in m) \Rightarrow (\exists (c, f, \mathsf{Instance}, q) \in m) \\ \forall (c, f, r_u, r_c) \in m. \left(\begin{array}{c} (\forall (c, f, r'_u, r'_c) \in m.\ r'_u \leq_R r_u \Rightarrow r'_c \leq_R r_c) \\ \wedge\ (r_u \leq_R \mathsf{Instance}) \wedge (f \in \mathbf{Regions}_S \Rightarrow r_c \in \mathbf{Regions}_S) \end{array} \right) \end{array}$$

Inclusion Given the region hierarchy $(\mathit{rgns}, \mathsf{parent}_R, m)$, we create the reflexive transitive closure of parent_R to define the inclusion relation over regions.

$$\begin{aligned} &\leq_R \subseteq \mathbf{Regions} \times \mathbf{Regions} \\ &\leq_R = (\mathsf{parent}_R \cup \{(r, r) \mid r \in \mathit{rgns}\})^* \end{aligned}$$

The inclusion relation over targets, $\leq_T$, is defined to be the reflexive transitive closure of a relation, parent_T.

$$
\begin{aligned}
\mathsf{unshared}\, e &\Leftrightarrow e = e'.f \wedge \exists(c, f, r_u, r_c) \in m \\
\mathsf{shared}\, e &\Leftrightarrow \neg\mathsf{unshared}\, e
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{parent}_T &\subseteq \mathit{target} \times \mathit{target} \\
\mathsf{local}\, v \;\mathsf{parent}_T\; \mathsf{local}\, w &\Leftrightarrow v = w \\
\mathsf{instance}\,(e, r) \;\mathsf{parent}_T\; \mathsf{instance}\,(e, r') &\Leftarrow r\, \mathsf{parent}_R\, r' \\
\mathsf{instance}\,(e.f, r) \;\mathsf{parent}_T\; \mathsf{instance}\,(e, r') &\Leftarrow \mathsf{lookup}\ \mathit{rgns}\ (\mathsf{typeof}\ e)\ f \in \mathbf{Regions}_I \\
&\quad \wedge\ \exists(c, f, r, r') \in m \\
\mathsf{instance}\,(e, r) \;\mathsf{parent}_T\; \mathsf{anyInstance}\, r &\Leftarrow \mathsf{shared}\, e \\
\mathsf{instance}\,(e.f, r) \;\mathsf{parent}_T\; \mathsf{static}\, s &\Leftarrow \mathsf{lookup}\ \mathit{rgns}\ (\mathsf{typeof}\ e)\ f \in \mathbf{Regions}_S \\
&\quad \wedge\ \exists(c, f, r, s) \in m \\
\mathsf{anyInstance}\, r \;\mathsf{parent}_T\; \mathsf{anyInstance}\, r' &\Leftarrow r\, \mathsf{parent}_R\, r' \\
\mathsf{anyInstance}\, r \;\mathsf{parent}_T\; \mathsf{static}\, s &\Leftarrow r\, \mathsf{parent}_R\, s \\
\mathsf{static}\, s \;\mathsf{parent}_T\; \mathsf{static}\, s' &\Leftarrow s\, \mathsf{parent}_R\, s'
\end{aligned}
$$

$$
\begin{aligned}
\leq_T &\subseteq \mathit{target} \times \mathit{target} \\
\leq_T &= (\mathsf{parent}_T \cup \{(t, t) \mid t \in \mathit{target}\})^*
\end{aligned}
$$

Finally, effect inclusion is defined using $\leq_T$ in the obvious manner.

$$
\begin{aligned}
\leq_E &\subseteq \mathit{effect} \times \mathit{effect} \\
\mathsf{read}(t) \leq_E \mathsf{read}(t') &\Leftrightarrow t \leq_T t' \\
\mathsf{write}(t) \leq_E \mathsf{write}(t') &\Leftrightarrow t \leq_T t' \\
\mathsf{read}(t) \leq_E \mathsf{write}(t') &\Leftrightarrow t \leq_T t'
\end{aligned}
$$

Method Annotations The method annotations are used to construct a method dictionary: a set of tuples (*class*, *method*, *params*, *effects*), where *class.method* is the method (or constructor) name, *params* is a vector of type–identifier pairs that captures the method's signature and formal parameter names, and *effects* is the set of effects that the method produces. The method dictionary is produced by analyzing the annotations of the methods in each defined class. For a program p, with region hierarchy $(r, \mathsf{parent}_R, m)$, the set of method dictionary is $\mathcal{A}_P[\![p]\!]\ r\ \emptyset$.

$$
\begin{aligned}
\mathbf{Method} &= \mathbf{Ide} \times \mathbf{Ide} \times (\mathbf{Ide} \times \mathbf{Ide})^* \times 2^{\mathit{effect}} \\
\mathcal{A}_P &: \mathit{program} \to 2^{\mathbf{Regions}} \to 2^{\mathbf{Method}} \to 2^{\mathbf{Method}} \\
\mathcal{A}_C &: \mathit{classDecl} \to 2^{\mathbf{Regions}} \to 2^{\mathbf{Method}} \to 2^{\mathbf{Method}} \\
\mathcal{A}_B &: \mathit{classBody} \to 2^{\mathbf{Regions}} \to \mathbf{Ide} \to 2^{\mathbf{Method}} \to 2^{\mathbf{Method}} \\
\mathcal{A}_D &: \mathit{classBodyDecl} \to 2^{\mathbf{Regions}} \to \mathbf{Ide} \to 2^{\mathbf{Method}} \to 2^{\mathbf{Method}} \\
\mathcal{A}_A &: \mathit{annotation} \to 2^{\mathbf{Regions}} \to 2^{\mathit{effect}} \\
\mathcal{A}_T &: \mathit{locations} \to 2^{\mathbf{Regions}} \to 2^{\mathit{target}} \\
\mathcal{A}_F &: \mathit{params} \to (\mathbf{Ide} \times \mathbf{Ide})^*
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}_P[\![\mathit{classDecl}^*]\!]\, r\, a \;&=\; (\mathcal{A}_C[\![\mathit{classDecl}_n]\!] r) \cdots (\mathcal{A}_C[\![\mathit{classDecl}_1]\!] r\, a) \\
\mathcal{A}_C[\![\texttt{class}\, c\, \texttt{extends}\, p\ \mathit{classBody}]\!]\, r\, a \;&=\; \mathcal{A}_B[\![\mathit{classBody}]\!] r\, c\, a \\
\mathcal{A}_B[\![\{\mathit{classBodyDecl}^*\}]\!]\, r\, c\, a &= (\mathcal{A}_D[\![\mathit{classBodyDecl}_n]\!] r\, c) \\
&\quad \cdots (\mathcal{A}_D[\![\mathit{classBodyDecl}_1]\!] r\, c)\, a \\
\mathcal{A}_D[\![\mathit{region}]\!]\, r\, c\, a &= a \\
\mathcal{A}_D[\![\mathit{field}]\!]\, r\, c\, a &= a \\
\mathcal{A}_D[\![\mathit{type}\, \mathit{name}\ \texttt{(}\mathit{params}\texttt{)}\ \mathit{anno}\ b]\!]\, r\, c\, a &= a \cup \{(c, \mathit{name}, \mathcal{A}_F[\![\mathit{params}]\!], \mathcal{A}_A[\![\mathit{anno}]\!] r)\} \\
\mathcal{A}_D[\![c\ \texttt{(}\mathit{params}\texttt{)}\ \mathit{anno}\ b]\!]\, r\, c\, a &= a \cup \{(c, c, \mathcal{A}_F[\![\mathit{params}]\!], \mathcal{A}_A[\![\mathit{anno}]\!] r)\} \\
\mathcal{A}_A[\![\epsilon]\!]\, r &= \{\mathsf{writes}(\mathsf{All})\} \\
\mathcal{A}_A[\![\texttt{reads}\, \mathit{locs}_R\ \texttt{writes}\, \mathit{locs}_W]\!]\, r \;&=\; \textstyle\bigcup_{t \in \mathcal{A}_T[\![\mathit{locs}_R]\!] r} \mathsf{read}(t) \cup \bigcup_{t \in \mathcal{A}_T[\![\mathit{locs}_W]\!] r} \mathsf{write}(t) \\
\mathcal{A}_T[\![\texttt{nothing}]\!]\, r &= \emptyset \\
\mathcal{A}_T[\![\mathit{location}\texttt{,}\ \mathit{targetSeq}]\!]\, r &= \mathcal{A}_T[\![\mathit{location}]\!] r \cup \mathcal{A}_T[\![\mathit{targetSeq}]\!] r \\
\mathcal{A}_T[\![\mathit{var}\texttt{.}\,\mathit{region}]\!]\, r &= \{\mathsf{instance}\,(\mathit{var}, \mathsf{lookup}\ r\ (\mathsf{typeof}\ \mathit{var})\ \mathit{region})\} \\
\mathcal{A}_T[\![\mathit{class}\texttt{.}\,\mathit{region}]\!]\, r &= \{\mathsf{static}\,(\mathsf{lookup}\ r\ \mathit{class}\ \mathit{region})\} \\
\mathcal{A}_T[\![\texttt{any(}\mathit{class}\texttt{)}\texttt{.}\,\mathit{region}]\!]\, r &= \{\mathsf{anyInstance}\,(\mathsf{lookup}\ r\ \mathit{class}\ \mathit{region})\} \\
\mathcal{A}_F[\![\epsilon]\!] &= () \\
\mathcal{A}_F[\![\mathit{type}\, \mathit{name}]\!] &= (\mathit{type}, \mathit{name}) \\
\mathcal{A}_F[\![\mathit{type}\, \mathit{name}\texttt{,}\ \mathit{params}]\!] &= ((\mathit{type}, \mathit{name}) : \mathcal{A}_P[\![\mathit{params}]\!])
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{typeof} \;&:\; \mathit{expr} \to \mathbf{Ide} \\
\mathsf{typeof}\ e \;&=\; \text{static class of expression } e
\end{aligned}
$$

A.4 Computing Effects

The effects of an expression e in program p, with region hierarchy $(r, \mathsf{parent}_R, m)$ and effects dictionary d, are $\mathcal{E}[\![e]\!]\ (r, \mathsf{parent}_R, m)\ d$.

$$
\mathcal{E}[\![-]\!] : \mathbf{Hier} \to 2^{\mathbf{Method}} \to 2^{\mathit{effect}}
$$

Features that do not have effects

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{;}]\!]\ h\ d = \emptyset && \mathcal{E}[\![\mathit{type}]\!]\ h\ d = \emptyset && \mathcal{E}[\![\texttt{true}]\!]\ h\ d = \emptyset \\
&\mathcal{E}[\![\texttt{return;}]\!]\ h\ d = \emptyset && \mathcal{E}[\![\texttt{1}]\!]\ h\ d = \emptyset && \mathcal{E}[\![\texttt{null}]\!]\ h\ d = \emptyset
\end{aligned}
$$

Features that do not have direct effects

$$
\begin{aligned}
\mathcal{E}[\![\texttt{\{}\ \mathit{stmt}^*\ \texttt{\}}]\!]\ h\ d &= \textstyle\bigcup \mathcal{E}[\![\mathit{stmt}_i]\!] h\ d \\
\mathcal{E}[\![\texttt{return}\ \mathit{expr}\texttt{;}]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}]\!] h\ d \\
\mathcal{E}[\![\mathit{expr}_1\ \texttt{=}\ \mathit{expr}_2\texttt{;}]\!]\ h\ d &= \mathcal{E}_{\mathrm{LVal}}[\![\mathit{expr}_1]\!] h\ d \cup \mathcal{E}[\![\mathit{expr}_2]\!] h\ d \\
\mathcal{E}[\![\texttt{if(}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}]\!] h\ d \cup \mathcal{E}[\![\mathit{stmt}_1]\!] h\ d \cup \mathcal{E}[\![\mathit{stmt}_2]\!] h\ d \\
\mathcal{E}[\![\texttt{while(}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt}]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}]\!] h\ d \cup \mathcal{E}[\![\mathit{stmt}]\!] h\ d \\
\mathcal{E}[\![\mathit{expr}_1 \oplus \mathit{expr}_2]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}_1]\!] h\ d \cup \mathcal{E}[\![\mathit{expr}_2]\!] h\ d \\
\mathcal{E}[\![\ominus \mathit{expr}]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}]\!] h\ d \\
\mathcal{E}[\![\mathit{expr}\texttt{,}\ \mathit{argsSeq}]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}]\!] h\ d \cup \mathcal{E}[\![\mathit{argsSeq}]\!] h\ d \\
\mathcal{E}[\![\texttt{new}\ \mathit{type}\,(\texttt{[}\mathit{expr}\texttt{]})^+]\!]\ h\ d &= \textstyle\bigcup \mathcal{E}[\![\mathit{expr}_i]\!] h\ d
\end{aligned}
$$

Features that have direct effects Reading or writing to a local variable produces an effect on a local target. Array subscripting produces an effect on the array's region `[]`. Accessing a field produces an effect on an instance or static target, depending on whether the field is an instance or static field, respectively.

$$
\begin{aligned}
\mathcal{E}[\![var]\!]\ h\ d &= \{\mathsf{read}(\mathsf{local}\ var)\} \\
\mathcal{E}[\![constructObj]\!]\ h\ d &= \{\mathsf{read}(\mathsf{local}\ \mathtt{this})\} \\
\mathcal{E}[\![expr_1\,\mathtt{[}expr_2\mathtt{]}]\!]\ h\ d &= \mathcal{E}[\![expr_1]\!] h\ d \cup \mathcal{E}[\![expr_2]\!] h\ d \\
&\quad \cup \{\mathsf{read}(\mathsf{instance}\,(expr_1, \mathsf{Element}))\} \\
\mathcal{E}[\![expr\,\mathtt{.}\,field]\!]\ (r, \mathsf{parent}_R, m)\ d &= \mathcal{E}[\![expr]\!](r, \mathsf{parent}_R, m)\ d \cup \{\mathsf{read}(t)\} \\
\text{where } rgn &= \mathsf{lookup}\ r\ (\mathsf{typeof}\ expr)\ field \\
t &= \begin{cases} \mathsf{instance}\,(expr, rgn) & \text{if } rgn \in \mathbf{Regions}_I \\ \mathsf{static}\,rgn & \text{if } rgn \in \mathbf{Regions}_S \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}_{\mathrm{LVal}}[\![var]\!]\ h\ d &= \{\mathsf{write}(\mathsf{local}\ var)\} \\
\mathcal{E}_{\mathrm{LVal}}[\![expr_1\,\mathtt{[}expr_2\mathtt{]}]\!]\ h\ d &= \mathcal{E}[\![expr_1]\!] h\ d \cup \mathcal{E}[\![expr_2]\!] h\ d \\
&\quad \cup \{\mathsf{write}(\mathsf{instance}\,(expr_1, \mathsf{Element}))\} \\
\mathcal{E}_{\mathrm{LVal}}[\![expr\,\mathtt{.}\,field]\!]\ (r, \mathsf{parent}_R, m)\ d &= \mathcal{E}[\![expr]\!](r, \mathsf{parent}_R, m)\ d \cup \{\mathsf{write}(t)\} \\
\text{where } rgn &= \mathsf{lookup}\ r\ (\mathsf{typeof}\ expr)\ field \\
t &= \begin{cases} \mathsf{instance}\,(expr, rgn) & \text{if } rgn \in \mathbf{Regions}_I \\ \mathsf{static}\,rgn & \text{if } rgn \in \mathbf{Regions}_S \end{cases}
\end{aligned}
$$

Method and constructor calls In addition to any effects produced by evaluating the actual parameters and the receiver, method calls also have the effects produced by the method. These are found by first looking up the method being called in the method dictionary (using find, which uses the language specific static method resolution semantics), and then substituting the expression of the actual parameter for the formal in any effects with instance targets. The helper function process creates a vector of type–expression pairs out of the argument list; the types are used in find, while the expressions are used in pair. The function pair is used to create a set of formal–actual parameter pairs. The actual substitutions are made by the function replace.

$$
\begin{aligned}
\mathcal{E}[\![expr\,\mathtt{.}\,method\,\mathtt{(}args\mathtt{)}]\!]\ h\ d &= \mathcal{E}[\![expr]\!] h\ d \cup \mathcal{E}[\![args]\!] h\ d \\
&\quad \cup \mathsf{replace}\ e\ (\mathsf{pair}\ p\ l) \cup \{(\mathtt{this}, expr)\} \\
\text{where } l &= \mathsf{process}\ args \\
(c, method, p, e) &= \mathsf{find}\ d\ (\mathsf{typeof}\ expr)\ method\ l
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\mathtt{new}\ class\,\mathtt{(}args\mathtt{)}]\!]\ h\ d &= \mathcal{E}[\![args]\!] h\ d\ \cup\ \mathsf{replace}\ e\ (\mathsf{pair}\ p\ l) \\
\text{where } l &= \mathsf{process}\ args \\
(class, class, p, e) &= \mathsf{find}\ d\ class\ class\ l
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![constructObj\,\mathtt{(}args\mathtt{);}]\!]\ h\ d &= \mathcal{E}[\![args]\!] h\ d\ \cup\ \mathsf{replace}\ e\ (\mathsf{pair}\ p\ l) \\
\text{where } l &= \mathsf{process}\ args \\
(c, method, p, e) &= \mathsf{find}\ d\ (\mathsf{typeof}\ constructObj)\ (\mathsf{typeof}\ constructObj)\ l
\end{aligned}
$$

$$\begin{aligned}\mathsf{find} &: 2^{\mathbf{Method}} \to \mathbf{Ide} \to \mathbf{Ide} \to (\mathbf{Ide} \times \mathbf{Ide})^* \to \mathbf{Method}\\ \mathsf{find}\ d\ c\ m\ l &= (c', m, p, e) \in d \text{ subject to language's}\\ &\quad\ \text{static method resolution mechanism}\end{aligned}$$

$$\begin{aligned}\mathsf{process} &: \mathit{args} \to (\mathbf{Ide} \times \mathit{expr})^*\\ \mathsf{process}\ \epsilon &= ()\\ \mathsf{process}\ \mathit{expr} &= (\mathsf{typeof}\ \mathit{expr}, \mathit{expr})\\ \mathsf{process}\ \mathit{expr}\ \mathbf{,}\ \mathit{argSeq} &= ((\mathsf{process}\ \mathit{expr}) : (\mathsf{process}\ \mathit{argSeq}))\end{aligned}$$

$$\begin{aligned}\mathsf{pair} &: (\mathbf{Ide} \times \mathbf{Ide})^* \to (\mathbf{Ide} \times \mathit{expr})^* \to 2^{\mathbf{Ide} \times \mathit{expr}}\\ \mathsf{pair}\ ()\ () &= \emptyset\\ \mathsf{pair}\ ((t, n) : p)\ ((t', e) : l) &= \{(n, e)\} \cup \mathsf{pair}\ p\ l\end{aligned}$$

$$\begin{aligned}\mathsf{replace} &: 2^{\mathit{effect}} \to 2^{\mathbf{Ide} \times \mathit{expr}} \to 2^{\mathit{effect}}\\ \mathsf{replace}\ e\ p &= R' \cup W' \cup (e - (R \cup W))\\ \text{where } R &= \{\mathsf{read}((f, r)) \in e \mid (f, a) \in p\}\\ W &= \{\mathsf{write}((f, r)) \in e \mid (f, a) \in p\}\\ R' &= \{\mathsf{read}((a, r)) \mid \mathsf{read}((f, r)) \in R \wedge (f, a) \in p\}\\ W' &= \{\mathsf{write}((a, r)) \mid \mathsf{write}((f, r)) \in W \wedge (f, a) \in p\}\end{aligned}$$

Elaborating Effects and Masking Sometimes we need to expand a set of effects to take into account the bindings of local variables and the mapping of unshared fields. In particular we need this *elaboration* when we check a method annotation.

$$\begin{aligned}\mathsf{elaborate}(E) = &\left\{\mathsf{read}(t) \mid t \in \mathsf{elaborate}(\left\{t'\right\}), \mathsf{read}(t') \in E\right\}\\ \cup &\left\{\mathsf{write}(t) \mid t \in \mathsf{elaborate}(\left\{t'\right\}), \mathsf{write}(t') \in E\right\}\end{aligned}$$

$$\begin{aligned}\mathsf{elaborate}(T) = &\text{ smallest set } T' \supseteq T \text{ such that}\\ &(\mathsf{instance}\,(v, r) \in T', e \in B\,v) \Rightarrow \mathsf{instance}\,(e, r) \in T'\\ &\mathsf{instance}\,(e.f, r) \in T' \Rightarrow \mathsf{instance}\,(e, r') \in T'\\ &\ \text{where } r' = \mathit{min}_{\leq_R}\{r_c \mid (c, f, r_u, r_c) \in m, r \leq_R r_u\}\end{aligned}$$

Effect masking is accomplished by dropping out effects on local variables, regions of local variables (handled in elaboration), regions in newly allocated objects, and regions of unshared fields (which are redundant after elaboration):

$$\begin{aligned}\mathsf{mask}(E) = &\left\{\mathsf{read}(t) \mid t \in \mathsf{mask}(\left\{t'\right\}), \mathsf{read}(t') \in E\right\}\\ \cup &\left\{\mathsf{write}(t) \mid t \in \mathsf{mask}(\left\{t'\right\}), \mathsf{write}(t') \in E\right\}\end{aligned}$$

$$\begin{aligned}\mathsf{mask}(T) = \{t \in T \mid t \text{ not of the forms } &\mathsf{local}\,v \text{ (where } v \text{ is any local or parameter)},\\ &\mathsf{instance}\,(v,r) \text{ (where } v \text{ is a local variable, but not a parameter)}\\ &\mathsf{instance}\,(allocExpr, r), \text{ or } \mathsf{instance}\,(e,r) \text{ (where } \mathsf{unshared}\,e)\}\end{aligned}$$

A.5 Checking Annotated Effects

The annotated effects of a method, E_A, need to be checked against the computed effects of the method, E_C. If E_A does not account for every possible effect of the method, the annotation is invalid. Formally, the annotated effects of a method are valid if and only if $\forall e \in \mathsf{mask}(\mathsf{elaborate}(E_C)).\ \exists e' \in E_A.\ e \leq_E e'$

A.6 Conflicts and Interference

Two effects *conflict* if and only if $\mathsf{read}(t_1)$ is included (by $\leq_E$) in one of the effects, $\mathsf{write}(t_2)$ is included in the other effect, and $t_1\,\mathsf{overlap}_T\,t_2$. This can be extended to sets of effects: E_1 and E_2 conflict if and only if $\exists e_1 \in \mathsf{elaborate}(E_1)$. $\exists e_2 \in \mathsf{elaborate}(E_2)$. e_1 and e_2 conflict. Finally, two computations *interfere* if and only if they have conflicting sets of effects. Target overlap, $\mathsf{overlap}_T$, is the symmetric closure of $\mathsf{overlap}_0$.

$$\mathsf{overlap}_R \subseteq \mathbf{Regions} \times \mathbf{Regions}$$
$$r\,\mathsf{overlap}_R\,r' \Leftrightarrow r \leq_R r' \vee r' \leq_R r$$

$$\mathsf{overlap}_T \subseteq target \times target$$
$$\mathsf{overlap}_T = \mathsf{overlap}_0 \cup \{(t',t) \mid (t,t') \in \mathsf{overlap}_0\}$$

$$\begin{aligned}\mathsf{overlap}_0 &\subseteq target \times target\\ \mathsf{local}\,v\ \mathsf{overlap}_0\ \mathsf{local}\,w &\Leftarrow v = w\\ \mathsf{instance}\,(e,r)\ \mathsf{overlap}_0\ \mathsf{instance}\,(e',r') &\Leftarrow MayEqual\,(e,e') \wedge r\,\mathsf{overlap}_R\,r'\\ \mathsf{instance}\,(e,r)\ \mathsf{overlap}_0\ \mathsf{anyInstance}\,r' &\Leftarrow \mathsf{shared}\,e \wedge r\,\mathsf{overlap}_R\,r'\\ \mathsf{instance}\,(e,r)\ \mathsf{overlap}_0\ \mathsf{static}\,s &\Leftarrow \mathsf{shared}\,e \wedge r\,\mathsf{overlap}_R\,s\\ \mathsf{anyInstance}\,r\ \mathsf{overlap}_0\ \mathsf{anyInstance}\,r' &\Leftarrow r\,\mathsf{overlap}_R\,r'\\ \mathsf{anyInstance}\,r\ \mathsf{overlap}_0\ \mathsf{static}\,s &\Leftarrow r\,\mathsf{overlap}_R\,s\\ \mathsf{static}\,s\ \mathsf{overlap}_0\ \mathsf{static}\,s' &\Leftarrow s\,\mathsf{overlap}_R\,s'\end{aligned}$$

Providing Persistent Objects in Distributed Systems*

Barbara Liskov, Miguel Castro, Liuba Shrira**, and Atul Adya

Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square, Cambridge, MA 02139
{liskov,castro,liuba,adya}@lcs.mit.edu

Abstract. THOR is a persistent object store that provides a powerful programming model. THOR ensures that persistent objects are accessed only by calling their methods and it supports atomic transactions. The result is a system that allows applications to share objects safely across both space and time.
The paper describes how the THOR implementation is able to support this powerful model and yet achieve good performance, even in a wide-area, large-scale distributed environment. It describes the techniques used in THOR to meet the challenge of providing good performance in spite of the need to manage very large numbers of very small objects. In addition, the paper puts the performance of THOR in perspective by showing that it substantially outperforms a system based on memory mapped files, even though that system provides much less functionality than THOR.

1 Introduction

Persistent object stores provide a powerful programming model for modern applications. Objects provide a simple and natural way to model complex data; they are the preferred approach to implementing systems today. A persistent object store decouples the object model from individual programs, effectively providing a persistent heap that can be shared by many different programs: objects in the heap survive and can be used by applications so long as they can be referenced. Such a system can guarantee *safe sharing*, ensuring that all programs that use a shared object use it in accordance with its type (by calling its methods). If in addition the system provides support for atomic transactions, it can guarantee that concurrency and failures are handled properly. Such a platform allows sharing of objects across both space and time: Objects can be shared between applications running now and in the future. Also, objects can be used concurrently by applications running at the same time but at different locations: different processors within a multiprocessor; or different processors within a distributed environment.

* This research was supported in part by DARPA contract DABT63-95-C-005, monitored by Army Fort Huachuca, and in part by DARPA contract N00014-91-J-4136, monitored by the Office of Naval Research. M. Castro is supported by a PRAXIS XXI fellowship.

** Current address: Department of Computer Science, Brandeis University, Waltham, MA 02254

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 230–257, 1999.
© Springer-Verlag Berlin Heidelberg 1999

A long-standing challenge is how to implement this programming model so as to provide good performance. A major impediment to good performance in persistent object stores is the need to cope with large numbers of very small objects. Small objects can lead to overhead at multiple levels in the system and affect the cost of main memory access, cache management, concurrency control and recovery, and disk access.

This paper describes how this challenge is met in the THOR system. THOR provides a persistent object store with type-safe sharing and transactions. Its implementation contains a number of novel techniques that together allow it to perform well even in the most difficult environment: a very large scale, wide-area distributed system. This paper pulls together the entire THOR implementation, explaining how the whole system works.

THOR is implemented as a client/server system in which servers provide persistent storage for objects and applications run at client machines on cached copies of persistent objects. The paper describes the key implementation techniques invented for THOR: the CLOCC concurrency control scheme, which provides object-level concurrency control while minimizing communication between clients and servers; the MOB disk management architecture at servers, which uses the disk efficiently in the presence of very small writes (to individual objects); and the HAC client caching scheme, which provides the high hit rates of an object caching scheme with the low overheads of a page caching scheme. In effect, the THOR implementation takes advantage of small objects to achieve good performance, thus turning a liability into a benefit.

The paper also presents the results of experiments that compare the performance of THOR to that of C++/OS. C++/OS represents a well-known alternative approach to persistence; it uses memory mapped files to provide persistent storage for objects, and a 64-bit architecture to allow addressing of very large address spaces. This system does not, however, provide the functionality of THOR, since it supports neither safe sharing nor atomic transactions. The performance comparison between THOR and C++/OS is interesting because the latter approach is believed to deliver very high performance. However, our results show that THOR substantially outperforms C++/OS in the common cases: when there are misses in the client cache, or when objects are modified.

The rest of this paper is organized as follows. We describe the THOR model in Section 2. Section 3 presents an overview of our implementation architecture. Sections 4, 5, and 6 describe the major components of the implementation. Our performance experiments are described in Section 7, and conclusions are presented in Section 8.

2 Thor

THOR provides a universe of persistent objects. Each object in the universe has a unique identity, a state, and a set of methods; it also has a type that determines its methods and their signatures. The universe is similar to the heap of a strongly-typed language, except that the existence of its objects is not linked to the running of particular programs. The universe has a persistent root object. All objects reachable from the root are persistent; objects that are no longer accessible from the root are garbage collected.

Applications use THOR objects by starting a THOR *session*. Within a session, an application performs a sequence of *transactions*; a new transaction is started each time

the previous one completes. A transaction consists of one or more calls to methods of THOR objects. The application code ends a transaction by requesting a commit or abort. A commit request may fail (causing an abort); if it succeeds, THOR guarantees that the transaction is serialized with respect to all other transactions and that all its modifications to the persistent universe are recorded reliably. If the transaction aborts, it is guaranteed to have no effect and all its modifications are discarded.

THOR objects are implemented using a type-safe language called Theta [LCD+94], [DGLM95]. A different type-safe language could have been used, and we have subsequently provided a version of THOR in which objects are implemented using a subset of Java. A number of issues arise when switching to Java, e.g., what to do if an object that cannot be made persistent, such as a thread in the current application, becomes reachable from the persistent root. These issues are discussed in [Boy98].

Applications that use THOR need not be written in the database language, and in fact can be written in many different languages, including unsafe ones like C and C++. THOR supports such *heterogeneous sharing* by providing a small layer of code called a *veneer*. A veneer consists of a few procedures that the application can call to interact with THOR (e.g., to start up a session or commit a transaction), together with a *stub* for each persistent type; to call a method on a THOR object, the application calls the associated method on a stub object. More information about veneers can be found in [LAC+96,BL94].

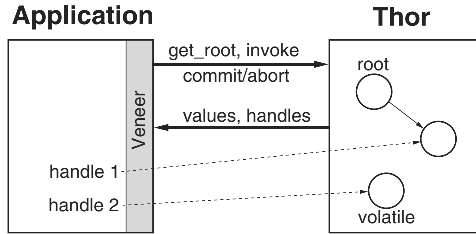


Fig. 1. The THOR Interface

Figure 1 illustrates the THOR interface. Note that THOR objects remain inside THOR; this is an important way in which THOR differs from other object-oriented databases [LLOW91,C+]. Furthermore, the distinction is critical to safe sharing because this way we can ensure that objects are accessed properly, even though the language used in the application may not be type safe.

Theta is a strongly-typed language and therefore once a method call starts running within THOR, any calls it makes will be type safe. However, no similar guarantee exists for calls coming into THOR from the application. Therefore all calls into THOR are type-checked, which is relatively expensive. To reduce the impact of this overhead, each call to THOR should accomplish quite a bit of work. This can be achieved by *code shipping*: portions of the application are pushed into THOR and run on the THOR side of the boundary. Of course such code must first be checked for type-safety; this can be accomplished by verifying the code using a bytecode or assembler verifier [MWCG98,M+99].

Once verified the code can be stored in THOR so that it can be used in the future without further checking. Information about the benefit of code shipping and other optimizations at the THOR boundary can be found in [LACZ96,BL94].

3 Implementation Architecture

The next several sections describe how we achieve good performance. Our implementation requirements were particularly challenging because we wanted a system that would perform well in a large-scale, wide-area, distributed environment. We wanted to support very large object universes, very small objects, very large numbers of sessions running concurrently, and world-wide distribution of the machines concurrently accessing THOR.

Our implementation makes use of a client/server architecture. Persistent objects are stored at servers; each object resides at a particular server, although objects can be moved from one server to another. We keep persistent objects at servers because it is important to provide continuous access to objects and this cannot be ensured when persistent storage is located at client machines (e.g., the machine's owner might turn it off). There can be many servers; in a large system there might be tens of thousands of them. Furthermore, an application might need to use objects stored at many different servers.

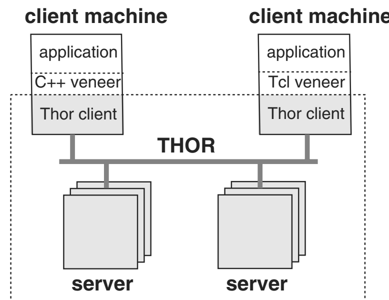


Fig. 2. Architecture of THOR Implementation

Applications run at clients on cached copies of persistent objects. This architecture is desirable because it supports scalability; it reduces the load on servers by offloading work to clients, thus allowing servers to handle more clients. Figure 2 shows the THOR architecture. The figure shows each server with a number of replicas; THOR uses replication to provide highly-available access to persistent objects.

3.1 Object Format

Servers store objects on disk in pages. To simplify cache management, objects are required not to span page boundaries. Pages are large enough that this does not cause

significant internal fragmentation; for example, the average size of objects accessed by most traversals of the OO7 benchmark [CDN93] in THOR is 29 bytes, while our current page size is 8 KB. Objects larger than a page are represented using a tree.

Our design for the format of objects had the goal of keeping objects small; this is important because it has a large impact on performance [WD94,MBMS95]. Our objects are small primarily because object references (or *orefs*) are only 32 bits. Orefs refer to objects at the same server; objects point to objects at other servers indirectly via *surrogates*. A surrogate is a small object that contains the identifier of the target object's server and its oref within that server; this is similar to designs proposed in [Bis77,Mos90,DLMM94]. Surrogates will not impose much penalty in either space or time, assuming the database can be partitioned among servers so that inter-server references are rare and are followed rarely; we believe these are realistic assumptions.

Object headers are also 32 bits. They contain the oref of the object's class object, which contains information such as the number and types of the object's instance variables.

An oref is a pair consisting of a 22-bit *pid* and a 9-bit *oid* (the remaining bit is used at the client as discussed in Section 5.1). The *pid* identifies the object's page and allows fast location of the page both on disk and in the server cache. The *oid* identifies the object within its page but does not encode its location. Instead, a page contains an *offset table* that maps *oids* to 16-bit offsets within the page. The offset table has an entry for each existing object in a page; this 2-byte extra overhead, added to the 4 bytes of the object header, yields a total overhead of 6 bytes per object. The offset table is important because it allows servers to compact objects within their pages independently from other servers and clients, e.g., when doing garbage collection. It also provides a larger address space, allowing servers to store a maximum of 2 G objects consuming a maximum of 32 GB; this size limitation does not unduly restrict servers, since a physical server machine can implement several logical servers.

Our design allows us to address a very large database. For example, a server identifier of 32 bits allows 2^{32} servers and a total database of 2^{67} bytes. However, our server identifiers can be larger than 32 bits; the only impact on the system is that surrogates will be bigger. In contrast, most systems that support large address spaces use very large pointers, e.g., 64-bit [CLFL94,LAC+96], 96 [Kos95], or even 128-bit pointers [WD92]. In Quickstore [WD94], which also uses 32-bit pointers to address large databases, storage compaction at servers is very expensive because all references to an object must be corrected when it is moved (whereas our design makes it easy to avoid fragmentation).

3.2 Implementation Overview

Good performance for a distributed object storage system requires good solutions for client cache management, storage management at servers, and concurrency control for transactions. Furthermore, all of our techniques have to work properly when there are crashes; in particular there must not be any loss of persistent information, or any incorrect processing of transactions when there are failures. Our solutions in these areas are described in subsequent sections. Here we discuss our overall strategy.

When an application requests an object that is not in the client cache, the client fetches that object's page from the server. Fetching an entire page is a good idea because

there is likely to be some locality within a page and therefore other objects on the page are likely to be useful to the client; also page fetches are cheap to process at both clients and servers.

However, an application's use of objects is not completely matched to the way objects are clustered on disk and therefore not all objects on a page are equally useful within an application session. Therefore our caching strategy does not retain entire pages; instead it discards unuseful objects but retains useful ones. This makes the effective size of the client cache much larger and allows us to reduce the number of fetches due to capacity misses. Our cache management strategy, hybrid adaptive caching, or HAC, is discussed in Section 5.

To avoid communication between clients and servers, we use an optimistic concurrency control scheme. This scheme is called *Clock-based Lazy Optimistic Concurrency Control* or CLOCC. The client performs the application transaction and tracks its usage of objects without doing any concurrency control operations; at the commit point, it communicates with the servers, and the servers decide whether a commit is possible. This approach reduces communication between clients and servers to just fetches due to cache misses, and transaction commits. Our concurrency control scheme is discussed in Section 4.

When a transaction commits, we send the new versions of objects it modified to the servers. We cannot send the pages containing those objects, since the client cache does not necessarily contain them, and furthermore, sending entire pages would be expensive since it would result in larger commit messages. Therefore we do *object shipping* at commit time. The server ultimately needs to store these objects back in their containing pages, in order to preserve spatial locality; however, if it accomplished this by immediately reading the containing pages from disk, performance would be poor [OS94]. Therefore we have developed a unique way of managing storage, using a modified object buffer, or MOB, that allows us to defer writing back to disk until a convenient time. The MOB is discussed in Section 6.

In addition to CLOCC, HAC, and the MOB, THOR also provides efficient garbage collection, and it provides support for high availability via replication. Our garbage collection approach partitions the heap into regions that are small enough to remain in main memory while being collected; in addition we record the information about inter-partition references in a way that avoids disk access or allows it to be done in the background. More information about our techniques, and also about how we do distributed collection, can be found in [ML94,ML97b,ML97a,ML97c]. Our replication algorithm is based on that used in the Harp file system [LGG+91,Par98].

4 Concurrency Control

Our approach to concurrency control is very fine-grained; concurrency control is done at the level of objects to avoid false conflicts [CFZ94]. In addition our approach maximizes the benefit of the client cache by avoiding communication between clients and servers for concurrency control purposes.

We avoid communication by using optimistic concurrency control. The client machine runs an application transaction assuming that reads and writes of objects in the

cache are permitted. When the transaction attempts to commit, the client informs a server about the reads and writes done by that transaction, together with the new values of any modified objects. The server determines whether the commit can occur by seeing whether it can serialize the transaction relative to other transactions; if the transaction used objects at multiple servers, a two-phase commit protocol will be used[GR93]. If the commit can occur, the modifications are made persistent; otherwise the client is told to abort the transaction.

When a commit happens, if copies of objects modified by this transaction exist in other client caches, those copies will become stale. Transactions running at those other client machines will be unable to commit if they use the stale objects. Therefore, servers track such modifications and send *invalidations* to the affected clients. An invalidation identifies objects that have been modified since they were sent to the client; the client discards these objects from its cache (if they are still there) and aborts its current transaction if it used them. Invalidations are piggybacked on other messages sent to the client, so that extra communication is not needed for them.

Now we describe in more detail how our concurrency control scheme works. Even more detail can be found in [Gru97,Ady94,AGLM95].

4.1 Clients

As an application transaction runs, the client keeps track of the objects it reads and writes in the ROS (read object set) and MOS (modified object set), respectively; the MOS is always a subset of the ROS (modified objects are entered in both sets). We also store a copy of the current (pre-transaction) state of each modified object in an *undo log*. This tracking is done as part of running methods on the object. The ROS, MOS, and undo log are cleared at the start of each transaction.

If the application requests a commit, the MOS, ROS, and copies of all modified objects are sent to one of the servers that stores objects used by that transaction. The server will either accept or reject the commit. If the commit is rejected, or if the application requests an abort, the client uses the undo log to restore the pre-transaction states of modified objects. The undo log is an optimization; it may be discarded by the cache replacement algorithm. If a transaction aborts after its undo log is discarded, the client invalidates the cached copies of the objects modified by the transaction.

The client also processes invalidation messages. It discards the invalid objects if they are present in the cache, and then uses the MOS and ROS to determine whether the current transaction used any invalid objects. In this case, the transaction is aborted but the states of modified invalidated objects are not restored.

The client notifies servers when it has processed invalidations and when pages are discarded from its cache. This information is sent in the background, piggybacked on other messages (e.g., fetches) that the client is sending to the server.

4.2 Servers

When a server receives a commit request, it assign the transaction a timestamp. This timestamp is obtained by reading the time of the server's clock and concatenating it

with the server's id to obtain a unique number. We assume that server clocks are loosely synchronized to within a few tens of milliseconds of one another. This assumption is not needed for correctness, but improves performance since it allows us to make time-dependent decisions, e.g., we are able to discard old information. The assumption about loosely-synchronized clocks is a reasonable one for current systems [Mil92].

The server then acts as *coordinator* of a two-phase commit protocol; all servers where objects used by the transaction reside act as *participants*. The coordinator sends prepare messages to the participants, which *validate* the transaction by checking locally whether the commit is acceptable; participants then send an acceptance or refusal to the coordinator. If all participants accept, the transaction commits and otherwise it aborts; in either case the coordinator notifies the client about the decision. Then the coordinator carries out a second phase to inform the participants about the decision. (If only one server is involved, we avoid the two-phase protocol entirely, and read-only participants never participate in phase two.)

With this protocol, the client learns of the commit/abort decision after four message delays. In fact we run an optimized protocol in which the client selects the transaction's timestamp and communicates with all participants directly; this reduces the delay to three messages for read/write transactions. For read-only transactions, participants send their decision directly to the client, so that there are just two message delays. Furthermore, we have developed a way to commit read-only transactions entirely at the client almost all the time [Ady99].

Now we discuss how validation works. We use backward validation[Hae84]: the committing transaction is compared with other committed and committing transactions but not with active transactions (since that would require additional communication between clients and servers).

A transaction's timestamp determines its position in the serial order and therefore the system must check whether it can commit the transaction in that position. For it to commit in that position the following conditions must be true:

1. for each object it used (read or modified), it must have used the latest version, i.e., the modification installed by the latest committed transaction that modified that object and that is before it in the serialization order.
2. it must not have modified any object used by a committing or committed transaction that follows it in the serialization order. This is necessary since otherwise we cannot guarantee condition (1) for that later transaction.

A server validates a transaction using a *validation queue*, or VQ, and *invalid sets*. The VQ stores the MOS and ROS for committed and committing transactions. For now we assume that the VQ grows without bound; we discuss how its entries are removed below. If a transaction passes validation, it is entered in the VQ as a committing transaction; if it aborts later it is removed from the VQ while if it commits, its entry in the VQ is marked as committed.

The invalid set lists pending invalidations for a client. As soon as a server knows about a commit of a transaction, it determines what invalidations it needs to send to what clients. It makes this determination using a *directory*, which maps each client to a list of pages that have been sent to it. When a transaction commits, the server adds a modified object to the invalid set for each client that has been sent that object's page;

then it marks the VQ entry for the transaction as committed. As mentioned, information about invalid sets is piggybacked on messages sent to clients. An object is removed from a client's invalid set when the server receives an ack for the invalidation from the client. A page is removed from the page list for the client when the server is informed by the client that it has discarded the page.

A participant validates a transaction as follows. First, it checks whether any objects used by a transaction T are in the invalid set for T's client; if so T must abort because it has used a stale version of some object. Otherwise, the participant does the following VQ checks:

1. For each uncommitted transaction S with an earlier timestamp than T, if S's MOS intersects T's ROS, T must abort. We abort T only if S is uncommitted since otherwise T could be aborted unnecessarily (recall that the check against the invalid set ensures that T read the last versions produced by committed transactions). However, S might commit and if it did we would not be able to commit T since it missed a modification made by S. Of course, S might abort instead, but since this is an unlikely event, we simply abort T immediately rather than waiting to see what happens to S.
2. For each transaction S with later timestamp than T, T must abort if:
   (a) T's MOS intersects S's ROS. T cannot commit because if it did, a later transaction S would have missed its modifications. Again, we abort T even if S has not yet committed because S is highly likely to commit.
   (b) T's ROS intersects S's MOS. If S is committed, the abort is necessary since in this case T has read an update made by a later transaction (we know this since T passed the invalid-set test). If S is uncommitted, we could allow T to commit; however, to ensure that external consistency [Gif83] is also provided, we abort T in this case as well.

   If validation fails because of test 2b (when S is committed) or test 2a, it is possible that T could commit if it had a later timestamp. Therefore, we retry the commit of T with a later timestamp.

We use time to keep the VQ small. We maintain a *threshold timestamp*, VQ.t, and the VQ does not contain any entries for committed transactions whose timestamp is less than VQ.t. An attempt to validate a transaction whose timestamp is less than VQ.t will fail, but such a situation is unlikely because of our use of synchronized clocks. We keep VQ.t below the current time minus some delta that is large enough to make it highly likely that prepare messages for transactions for which this server is a participant will arrive when their timestamp is greater than the threshold. For example, a threshold delta of five to ten minutes would be satisfactory even for a widely-distributed network.

When transactions prepare and commit we write the usual information (the ROS, MOS, and new versions of modified objects) to the transaction log. This is necessary to ensure that effects of committed transactions survive failures. The log can be used during recovery to restore the VQ; communication with clients is necessary to restore the directories, and the invalid sets can then be recovered using information in the log. Recovery of the invalid sets is conservative and might lead to some unnecessary aborts, but failures are rare so that this possibility is not a practical problem.

4.3 Discussion

Our optimistic scheme is efficient in terms of space and processing time because our data structures are small (the VQ and the invalid sets). Directories could be large, but in this case we can simply keep coarser information; the cost would be larger invalid sets, and more invalidations piggybacked on messages sent to the client, but no extra aborts would result.

We have shown that our scheme performs well in practice by comparing its performance to that of other schemes; the results are reported in [AGLM95,Gru97]. In particular we compared its performance to that of adaptive callback locking [CFZ94,ZCF97], which is considered to be the strongest competitor. Our results show that our approach outperforms adaptive callback locking in all reasonable environments and almost all workloads. We do better both on low contention workloads, which are likely to be the common case, and also on high contention workloads. These experiments were performed under simulation [AGLM95,Gru97]; they allowed us to show that our results scale to systems with lots of clients and servers.

When there is high contention, our optimistic scheme has more aborts while a locking scheme has more delay. Our scheme performs well because the cost of aborts is low: we abort early (e.g., when an invalidation arrives at the client), and when we rerun the transaction we are able to run quickly because much of what is needed is already in the client cache, including the previous states of modified objects. Furthermore, most of the extra work due to aborts occurs at the clients, rather than at the servers, which are the scarce resource. Locking schemes have a larger impact on the servers, and the delays due to lock contention can be longer than our delays due to rerunning transactions after aborts.

Our scheme assumes that a transaction's modifications fit in the client cache; we believe this is reasonable for today's machines given the very efficient way we manage the cache (see Sections 5 and 7). Our results apply only to client/server systems in which transactions run at the clients; if transactions ran at servers, the extra work due to aborts would slow down all clients rather than just the client whose transaction aborted, so that optimism may not be the best approach.

5 Hybrid Adaptive Caching

Most persistent object systems manage the client cache using *page caching* [LLOW91], [WD94,SKW92]; such systems fetch and discard entire pages. These systems have low miss penalties because it is simple to fetch and replace fixed-size units. Also, page caching can achieve low miss rates provided clustering of objects into pages is good. However, it is not possible to have good clustering for all application access patterns [TN91,CS89,Day95]. Furthermore, access patterns may evolve over time, and reclustering will lag behind because effective clustering algorithms are very expensive [TN91] and are performed infrequently. Therefore, pages contain both *hot* objects, which are likely to be used by an application in the near future, and *cold* objects, which are not likely to be used soon. Bad clustering, i.e., a low fraction of hot objects per page, causes page caching to waste client cache space on cold objects that happen to reside in the same pages as hot objects.

Object caching systems [Ont92,D$^+$90,LAC$^+$96,C$^+$,TG90,WD92,K$^+$89] allow clients to cache hot objects without caching their containing disk pages and can thus achieve lower miss rates than page caching when clustering is bad. However, object caching has two problems: objects are variable-sized units, which leads to storage fragmentation, and there are many more objects than pages, which leads to high overhead for bookkeeping and for maintaining per-object usage statistics.

Our cache management scheme is called HAC, for hybrid adaptive caching. HAC is a hybrid between page and object caching that combines the virtues of each — low overheads and low miss rates — while avoiding their problems. It partitions the cache between objects and pages adaptively based on the current application behavior: pages in which locality is high remain intact, while only hot objects are retained for pages in which locality is poor. Hybrid object and page caching was introduced in [OS95,Kos95] but this earlier work did not provide solutions to the crucial problems of cache partitioning and storage fragmentation.

HAC partitions the client cache into page-sized *frames* and fetches entire pages from the server. To make room for an incoming page, it

- selects some page frames for *compaction*,
- discards the cold objects in these frames,
- compacts the hot objects to free one of the frames.

The approach is illustrated in Figure 3.

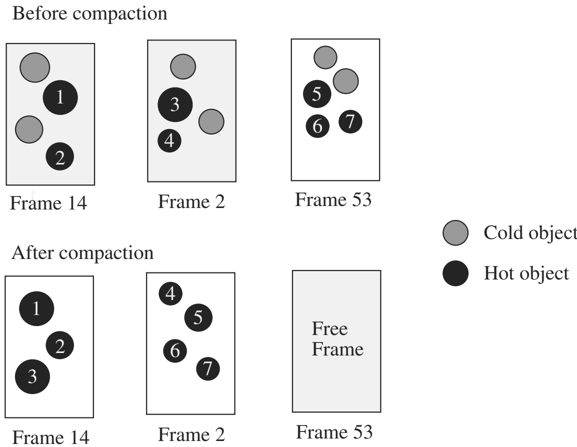


Fig. 3. Compaction of pages by HAC

Now we explain the scheme in more detail; even more information can be found in [CALM97].

5.1 Client Cache Organization

Pages at the client have the same size and structure as at the server to avoid extra copies. However, it is not practical to represent object pointers as *orefs* in the client cache because each pointer dereference would require an expensive computation to obtain the object's memory location. Therefore, clients perform *pointer swizzling* [TG90,Mos92], [WD92], i.e., replace the *orefs* in objects' instance variables by virtual memory pointers to speed up pointer traversals. HAC uses *indirect* pointer swizzling [TG90]; the *oref* is translated to a pointer to an entry in an *indirection table* and the entry points to the target object. (In-cache pointers are 32 bits just like orefs. On 64-bit machines; HAC simply ensures that the cache and the indirection table are located in the lower 2^{32} bytes of the address space.) Indirection allows HAC to move and evict objects from the client cache with low overhead; indirection has also been found to simplify page eviction in a page-caching system [MS95].

Both pointer swizzling and *installation* of objects, i.e., allocating an entry for the object in the indirection table, are performed *lazily*. Pointers are swizzled the first time they are loaded from an instance variable into a register [Mos92,WD92]; the extra bit in the *oref* is used to determine whether a pointer has been swizzled or not. Objects are installed in the indirection table the first time a pointer to them is swizzled. The size of an indirection table entry is 16 bytes. Laziness is important because many objects fetched to the cache are never used, and many pointers are never followed. Furthermore, lazy installation reduces the number of entries in the indirection table, and it is cheaper to evict objects that are not installed.

HAC uses a novel lazy reference counting mechanism to discard entries from the indirection table [CAL97]. The reference count in an entry is incremented whenever a pointer is swizzled and decremented when objects are evicted, but no reference count updates are performed when objects are modified. Instead, reference counts are corrected lazily when a transaction commits, to account for the modifications performed during the transaction.

5.2 Compaction

HAC computes usage information for both objects and frames as described in Section 5.3, and uses this information to select a victim frame V to compact, and also to identify which of $V's$ objects to retain and which to discard. Then it moves retained objects from V into frame T, the current *target* for retained objects, laying them out contiguously to avoid fragmentation. Indirection table entries for retained objects are corrected to point to their new locations; and entries for discarded objects are modified to indicate that the objects are no longer present in the cache. If all retained objects fit in T, the compaction process ends and V can be used to receive the next fetched page. If some retained objects do not fit in T, V becomes the new target and the remaining objects are compacted inside V to make all the available free space contiguous. Then, another frame is selected for compaction and the process is repeated for that frame.

This compaction process preserves locality: retained objects from the same disk page tend to be located close together in the cache. Preserving locality is important

because it takes advantage of any spatial locality that the on-disk clustering algorithm may be able to capture.

When disk page P is fetched, some object o in P may already be in use, cached in frame F. HAC handles this in a simple and efficient way. No processing is performed when P is fetched. Since the copy of o in F is installed in the indirection table, o's copy in P will not be installed or used. If there are many such unused objects in P, its frame will be a likely candidate for compaction, in which case all its uninstalled copies will simply be discarded. If instead F is freed, its copy of o is moved to P (if o is retained) instead of being compacted as usual. In either case, we avoid both extra work and foreground overhead.

5.3 Replacement

Now we discuss how we do page replacement. We track object usage, use this to compute frame usage information from time to time, select frames for removal based on the frame usage information, and retain or discard objects within the selected frame based on how their usage compares to that of their frame. Replacement is done in the background. HAC always maintains a free frame, which is used to store the incoming page. Another frame must be freed before the next fetch, which can be done while the client waits for the fetch response.

Object Usage Computation Our object usage calculation takes both recency and frequency of access into account, since this has been shown to outperform LRU [JS94], [OOW93,RD90], but we do this with very low overhead. Headers of installed objects contain 4 usage bits. The most significant usage bit is set each time a method is invoked on the object. Usage bits are cheap in both space and time; only two extra instructions and no extra processor cache misses are needed to do a method call. They are much cheaper in both space and time than maintaining either an LRU chain or the data structures used in [JS94,OOW93,RD90].

The usage value is decayed periodically by shifting right by one; thus, each usage bit corresponds to a decay period and it is set if the object was accessed in that period. Our scheme considers objects with higher usage (interpreting the usage as a 4-bit integer) as more *valuable*, i.e., objects that were accessed in more recent periods are more valuable and when the last accesses to two objects occurred in the same period, their value is ordered using the history of accesses in previous periods. Therefore, our scheme acts like LRU but with a bias towards protecting objects that were frequently accessed in the recent past. To further increase this bias and to distinguish objects that have been used in the past from objects that have never been used, we add one to the usage bits before shifting; we found experimentally that this increment reduces miss rates by up to 20% in some workloads. We set the usage of invalid objects to 0, which ensures their timely removal from the cache.

Frame Usage Computation We could implement replacement by evicting the objects with the lowest usage in the cache, but this approach may pick objects from a large

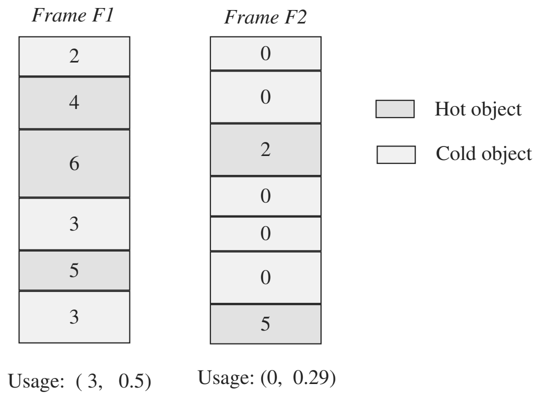


Fig. 4. Usage statistics for frames

number of frames, which means we would need to compact all these frames. Therefore, we compute usage values for frames and use these values to select frames to compact.

Our goals in freeing a frame are to retain hot objects and to free space. The frame usage value reflects these goals. It is a pair $<T, H>$. T is the *threshold*: when the frame is discarded, only *hot* objects, whose usage is greater than T, will be retained. H is the fraction of objects in the frame that are hot at threshold T. We require H to be less than the *retention fraction*, R, where R is a parameter of our system; we have found experimentally that $R = 2/3$ works well. T is the minimum usage value that results in an H that meets this constraint. Frame usage is illustrated (for R = 2/3) in Figure 4. For frame F1, T = 2 would not be sufficient since this would lead to H = 5/6; therefore we have T = 3. For frame F2, T = 0 provides a small enough value for H.

We use object count as an estimate for the amount of space occupied by the objects because it is expensive to compute this quantity accurately; it is a reasonable estimate if the average object size is much smaller than a page.

HAC uses a *no-steal* [GR93] cache management policy: objects that were modified by a transaction cannot be evicted from the cache until the transaction commits. (Objects read by the current transactions, and old versions in the undo log, can be discarded.) The frame usage is adjusted accordingly, to take into account the fact that modified objects are retained regardless of their usage value: when computing the usage of a frame, we use the *maximum usage value* for modified objects rather than their actual usage value.

Selection of Victims The goal for replacement is to free the least valuable frame. Frame F is less valuable than frame G if its usage is lower:

$$F.T < G.T \text{ or } (F.T = G.T \text{ and } F.H < G.H)$$

i.e., either F's hot objects are likely to be less useful than G's, or the hot objects are equally useful but more objects will be evicted from F than from G. For example, in Figure 4, $F2$ has lower usage than $F1$.

Although in theory one could determine the least valuable frame by examining all frames, such an approach would be much too expensive. Therefore, HAC selects the victim from among a *set of candidates*. Frames are added to this set at each fetch; we select frames to add using a variant of the clock algorithm [Cor69]. A frame's usage is computed when it is added to the set; since this computation is expensive, we retain frames in the candidate set, thus increasing the number of candidates for replacement at later fetches without increasing replacement overhead. We remove frames from the candidate set if they survive enough fetches; we have found experimentally that removing frames after 20 fetches works well.

The obvious strategy is to free the lowest-usage frame in the candidate set. However, we modify this strategy a little to support the following important optimization.

As discussed earlier, HAC relies on the use of an indirection table to achieve low cache replacement overhead. Indirection can increase hit times, because each object access may require dereferencing the indirection entry's pointer to the object, and above all, may introduce an extra cache miss. This overhead is reduced by ensuring the following invariant: *an object for which there is a direct pointer in the stack or registers is guaranteed not to move or be evicted.* The Theta compiler takes advantage of this invariant by loading the indirection entry's pointer into a local variable and using it repeatedly without the indirection overhead; other compilers could easily do the same. We ran experiments to evaluate the effect of pinning frames referenced by the stack or registers and found it had a negligible effect on miss rates.

To preserve the above invariant, the client scans the stack and registers and conservatively determines the frames that are being referenced from the stack. It frees the lowest-usage frame in the candidate set that is not accessible from the stack or registers; if several frames have the same usage, the frame added to the candidate set most recently is selected, since its usage information is most accurate.

When a frame fills up with compacted objects, we compute its usage and insert it in the candidate set. This is desirable because objects moved to that frame may have low usage values compared to pages that are currently present in the candidate set.

The fact that the usage information for some candidates is old does not cause valuable objects to be discarded. If an object in the frame being compacted has been used since that frame was added to the candidate set, it will be retained, since its usage is greater than the threshold. At worst, old frame-usage information may cause us to recover less space from that frame than expected.

6 The Modified Object Buffer

Our use of HAC, and also the fact that we discard invalid objects from the client cache, mean that it is impossible for us to send complete pages to the server when a transaction commits. Instead we need to use object shipping. Ultimately, however, the server must write objects back to their containing page in order to preserve clustering. Before writing the objects it is usually necessary to read the containing page from disk, since it is unlikely in a system like ours that the containing page will be in memory when it is needed [MH92]. Such a read is called an *installation read* [OS94]. Doing installation

reads while committing a transaction performs poorly [WD95,OS94] so that some other approach is needed.

Our approach is to use a volatile buffer in which we store recently modified objects; the buffer is called the MOB (for modified object buffer). When modified objects arrive at the server they are stored in the MOB instead of being installed in a page cache. The modifications are written to disk lazily as the MOB fills up and space is required for new modifications. Only at this point are installation reads needed.

The MOB architecture has several advantages over a conventional page buffer. First, it can be used in conjunction with object shipping, and yet installation reads can be avoided when transactions commit. Second, less storage is needed to record modifications in the MOB than to record entire modified pages in a page cache. Therefore the MOB can record the effects of more transactions than a page cache, given the same amount of memory; as a result, information about modifications can stay in the MOB longer than in a page cache. This means that by the time we finally move an object from the MOB to disk, there is a high probability that other modifications have accumulated for its page than in the case of a page cache. We call this effect *write absorption*. Write absorption leads to fewer disk accesses, which can improve system performance.

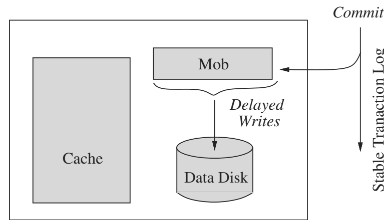


Fig. 5. Server Organization

Now we describe how the MOB works; more information can be found in [Ghe95]. The server contains some volatile memory, disk storage, and a stable transaction log as shown in Figure 5. The disk provides persistent storage for objects; the log records modifications of recently committed transactions. The volatile memory is partitioned into a page cache and a MOB. The page cache holds pages that have been recently fetched by clients; the MOB holds recently modified objects that have not yet been written back into their pages on disk.

Modifications of committed transactions are inserted into the MOB as soon as the commit has been recorded in the log. They are not written to disk immediately. Instead a background *flusher* thread lazily moves modified objects from the MOB to disk. The flusher runs in the background and does not delay commits unless the MOB fills up completely with pending modifications.

The MOB does not replace the transaction log, which is used to ensure that transactions commit properly in spite of failures. After a failure the MOB can be restored from the log. When objects are removed from the MOB, the corresponding records can also be removed from the log.

Pages in the cache are completely up-to-date: they contain the most current versions of their objects. However, pages on disk do not reflect modifications of recent transactions, i.e., modifications in the MOB. Therefore a fetch request is processed as follows: If the needed page is not in the page cache, it is read into the cache and then updated to reflect all the recent modifications of its objects in the MOB. Then the page is sent to the client. When the page is evicted from the cache, it is not written back to disk even if it contains recent modifications; instead we rely on the flusher to move these modifications to disk.

The page cache is managed using a LRU policy, but the MOB is managed in a FIF0 order to allow the log to be truncated in a straightforward way. The flusher scans the MOB and identifies a set of pages that should be written to disk to allow a prefix of the log to be truncated. This set of pages is read into the page cache if necessary (these are the installation reads). Then *all* modifications for those pages are installed into the cached copies and removed from the MOB. Then the pages are written to disk, and finally a prefix of the log is truncated. Therefore the operation of the flusher replaces the checkpointing process used for log truncation in most database systems [GR93]. The log may contain records for modifications that have already been installed on disk but this is not a problem: the modifications will be re-installed if there is a failure (and re-entered in the MOB), but failures are rare so that the extra cost to redo the installs is not an issue and correctness is not compromised since installation is idempotent.

The MOB is implemented in a way that makes both fetches and the flusher run fast. The flusher needs to identify pages to be flushed; it also needs to find all modifications for these pages, and remove them from the MOB without leaving any holes. Fetching requires a quick way to find any modifications for the requested page. We accommodate these requirements by storing the modified objects in a heap that is managed using a standard malloc/free memory allocator. The contents of the heap are chained together in a *scan list* that can be used to identify the pages that need to be written out so that the MOB contents can be discarded in log order. Finally, a hash table maps from page identifiers to the set of objects in the MOB that belong to that page. This hash table is used to handle client fetch requests, and also by the flusher to find the set of all objects belonging to a page that is being written to disk.

Writes to disk are actually done using units called segments, which are larger than pages and correspond roughly in size to a disk sector. Using segments instead of pages allows us to take advantage of disk characteristics. Furthermore, segments are useful from an application perspective since they provide a larger unit for clustering, and when such clustering exists there can be more write absorption using segments than pages. Thus segments improve performance in two ways: by using the disk more efficiently and by improving write absorption.

7 Performance Evaluation

This section evaluates the performance of THOR. It compares THOR with a system, called C++/OS, that does not implement safe sharing or transactions and is directly based on abstractions and mechanisms offered by current operating systems. This sys-

tem does not provide the full functionality of THOR. It provides a form of transparent persistency for objects but it does not support transactions or safe sharing.

The comparison between the two systems is interesting because C++/OS indicates the kind of performance that could be obtained by running a persistent object store directly on current operating systems; also, C++/OS is believed to perform extremely well because the mechanisms it uses have been the focus of extensive research and development and are fairly well optimized. Our results show that THOR significantly outperforms C++/OS in the common case when there are cold or capacity misses in the client cache, or when objects are modified. Furthermore, THOR outperforms C++/OS even though C++/OS implements a much less powerful programming model.

7.1 Experimental Setup

Before presenting the analysis, we describe the experimental setup. Our workloads are based on the OO7 benchmark [CDN94]; this benchmark is intended to match the characteristics of many different CAD/CAM/CASE applications. The OO7 database contains a tree of *assembly* objects, with leaves pointing to three *composite parts* chosen randomly from among 500 such objects. Each composite part contains a graph of *atomic parts* linked by *connection* objects; each atomic part has 3 outgoing connections. All our experiments ran on the *medium* database, which has 200 atomic parts per composite part. The traversals we ran and the environment are described below.

C++/OS The C++/OS system uses a C++ implementation of OO7. This implementation uses a modified memory allocator that creates objects in a memory-mapped file in the client machine. This file is stored by a server and it is accessed by the client operating system using the NFS Version 3 distributed file system protocol.

The operating system manages the client cache by performing replacement at the page granularity (8 KB) and it also ships modified pages back to the server. We added an `msync` call at the end of each traversal to force any buffered modifications back to the server.

C++/OS uses 64-bit object pointers so that more than 4 GB of data can be accessed; this is the same approach taken in [CLFL94]. The result is a database that is 42% larger than THOR's, but the system does not incur any space or time overhead for swizzling pointers.

Environment The objects in the databases in both systems are clustered into 8 KB. In THOR, the database was stored by a replicated server with two replicas and a witness [LGG$^+$91], i.e., a server that can tolerate one fault. Each replica stored the database on a Seagate ST-32171N disk, with a peak transfer rate of 15.2 MB/s, an average read seek time of 9.4 ms, and an average rotational latency of 4.17 ms [Sea97]. The THOR database takes up 38 MB in our implementation. The C++/OS database is also stored on a Seagate ST-32171N disk; it takes up 54 MB since it uses 64-bit pointers.

The database was accessed by a single client. Both the server and client machines were DEC 3000/400 Alpha workstations, each with a 133 MHz Alpha EV4 (21064) processor, 160 MB of memory and Digital Unix. They were connected by a 10 Mb/s

Ethernet and had DEC LANCE Ethernet interfaces. In THOR, each server replica had a 36 MB cache (of which 6 MB were used for the MOB); in C++/OS, the NFS server had 137 MB of cache space. We experimented with various sizes for the client cache.

All the results we report are for *hot* traversals: we preload the caches by running a traversal twice and timing the second run. There are no cold-cache misses in either the server or the client cache during a hot traversal. This is conservative; THOR outperforms C++/OS in cold cache traversals because its database is much smaller.

The C code generated by the Theta compiler for the traversals, the THOR system code, and the C++ implementation of OO7 were all compiled using GNU's gcc with optimization level 2.

Traversals The OO7 benchmark defines several database traversals; these perform a depth-first traversal of the assembly tree and execute an operation on the composite parts referenced by the leaves of this tree. Traversals T1 and T6 are read-only; T1 performs a depth-first traversal of the entire composite part graph, while T6 reads only its *root atomic part*. Traversals T2a and T2b are identical to T1 except that T2a modifies the root atomic part of the graph, while T2b modifies all the atomic parts.

In general, some traversals will match the database clustering well while others will not, and we believe that on average, one cannot expect traversals to use a large fraction of each page. For example, Tsangaris and Naughton [TN91] found it was possible to achieve good average use only by means of impractical and expensive clustering algorithms; an $O(n^{2.4})$ algorithm achieved average use between 17% and 91% depending on the workload, while an $O(n \log n)$ algorithm achieved average use between 15% and 41% on the same workloads. Chang and Katz [CS89] observed that real CAD applications had similar access patterns. Furthermore, it is also expensive to collect the statistics necessary to run good clustering algorithms and to reorganize the objects in the database according to the result of the algorithm [GKM96,MK94]. These high costs bound the achievable frequency of reclusterings and increase the likelihood of mismatches between the current workload and the workload used to train the clustering algorithm; these mismatches can significantly reduce the fraction of a page that is used [TN91].

The OO7 database clustering matches traversal T6 poorly but matches traversals T1, T2a and T2b well; our results show that on average T6 uses only 3% of each page whereas the other traversals use 49%. We only present results for traversals T1, T2a and T2b. Since the performance of THOR relative to a page-based system improves when the clustering does not match the traversal, this underestimates the performance gain afforded by our techniques.

7.2 Read-Only Traversals

This section evaluates the performance of THOR running a hot read-only traversal, T1. It starts by presenting a detailed analysis of the overhead when all the pages accessed by T1 fit in the client cache and there are no cold cache misses. Then, it analyzes the performance for the common case when not all the pages fit in the client cache.

Traversals without cache management Our design includes choices (such as indirection) that penalize performance when all the pages accessed fit in the client cache to improve performance in the common case when they do not; this section shows that the price we pay for these choices is reasonable.

We compare THOR with C++/OS running a T1 traversal of the database. Both systems run with a 55 MB client cache to ensure that there are no client cache misses.

Table 1 shows where the time is spent in THOR. This breakdown was obtained by removing the code corresponding to each line and comparing the elapsed times obtained with and without that code. Therefore, each line accounts not only for the overhead of executing extra instructions but also for the performance degradation caused by code blowup. To reduce the noise caused by conflict misses in the direct-mapped processor caches, we used *cord* and *ftoc*, two Digital Unix utilities that reorder procedures in an executable to reduce conflict misses. We used cord on all THOR executables and on the C++/OS executable.

	T1 (sec)
Exception code	0.86
Concurrency control checks	0.64
Usage statistics	0.53
Residency checks	0.54
Swizzling checks	0.33
Indirection	0.75
C++/OS traversal	4.12
Total (THOR traversal)	7.77

Table 1. Breakdown, Hot T1 Traversal, Without Cache management

The first two lines in the table are not germane to cache management. The *exception code* line shows the cost introduced by code to generate or check for various types of exceptions (e.g., array bounds and integer overflow). This overhead is due to our implementation of the type-safe language Theta [LAC+96]. The *concurrency control checks* line shows what we pay for providing transactions. As we discussed before these features are very important for safe sharing of persistent objects. Since the C++/OS system does not offer these features, it incurs no overhead for them.

The next four lines are related to our cache management scheme: *usage statistics* accounts for the overhead of maintaining per-object usage statistics, *residency checks* refers to the cost of checking indirection table entries to see if the object is in the cache; *swizzling checks* refers to the code that checks if pointers are swizzled when they are loaded from an object; and *indirection* is the cost of accessing objects through the indirection table. The indirection costs were computed by subtracting the elapsed times for the C++/OS traversals from the elapsed times obtained with a THOR front-end executable from which the code corresponding to all the other lines in the table had been removed.

Note that the OO7 traversals exacerbate our overheads, because *usage statistics*, *residency checks* and *indirection* are costs that are incurred once per method call, and methods do very little in these traversals: assuming no stalls due to the memory hierarchy, the average number of cycles per method call in the C++ implementation is only 24 for T1.

Figure 6 presents elapsed time results for the hot T1 traversals without any cache misses. The results show that the overheads introduced by THOR on hit time are quite reasonable; THOR adds an overhead relative to C++/OS of 89% on T1. Furthermore, our results show that this overhead is quickly offset by the benefits of our implementation techniques in the presence of client cache misses or modifications .

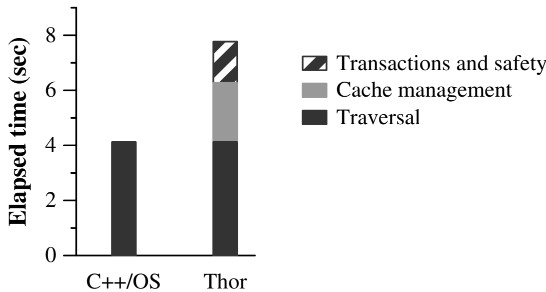


Fig. 6. Elapsed time, Hot T1 traversal, Without Cache Management

Traversals with cache management The previous experiments compared the performance of THOR and C++/OS when the client cache can hold all the pages touched by the traversal. Now we analyze the performance for smaller cache sizes and show that THOR performs better than C++/OS in this region.

Figure 7 shows elapsed times for the hot T1 traversals with cache management. We measured running a hot T1 traversal on both systems while varying the amount of memory devoted to caching at the client. For THOR, this includes the memory used by the indirection table. The amount of memory available to the client in the C++/OS system was restricted by using a separate process that locked pages in physical memory. The elapsed times in THOR do not include the time to commit the transaction. This time was approximately 1.8 seconds.

The performance gains of THOR relative to C++/OS are substantial because it has a much lower miss rate. For example, for a cache size of 18 MB, THOR has 4132 client cache misses whereas C++/OS has 16945. The maximum performance difference between THOR and C++/OS occurs for the minimum cache size at which all the objects used by the traversal fit in the client cache; THOR performs approximately 15 times faster than C++/OS in traversal T1. From another perspective, THOR requires less than half the memory as C++/OS to run traversal T1 without cache management activity.

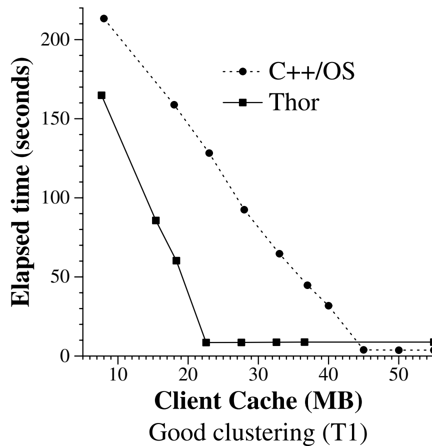


Fig. 7. Elapsed Time, Hot T1 Traversal, With Cache Management

The miss rate is lower because of HAC and because our objects are smaller. HAC allows the client to cache only the objects accessed by the traversal rather than their pages. Since on average T1 uses only 49% of each page, HAC improves performance significantly. The impact is even bigger in transactions that match the database clustering poorly, e.g., HAC improves performance by up to three orders of magnitude in T6 [CALM97].

THOR's swizzling technique and the use of surrogates allow it to use 32-bit pointers, but at the cost of extra computation and extra space to store the indirection table. As discussed previously, the C++/OS system avoids these costs but needs to use 64-bit pointers instead. The result is a space overhead for the database that is two times larger than the space used up by our indirection table during traversal T1.

C++/OS does not incur any disk reads in the experiments reported in this section whereas THOR does; the time to read from disk accounts for approximately 20% of THOR's miss penalty. This happens because we conservatively allowed the NFS server in C++/OS to use a 137 MB cache while the THOR server used a 36 MB cache (from which 6 MB were dedicated to the MOB). THOR's performance relative to C++/OS would have been even better if we had not been conservative. More generally, as discussed in [Bla93], we expect the miss rate in the server cache to be high in real applications. This will lead to high miss penalties and will further increase our performance gains.

7.3 Read-Write Traversals

All experiments presented so far ran traversal T1, which is read-only. This section shows that THOR outperforms C++/OS for traversals with updates even when all the pages accessed fit in the cache. Figure 9 presents elapsed times for hot traversals T2a and T2b running with a 55 MB client cache. For this cache size, there is no cache replacement in either of the two systems.

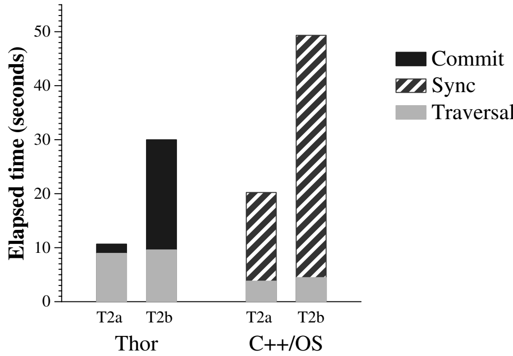


Fig. 8. Elapsed time, Hot Read-Write traversals, Without Cache Management

The results show that THOR outperforms C++/OS. The time to run the traversal is lower in the C++/OS system mainly for the causes discussed in Section 7.2. But the time to commit the modifications to the server is much lower in THOR for two reasons. First, THOR ships only the modified objects back to the server whereas C++/OS ships the modified pages. For example, in T2b, THOR ships back 4.5 MB whereas C++/OS ships back 25.7 MB. Second, C++/OS installs all the modified pages on disk at the server; the THOR server only inserts the modified objects in the MOB and in the stable log (i.e., forces the modified objects to the backup replica). This performance gain can be attributed to the MOB, which enables efficient object shipping by allowing installation reads to be performed in the background as discussed in Section 6.

The small 6 MB MOB used in these experiments is more than sufficient to absorb all the modifications in the OO7 traversals. Therefore, there are no installation reads in these experiments. The detailed study of the MOB architecture in [Ghe95] analyzes workloads with installation reads and shows that this architecture outperforms a page shipping system for almost all workloads. The only exception are the rare workloads that modify almost all the objects they access.

We also ran traversals T2a and T2b in a configuration with cache management. Figure 9 presents elapsed times for a configuration with a 18 MB client cache. The results show that with cache management and modifications THOR significantly outperforms C++/OS. This happens because, in the C++/OS system, the cache replacement mechanism is forced to write modified pages back to the server to make room for missing pages while the traversal is running. This increases the number of page writes relative to what was shown in Figure 8 and these writes are to random locations. For example, C++/OS performs 12845 page writes in T2b with cache management and only 3285 writes without cache management. On the other hand, THOR keeps modified objects in the cache until the transaction commits.

As discussed previously, THOR uses a *no-steal* [GR93] cache management policy: modified objects cannot be evicted from the client cache until the current transaction commits. We claim there is no significant loss in functionality or performance in our system due to the lack of a steal approach; since object caching retains only the modified objects and not their pages, it is unlikely that the cache will fill up. Our claim is

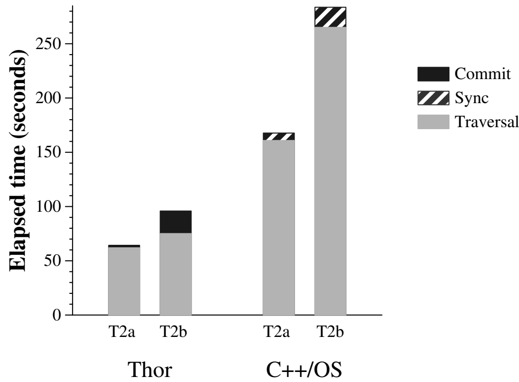


Fig. 9. Elapsed time, Hot Read-Write traversals, With Cache Management (18 MB)

supported by the results presented in Figure 9: HAC allows THOR to run traversal T2b in a single transaction even though this transaction reads and writes an extremely large number of objects (it reads 500000 objects and writes 100000). Evicting modified pages is needed in a page-caching system, since the cache is used much less efficiently.

8 Conclusions

THOR provides a powerful programming model that allows applications to safely share objects across both space and time. It ensures that sharing is safe by allowing objects to be accessed only by calling their methods. In addition, it provides atomic transactions to guarantee that concurrency and failures are handled properly.

The paper describes three implementation techniques that enable THOR to support this powerful model while providing good performance: the CLOCC concurrency control scheme; the HAC client caching scheme; and the MOB disk management architecture at servers. It also presents the results of experiments that compare the performance of THOR to that of C++/OS, which provides persistency using memory mapped files but does not support safe sharing or transactions. The performance comparison between THOR and C++/OS is interesting because the latter approach is believed to deliver very high performance. However, our results show that THOR substantially outperforms C++/OS in the common cases: when there are misses in the client cache, or when objects are modified.

The high performance of THOR is due to all three of its implementation techniques. The MOB provides good performance in modification workloads because it handles small writes efficiently and reduces communication overhead by allowing object shipping at transaction commits. HAC has low overhead and takes advantage of object shipping to reduce capacity misses by managing the client cache at an object granularity. And, although CLOCC does not show up to full advantage in these experiments (since there is no concurrency), our performance benefits from its low overhead and the fact that it avoids introducing extra communication between clients and servers.

We believe these results have important implications for future architectures for supporting persistence. As object systems become the accepted base for building modern distributed applications, the semantic gap between file systems and applications will keep widening. This gap translates into loss of safety and, as indicated by our results, a loss of performance. The paper shows that there is a very attractive alternative: a persistent object storage system that provides both a powerful semantic model and high performance.

Acknowledgements

THOR was the result of the work of many researchers in the Programming Methodology Group at MIT. The following people were instrumental in designing and implementing various parts of THOR: Phillip Bogle, Chandrasekhar Boyapati, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Umesh Maheshwari, Andrew Myers, Tony Ng, Arvind Parthasarathi, Quinton Zondervan.

References

Ady94. A. Adya. Transaction Management for Mobile Objects Using Optimistic Concurrency Control. Master's thesis, Massachusetts Institute of Technology, Jan. 1994. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-626.

Ady99. A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Mar. 1999.

AGLM95. A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, CA, May 1995.

Bis77. P. B. Bishop. Computer Systems with a Very Large Address Space and Garbage Collection. Technical Report MIT/LCS/TR-178, Laboratory for Computer Science, MIT, Cambridge, MA, May 1977.

BL94. P. Bogle and B. Liskov. Reducing Cross-Domain Call Overhead Using Batched Futures. In *Proc. OOPSLA '94*, pages 341–359. ACM Press, 1994.

Bla93. M. Blaze. Caching in Large-Scale Distributed File Systems. Technical Report TR-397-92, Princeton University, January 1993.

Boy98. C. Boyapati. JPS: A Distributed Persistent Java System. Master's thesis, Massachusetts Institute of Technology, Sept. 1998.

C^+. M. J. Carey et al. Shoring Up Persistent Applications. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 383–394, Minneapolis, MN, May 1994. ACM Press.

CAL97. M. Castro, A. Adya, and B. Liskov. Lazy Reference Counting for Transactional Storage Systems. Technical Report MIT-LCS-TM-567, MIT Lab for Computer Science, June 1997.

CALM97. M. Castro, A. Adya, B. Liskov, and A. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 102–115, St. Malo, France, Oct. 1997.

CDN93. M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington D.C., May 1993.

CDN94. M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. Technical Report; Revised Version dated 7/21/1994 1140, University of Wisconsin-Madison, 1994. At `ftp://ftp.cs.wisc.edu/OO7`.

CFZ94. M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 359–370, Minneapolis, MN, June 1994.

CLFL94. J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. In *ACM Transactions on Computer Systems*, volume 12, Feb. 1994.

Cor69. F. J. Corbato. A Paging Experiment with the Multics System*, in Festschrift: In Honor of P. M. Morse*, pages 217–228. MIT Press, 1969.

CS89. W. W. Chang and H. J. Schek. A Signature Access Method for the Starburst Database System. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 145–153, Amsterdam, Netherlands, August 1989.

D^+90. O. Deux et al. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

Day95. M. Day. *Client Cache Management in a Distributed Object Database*. PhD thesis, Massachusetts Institute of Technology, 1995. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-652.

DGLM95. M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In *Proc. OOPSLA '95*, pages 156–168, Austin TX, Oct. 1995. ACM SIGPLAN Notices 30(10).

DLMM94. M. Day, B. Liskov, U. Maheshwari, and A. C. Myers. References to Remote Mobile Objects in Thor. *ACM Letters on Programming Languages and Systems*, Mar. 1994.

Ghe95. S. Ghemawat. *The Modified Object Buffer: a Storage Management Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, 1995. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-656.

Gif83. D. Gifford. Information Storage in a Decentralized Computer System. Technical Report CSL-81-8, Xerox Corporation, March 1983.

GKM96. C. Gerlhof, A. Kemper, and G. Moerkotte. On the Cost of Monitoring and Reorganization of Object Bases for Clustering. *SIGMOD Record*, 25(3):22–27, September 1996.

GR93. J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

Gru97. R. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases*. PhD thesis, M.I.T., Cambridge, MA, 1997.

Hae84. T. Haerder. Observations on Optimistic Concurrency Control Schemes. *Information Systems*, 9(2):111–120, June 1984.

JS94. T. Johnson and D. Shasha. A Low Overhead High Performance Buffer Replacement Algorithm. In *Proceedings of International Conference on Very Large Databases*, pages 439–450, 1994.

K^+89. W. Kim et al. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, June 1989.

Kos95. D. Kossmann. *Efficient Main-Memory Management of Persistent Objects*. Shaker-Verlag, 1995. Dissertation, RWTH Aachen.

LAC+96. B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 318–329, Montreal, Canada, June 1996.

LACZ96. B. Liskov, A. Adya, M. Castro, and Q. Zondervan. Type-safe Heterogenous Sharing Can Be Fast. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, NJ, May 1996.

LCD+94. B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, Feb. 1994. Available at http://www.pmg.lcs.mit.edu/papers/thetaref/.

LGG+91. B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, pages 226–238. ACM Press, 1991.

LLOW91. C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Comm. of the ACM*, 34(10):50–63, October 1991.

M+99. G. Morrisett et al. TALx86: A Realistic Typed Assembly Language. Submitted for publication, 1999.

MBMS95. J. C. Mogul, J. F. Barlett, R. N. Mayo, and A. Srivastava. Performance Implications of Multiple Pointer Sizes. In *USENIX 1995 Tech. Conf. on UNIX and Advanced Computing Systems*, pages 187–200, New Orleans, LA, 1995.

MH92. D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems or Your Cache ain't nothin' but trash. In *Winter Usenix Technical Conference*, 1992.

Mil92. D. L. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. Network Working Report RFC 1305, March 1992.

MK94. W. J. McIver and R. King. Self Adaptive, On-Line Reclustering of Complex Object Data. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 407–418, Minneapolis, MN, May 1994.

ML94. U. Maheshwari and B. Liskov. Fault-Tolerant Distributed Garbage Collection in a Client-Server Object-Oriented Database. In *Third International Conference on Parallel and Distributed Information Systems, Austin*, Sept. 1994.

ML97a. U. Maheshwari and B. Liskov. Collecting Cyclic Distributed Garbage by Controlled Migration. *Distributed Computing*, 10(2):79–86, 1997.

ML97b. U. Maheshwari and B. Liskov. Partitioned Collection of a Large Object Store. In *Proc. of SIGMOD International Conference on Management of Data*, pages 313–323, Tucson, Arizona, May 1997. ACM Press.

ML97c. U. Maheswari and B. Liskov. Collecting Cyclic Distributed Garbage using Back Tracing. In *Proc. of the ACM Symposium on Principles of Distributed Computing*, Santa Barbara, California, Aug. 1997.

Mos90. J. E. B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Office Information Systems*, 8(2):103–139, March 1990.

Mos92. J. E. B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Transactions on Software Engineering*, 18(3), August 1992.

MS95. M. McAuliffe and M. Solomon. A Trace-Based Simulation of Pointer Swizzling Techniques. In *Proc. International Conf. on Data Engineering*, Mar. 1995.

MWCG98. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, Jan. 1998.

Ont92. Ontos. Inc. Ontos reference manual, 1992.

OOW93. E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proc. of ACM SIGMOD International Conference on Management of Data*, Washington, D.C., May 1993.

OS94. J. O'Toole and L. Shrira. Opportunistic Log: Efficient Reads in a Reliable Storage Server. In *Proc. of First Usenix Symposium on Operating Systems Design and Implementation*, pages 119–128. ACM Press, 1994.

OS95. J. O'Toole and L. Shrira. Shared Data Management Needs Adaptive Methods. In *In Proc. of IEEE Workshop on Hot Topics in Operating Systems*, May 1995.

Par98. A. Parthasarathi. The NetLog: An Efficient, Highly Available, Stable Storage Abstraction. Master's thesis, Massachusetts Institute of Technology, June 1998.

RD90. J. Robinson and N. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, 1990.

Sea97. Seagate Technology, Inc. http://www.seagate.com/, 1997.

SKW92. V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An Efficient, Portable Persistent Store. In *5th Int'l Workshop on Persistent Object Systems*, San Miniato, Italy, Sept. 1992.

TG90. K. T. and K. G. *LOOM—Large Object-Oriented Memory for Smalltalk-80 Systems*, pages 298–307. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

TN91. M. Tsangaris and J. Naughton. A stochastic approach for clustering in object bases. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 12–21, Denver, CO, 1991. ACM.

WD92. S. J. White and D. J. Dewitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 419–431, Vancouver, BC, Canada, 1992.

WD94. S. J. White and D. J. Dewitt. Quickstore: A high performance mapped object store. In *SIGMOD '94*, pages 187–198, 1994.

WD95. S. J. White and D. J. Dewitt. Implementing crash recovery in QuickStore: A performance study. In *SIGMOD '95*, pages 187–198. ACM Press, 1995.

ZCF97. M. Zaharioudakis, M. J. Carey, and M. J. Franklin. Adaptive, Fine-Grained Sharing in a Client-Server OODBMS: A Callback-Based Approach. *ACM Transactions on Database Systems*, 22(4):570–627, Dec. 1997.

Inlining of Virtual Methods

David Detlefs and Ole Agesen

Sun Microsystems Laboratories*
1 Network Drive
Burlington, MA 01803-0902, USA
{david.detlefs,ole.agesen}@sun.com

Abstract. We discuss aspects of inlining of virtual method invocations. First, we introduce a new *method test* to guard inlinings of such invocations, with a different set of tradeoffs from the class-equality tests proposed previously in the literature. Second, we consider the problem of inlining virtual methods directly, with no guarding test, in *dynamic* languages such as Self or the Java™ programming language, whose semantics prohibit a static identification of the complete set of modules that comprise a program. In non-dynamic languages, a whole-program analysis might prove the correctness of a direct virtual inlining. In dynamic languages, however, such analyses can be invalidated by later class loading, and must therefore be treated as assumptions whose later violation must cause recompilation. In the past, such systems have required an *on-stack replacement* mechanism to update currently-executing invocations of methods containing invalidated inlinings. This paper presents analyses that allow some virtual calls to be inlined directly, while ensuring that invocations in progress may complete safely even if class loading invalidates the inlining for future invocations. This provides the benefits of direct inlining without the need for on-stack replacement, which can be complicated and require space-consuming data structures.

1 Introduction

One of the most important jobs of a programming language is to provide good semantic abstraction boundaries. One of the most important jobs of a programming language *implementation* is to remove these abstraction boundaries to the extent necessary to permit efficient execution. Methods, one of the distinguishing characteristics of object-oriented languages, are a very important abstraction mechanism. By encapsulating functionality in a method, a programmer leaves open the possibility of reimplementation of the method, adding new functionality without requiring changes at call sites. For example, consider a `Point` class

* Sun, Sun Microsystems, Java, and HotJava are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 258–277, 1999.
© Springer-Verlag Berlin Heidelberg 1999

whose Cartesian coordinates are given in publicly accessible `x` and `y` fields. In some application it is found desirable to obtain the polar coordinates of points. In extending `Point` to accommodate this requirement, we decide to cache the result for efficiency, and only recompute it when the `x` or `y` coordinate changes. Since the fields are public, this requires some editing everywhere `x` or `y` are modified. If, on the other hand, the fields had been private, and public uses had been mediated by `get` and `set` methods, then the recomputation of the polar form would be a simple local change to the `Point` class.

At present, many programmers recognize the wisdom of such observations, but avoid the use of the extra abstraction. For immature programmers the reason may be to avoid some typing, but even mature programmers may avoid abstraction because they are (legitimately) wary of extra cost. Thus, if we are to allow programmers to write in the most modular and robust style without incurring a performance penalty, we must ensure that simple methods execute as fast as if the extra abstraction layer were not present. In practice, this dictates that such methods be inlined.

The above remarks apply generally to most programming languages. Object-oriented languages, such as Simula [3], C++ [26], Modula-3 [22], Smalltalk [14], Eiffel [21], Trellis [24], CLOS [13], and the Java™ programming language [15], to name just a few, complicate inlining, because methods are usually *virtual*. Virtual methods are defined in one class, but may be *overridden* in subclasses of that class. The method actually invoked at a call site depends on the dynamic type of the *receiver* object.

Inlining of virtual methods is difficult because a given call site may invoke several different actual methods over the course of a program execution. Thus, it may be impossible to uniquely identify code to inline. However, in many programs some virtual call sites actually execute only one method, i.e., are *monomorphic* rather than *polymorphic*. Some call sites are provably monomorphic; others are "almost monomorphic," in that several methods might be executed, but one is executed much more frequently than the others.

In the latter situation, a strategy that has been used previously in the literature is to select code to be inlined, then generate a test to guard the inlined code to ensure that it is correct for the dynamic type of the current receiver. If the test fails, the normal virtual call mechanism is used. The guard test is usually a *class test*, verifying that the class of the receiver object matches a particular class. One contribution of this paper is to introduce an alternative test, a *method test* that compares the address of the method inlined with the address of the virtual method to be executed. This incurs the overhead of an extra load, but is more robust, and in some ways more suited to dynamic compilation.

Most analyses that prove a call site monomorphic require interprocedural techniques. Such analyses work best for languages where the full set of components that comprise a program is known statically. Languages with more dynamic semantics make such analyses more difficult. In the Java programming language, for example, `Class.forName` finds and loads a class with a dynamically com-

puted name. Use of this facility complicates interprocedural analyses, since new classes and methods may be added at runtime.

This liability can turn into something of an asset. In an environment with dynamic compilation, we have the freedom to make assumptions based on the code that has been executed so far, and recompile when these assumptions are violated. Thus, if some method invocation `o.m2()` in a calling method `m1` can be bound to a single implementation among the classes loaded at the time `m1` is compiled, we might choose to inline this invocation *directly* (i.e., without a guard test), recording the fact that the compilation of `m1` depends on this assumption about `m2`. If some class that overrides `m2` is loaded later, and an instance of that class could become the receiver `o` in the invocation `o.m2()`, then we must recompile the caller `m1`.

There is a fly in the ointment here: what if an invocation of `m1` is being executed when the assumption about `m2` is violated? Concretely, consider `m1` of the form:

```
void m1() {
  while (true) {
     O o = getSomeO();
     o.m2();    // Call site is inlined.
  }
}
```

In the worst case, `getSomeO` could query the user for the name of a new subclass of `O`, load that class, and create and return an instance. Such a class may of course override `m2`, invalidating the inlining of `m2`.

In the Self system, which did much pioneering work in the field of dynamic compilation [16], this complication is dealt with by a mechanism called *on-stack replacement* [18]. In Self, there are *deoptimization points* within each method, at which the *source state* of the method, the state of the method's variables as defined by the interpretation of the source code, can be recovered from the *machine state* maintained by the compiled code. When a compilation assumption is violated, any method currently in progress must be at such a deoptimization point (or, in a multi-threaded system, reach one within bounded time). The Self system then recovers the source state, recompiles the method without the violated assumption, and also computes from the source state the corresponding machine state at the deoptimization point for the new compilation (which may, of course, have completely different machine state variables.) The new state replaces the old state for the method "on the stack."

There are several reasons to be cautious about such a system. In the Self implementation, the compiler produces voluminous data structures to enable deoptimization.[1] Deoptimization points also introduce constraints on code motion: for example, if a deoptimization point separates two source code writes,

[1] Though, in fairness, deoptimization was also used to enable debugging of optimized code, and was made possible more frequently than would be required for the purposes of this paper.

then they can't be reordered by a code scheduler. Finally, veterans of the Self project recount how much of the complexity of the system is related to deoptimization and reoptimization.

This brings us, finally, to the second contribution of this paper. We present a property of receiver expressions called *preexistence*. Virtual method invocations whose receivers have this property can be inlined directly, and no on-stack replacement mechanism is needed to handle invocations in progress when the assumptions on which the direct inlinings depend are broken. We present two static analysis techniques that can prove preexistence in a significant number of cases. We present measurements indicating the efficacy and costs of these analyses, and the speedups they obtain in an actual Java virtual machine implementation.

2 Related Work

Several techniques have been used to implement virtual calls. The original "Blue Book" implementation of Smalltalk [14] used a hash table mechanism to look up method names at runtime. Object-oriented languages with more constrained type systems were also developed; the type constraints allow more efficient virtual method invocation, using what are called *vtables* in C++ [26]. With this technique, each method is statically assigned an index that is constant across all classes that implement the method. A virtual call jumps indirectly to the code address in the vtable of its receiver object at the index corresponding to the invoked method.

Smalltalk systems pioneered dynamic compilation techniques [8], increasing the efficiency of execution. Thus, the relative cost of method lookup increased, so *inline cache* techniques were developed to speed up calls. An inline cache records the class of the last receiver object observed at the call site, and jumps directly to the implementation of the method for that class. A method prologue validates that the dynamic type of the receiver matches the expected type; if this test fails, a slower method lookup forwards the call and rebinds the call site cache to the class of the current receiver. Hölzle presents a generalized *polymorphic* form of inline caches [17]. It turns out that on some modern architectures the high costs of indirect calls make inline cache techniques attractive even for languages for which vtables could be used [10].

Several systems perform analyses to statically bind virtual calls, allowing them to be inlined or implemented with fast direct calls. Dean et al. use a *class hierarchy analysis* to statically bind virtual calls [7] in the Vortex system [6]. Class hierarchy analysis is a fairly inexpensive process, that, given a complete program, determines when the static type of a receiver implies that an invoked method has only a single implementation in the set of classes used in the program. More expensive *type flow* analyses attempt to tighten the static type constraint on the receiver object, ideally to a set small enough to determine the invoked method. Chambers et al. present such an analysis [1]. Fernandez [12] describes a link-time optimization system, and Diwan et al. [9] describe WPO, a "Whole Program Optimizer." Both systems are for Modula-3 programs and perform both

of kinds of analyses. In addition, several languages, such as Trellis, Dylan [11], and the Java programming language, have linguistic mechanisms that allow the programmer to declare a class *sealed*, that is, ineligible for further subclassing. This enables static binding.

The Self system continues the Smalltalk research path of run-time compilation, adding more aggressive optimization, including extensive method inlining [16], whose correctness is often maintained by the previously mentioned on-stack replacement mechanism. Dean's thesis [4] describes several inlining techniques that use class-based guard tests to ensure correctness. These techniques include efficient subclass tests that offer some of the same benefits as the method test of the present paper. Also, Dean defines the concept of the set of classes sharing the same implementation of a method, but uses this concept only in static analyses, not dynamically to implement a guard test. Vitek, Krall, and Horspool [27] describe data structures that support time- and space-efficient implementations of subtype tests. Some of these techniques require potentially expensive recomputation when new classes are added to the system.

The current work uses contributions from these research directions. The system in which we did our experiments speeds virtual calls with inline caches. The work on different kinds of inlining guards extends the previous work on inlining in dynamic systems. The work on direct inlining of virtuals combines a new static analysis with dynamic recompilation in a novel way.

Another line of related work which the current paper does not explore is *specialization.* The idea is that methods of a class are cloned, so that if method `foo`, invoked on some object of class `C`, calls method `bar` on the same object, it may call the clone of method `bar` for `C` directly, rather than indirectly, and may possibly inline the call. In a variation, a compiler may introduce type tests early in a method, creating two versions of downstream code, one of which assumes the test. Plevyak and Chien [23] describe a whole-program analysis that employs specialization to improve the precision of type-flow analysis. This enables the computation of a precise global control-flow graph from which most method calls can be statically bound. Chambers and Ungar [2] describe the use of *customization* (i.e., specialization on the type of the receiver object) in an early version of Self. Another example of specialization is described by Dean et al. [5]. Specialization is an interesting area of research, but one where a number of competing concerns (execution speed vs. code size, for example) must be balanced, and we did not explore it.

3 Method Tests and Class Tests.

As mentioned in the introduction, most systems that use virtual inlining have guarded the inlined code with a class test verifying that the receiver has a particular class. In pseudo-machine code, the code generated at such an inlined call site could be:

```
r0 := <receiver object>
r1 := load(r0 + <offset-of-class-in-object>)
if (r1 == <address-of-expected-class>) {
   <method inlining>
} else {
   r2 := load(r1 + <offset-of-method-in-class>);
   call r2
}
```

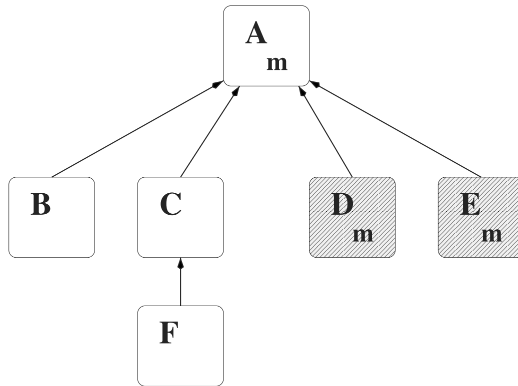


Fig. 1. An inheritance hierarchy

The basic problem with the class test is that it sometimes forces the non-inlined path to be taken unnecessarily. Assume we have the classes shown in figure 1. Class `A` defines a method `m`, classes `B` and `C` each extend `A` without overriding `m`, class `F` extends `C` still without overriding `m`, and classes `D` and `E` extend `A` and *do* override `m`. Consider a call site that invokes the method `m` on a receiver whose static type is `A`. We might wish to inline such an invocation. The inlined code for `A.m` is appropriate when the dynamic type of the receiver is any of `A`, `B`, `C`, or `F`, but incorrect if the dynamic type is `D` or `E`. A single class test will cover only one of the permitted classes `A`, `B`, `C` or `F`. One could convert the test into a disjunction of these possibilities, but only at some cost in speed and code density.

To solve this problem, we have invented an alternate test, called the *method test*, to guard inlined virtual methods. The class test imposes the reasonable requirement that each object contain a pointer to its class information. The method test imposes a further assumption, that the class information includes a vtable. A call site inlined with a method test guard will look like the following:

```
r0 := <receiver object>
r1 := load(r0 + <offset-of-class-in-object>)
r2 := load(r1 + <offset-of-method-in-class>)
if (r2 == <address-of-inlined-method>) {
   <method inlining>
} else {
   call r2
}
```

The method test obtains a pointer to the class information from the object, and then loads, from the class' vtable, the address of the code that would be invoked by an ordinary virtual call. The test compares this code address with the address of the method that was inlined. If they match, it is obviously permissible to execute the inlining. A method test can be used to good effect in the situation described previously. Suppose we decide to inline `A.m` because only classes `A`, `B`, and `C`, which share a single implementation of method `m`, are loaded when the caller is compiled. We can guard the inlining with a method test that covers all three receiver classes. If the overriding classes `D` and `E` are loaded later, the method test correctly chooses not to execute the inlined code for these receiver classes. Further, if the non-overriding class `F` is loaded later, the inlining *will* be executed for this receiver class. It is hard to see how this could be accomplished using class tests. Thus, the method test has two advantages over class tests: an *efficiency* advantage, in cases where one method test succinctly represents the results of several class tests, and a *robustness* advantage, in cases where a method test generated when only some of the classes in the program have been loaded stays effective as more classes are loaded.

The method test has another advantage in the particular case of the Java platform. It is a perhaps seldom-recognized property of the Java virtual machine's bytecode instruction set that a method invocation names the *definition* of the method that is invoked; the static type of the receiver expression, which may be more specific than the class that defines the method, is not apparent in the bytecodes. Consider the following situation:

```
class A {   void m() { ... };  }

class B extends A {            // Overrides m.
  void m() { ... };
}

class C extends A { ... }      // Does not override m

class X {
  void y(C c) { ... c.m(); ... }
}
```

The bytecodes of the method `X.y` indicate only that the invocation `c.m()` invokes the method `A.m`; the more specific type `C` of the receiver can be recovered through

an abstract interpretation similar to that performed in the bytecode verification process [19]. While it may be argued that excellent compilation of JVM bytecodes will have to recover this information, some JIT compilers will want to optimize speed of compilation over quality of code produced, and this analysis will impose some cost.

The missing type information is relevant to the comparison of method tests and class tests because it makes implementation of a useful class test problematical. In the situation above, use of a class test to guard an inlining of `c.m()` would be worse than useless unless we knew the most precise static type of the receiver: a test against `A` would always fail! In systems that do not implement a type-recovery pass, including the system we use, the method test may be used profitably where the class test cannot.

The imprecise type information may also lead to missed inlining opportunities. In the example above, if the compiler knows only that the receiver of `m` is an `A`, it would conclude that two methods might be invoked, and possibly reject the invocation `c.m()` as a candidate for inlining. However, if the precise static type of the receiver were known, the compiler would know that the invocation `c.m()` always invokes `A`'s definition of `m`, and would consider the call site a better candidate for inlining.

The method test has one obvious disadvantage when compared to the class test: an extra dependent load on the fast path. Our colleague Alex Garthwaite notes that instruction scheduling might ameliorate this problem. If several arguments to a method are being computed and moved into argument registers, the two loads required by the method test can often be separated.

3.1 Method and Class Test Measurements

We now present some measurements of the efficacy of virtual inlining in our system. Our experiments were performed in a version of the Sun Java® 2 SDK for the Solaris™ operating system. This environment includes a JIT compiler that performs inlining. Table 1 shows the programs for which we report measurements. For each benchmark, we report lines of source code (including comments and other non-code, excluding class libraries), and give a brief description. The first seven benchmarks constitute the SPECjvm98 suite [25], and the next three benchmarks were candidates considered for inclusion in SPECjvm98, but were not chosen for the final suite. The last two benchmarks are derived from real applications. The VolanoMark benchmark (**volano** in the tables) consists of a version of Volano LLC's VolanoChat internet chat server [20], and a client program that provides a simulated workload. We ran the client and server on the same machine, and added together measurements for both programs. The **portBOB** benchmark was created by IBM to predict the performance of object databases written in the Java language. We modified this program to make it run deterministically.

The measurements are performed on a Sun Ultra™ 1 Creator 3D desktop, which has a 167 MHz UltraSPARC™ processor and 512 megabytes of memory.

benchmark	lines of code	description
mtrt	3799	Multi-threaded image rendering
jess	10579	Version of NASA's CLIPS expert system shell
db	1028	Search and modify a database
jack	8194	Parser generator generating itself
mpegaudio	n/a	Decompress audio file
javac	25211	Source to bytecode compiler
compress	927	LZW compression and decompression
richards	3637	Threads running five versions of OS simulator
tsgp	894	Genetic program for traveling salesman problem
si	1707	Interpreter for a simple language
volano	13811	Internet chat server
portBOB	n/a	Transaction processing benchmark

Table 1. Descriptions of Benchmark Programs.

The compiler only inlines methods that have straight-line control flow, contain no exception handlers or `try...finally` constructs, and are shorter than some maximum length (15 bytecode instructions). Recursive inlining halves the length limit to bound code expansion. Inlining of virtual method invocations further requires the call site to be monomorphic with respect to the classes loaded at the time of compilation. We currently do not use on- or off-line profile information to determine when one method dominates at a polymorphic call site. We also do not inline invocations of methods of interface types, though this would be a straightforward extension.

We measured three system configurations for each program. The first, our baseline system, does no inlining of virtual methods. It uses an inline cache scheme to speed up virtual invocations, so this configuration is no straw man. The second configuration uses the method test. The third *hybrid* configuration combines method tests and class tests: a virtual invocation `o.m()` is inlined with a class test when the defining class of `m` is a *leaf class*, one with no subclasses (at the time of compilation), and with a method test otherwise. If no subclasses of the defining class are loaded later, the class test should be as accurate as a method test and somewhat cheaper. However, if subclasses are loaded later, the test may become inaccurate. This hybrid configuration gives us some efficiency/accuracy comparison of class and method tests.

For each benchmark program, table 2 details the number of virtual call sites inlined, the fraction that were inlined with a class test in hybrid mode, the number of executions of all inlined call sites, and the number of invocations that took the non-inline path using the pure method test and using the hybrid strategy. The benchmarks are ordered by number of executions of inlined virtual call sites.

We can make several remarks about the data in table 2. First, the number of inlined virtual sites varies considerably, by more than an order of magnitude, from program to program, but the number of executions of those sites varies

benchmark	inlined virtual call sites	hybrid class-test %	executions *millions*	method-test non-inline execs	hybrid non-inline execs
mtrt	781	30.9	235	0	0
richards	244	41.8	232	0	0
portBOB	359	39.0	18.7	0	2
tsgp	58	51.7	13.2	0	0
jess	211	46.9	8.8	0	0
si	63	54.0	6.3	0	0
db	69	37.7	5.8	0	0
jack	148	27.0	4.3	0	0
mpegaudio	73	31.5	3.2	0	0
volano	329	53.8	2.0	0	6
javac	315	19.0	1.8	412	412
compress	55	47.3	0.001	0	0

Table 2. Comparison of class and method tests for virtual inlining.

much more, by six orders of magnitude. The importance of the efficiency of virtual inlining tests is magnified by this second number, so we expect to see larger effects for the programs at the top of this table. Second, the fraction of call sites at which the class test is used in hybrid mode varies, averaging a little less than half of inlined call sites for these programs. Third, both kinds of tests are very accurate for these programs. The hybrid test failed a handful of times in **portBOB** and **volano**; the method test failed only in **javac**, and the hybrid test failed identically for that program.

Table 3 gives the performance results for the three configurations. For each benchmark and configuration we give user time and instruction count. The former is measured by the operating system, and the latter by accessing on-chip performance counters. For each of these measurements, we give the percentage difference with respect to no virtual inlining for the two virtual inlining modes. Finally, we also measure JIT compilation time and resulting code size. We omit compilation time for **volano**, since the server and client processes were running concurrently, distorting this measurement. The user time measurements should be treated with some skepticism, since they can vary significantly from run to run because of random factors such as instruction cache placement. Instruction counts are usually, if not always, a good predictor of elapsed time, and have the virtue of being highly repeatable, so that small differences indicate real effects.

The most important conclusion to draw from table 3 is that virtual inlining with either flavor of test is sometimes quite effective, and in no case decreases performance. Again, the benchmarks are ordered by frequency of execution of inlined virtual call sites. As one might expect, the performance improvement correlates well with this measure. Virtual inlining adds some compilation cost in most cases, but nothing we consider extreme. Similarly, virtual inlining adds a small increment in compiled code footprint. Since the previous measurements

indicated high accuracy for both tests, one would expect the hybrid test, which is less expensive when the class test is used, to perform better. This expectation is borne out by the measurements, but the magnitude of the difference is fairly small.

benchmark	virtual inlining	user time *(sec)*	percent diff.	instructions *(millions)*	percent diff.	compilation *(ms)*	code size *(KBytes)*
mtrt	none	52.8		5731		631	352
	method	43.4	-17.8	4595	-19.8	797	384
	hybrid	41.8	-20.8	4439	-22.5	749	382
richards	none	92.2		9645		846	377
	method	72.0	-21.9	9829	-8.4	688	371
	hybrid	69.6	-24.5	8697	-9.8	837	370
portBOB	none	76.5		6156		899	510
	method	76.9	0.5	6122	-0.6	946	515
	hybrid	76.6	0.1	6093	-1.0	930	514
tsgp	none	209.2		29408		434	244
	method	208.4	-0.4	29305	-0.3	432	246
	hybrid	208.1	-0.5	29279	-0.4	435	246
jess	none	39.1		4072		758	410
	method	36.4	-6.9	4034	-0.9	769	416
	hybrid	36.3	-7.2	4031	-1.0	768	416
si	none	93.0		9805		510	290
	method	90.1	-3.1	9776	-0.3	518	291
	hybrid	89.5	-3.8	9764	-0.4	512	290
db	none	144.8		8475		449	263
	method	142.3	-1.7	8473	0.0	453	264
	hybrid	141.8	-2.1	8473	0.0	454	264
jack	none	40.9		4161		793	470
	method	41.5	1.5	4141	-0.5	800	476
	hybrid	41.4	1.2	4141	-0.5	799	476
mpegaudio	none	100.7		13155		647	332
	method	106.3	5.6	13146	-0.1	633	333
	hybrid	108.6	7.8	13146	-0.1	632	333
volano	none	296.7		16861		n/a	756
	method	295.5	-0.7	16861	0.0	n/a	758
	hybrid	297.2	0.2	16849	-0.1	n/a	757
javac	none	70.5		8143		1424	793
	method	72.4	2.7	8129	-0.2	1453	815
	hybrid	72.0	2.1	8128	-0.2	1456	814
compress	none	81.4		10268		448	258
	method	77.4	-4.9	10267	0.0	450	258
	hybrid	77.5	-4.8	10268	0.0	450	258

Table 3. Performance comparison of class and method tests for virtual inlining

A question is raised by these measurements: if the hybrid test performs better than the method test, is the "method" part actually responsible for any benefit? That is, could we get the same performance improvements from a purely class-test-based system? We can run the system in two class test configurations to help answer this. In the first, *class-def*, we inline all virtual sites with a class test on the class that defines the method. This test will fail when the receiver class is a subclass of the defining class. In the second, *class-leaf*, we inline the virtual site with a class test only if the defining class has no subclasses (at the time of compilation). This test will fail less often, but fewer sites will be inlined.

We run these configurations on four of the benchmarks: **mtrt**, **richards**, and **portBOB**, because they are most affected by virtual inlining, and **javac**, which had some failed tests. Table 4 is similar to table 2. It records the number of call sites inlined, the number of executions of these call sites, and the percentage of failed tests for each configuration. We see that quite a substantial fraction of class tests fail when the class-def configuration is used, and many fewer sites are inlined when class-leaf is used. For concurrent programs such as **richards**, behavior can vary from run to run; for example, the number of inlined call sites differs slightly from the number in table 2.

benchmark	inlining mode	inlined virtual call sites	executions *millions*	failed tests
mtrt	class-def	781	243	27.1 %
	class-leaf	241	76.9	0.0%
richards	class-def	250	232	73.7%
	class-leaf	102	64.9	0.0%
portBOB	class-def	359	18.7	76.5%
	class-leaf	140	4.4	0.0%
javac	class-def	315	1.9	54.6%
	class-leaf	60	0.3	0.0%

Table 4. Two class tests for virtual inlining.

Table 5 is similar to table 3, and compares the performance of these two class test configurations with the baseline of no virtual inlining at all. For **mtrt**, the class-def configuration does almost as well as the method or hybrid tests (see table 3), but the class-leaf configuration does substantially worse. For **richards**, both configurations do poorly; in fact, class-def, because of the high failure rate of the test, does worse (in instruction counts) than no inlining at all.

These measurements show that class tests can obtain a significant fraction of the potential performance increase from inlining virtuals for some programs (like **mtrt**), but for others (like **richards**), the inaccuracy of such tests makes at least some method tests necessary to get significant benefit.

It might seem that the data we have presented argues unequivocally for the hybrid test. However, that is not so clear: whenever a call site is inlined with a class test, future class loading may render the test inaccurate in cases where the method test would continue to execute the inlined code.

4 Possible Benefit of Direct Virtual Inlining

The measurements in the last section indicate that inlining of selected virtual calls with a method test guard can have significant benefits for some programs. They also suggest an experiment. For all programs except **javac**, the method

benchmark	virtual inlining	user time *(sec)*	percent diff.	instructions *(millions)*	percent diff.
mtrt	none	52.6		5626	
	class-def	43.9	-16.5	4752	-15.5
	class-leaf	50.0	-4.9	5169	-8.1
richards	none	93.7		9644	
	class-def	88.3	-5.8	9741	1.0
	class-leaf	89.5	-4.5	9184	-4.8
portBOB	none	76.7		6173	
	class-def	75.6	-1.4	6161	-0.2
	class-leaf	75.0	-2.2	6107	-1.1
javac	none	70.9		8142	
	class-def	72.3	2.0	8149	0.1
	class-leaf	71.1	0.3	8146	0.0

Table 5. Performance comparison of class tests for virtual inlining

test never failed, so we altered the compiler to directly inline virtual calls. While incorrect in general, this altered compiler produces code that executes correctly for these programs. This experiment determines an upper bound on how much further efficiency can be gained by schemes that inline virtual invocations directly. Table 6 shows the results of this experiment, for the four benchmarks out of the set where direct inlining results in a noticeable improvement. We ran each benchmark twice, once using the method test, and once inlining virtual calls directly. For each run, we show user time, instruction count, and percentage differences between the two runs.

benchmark	virtual inlining	user time *(sec)*	percent diff.	instructions *(millions)*	percent diff.
mtrt	method	43.7		4693	
	direct	27.5	-38.7	2280	-51.4
richards	method	71.4		8829	
	direct	62.1	-13.0	6838	-22.6
portBOB	method	76.0		6095	
	direct	73.1	-3.8	5944	-2.5
jess	method	36.5		4034	
	direct	37.0	1.4	3956	-1.9

Table 6. Potential benefit of direct virtual inlining

The comparison in table 6 shows that there are programs for which direct virtual inlining yields significantly better performance. Direct inlining both eliminates the overhead of a guard test and enables additional optimizations because

the inlined code can be assumed always executed, instead of just possibly executed.

5 Avoiding On-Stack Replacement: Preexistence

Section 4 presented an experiment that quantified the potential benefits of direct inlining of virtual calls, using a system known to be incorrect in general but safely applicable to the particular programs tested. In the next sections, we discuss techniques for safe direct inlining of virtual calls, without requiring an on-stack replacement mechanism.

Assume that we only inline virtual invocations directly when the called method has only a single implementation at the time of compilation of the caller. Assume further that we record a dependency of the caller on this *single-implementation assumption*, as is done in Self. It is easy to guarantee that any such assumptions made by a method are true on entry to the method—we simply recompile the method without performing the direct inlining, and thus without the assumption, before we allow the assumption to be violated. In the case of direct inlining, the associated single-implementation assumption is violated by the loading of a class that provides a second implementation of the inlined method. Thus, we augment the class loading process to detect violations of single-implementation assumptions, and to recompile the effected methods, or revert them to interpretation, before making the loaded class available.

The problem remains that even when a single-implementation assumption is satisfied on entry to a method, it could become invalid during the method's execution. This problem prompted the invention of on-stack replacement. Now we propose an alternative solution.

Consider a method `m1` that contains a method invocation `o.m2()`. We say that the receiver expression `o` is *preexisting in m1* when the object denoted by `o` was allocated before execution of the calling method `m1` began. (When `m1` is clear from context, we will simply say that `o` is preexisting.) Because `o` is allocated prior to the start of an execution of `m1`, the dynamic type of `o` is clearly in the set S of classes extant in the system at the start of `m1`. If a single-implementation assumption about `m2` is true over the classes in S, and the dynamic type of `o` is in S, then even as the total set of classes in the system grows, the type of the preexisting receiver `o` remains in the set S for the duration of the execution of `m1`. Thus, if `m2` has a single implementation in S, it can be inlined directly without a guard test.

If we directly inline virtual calls only when the receiver is preexisting, then when some class loading operation provides a second implementation of `m2` that necessitates a recompilation of `m1`, as described above, we can allow any in-progress invocations of the original compilation of `m1` to continue to execute the original code without risking incorrect behavior.

6 Proving Preexistence

The concept of preexistence is useful only if it is possible to prove that receiver expressions denote preexisting objects at a significant fraction of call sites, and do so at low cost (always an issue in dynamic compilation systems). Our experience shows that this is possible. We present two analyses for proving preexistence. The first is quite simple, and gets a surprisingly large fraction of the possible benefit. The second is more complicated, and adds little benefit in the programs we tried; still, it illustrates the generality of the approach, and it may be that other programs benefit significantly.

6.1 Invariant Argument Analysis

The first technique, *invariant argument analysis*, examines a method to identify input arguments to which no assignments are made. Clearly, such constant arguments refer to objects that were allocated before the method began executing, i.e., are preexisting. A slightly more ambitious analysis tracks the values of such arguments, rather than simply pattern matching on their textual occurrences, to prove that the receiver expression in

```
void m1(Foo f) {
   Foo f2;
   ...
   f2 = f;
   f2.m2();   // f2 is preexisting.
}
```

is preexisting.[2]

In many object-oriented languages, the receiver argument of a method is passed with a different syntax than is used for other arguments to the method, e.g., `o.m(...)` instead of `m(o, ...)`, and members of the current object may be accessed with a syntax different from that used to access members of other objects. All such syntactic sugars are irrelevant to this analysis.

6.2 Effectiveness of Invariant Argument Analysis

We augmented the compiler in our system to prove preexistence using invariant argument analysis. As before, a virtual call site with a single implementation may be inlined, but now, when the receiver is preexisting, the call is inlined directly. The compiler records a dependency linking the validity of the compiled code

[2] Note that this observation could be extended to the `clone` operation of the Java programming language: if we clone an object we can prove preexisting through other means, we obtain an object which, while *not* preexisting as defined in this paper, still has a dynamic type that is a member of the set of classes extant at the start of the calling method, which is what direct inlining requires.

to the assumption of single implementation of the called method, as discussed previously.

Table 7 shows the fraction of inlined virtual call sites at which invariant argument analysis proves the receiver to be preexisting, as well as the number of recompilations caused by dependency violations over the course of the run. Table 8 compares the performance of the system using this analysis with method-test virtual inlining. In both tables, we present results for the four benchmarks of table 6, the ones for which direct virtual inlining has a potential benefit. We also include **javac**, which, because it had method test failures, could not be included in table 6. Finally, table 7 includes a run of the HotJava^{TM} browser accessing the ECOOP home page. Unlike the SPEC benchmarks, this is an interactive program, and compiles considerably more code. The interactive nature of this program complicates performance measurements, but we include it here to see if recompilation will be a more significant issue for interactive programs, which use the deeper class hierarchies common to user-interface toolkits.

benchmark	inlined virtual call sites	sites with preexisting receiver	percentage	recompilations
mtrt	784	324	41.3	0
richards	246	154	62.5	0
portBOB	359	166	46.2	0
jess	211	54	25.6	0
javac	323	218	67.5	1
hotjava	1548	723	46.7	40

Table 7. Effectiveness of invariant argument analysis, by call sites

benchmark	virtual inlining	user time *(sec)*	percent diff.	instructions *(millions)*	percent diff.	compilation *(ms)*	code size *(KBytes)*
mtrt	method	43.6		4697		893	383
	preexist	32.2	-26.1	3227	-31.3	684	364
richards	method	72.0		8829		795	371
	preexist	62.4	-13.3	7279	-17.6	619	362
portBOB	method	76.7		6115		983	515
	preexist	75.4	-1.7	5978	-2.2	920	505
jess	method	36.5		4034		810	417
	preexist	37.7	3.3	4010	-0.6	775	413
javac	method	72.2		8129		1461	815
	preexist	72.6	0.6	8114	-0.2	1469	801

Table 8. Performance of direct virtual inlining via invariant argument analysis

Table 7 shows that, for these programs, approximately half of virtual call sites are proven to have preexisting receivers by invariant argument analysis.

(The average over the complete benchmark set is 40.9%.) The **javac** benchmark is the only SPEC benchmark that performs recompilations, and performs just one. The HotJava browser performed 40 recompilations, but it compiled 2269 methods in total, so this is a relatively small fraction.

A comparison of table 6 and table 8 shows that a surprisingly large fraction of the potential performance gains from direct virtual inlining can be obtained by using invariant argument analysis to prove receiver object preexistence. In the case of **mtrt**, we obtained (roughly) a 30% speedup out of a possible 50%. Thus, with the caveat that we are extrapolating from a small number of benchmarks, it seems possible to obtain many the benefits of direct virtual inlining without making "closed-world" assumptions, and without implementing an on-stack replacement mechanism. Further, the dependency mechanism does not seem to trigger an excessive number of recompilations.

6.3 Immutable Field Analysis

To understand the second technique for proving preexistence, consider a virtual method invocation of the form `o.f.m()`. That is, the receiver expression is of the form `o.f`, where `o` is an expression of static type `O`, and `f` is a field of that class. Call a field `f` *immutable* if it is assigned to only in constructors. If `f` is proven immutable, and one has access to a fully-constructed instance of a class in which the field `f` appears, then one can assume that `f` will not change subsequently.[3] Consequently, preexistence of `o.f` follows from preexistence of `o` and immutability of `f`. (We assume that "`o` is preexisting" is taken to mean not only that `o` is allocated, but also that it is constructed—this is an easy consequence of the rules of the C++ and Java programming languages.)

In our implementation, we attempt to prove immutability only of private fields of classes, fields that cannot be accessed by methods of other classes. In this case, the analysis is easy and efficient: each method of the class that declares the private field is searched for assignments to that field; the field is immutable if such assignments are found only in constructors. (A refinement of this analysis identifies private methods that are called only from constructors, and treats field assignments in such methods as if they occur in constructors.)

It is easy to see why immutable field analysis for non-private fields would not be useful for proving preexistence. The proof that the preexistence of `o.f` follows from the preexistence of `o` and the immutability of `f` requires a *proof* of both of the latter properties, not an *assumption* that might later be violated. In systems that cannot make closed-world assumptions, even if one examines all the classes extant to determine that a non-private field `f` is immutable "so far," a class might be loaded later that writes to `f`. Thus, even if an expression `o` is preexisting and a non-private field `f` is immutable "so far," modifications to `f` by methods loaded later may make `o.f` denote a non-preexisting value.

It is sometimes suggested that the reflection API of the Java platform invalidates this analysis, by allowing classes to modify private fields of other classes.

[3] At least in languages in which constructors are executed only once on a given object.

We believe that the most current specification of the reflection API allows an implementation to prevent such modifications by enforcing access constraints. An implementation whose compiler uses immutable field analysis would have to implement reflection in this manner. The Java Native Interface, on the other hand, does allow such uncontrolled updating. Our view is that since native code can do *anything,* authors of native code are on their honor to respect access constraints. Even if native code respects privacy, the inability to analyze native code requires immutable field analysis to assume that native methods update all private fields of their class.

6.4 Effectiveness of Immutable Field Analysis

We have implemented immutable field analysis, but will only summarize performance results. There are a noticeable number of call sites at which immutable field analysis can prove preexistence but invariant argument analysis cannot. The fraction of such call sites varies from 1.2% to 21.3% of all inlined virtual call sites, averaging 9.5%. However, this increase in the number of call sites directly inlined does not translate, at least for these benchmarks, into significantly better performance. The **mtrt** benchmark's instruction count decreases by 1.4%, and **portBOB**'s by 0.4%, but no other benchmark's performance changes appreciably. Compilation time does not increase noticeably from the cost of the analysis. Despite these somewhat disappointing end-to-end results, the increase in the fraction of call sites inlined causes us to believe that immutable field analysis may be important for some programs; if immutable field analysis enables direct inlining in a frequently-executed inner loop of some program, it could have a large effect on that program.

7 Conclusions and Future Work

There are two main contributions in this paper. First, we presented an alternative test for guarding inlining of virtual invocations. This method test is less efficient than a class test when there is only one possible receiver type, but more efficient when several possible receiver types can be covered in a single test. It may also be more robust than class tests, since a method test allows execution of inlined code even for receiver types that weren't loaded when the test was generated. Finally, method tests may be more convenient to generate in some some systems, such as just-in-time compilers for Java virtual machines.

The second contribution allows the direct inlining of some virtual calls without an on-stack replacement mechanism. The scheme restricts direct inlining to call sites whose receivers have a preexistence property, and records single-implementation assumptions on which correctness of the inlining depends. Callers must be recompiled when these assumptions are invalidated, but currently-executing invocations of such callers may continue, by virtue of the preexistence restriction. We described two analyses, with different cost/precision tradeoffs, that prove preexistence.

We presented several measurements. Even though the base system against which comparisons were made included an inline cache mechanism to speed up virtual calls, inlining of virtuals with a method test guard offers significant improvements for programs that execute inlined virtual calls frequently. We determined an upper bound on the speedup available by inlining all eligible virtual calls directly. Our simplest analysis, invariant argument analysis, allowed sufficient direct inlining to realize a large fraction of this potential speedup. Immutable field analysis increased the fraction of call sites inlined directly, but did not improve the overall performance significantly for the programs measured.

In future work, we hope to eliminate some of the restrictions our compiler imposes on what may be inlined. The resulting increase in inlining opportunities may extend the benefits of virtual inlining to more programs. We also hope to investigate the application of preexistence to other assumptions that compilers may make to enable optimizations.

Acknowledgments

We would like to thank Christine Flood and Steve Heller for careful readings of the paper. Craig Chambers and Jeff Dean helped clarify the relationship between their work and our's. Thanks also to the anonymous referees for helpful comments.

References

1. Craig Chambers, David Grove, Greg DeFouw, and Jeffrey Dean. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 32(10):108–124, October 1997.
2. Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In Bruce Knobe, editor, *Proceedings of the SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 146–160, Portland, OR, USA, June 1989. ACM Press.
3. Ole-Johan Dahl, Bjørn Myrhaug, and Kristen Nygaard. Simula common base language. Technical Report S-22, Norwegian Computing Center, Oslo, Norway, 1970.
4. Jeffrey Dean. *Whole-Program Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, Seattle, Washington, 1996.
5. Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. *ACM SIGPLAN Notices*, 30(6):93–102, June 1995.
6. Jeffrey Dean, Greg DeFouw, David Grove, Vassily Livinov, and Craig Chambers. Vortex: an optimizing compiler for object-oriented languages. *ACM SIGPLAN Notices*, 31(10):83–100, October 1996.
7. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *Proceedings of the Ninth European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Århus, Denmark, 7–11 August 1995. Springer-Verlag.

8. L. Peter Deutsch and Allan Schiffman. Efficient implementation of a Smalltalk-80 system. In *Proceedings of the 11th Symposium on the Principles of Programming Languages*, pages 297–302, Salt Lake City, 1984. ACM SIGPLAN.
9. Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the 1996 Conference on Object-Oriented Programs, Systems, Languages, and Applications*, pages 292–305. ACM SIGPLAN, October 1996.
10. Karel Driesen and Urs Hölzle. Accurate indirect branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 167–178, New York, June 27–July 1 1998. ACM Press.
11. Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass., 1997.
12. Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. *ACM SIGPLAN Notices*, 30(6):103–115, June 1995.
13. Richard Gabriel, Jon White, and Daniel Bobrow. CLOS: Integrating object-oriented and functional programming. *CACM: Communications of the ACM*, 34, 1991.
14. Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.
15. James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification*. The Java Series. Addison-Wesley, 1.0 edition, August 1996.
16. Urs Hölzle. Adaptive optimization for SELF: Reconciling high performance with exploratory programming. Ph.D. Thesis CS-TR-94-1520, Stanford University, Department of Computer Science, August 1994.
17. Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings of the 1991 European Conference on Object-oriented Programming*, LNCS 512, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag.
18. Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In Christopher W. Fraser, editor, *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 32–43, San Francisco, CA, June 1992. ACM Press.
19. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.
20. Volano LLC. Volanomark benchmark. http://www.volano.com/benchmarks.html, Mar. 1999.
21. Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
22. Greg Nelson, editor. *Systems Programming in Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
23. John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–324, October 1994.
24. Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis object-based environment language reference manual. DEC-TR 372, Digital Equipment Corp., Object-Based Systems Group, Hudson, Massachusetts, Nov. 1985.
25. SPEC. SPECjvm98 benchmarks. http://www.spec.org/osg/jvm98, August 1998.
26. Bjarne Stroustrup. *The C++ Programming Language: Second Edition*. Addison-Wesley, Reading, Massachusetts, 1991.
27. Jan Vitek, Nigel R. Horspool, and Andreas Krall. Efficient type inclusion tests. In *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Atlanta, GA, October 1997. ACM Press.

Modular Statically Typed Multimethods

Todd Millstein and Craig Chambers

Department of Computer Science and Engineering
University of Washington
{todd,chambers}@cs.washington.edu

Abstract. Multimethods offer several well-known advantages over the single dispatching of conventional object-oriented languages, including a simple solution to the "binary method" problem, cleaner implementations of the "visitor," "strategy," and similar design patterns, and a form of "open objects." However, previous work on statically typed multimethods whose arguments are treated symmetrically has required the whole program to be available in order to perform typechecking. We describe Dubious, a simple core language including first-class generic functions with symmetric multimethods, a classless object model, and modules that can be separately typechecked. We identify two sets of restrictions that ensure modular type safety for Dubious as well as an interesting intermediate point between these two. We have proved each of these modular type systems sound.

1 Introduction

In object-oriented languages with multimethods (such as Common Lisp, Dylan, and Cecil), the appropriate method to invoke for a message send can depend on the run-time class of any subset of the message arguments, rather than a distinguished receiver argument. Multimethods unify the otherwise distinct concepts of functions, methods, and static overloading, leading to a potentially simpler language. They also support safe covariant overriding in the face of subtype polymorphism, providing a natural solution to the "binary method" problem [Bruce *et al.* 95] and a simple implementation of the "strategy" design pattern [Gamma *et al.* 95]. Finally, multimethods allow clients to add new operations that dynamically dispatch on existing classes, supporting a form of what we call "open objects" [Chambers 98] that enables easy programming of the "visitor" design pattern [Gamma *et al.* 95, Baumgartner *et al.* 96] and is a key element of aspect-oriented programming [Kiczales *et al.* 97]. Open objects also relieve the tension observed by others [Cook 90, Odersky & Wadler 97, Findler & Flatt 98] between ease of adding operations to existing classes and ease of adding subclasses.

A key challenge for multimethods is separate static typechecking: it is possible for two modules containing arbitrary multimethods to typecheck successfully in isolation but generate type errors when linked together. Previous work on statically typed multimethods has dealt with this problem either by forcing programs to be typechecked as a whole [Mugridge *et al*. 91, Castagna *et al.* 92, Chambers 92, Bourdoncle & Merz 97, Castagna 97, Chambers & Leavens 97, Leavens & Millstein 98] or by sacrificing symmetric treatment of multimethod arguments to ensure the safety of modular typechecking [Agrawal *et al.* 91, Bruce *et al.* 95, Boyland & Castagna 97].

We have designed Dubious, a simple core language supporting both symmetric multimethods and separately typechecked modules. We have identified two sets of restrictions on modules that achieve modular type safety, representing the endpoints in a range of modular type systems trading off flexibility and expressiveness for modularity. We also identify an intermediate point in this range that provides programmers with fine-

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 279-303, 1999.
© Springer-Verlag Berlin Heidelberg 1999

grained control over this tradeoff. We have proved all three of these modular type systems sound.

Section 2 describes the language's features, along with a simple global typechecking algorithm. Section 3 presents a set of important programming idioms that we use as expressiveness benchmarks for our various modular typechecking algorithms. Section 4 focuses on the challenges and solutions for modular typechecking, describing the two main sets of restrictions for modular type safety as well as an interesting intermediate point. Section 5 sketches several extensions to Dubious that are necessary for a practical language. Section 6 discusses related work, and section 7 concludes. A separate technical report provides formal dynamic and static semantics for Dubious, as well as soundness proofs for the various type systems presented [Millstein & Chambers 99].

2 The Dubious Language

The design of Dubious is focused on the issue of modular typechecking of symmetric multimethods; we consciously omit many useful but less relevant features. Dubious includes:

- a classless object model with explicitly declared objects and inheritance (but not dynamically created objects and state),
- first-class generic functions (but not lexically nested closures),
- explicitly declared function types (but not explicitly declared object types nor subtyping independent of objects and inheritance*), and
- modules to support separate typechecking and namespace management (but not nested modules, parameterized modules, nor encapsulation mechanisms).

Section 5 sketches how Dubious could be extended to handle many of these omissions.

Dubious's syntax appears in figure 1; brackets delimit optional parts of the syntax. The following subsections present Dubious's semantics and typechecking rules informally.

2.1 Informal Semantics

The **object** declaration creates a fresh object with a unique, statically known identity and binds it to the given name. The declaration also names the objects from which the new object inherits (its parents). The *descends* relation among objects is the reflexive, transitive closure of the declared inheritance relation. As in other classless languages [LaLonde *et al.* 86, Lieberman 86, Ungar & Smith 87, Chambers 92, Abadi & Cardelli 96], objects play the roles of both classes and instances, and **isa** accordingly plays the

P ::= M_1 ... M_n **import** I **in** E **end**

M ::= **module** I **imports** I_1,...,I_m { D_1 ... D_n }

D ::= [**abstract** | **interface**] **object** I **isa** O_1,...,O_n
  | I **has** **method**(F_1,...,F_n) { E }

E ::= I | E_0(E_1,...,E_n)

O ::= I | (O_1,...,O_n)$\rightarrow O$

F ::= I_1 [**@** I_2]

I ::= *identifier*

Figure 1: Syntax of Dubious

* An earlier version of this work separated objects and types, but we removed this feature to simplify the presentation of the language and its semantics.

```
module GraphicMod imports StdLibMod {
  abstract object graphic
  object draw isa (graphic, display)→void
}
module PointMod imports GraphicMod {
  object point isa graphic
  object x isa (point)→int
  x has method(p@point) { one }
  object y isa (point)→int
  y has method(p@point) { two }
  draw has method(p@point, d) { ... }
  object equal isa (point, point)→bool
  equal has method(p1@point, p2@point) {
    and(=(x(p1), x(p2)), =(y(p1), y(p2))) }
}
module ColorPointMod imports PointMod, ColorMod {
  object colorPoint isa point
  object col isa (colorPoint)→color
  col has method(cp@colorPoint) { red }
  draw has method(cp@colorPoint, d) { ... -- draw in color }
  equal has method(cp1@colorPoint, cp2@colorPoint) {
    and(and(=(x(cp1), x(cp2)), =(y(cp1), y(cp2))),
        eq_color(col(cp1), col(cp2))) }
}
module OriginMod imports PointMod {
  object origin isa point
  x has method(o@origin) { zero }
  y has method(o@origin) { zero }
  equal has method(o1@origin, o2@origin) { true }
}
```

Figure 2: A simple Dubious program fragment

roles of both inheritance and instantiation. For example, in the modules of figure 2, the `colorPoint` object represents a subclass of `point` that has a color, while the `origin` object represents a particular `point` instance.

An object can also act like a (generic) function by inheriting from an *arrow object* that defines the legal arguments and return values of the function. For example, the `equal` object declared in the `PointMod` module in figure 2 inherits from the `(point, point)→bool` arrow object, specifying that `equal` can act like a generic function accepting two objects, each of which is `point` or a descendant, and returning `bool` or a descendant. The ordinary contravariant subtyping rule for function types [Cardelli 84] is used as the descends relation among arrow objects. For simplicity, we require that every generic function have a unique most-specific arrow object from which it inherits. We refer to this most-specific arrow object as the arrow object of the generic function.

A generic function is implemented by adding methods to it, via the **`has method`** declaration. In the `PointMod` module example in figure 2, the method added to the `draw` object has two formal parameters, named `p` and `d`. The first formal is *specialized* by providing the `@point` suffix, which specifies the object on which the argument is dynamically dispatched. We require that a specializer object on a method descend from the object in the corresponding position of the associated generic function's arrow object. For uniformity, unspecialized formals are treated as if they specialize on the object in the corresponding position of the generic function's arrow object. Therefore, the second formal in the `draw` method implicitly specializes on the `display` object. While this method is (explicitly) specialized only on its first argument, the method added

to the `equal` object is a true multimethod, dynamically dispatching on both of its arguments. The **has method** declaration is an imperative, side-effecting operation, like the method update construct in the calculi of Abadi and Cardelli [Abadi & Cardelli 95, Abadi & Cardelli 96]. For example, the `ColorPointMod` module adds a second method to the `draw` object created in the `GraphicMod` module.

If an object is declared **abstract**, it is used solely as a template for other objects and may not be referred to in expressions. For example, the abstract `graphic` object in figure 2 is a template for objects that can be drawn, which is then implemented by the `point` object and descendants. Methods can be specialized on abstract objects, allowing such objects to be partially implemented. These methods are then inherited for use by descendants. An **interface** object is an abstract object that additionally may not act as a specializer (although it may be an implicit specializer on an unspecialized position, as described above). This is similar to the interface construct in Java [Arnold & Gosling 98, Gosling et al. 96]. Arrow objects are implicitly treated as interface objects. (Distinguishing between abstract and interface objects will become useful when considering modular typechecking algorithms in section 4.) An object that is neither abstract nor an interface is called *concrete*. (A *non-concrete* object is either abstract or an interface, and similarly for *non-abstract* and *non-interface*.)

To evaluate a generic function application (message send) $E_0(E_1, \ldots, E_n)$, we evaluate each expression E_i to some object o_i, extract the methods that were added to the generic function object o_0, and then select and invoke the *most-specific applicable method* for $(o_1, \ldots, o_n)$. A method in o_0 is *applicable* to $(o_1, \ldots, o_n)$ if the method has n arguments and if $(o_1, \ldots, o_n)$ pointwise descends from the tuple of the method's specializers. The *most-specific* applicable method is the unique applicable method whose specializers pointwise descend from the specializers of every applicable method. If there are no applicable methods, a *message-not-understood* error occurs, while if there are applicable methods but no most-specific one, a *message-ambiguous* error occurs. For example, consider the application `equal(colorPoint, colorPoint)`, evaluated in some context that imports all the modules in figure 2. First, the `equal` and `colorPoint` expressions are evaluated, yielding the objects with those names.* Then the methods that were added to the `equal` object are extracted. Of the three methods, the `(@point, @point)` and `(@colorPoint, @colorPoint)` methods are applicable, and the `(@colorPoint, @colorPoint)` method is the most-specific one. Therefore, this most-specific applicable method is selected and invoked.

One module **imports** another in order to access its objects. The import relation is transitive; for example, the `ColorPointMod` module may refer to objects in the `StdLibMod` module.

A program is simply an expression evaluated in the context of a single module. (Restricting this top-level expression to import a single module is no loss of expressiveness, as that module can import as many modules as are needed.)

2.2 Static Type Checking

Dubious's static type system ensures that legal programs do not have message-not-understood or message-ambiguous errors. Ruling out these errors involves two kinds of checks: *client-side* and *implementation-side* [Chambers & Leavens 95]. Client-side

* A different object could be named `equal` in a different scope, so this application expression would invoke a different generic function in that other scope. Message names are not special, but are simply identifiers that evaluate to some object via regular scoping rules. The explicit distinction in Dubious between introducing a new generic function and adding a method to an existing generic function clarifies a number of issues, such as overriding versus overloading, that are often confusing in traditional object-oriented languages lacking explicit generic function declarations.

checks are local checks on declarations and expressions. The most important of these checks relate to invoking and implementing generic functions. For each message send expression $E(E_1, ..., E_n)$ in the program, we check that E descends from an arrow object $(O_1, ..., O_n) \rightarrow O$ and each E_i descends from O_i. The message send expression is then known to yield a value that descends from O. For each method declaration I **has method**$(I_1$ @ $I_1', ..., I_n$ @ $I_n')$ { E } in the program, we check that I descends from an arrow object $(O_1, ..., O_n) \rightarrow O$, each I_i' descends from O_i, and E descends from O when typechecked in an environment where each I_i is known to descend from I_i'.

Implementation-side checks ensure that each concrete generic function o correctly implements its arrow object $(O_1, ..., O_n) \rightarrow O$. For every tuple of concrete objects $(o_1, ..., o_n)$ such that each o_i descends from O_i, there must exist a most-specific applicable method in o. For example, consider implementation-side checks on the `equal` generic function. `equal` is declared to inherit from `(point, point)`→`bool`. Nine concrete argument tuples pointwise descend from (`point`, `point`): all possible pairs of the objects `point`, `colorPoint`, and `origin`. The (`@colorPoint`, `@colorPoint`) method is most-specific for two `colorPoints`, the (`@origin`, `@origin`) method is most-specific for two `origins`, and the (`@point`, `@point`) method is most-specific for all other tuples.

This description suggests a straightforward typechecking algorithm. Client-side checks on the declarations in a module require only inheritance information from imported modules, and they can therefore be performed on a module-by-module basis. The implementation-side checks as described above assume global knowledge of the program's objects and methods. Therefore, implementation-side typechecking on all generic functions is deferred until link-time, when all modules are present. This link-time typechecking approach is used for typechecking previous languages with symmetric multimethods, including Kea [Mugridge *et al.* 91], Cecil [Chambers 92, Chambers 95], $ML_{\leq}$ [Bourdoncle & Merz 97], and Tuple [Leavens & Millstein 98]. We refer to this global typechecking algorithm as *System G*.

3 Example Programming Idioms

There are several flexible programming idioms that we would like to be able to express and statically typecheck in Dubious. These idioms will serve as benchmarks for evaluating the various typechecking algorithms presented in this paper. The global typechecking of System G supports all these idioms.

First, we wish to support traditional receiver-oriented programming. In this idiom, an object is declared with its associated methods in its own module, which imports the modules defining the parent objects. More generally, we can allow multiple objects and their associated methods to be declared in a single module. We refer to this idiom, in which each method is specialized solely on its first argument to an object declared in the same module, as *single dispatching*.

We also wish to allow a module to define abstract objects, whose operations are not required to be implemented, and to provide concrete implementations of these objects in separate modules. For example, the `GraphicMod` module in figure 2 defines an abstract template for graphical objects. Because the `graphic` object is abstract, its `draw` generic function need not be completely implemented. Concrete descendants of the `graphic` object can be declared in other modules, which must provide appropriate implementations for the generic functions declared in the `GraphicMod` module. For example, the `point` object is a legal implementation of `graphic`. Clients of the abstract object need not be aware of the various concrete implementations.

Several expressive idioms exploit multimethods. If single dispatching is generalized to allow method arguments in addition to the first to be specialized on objects declared in the enclosing module, *all-local (multi)methods* result, which enable a simple solution to the "binary method" problem [Bruce *et al.* 95]. For example, the three `equal` methods in figure 2 are all-local multimethods, each specialized on two objects declared in the enclosing module. All-local multimethods allow easy programming of one reasonable semantics for equality on various combinations of points. At the same time, `colorPoint` and `origin` remain safe subtypes of `point`, so subtype polymorphism over the `point` hierarchy is still available.

A further generalization of single dispatching and all-local multimethods allows arguments other than the first to be specialized on any object, including objects declared in imported modules; the first argument is still restricted to be specialized on a locally declared object. We refer to this kind of multimethod as an *encapsulated-style multimethod*; it is similar to the style of multimethods allowed by encapsulated multimethods [Castagna 95, Bruce *et al.* 95] and parasitic multimethods [Boyland & Castagna 97]. Using encapsulated-style multimethods, new objects can interact with imported objects in interesting ways, without modifying the code for the imported module. For example, the `ColorPointMod` module could include the following declaration to program how colored points should be compared to uncolored points:

```
equal has method(cp@colorPoint, p@point) { ... }
```

If we remove all restrictions on method specializers, allowing any or all to be imported objects, we obtain what we call *arbitrary (multi)methods*. Using arbitrary multimethods, we can program special behavior for equality on one `point` and one `colorPoint`, without modifying either the `PointMod` or `ColorPointMod` modules, with the following code:

```
module PCPMod imports ColorPointMod {
  equal has method(p@point, cp@colorPoint) { ... }
}
```

A final desirable idiom is *open objects* [Chambers 98], where an object declared in an imported module is extended by adding new operations that dispatch on the object, without modifying any existing modules. For example, the following code introduces a module that defines a `lineseg` object representing a line segment. The `distance` generic function extends the `point` object and descendants with a new operation, without modifying existing code.

```
module LineSegMod imports OriginMod {
  object lineseg
  object p1 isa (lineseg)→point
  p1 has method(ls@lineseg) { ... }
  object p2 isa (lineseg)→point
  p1 has method(ls@lineseg) { ... }
  object distance isa (point, lineseg)→real
  distance has method (p@point, ls) {
    ... -- distance from a point to a line segment }
  distance has method (p@origin, ls) {
    ... -- faster algorithm for the origin }
}
```

Open objects arise naturally in languages based on multimethods, but they are useful even for singly dispatched methods, as in the case of the `distance` generic function above. Open objects support simpler programming of the "visitor" design pattern [Gamma *et al.* 95]. Client-specific visitors are programmed directly, without needing to build a special visitor infrastructure for each object hierarchy. In addition, the open

object idiom retains the ability to add new subclasses without modifying existing code, while the visitor pattern does not. In the `lineseg` example, the `distance` generic function is a kind of client-specific visitor of the `point` hierarchy. New descendants of `point` can be added in other modules without modifying or breaking the `distance` method (as long as the implementations of `distance` inherited by new descendants are still appropriate), and new operations can be added to existing object hierarchies without modifying or breaking those hierarchies. Aspect-oriented programming [Kiczales *et al.* 97] relies heavily on open objects to implement cross-cutting concerns.

4 Modular Typechecking

We would like a modular approach to implementation-side typechecking of generic functions, unlike the global algorithm of System G. In particular, we would like a typechecking scheme with the following properties for each module:

- The generic functions created in the module can be safely implementation-side typechecked given only the interfaces of imported modules: their objects, associated inheritance hierarchy, and the headers of each generic function's methods.
- Implementation-side typechecking of an imported generic function is only necessary when the importer adds a new method to the generic function, or when the importer creates a concrete object directly inheriting from a non-concrete object, and the concrete object is applicable to one of the generic function's argument positions. Further, this checking does not re-examine legal argument tuples already checked by the importee.

This typechecking scheme is a generalization of the modular typechecking scheme of conventional statically typed, singly dispatched object-oriented languages. Unfortunately, applying such a modular typechecking scheme to the unrestricted Dubious language is unsound. It is possible for two importers of a module to pass these checks in isolation but still cause message-not-understood or message-ambiguous errors when combined in a single program [Chambers & Leavens 95]. Subsection 4.1 describes the situations that can lead to such errors. Subsections 4.2, 4.3, and 4.4 describe three different sets of restrictions that support safe modular typechecking, representing different tradeoffs between expressiveness, modularity, and complexity. Subsection 4.5 summarizes the key features of the various type systems discussed in this paper.

4.1 Challenges for Modular Typechecking

The unrestricted Dubious language poses several problems for modular typechecking. In particular, there are four scenarios where a modular typechecking approach applied to the unrestricted language will fail to statically detect errors that can occur at run-time. The first two problems are specific to multimethods, while the other two involve open objects. In this subsection, we give examples of each of these kinds of problems.

First, the ability to add arbitrary multimethods anywhere in the program can cause undetected ambiguities. A simple example of the problems that can occur appears in figure 3, where each of the `ColorPointMod` and `OriginMod` modules from figure 2 is augmented with a second `equal` method. Each module typechecks in isolation, given only information about its imported modules. From the `ColorPointMod` module's point of view, every visible legal argument tuple to the `equal` generic function has a single, most-specific method implementation, and similarly for the `OriginMod` module. However, when the two modules are combined in a single program, a run-time message-ambiguous error will occur if the message `equal(colorPoint,origin)` is ever sent, since neither of the methods in the example is more specific than the other.

```
module ColorPointMod imports PointMod, ColorMod {
  ... -- as in figure 2
  equal has method(cp@colorPoint, p@point) { ... }
}
module OriginMod imports PointMod {
  ... -- as in figure 2
  equal has method(p@point, o@origin) { ... }
}
```

Figure 3: Ambiguity problems with arbitrary multimethods

```
module AbstractPointMod {
  abstract object point
  object equal isa (point, point)→bool
}
module ColorPointMod imports AbstractPointMod, ColorMod {
  object colorPoint isa point
  equal has method(cp1@colorPoint, cp2@colorPoint) { ... }
}
module OriginMod imports AbstractPointMod {
  object origin isa point
  equal has method(o1@origin, o2@origin) { true }
}
```

Figure 4: Incompleteness problems with multimethods on abstract objects

```
module PrintMod imports ColorPointMod, OriginMod {
  object print isa (point)→void
  print has method(p@point) { ... -- print points }
  print has method(cp@colorPoint) { ... -- print in color }
  print has method(o@origin) { ... -- print the origin specially }
}
module ColorOriginMod imports ColorPointMod, OriginMod {
  object colorOrigin isa colorPoint, origin
}
```

Figure 5: Ambiguity problems combining open objects with multiple inheritance

```
module EraseMod imports PointMod {
  object erase isa (graphic, display)→int
  erase has method(p@point, d) { ... -- erase points }
}
module MyGraphicMod imports GraphicMod {
  object myGraphic isa graphic
  draw has method(g@myGraphic, d) { ... }
}
```

Figure 6: Incompleteness problems combining open objects with abstract objects

One way to solve this problem is to break the symmetry of the dispatching semantics. For example, if we linearized the specificity of argument positions, comparing specializers lexicographically left-to-right (rather than pointwise) as is done in Common Lisp [Steele 90, Paepcke 93] and Polyglot [Agrawal *et al.* 91], then the `(@colorPoint, @point)` method would be more specific than the `(@point, @origin)` method. However, one of our major design goals for Dubious is to retain the symmetric multimethod dispatching semantics, which we believe is more natural and less error-prone, since it reports potential ambiguities rather than silently resolving them. The symmetric semantics is used in the languages Cecil [Chambers 92, Chambers 95], Dylan [Shalit 97, Feinberg *et al.* 97], the λ&-calculus [Castagna *et al.* 92, Castagna 97],

Kea [Mugridge *et al.* 91], $ML_{\leq}$ [Bourdoncle & Merz 97], and Tuple [Leavens & Millstein 98].

A second unsafe scenario involves the combination of non-concrete objects with multimethods. For example, suppose we want to make the `point` object in figure 2 be abstract, so that it need not be fully implemented. Figure 4 shows the relevant parts of the revised modules. Each module passes implementation-side typechecks on the `equal` generic function in isolation, as all legal tuples of concrete objects have a single, most-specific implementation from each module's point of view. However, at run-time a message-not-understood error will occur if the message `equal(colorPoint, origin)` or `equal(origin, colorPoint)` is sent.

The last two unsafe scenarios involve the ability to program open object idioms. Figure 5 shows an example of the problems that can occur when the open object idiom is combined with multiple inheritance. The `PrintMod` module extends the interface of points from figure 2 with a function for printing points. From this module's point of view, the `print` generic function is completely and unambiguously implemented. Independently, the `ColorOriginMod` module creates a new point object that multiply inherits from `colorPoint` and `origin`. Since this module does not know about the `print` generic function, it has no way of statically detecting the ambiguity for `print(colorOrigin)`, which can therefore cause a run-time error.

One way to fix this ambiguity is to linearize the inheritance hierarchy, as is done in Common Lisp [Steele 90, Paepcke 93] and Dylan [Shalit 97, Feinberg *et al.* 97], or to provide some other total ordering over methods, as in parasitic multimethods [Boyland & Castagna 97]. However, we reject these solutions for reasons similar to our rejection of argument-position linearization for multimethods, preferring the simpler and less error-prone semantics.

The final unsafe scenario involves the combination of open object idioms with non-concrete objects. In figure 6, the `EraseMod` module extends the `graphic` interface from figure 2 with an operation for erasing the graphical object. The `erase` generic function is completely implemented for all the concrete implementations of `graphic` that are visible to that module. The `MyGraphicMod` creates a concrete implementation of a `graphic` object. From its point of view, the `graphic` interface is completely implemented for `myGraphic`. However, at run-time there will be a message-not-understood error if the message send `erase(myGraphic, d)` occurs (where `d` is some `display` descendant).

4.2 System M: Maximizing Modularity

Because of the four unsafe programming scenarios described above, any completely modular typechecking scheme must restrict the usage of certain Dubious language constructs. In this subsection, we detail *System M*, a set of restrictions that allows a modular typechecking scheme with the desirable properties described at the beginning of this section. Our goal with System M is to provide the most flexible type system possible, subject to those strict modularity goals.

We say that a non-arrow object or a method is *local* if it is declared in the current module, and otherwise it is *non-local*. An arrow object is local if it has a local object in a positive[*] position, and otherwise it is non-local. Two objects are *related* if one object descends

[*] The result object of an arrow object is in a positive position, while the arguments are in negative positions. If an arrow object appears in a negative position, then the polarity of its positions is reversed [Canning *et al.* 89]. An arrow object with a local object in a positive position is local because no module other than importers of the current module can define functions that descend from the arrow. Due to contravariance, the same is not true if local objects appear only in negative positions.

from the other, and otherwise they are *unrelated*. Similarly, two modules are related if one imports the other, and otherwise they are unrelated. An *orphan* is a concrete object that directly inherits from a non-local, non-concrete object. *Implementation inheritance* is inheritance from a non-interface object, and *interface inheritance* is inheritance from an interface object.

The key insight of System M is that if two unrelated modules M_1 and M_2 each create an encapsulated-style method on the same generic function and multiple implementation inheritance across module boundaries is disallowed, then the two methods will apply to disjoint sets of legal argument tuples. Therefore, these restrictions safely remove method ambiguity problems. Potential incompleteness problems are removed by treating visible non-concrete objects as if they were concrete during the implementation-side typechecking of a generic function, thereby forcing the existence of appropriate method implementations to handle unseen concrete descendants of these objects. We treat all visible non-concrete objects in this way except for local non-concrete objects that apply to the first argument of the generic function. Such objects can remain safely unimplemented, with appropriate implementations for concrete descendants to be added by importers, thereby safely allowing abstract object idioms.

More precisely, System M imposes the following four restrictions on each module:

- (***M1***) Each method added to a non-local generic function must be an encapsulated-style method, i.e., the first argument must be specialized to a local object. (Strictly speaking, all that is required is that each generic function designate some argument position for which all methods of that generic function agree to have a local specializer; for simplicity in this paper, we assume this designated position is the first.)
- (***M2***) If a local non-interface object descends from two unrelated, non-local, non-interface objects o_1 and o_2, then it must also descend from a non-interface object that descends from o_1 and o_2 as well.
- (***M3***) If a generic function's arrow object has the object o in some argument position other than the first, then implementation-side typechecks of the generic function must consider any non-concrete visible descendants* of o to be concrete.
- (***M4***) If a generic function's arrow object has the object o in the first argument position, then implementation-side typechecks of the generic function must consider any non-local, non-concrete visible descendants of o to be concrete.

By imposing the above restrictions, we can ensure safety in Dubious while meeting the modularity goals described at the beginning of this section. In particular, each module implementation-side typechecks its local generic functions given only the interfaces of its importees. In addition, there are two scenarios in which a module must re-check imported generic functions:

- If the module adds methods to a non-local generic function, then this generic function is implementation-side typechecked. However, we only need to check argument tuples to which a local method applies.
- If the module creates an orphan o, then all non-local generic functions accepting o as an argument in the first position are implementation-side typechecked. However, only legal argument tuples containing o at that position need be checked. (This check ensures the safety of local, non-concrete descendants of the first argument in a generic function's arrow object, which was left unchecked by rule ***M4*** above. In this way, we safely allow abstract object idioms.)

* Recall that the *descends* relation is the reflexive, transitive closure of the declared inheritance relation.

We can use the restrictions to resolve the problems in the examples of the previous subsection. In figure 3, the `equal` method in the `OriginMod` would cause a static type error. In particular, it is not an encapsulated-style method because it has a non-local first specializer, so it violates restriction ***M1***. The restriction would be satisfied if the method instead had the form

```
equal has method(o@origin, p@point) { ... }
```

and indeed this removes the multimethod ambiguity. Forcing both methods in figure 3 to be encapsulated-style (in conjunction with the multiple inheritance limitations imposed by restriction ***M2***) ensures that no argument tuple will be applicable to both of them.

In figure 4, both the `colorPoint` and `origin` objects are orphans. Because `equal` accepts the `colorPoint` and `origin` objects on its first position, it is re-checked by the `ColorPointMod` and `OriginMod` modules. By restriction ***M3***, these checks must consider the abstract `point` object to be concrete for the second argument position. Therefore, re-checks from the `ColorPointMod` module will find an incompleteness for the argument tuple (`colorPoint,point`), and similarly for the `OriginMod` module's checks. Therefore, methods must be created to cover these cases safely. For example, the `ColorPointMod` module could include the method declaration

```
equal has method(cp@colorPoint, p@point) { ... }
```

and similarly for the `OriginMod` module, thereby safely allowing abstract object idioms.

In figure 5, the `colorOrigin` object fails restriction ***M2*** because it descends from two unrelated, non-local, non-interface objects without also descending from some non-interface descendant of both of these objects, so the `colorOrigin` object cannot be programmed. Because of the conflict between cross-module multiple implementation inheritance and open object idioms, we were forced to eliminate one of these idioms to ensure the safety of modular checking. We chose to disallow cross-module multiple implementation inheritance because of the importance of open objects. Multiple interface inheritance across import boundaries is still safe (because methods cannot specialize on interface objects and so cannot generate ambiguities), as is arbitrary multiple implementation inheritance within a module.

In addition, many cases of multiple implementation inheritance can be programmed safely within System M's restrictions if they are *anticipated* when at least one of the parents is implemented. For example, if the implementor of the `OriginMod` module anticipated that multiple inheritance between `colorPoint` and `origin` might be needed, then a placeholder abstract object could be added to `OriginMod` that multiply inherits from the two objects:

```
module OriginMod imports ColorPointMod {
  object origin isa point
  ...
  abstract object colorPointAndOrigin isa colorPoint, origin
}
```

Clients that wish to use multiple inheritance can then singly inherit from the placeholder object:

```
module ColorOriginMod imports OriginMod {
  object colorOrigin isa colorPointAndOrigin
}
```

Any modules that add new generic functions to existing objects, such as the `PrintMod` module in figure 5, will see the placeholder object whenever they see both its parents, so

they will be forced to write methods that resolve any potential multiple-inheritance ambiguities. In this example, restriction ***M4*** would force the existence of a `print` method to disambiguate printing a `colorPointAndOrigin`. This addition fixes the potential message-ambiguous error in the example. In our experience, gained largely from writing a 125,000-line optimizing compiler in Cecil, anticipated multiple implementation inheritance is common, while successful unanticipated multiple implementation inheritance is very rare.

In figure 6, the first argument of the `erase` generic function's arrow object is the non-local abstract `graphic` object. By restriction ***M4***, implementation-side typechecks on this generic function must assume that `graphic` is concrete. Therefore, an incompleteness is found, which is fixed by adding an appropriate default method implementation. Adding such a method allows the module to typecheck and ensures that the message send `erase(myGraphic, d)` now has a single, most-specific method implementation to invoke.

In summary, System M provides a completely modular and safe typechecking algorithm, while maintaining a high level of flexibility. Multimethods with a non-local first specializer are disallowed, but all other kinds of multimethods are safe. This provides several important multimethod idioms, including binary methods and encapsulated-style multimethods. The implementation of a non-concrete object across module boundaries and the use of arbitrary open objects are allowed, as long as the appropriate default method implementations are included to prevent unseen incompletenesses. The cost of allowing the open object idioms is a loss of unanticipated cross-module multiple implementation inheritance.

4.3 System E: Maximizing Expressiveness

Although the set of restrictions in System M provides completely modular typechecking for Dubious, there are certain idioms that cannot be expressed across module boundaries. These missing idioms include arbitrary multimethods, where the first argument position has a non-local specializer, and unanticipated multiple implementation inheritance. In this subsection, we present *System E*, whose fundamental requirement is to be able to express all idioms. Since a completely modular static type system cannot support all these idioms safely, System E includes a simple link-time check to ensure soundness of the more aggressive idioms, while striving to retain modular typechecking for as many idioms as possible.

A previous work informally introduced the idea of a module *extending* another instead of importing it [Chambers & Leavens 95] whenever the module needed to make use of one of the more aggressive idioms across that module boundary. If each module rechecks all the generic functions from modules it extends, and if at link-time each module in the program has a *unique most-extending module* in the reflexive, transitive closure of the direct module extends relation, then the program is guaranteed to be safe. In particular, the most-extending module's checks are enough to ensure run-time safety of all the modules it (directly or indirectly) extends.

We have formalized this notion of module extension in Dubious. We modify the module declaration to allow a set of extenders to be declared:

$$M \ ::= \ \textbf{module imports} \ I_1, \ldots, I_m \ \textbf{extends} \ I_1', \ldots, I_r' \ \{ \ D_1 \ \ldots \ D_n \ \}$$

In the context of System E, we say that a non-arrow object or a method is *local* if it was declared in the current module, *extended* if it was declared in an extended module, and *imported* otherwise. An arrow object is local if it has a local object in a positive position, otherwise it is imported if it has an imported object in a positive position, and otherwise it is extended. A *non-local* object is either imported or extended, and similarly for a *non-*

imported and *non-extended* object. A module m_1 is the most-extending module of m_2 if m_1 extends every module that extends m_2, using the reflexive, transitive closure of the module declarations' **`extends`** clauses.

A module is unrestricted in its interactions with extended objects and generic functions, and thus all of our benchmark idioms are available to extenders. However, because of the greater potential effects of unseen extenders, importers must obey stricter restrictions than those of System M. First, because extenders can use multiple implementation inheritance freely, open object idioms must be disallowed in importers. Second, because extenders can write arbitrary multimethods, we restrict modules to add only all-local methods (not just encapsulated methods) to imported generic functions.

If all methods added to imported generic functions were forced to be all-local, then very few idioms could be legally programmed in importing modules. To eliminate this problem, we introduce syntax for specifying, as part of a generic function declaration, which subset of its arguments may be specialized:

$$O \;::=\; I \;\mid\; (S_1, \ldots, S_n) \rightarrow O$$
$$S \;::=\; [\#]O$$

A **#** marker denotes a *marked argument position* of a generic function. All arrow objects a generic function inherits must agree on which argument positions are marked. Methods may not specialize at unmarked argument positions. This allows methods on imported generic functions to safely accept non-local objects on unmarked positions. Methods on imported generic functions must still have only local specializers on marked positions, as in an all-local method.

To avoid incompleteness, we use the same technique as in System M of considering all visible non-concrete objects to be concrete for the purposes of implementation-side typechecking a generic function. As with System M, there is one situation in which non-concrete objects may safely remain incompletely implemented. In particular, if the generic function has exactly one marked argument position, then any local non-concrete objects that apply to the marked position need not be considered concrete during implementation-side typechecking. This safely allows abstract object idioms for singly dispatched functions.

More formally, System E imposes the following four restrictions on modules, directly analogous to the four restrictions of System M:

- (***E1a***) A method may not specialize on an unmarked argument position of its generic function, and (***E1b***) a method added to an imported generic function must have a local specializer object at each marked argument position.
- (***E2***) If a local non-interface object *o* descends from an imported object, then every two non-interface parents of *o*, where at least one of the parents is non-local, must be related.
- (***E3a***) If a generic function's arrow object has the object *o* in some unmarked argument position, then implementation-side typechecks of the generic function must consider any non-concrete visible descendants of *o* to be concrete. (***E3b***) If a generic function's arrow object has the object *o* in some marked argument position and the generic function is not singly dispatched, then implementation-side typechecks of the generic function must consider any non-concrete visible descendants of *o* to be concrete.
- (***E4a***) If a local generic function's arrow object has the object *o* in a marked argument position, then no visible descendant of *o* may be an imported object. (***E4b***) If a singly dispatched generic function's arrow object has the object *o* in its marked

```
module ResolveColorPointAndOriginMod extends ColorPointMod,
                                             OriginMod {
    equal has method(cp@colorPoint, o@origin) { ... }
}
module ResolvePrintModAndColorOriginMod extends PrintMod,
                                                 ColorOriginMod
    print has method(co@colorOrigin) { ... }
}
```

Figure 7: Extension Modules

argument position, then implementation-side typechecks of the generic function must consider any non-local, non-concrete visible descendants of *o* to be concrete.

To typecheck a module, we need only the interfaces of extended and imported modules. A module re-implementation-side typechecks extended generic functions. In addition, there are two kinds of re-checks on imported generic functions, similar to the two cases in System M:

- If the module adds methods to an imported generic function, then this generic function is implementation-side typechecked. However, we only need to check argument tuples to which a local method applies.
- If the module creates an orphan *o*, then all imported, singly dispatched generic functions accepting *o* as an argument on the marked position are implementation-side typechecked. However, only legal argument tuples containing *o* at that position need be checked. (This check ensures the safety of local, non-concrete descendants of the argument in the marked position in a singly dispatched generic function's arrow object, which was left unchecked by rule ***E4b*** above. In this way, we safely allow abstract object idioms for singly dispatched generic functions.)

A link-time check ensures that every module has a unique, most-extending module. This is the only global check needed by System E, and it does not include any client-side or implementation-side typechecking; it merely ensures that modules exist that have already performed the necessary checks in their modular fashion. This check takes time $O(m + e)$, where *m* is the number of modules and *e* is the number of extends declarations in the program.

The restrictions on importers, while stronger than those of System M, still allow safe modular typechecking of singly dispatched programming idioms, the implementation of an imported non-concrete object for singly dispatched functions, and binary methods. For example, each of the modules in figure 2 satisfies these restrictions (assuming every generic function's first argument position is marked, and `equal` also has a marked second argument position). In particular, `point` is a safe implementation of the abstract `graphic` object, even though the `draw` method is not implemented for `graphic` objects and there are potentially other implementations of `graphic` that are unseen by the `PointMod` module. In addition, binary methods like `equal` are allowed by these rules, even though the `ColorPointMod` and `OriginMod` modules cannot see each other statically. For example, to ensure safety of the `equal` generic function, the `ColorPointMod` module simply needs to check that the (`colorPoint`, `colorPoint`) tuple has a most-specific method; the (`point`, `point`), (`colorPoint`, `point`), and (`point`, `colorPoint`) tuples need not be checked. The `OriginMod` module will perform analogous checks.

However, the more expressive programming idioms must use extenders, which can now resolve the problems from subsection 4.1 that System M could not solve. In figure 3, the revised `ColorPointMod` and `OriginMod` modules must use the clause **`extends`** `PointMod` instead of **`imports`** `PointMod` because they each violate restriction ***E1b***

by having an imported specializer on the equal method (the ColorPointMod module still need only import the ColorMod module). Further, if the two modules are combined in the same program, an additional module must be written that extends them both, in order for the PointMod module to have a unique most-extending module. In order for checks on the equal generic function to succeed in this new module, it must include the necessary declarations to fix the ambiguity, as shown in the ResolveColorPointAndOriginMod module in figure 7.

An analogous solution is used to resolve the problem illustrated in figure 5. Assuming the print generic function's first argument position is marked, print violates restriction ***E4a*** by having an imported object on a marked argument position in its arrow object, and colorOrigin violates restriction ***E2*** by multiply inheriting from unrelated, imported, non-interface objects. Therefore, both modules must extend the PointMod module rather than import it, requiring the existence of a most-extending module to fix the ambiguity. This module is shown in figure 7. Further, if the modules in figure 7 were combined in a single program, a module extending both of these would be required, in order to provide the PointMod module with a single most-extending module. (Because there are no conflicts, this module could be empty.)

In figure 4, the equal generic function in the AbstractPointMod module has multiple marked positions (assuming both argument positions are marked) and has the abstract point object on a marked position in its arrow object. Therefore, by restriction ***E3b***, implementation-side typechecks must consider point to be concrete. These checks will find an incompleteness for the argument tuple (point, point), forcing the creation of a default method implementation such as

```
equal has method(p1@point, p2@point) { ... }
```

to cover this case safely. This has the effect of disallowing abstract multimethods; singly dispatched abstract methods are still safe and can be implemented modularly.

In figure 6, the erase generic function, assuming that its first argument position is marked, violates restriction ***E4a*** by having the imported graphic object on a marked position in its arrow object. Therefore, the EraseMod module must extend GraphicMod. Then, by restriction ***E4b***, implementation-side checks on erase must assume graphic is concrete, requiring a default method declaration such as

```
erase has method(g@graphic, d) { ... }
```

This method safely handles the unseen myGraphic object.

The reflexive, transitive closure of the declared extension relation partitions the program into a set of module regions, the modules in each region connected to one another by extension and having a unique most-extending module. As opposed to the global typechecking of the naive algorithm and the local typechecking of importers, extenders represent a kind of *regional* typechecking. Implementation-side typechecking is still performed in a modular fashion, but each module re-checks its extended modules. The modular checks in the most-extending module of a region ensure the safety of that region, and each region's checking is separate from the others'. A simple link-time check ensures that each module has a most-extending module. By sacrificing complete modularity, System E provides all of our expressive programming idioms, including arbitrary multimethods and multiple implementation inheritance. Binary method idioms, abstract singly dispatched methods, and multiple interface inheritance are still completely modular.

4.4 System ME: Combining Modularity and Expressiveness

Systems M and E are the endpoints in a range of possible modular type systems. System M maximizes the language's modularity of typechecking. The cost for this modularity is a loss of certain expressive programming idioms across module boundaries, including arbitrary multimethods and unanticipated multiple implementation inheritance. System E maximizes the language's expressiveness, at the cost of some regional typechecking, a simple link-time check for unique most-extending modules, and more restrictions on what can be expressed in importers.

By selecting different subsets of restrictions from each of Systems M and E, it is possible to design safe modular type systems with intermediate abilities between these two extremes. In this subsection, we describe one interesting point in this range: *System ME*. In this system, each generic function uses System M's restrictions by default, but if a generic function is expected to need arbitrary methods added to it, it may be given an arrow object with # markers on argument positions, in which case System E's rules apply to that generic function. More precisely, System ME includes restrictions ***M1***-***M4*** and ***E1***-***E4***, with a few modifications. The generic functions referred to in restrictions ***M1***, ***M3***, and ***M4*** are only those generic functions that have no marked argument positions, while the generic functions referred to in restrictions ***E1***, ***E3***, and ***E4*** are only those generic functions that have at least one marked argument position. The combination of restrictions ***M2*** and ***E2*** allows multiple inheritance only if it is anticipated and if the two unrelated parent objects are non-imported.

This system gives implementors control over the tradeoff between modularity and flexibility at the granularity of an individual generic function. For example, with System E's rules for the `equal` generic function, we can use extension modules to resolve the ambiguity in figure 3, retaining the use of arbitrary multimethods. At the same time, by using System M's rules for the `erase` generic function in figure 6, we can keep the open object idiom in importers, with completely modular typechecking.

4.5 Summary

Table 1 summarizes the expressiveness of the various type systems we have described and compares them with the expressiveness of singly dispatched languages. System G expresses all of our benchmark programming idioms, at the cost of a global typechecking algorithm. System M has a substantial amount of expressive power while safely using solely local typechecking. The expressive power of the locally checked importers of System E is more limited, but its extenders allow all idioms to be expressed, including arbitrary multimethods and multiple implementation inheritance. System ME provides a nice balance between Systems M and E, maintaining local checking of all modular programming idioms in System M while allowing arbitrary multimethods to be programmed in extenders when needed. The main disadvantage of this system is its complexity, as the declarer of a generic function must decide *a priori* which kinds of extensibility are needed and thus which kinds of restrictions to impose. Finally, all of the type systems compare favorably to conventional singly dispatched languages, which provide fewer than half of these expressive programming idioms. However, such languages typecheck arbitrary multiple implementation inheritance modularly, which none of Dubious's type systems can do because of their support for open object idioms.

In summary, we have presented several alternatives for modular typechecking of multimethods, ranging from a completely modular approach that sacrifices certain programming idioms, to a completely expressive approach that requires some regional typechecking and a simple link-time check. Several issues must be better understood before a clear winner can emerge from among these type systems. Such issues include

	G	M	E		ME		singly dispatched languages
			importers	extenders	importers	extenders	
single dispatching	X	X	X	X	X	X	X
abstract objects	X	X	X	X	X	X	X
binary multimethods	X	X	X	X	X	X	
encapsulated multimethods	X	X		X	X	X	
arbitrary multimethods	X			X		X	
open objects	X	X		X	X	X	
multiple implementation inheritance	X			X			X
multiple interface inheritance	X	X	X	X	X	X	X
typechecking scope	global	local	local	regional	local	regional	local

Table 1: Overview of the expressiveness of the various type systems discussed

understanding which programming idioms are critical to be able to express in the language and which can be sacrificed, which idioms are critical to express purely modularly and which can require regional checking, and which kinds of restrictions are simpler or more intuitive than others. A practical evaluation of these systems is necessary to better understand these issues. We plan to undertake such an evaluation in the context of Diesel, a full programming language succeeding Cecil that will be based on Dubious's foundation for modular type systems for symmetric multimethods.

5 Extensions

This section sketches several extensions to Dubious, moving it closer to a full programming language. First we show how the modular typechecking rules can be exploited to safely allow arbitrarily nested declarations, which supports dynamic object creation and first-class nested functions. Then we describe how to add mutable state, how to add encapsulation, and how to generalize Dubious to the predicate dispatching model.

5.1 Nested Declarations

Dubious enforces a flat structure: modules are declared only at top-level (no modules may be nested in other modules or in methods), and object and method declarations exist only immediately within a module (not at top-level or within a method body). We can relax this restriction, allowing arbitrary nesting of modules and other declarations. For

```
module PointMod {
  import GraphicMod
  object point
  ...
  object newPoint isa (int, int)→point
  newPoint has method(newx, newy) {
    object newp isa point
    x has method(p@newp) { newx }
    y has method(p@newp) { newy }
    newp }
  object curried_equal isa (point)→((point)→bool)
  curried_equal has method(p1@point) {
    object eq_p1 isa (point)→bool
    eq_p1 has method(p2) {
      and(=(x(p1),x(p2)), =(y(p1),y(p2))) }
    eq_p1 }
}
import PointMod
curried_equal(newPoint(one,two))(newPoint(two,one))
```

Figure 8: Nested declarations in an extension of Dubious

example, Dubious's program, module, and declaration forms could be reorganized as follows:

P ::= B E

B ::= D_1 ... D_n

D ::= [**abstract** | **interface**] **object** I **isa** O_1,...,O_n
  | I **has method**(F_1,...,F_n) { B E }
  | **module** I **extends** I_1,...,I_m { B }
  | **import** I

In this reorganization, modules are just regular kinds of declarations, method bodies may begin with an arbitrary block of declarations, and programs simplify to a block of declarations followed by an expression. The **import** clause from the **module** declaration is now a separate declaration, allowing any scope to import a module. Nestable modules are largely a namespace-management convenience, but declarations nested in method bodies provide significant additional expressive power, as they are executed each time the enclosing method is invoked at run-time. In particular, this provides dynamic object creation and lexically nested functions, among other programming idioms.

Figure 8 shows a simple example of these two uses of nested declarations. The `newPoint` generic function is a constructor for point "instances" that creates and returns a fresh child of `point` each time it is invoked, initializing the new child's `x` and `y` coordinates appropriately. The `curried_equal` generic function is a curried version of the `equal` function on points, illustrating the creation of lexically nested functions.

Nested **has method** declarations within a method body provide a limited form of mutable state. The `newPoint` generic function illustrates the use of such nested methods for field initialization. In addition, mutable variables can be derived from zero-argument functions. A variable is simply a generic function with no arguments, with a single method whose body returns the variable's value. A variable assignment is simply a method update of this generic function to have a new method body. To make this work, we modify the semantics so that a new method declaration replaces any existing method

with the same tuple of specializers on the same generic function. The following example shows a mutable variable representing the screen size of a display:

```
object screenSize isa ()→point -- declare variable
screenSize has method() { newPoint(1024, 768) } -- set initial value
object update isa (point)→void
update has method(p) { screenSize has method() { p } -- update value }
...
screenSize() -- read current value
```

To typecheck nested declaration blocks (instances of the *B* nonterminal in the grammar above), we make use of the modular typechecking restrictions described in section 4. The fundamental idea is to consider a nested declaration block to be an *importer* of its enclosing scope. If the nested declaration block obeys the typechecking restrictions on importers, then we know that the block will not conflict with unseen importers or other nested declaration blocks of the enclosing scope. To ensure that we can statically check that the nested declaration block is a legal importer, we require that the inheritance parents in the **object** declaration as well as the generic function and specializer objects in the **has method** declaration be statically known objects, rather than potentially computed expressions such as formal arguments. (We also restrict modules nested in a method to extend only modules defined in that same method. Otherwise we could not verify the single most-extending module restriction at link-time.)

The nested declaration block needs to be typechecked in the context of the entire enclosing scope, however, not just the part of the enclosing scope that is visible when the nested declaration block is encountered. Otherwise, the nested declaration block may make unsafe assumptions about its enclosing scope. To handle this issue, we use a two-pass approach to typechecking a declaration block. On the first pass, we typecheck all declarations in textual order, but ignore the bodies of module and method declarations in the block. On the second pass, we typecheck the bodies of these skipped modules and methods, but now in the context of the full environment computed for the enclosing scope. No additional work is performed over a one-pass scheme; merely the order in which the individual typechecking steps are performed changes.

5.2 Mutable State

Nested declarations allow a limited form of mutable state for statically known objects. We incorporate arbitrary mutable state by introducing an alternative form of method update. In particular, we augment the syntax with an additional method declaration:

$$D ::= \ldots \mid I \ \textbf{has method}(I_1 \ \texttt{@=} \ E_1, \ldots, I_n \ \texttt{@=} \ E_n) \ \{ \ B \ E \ \}$$

Unlike the existing **has method** declaration, in this variant the specializer objects can be computed expressions rather than statically known objects (the generic function identifier *I* must still refer to a statically known object). In conjunction, we require all formals of such methods to use the **@=** specializer symbol. Dynamically, a formal of the form *I***@=***E*, where *E* evaluates to *o* when the method declaration is evaluated, applies only to *o* itself rather than to all descendants of *o*. The following code creates a “set” method illustrating the use of mutable state.

```
object setx isa (point, int)→point
setx has method(myP, newx) {
  x has method(p@=myP) { newx }
  myP }
```

By requiring that all argument positions use **@=**, we ensure that only a single argument tuple applies to the method. Consequently, there can be no possibility of ambiguity between this method and any other method declaration, so implementation-side typechecking can ignore **@=** methods.*

Regular client-side typechecking of the **has method** declaration will verify that the specializer expressions descend from the corresponding argument objects of the generic function's arrow object, and that the body of the method returns an object that descends from the result object of the generic function's arrow type. By requiring that the generic function to which the method is added be a statically known object rather than a computed expression, and by consistently using the generic function's most-specific arrow object when checking its associated **has method** declarations, we prevent soundness problems caused by subsumption in the face of method update.

5.3 Encapsulation

A simple approach to adding privacy to Dubious is to allow an optional **private** keyword to precede an **object** declaration in a module. Making an object private has the effect of disallowing importers of the module from seeing the object. However, it is unsafe in general for a module to be typechecked given only the public interfaces of its importees. The following is a simple example of the problems that can occur:

```
module BadMod {
  abstract object abs
  private object badFun isa (abs)→point
  object fun isa (abs)→point
  fun has method(a@abs) { badFun(a) }
}
module ImpMod imports BadMod {
  object conc isa abs
}
```

Because the ImpMod creates a new concrete implementation of the abstract abs object, it must check that the new implementation is correctly implemented (under both Systems M and E). If the typechecking of ImpMod does not see the private badFun generic function, then the conc object appears correctly implemented, because the fun generic function has an appropriate implementation. However, if this function is ever invoked on conc, the resulting invocation of badFun will cause a message-not-understood error to occur.

Our modular typechecking restrictions can be applied to overcome this problem. In addition to our normal checks on a module, we require that the private part pass the necessary checks as if it were a separate module that imported the public part of the module. (This means, for example, that a public object cannot inherit from a private object.) For the purposes of dividing up the module, a method is treated as private if its generic function or any of its specializer objects is private. If the private part obeys the rules on importers, then we know that importers of the module can be safely typechecked without seeing this private part. In our example above, the badFun object would need to pass implementation-side typechecking as if it were in its own module that imported a module containing only the public part of the module. Therefore, the object is considered to import the abs object, so badFun is subject to restrictions on open object idioms. In order to satisfy these restrictions, badFun must provide a default method implementation, thereby removing the potential type error that eludes typechecks from the ImpMod module.

---

* There is no conflict in having one @ and one @= method, each specialized to the same objects; both methods apply to that tuple of objects, but the @= method is treated as the more specific method. The @ method will still apply to any descending tuples.

5.4 Predicate Dispatching

A recent paper described predicate dispatching [Ernst *et al.* 98], a generalized dispatching model that subsumes single dispatching, multiple dispatching, pattern matching, and predicate classes. The basic idea is to specify a method's applicability by an arbitrary predicate over the method's formals, formed from conjunctions, disjunctions, and negations of descends-from tests (`@` specializers), equal-to tests (`@=` specializers), and arbitrary boolean-valued expressions. Method overriding is deduced from predicate implication. Under this model, a traditional multimethod is simply a method whose predicate is a conjunction of descends-from tests of the method's formals. Global static typechecking rules were presented for this model.

Modular typechecking of the predicate dispatching model follows from the observation that our modular typechecking rules for multimethods are already sufficient conditions for the safety of predicate methods. For example, System M requires that a method added to an imported generic function be an encapsulated-style method. In predicate dispatching, this corresponds to a method whose predicate is a conjunction where one conjunct is a test that the first formal descends from a local object. All other conjuncts of the predicate are unrestricted; such conjuncts can only further constrain the applicable argument tuples to the method.

6 Related Work

Chambers and Leavens made the first effort toward modular typechecking of symmetric multimethods [Chambers & Leavens 95], in the Cecil language [Chambers 92, Chambers 95]. They defined client-side and implementation-side typechecking and sketched informal ideas for modular typechecking, including the notions of extension modules and unique most-extending modules. In this sketch, objects are not allowed to conform to an imported type (objects and types are orthogonal in their model). When the object and type hierarchies parallel each other, as is the common case, this restriction disallows modules even from creating a singly inheriting child of an imported object. This makes importers unable to express most standard object-oriented programming idioms, forcing much of the program into extension modules and thereby largely giving up modular typechecking. They did not formalize the static or dynamic semantics of their modular language nor prove any soundness results for modular typechecking.

BeCecil [Chambers & Leavens 97] is a core language for multimethods, intended as a formalization of the work described above. It includes the same core features as Dubious, as well as types and subtyping separate from objects and inheritance, and a block structure that allows arbitrarily nested declarations. However, BeCecil does not have a module system. BeCecil's dispatching semantics is more complicated than Dubious's, with **`inherits`** (BeCecil's version of **`isa`**) and **`has method`** declarations associated with particular scopes and only visible to certain call sites. As a result, separate typechecking was not achieved. Dubious is in some ways a reaction to BeCecil's problems: Dubious treats **`isa`** and **`has method`** declarations as having global extent, simplifying the semantics enough for us to develop modular typechecking rules and prove their soundness.

The λ&-calculus and variants [Castagna *et al.* 92, Castagna 97] are a family of calculi based on overloading of symmetric multimethods. Dispatching is performed on types which, along with the subtyping relation, are predefined rather than programmer constructed. Some of the calculi have second-order type systems, which Dubious lacks. The language λ_object augments the calculi with the ability to define new types and subtyping relations. Neither the λ&-calculi nor λ_object address the issue of separate typechecking. Moreover, the functional nature of the & operator to add a method to a

generic function prevents spreading the definition of a generic function across unrelated modules, as would be needed to model independently developed subclasses.

$ML_{\leq}$ [Bourdoncle & Merz 97] is an ML-like language augmented with a form of classes and symmetric multimethods. The type system is more sophisticated than ours, including types separate from implementations and polymorphic multimethods. The authors sketch an extension to the language that adds modules for encapsulation. However, there is no separate typechecking, as these modules simply desugar into a global "letrec."

Kea [Mugridge *et al.* 91] is a statically typed, class-based language with symmetric multimethods. Kea has a notion of separate compilation, but this requires run-time checking of generic functions for type safety.

Tuple [Leavens & Millstein 98] is a language that provides dispatching on tuples in order to add symmetric multiple dispatch to existing singly dispatched languages. However, the modularity issue for multimethods is not addressed, as "tuple classes" must be typechecked globally. The syntax for tuple classes, which clearly separates the specialized and unspecialized argument positions of a generic function, has the same effect as the **#** markers in our System E.

Encapsulated multimethods [Castagna 95, Bruce *et al.* 95] are an attempt to solve the modularity problem for multimethods by embedding multimethods in the traditional object-oriented class model. An encapsulated multimethod first dispatches on the receiver argument, then dispatches on the remaining arguments. All the multimethods with a given receiver are encapsulated within the receiver class and are not extensible or inheritable outside this class. In the face of multiple inheritance, all the encapsulated multimethods within a class must be totally ordered. Given these restrictions, each class can be typechecked separately. The resulting kinds of allowable multimethods are similar to those allowed in our System M, although we are able to keep the symmetric multimethod dispatching semantics and to maintain ordinary inheritance of methods. In addition, encapsulated multimethods do not address open objects and abstract methods. Parasitic multimethods [Boyland & Castagna 97] are a variant of encapsulated multimethods adapted to Java. Parasitic multimethods are additionally complicated by the use of the textual order of methods to resolve ambiguities, the inheritance of parasitic methods in the presence of this textual ordering, and the need to retain backward compatibility with Java's blend of dynamic single dispatching and static overloading on arguments.

Common Lisp [Steele 90, Paepcke 93] and Dylan [Shalit 97, Feinberg *et al.* 97] are both multimethod-based languages with generic functions and module systems. To avoid run-time ambiguities, Common Lisp totally orders the arguments of a generic function; Dylan uses the symmetric model. Both Common Lisp and Dylan totally order the inheritance hierarchy, eliminating the potential for multiple-inheritance ambiguities. The module systems provide name-space management and encapsulation, allowing the creation of generic functions private to a module. The languages are dynamically typed, so they do not consider the issue of (separate) typechecking.

Polyglot [Agrawal *et al.* 91] is a database programming language akin to Common Lisp with a first-order static type system. There are no abstract methods and the type of a generic function is determined by the types of its methods, so there is no possibility of message-not-understood errors. Further, the dispatching uses Common Lisp-style total ordering of multimethod arguments and inheritance, avoiding ambiguities. Therefore, only the monotonicity of the result types [Castagna *et al.* 92, Reynolds 80] of multimethods needs to be checked to ensure type safety.

7 Conclusions and Future Work

Dubious is a statically typed core language that supports symmetric multimethods and separate typechecking. The core language includes explicit declaration of (possibly abstract) objects, (possibly multiple) inheritance, and first-class generic functions. We have defined several modular type systems for Dubious, with properties ranging from complete modularity at the cost of giving up certain programming idioms, to complete expressiveness at the cost of regional typechecking and a simple link-time check on regions, and we have proved the systems sound. Dubious represents the first formalization and the first proof of soundness of separate typechecking for symmetric multimethods.

In the future, we plan to formalize and prove sound the extensions to the language sketched in section 5. Other interesting extensions include supporting polymorphic types in the presence of separate typechecking, and supporting module types and first-class modules to program mixin classes [Bracha & Cook 90, Flatt *et al.* 98] and role-based programming [Reenskaug *et al.* 92, VanHilst & Notkin 96, Smaragdakis & Batory 98]. Finally, we are using the ideas in Dubious as a foundation for the design of Diesel, a practical programming language succeeding Cecil.

Acknowledgments

Thanks to Gary Leavens for his collaboration on BeCecil, the predecessor to Dubious, and for many discussions about typechecking for multimethods. Thanks to Vassily Litvinov for discussion on the formal semantics of Dubious. Thanks to Greg Badros, Michael Ernst, David Grove, Craig Kaplan, Gary Leavens, and Vassily Litvinov for helpful comments on an earlier draft of this paper.

This research is supported in part by an NSF grant (number CCR-9503741), an NSF Young Investigator Award (number CCR-9457767), and gifts from Sun Microsystems, IBM, Xerox PARC, Object Technology International, Edison Design Group, and Pure Software.

References

[Abadi & Cardelli 95] Martín Abadi and Luca Cardelli. An Imperative Object Calculus. *Theory and Practice of Object Systems*, **1**(3):151-166, 1995.

[Abadi & Cardelli 96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[Agrawal *et al.* 91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. *OOPSLA'91 Conference Proceedings*, Phoenix, AZ, October, 1991, volume 26, number 11 of *ACM SIGPLAN Notices*, pp. 113-128. ACM, New York, November, 1991.

[Arnold & Gosling 98] Ken Arnold and James Gosling. *The Java Programming Language*. Second Edition, Addison-Wesley, Reading, Mass., 1998.

[Baumgartner *et al.* 96] Gerald Baumgartner, Konstantin Läufer, and Vincent F. Russo. On the Interaction of Object-Oriented Design Patterns and Programming Languages. Technical Report CSD-TR-96-020, Department of Computer Science, Purdue University, February 1996.

[Bourdoncle & Merz 97] François Bourdoncle and Stephan Merz. Type Checking Higher-Order Polymorphic Multi-Methods. *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, pp. 302-315. ACM, New York, January 1997.

[Boyland & Castagna 97] John Boyland and Giuseppe Castagna. Parasitic Methods: An Implementation of Multi-Methods for Java. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 32, number 10 of *ACM SIGPLAN Notices*, pp. 66-76. ACM, New York, November 1997.

[Bracha & Cook 90] Gilad Bracha and William Cook. Mixin-Based Inheritance. *Proceedings of the Joint ACM Conference on Object-Oriented Programming Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, Ottawa, Canada, October 1990.

[Bruce *et al.* 95] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On Binary Methods. *Theory and Practice of Object Systems*, **1**(3):221-242, 1995.

[Canning *et al.* 89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, John C. Mitchell. F-Bounded Polymorphism for Object-Oriented Programming. *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273-280, September 1989.

[Cardelli 84] Luca Cardelli. A Semantics of Multiple Inheritance. *Semantics of Data Types Symposium*, LNCS 173, pp. 51-66, Springer-Verlag, 1984.

[Castagna *et al.* 92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, June, 1992, pp. 182-192, volume 5, number 1 of *LISP Pointers*. ACM, New York, January-March, 1992.

[Castagna 95] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431-447, 1995.

[Castagna 97] Giuseppe Castagna. *Object-Oriented Programming A Unified Foundation,* Birkhäuser, Boston, 1997.

[Chambers 92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. *ECOOP '92 Conference Proceedings*, Utrecht, the Netherlands, June/July, 1992, volume 615 of *Lecture Notes in Computer Science*, pp. 33-56. Springer-Verlag, Berlin, 1992.

[Chambers 95] Craig Chambers. The Cecil Language: Specification and Rationale: Version 2.0. Department of Computer Science and Engineering, University of Washington, December, 1995. http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html

[Chambers & Leavens 95] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. *ACM Transactions on Programming Languages and Systems*, **17**(6):805-843. November, 1995.

[Chambers & Leavens 97] Craig Chambers and Gary T. Leavens. BeCecil, A Core Object-Oriented Language with Block Structure and Multimethods: Semantics and Typing. *The Fourth International Workshop on the Foundations of Object-oriented Languages*, Paris, France, January 1997.

[Chambers 98] Craig Chambers. Towards Diesel, a Next-Generation OO Language after Cecil. Invited talk, *The Fifth Workshop on Foundations of Object-oriented Languages*, San Diego, California, January 1998.

[Cook 90] William Cook. Object-Oriented Programming versus Abstract Data Types. *Foundations of Object-Oriented Languages*, REX School/Workshop Proceedings, Noordwijkerhout, the Netherlands, May/June, 1990, volume 489 of *Lecture Notes in Computer Science*, pp. 151-178. Springer-Verlag, New York, 1991.

[Ernst *et al.* 98] Michael D. Ernst, Craig Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. *Twelfth European Conference on Object-Oriented Programming*, Brussels, Belgium, pp. 186-211, July, 1998.

[Feinberg *et al.* 97] Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass., 1997.

[Findler & Flatt 98] Robert Bruce Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. *International Conference on Functional Programming*, Baltimore, Maryland, September 1998.

[Flatt *et al.* 98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, pp. 171-183. ACM, New York, January 1998.

[Gamma *et al.* 95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[Gosling *et al.* 96] James Gosling, Bill Joy, Guy Steele, Guy L. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Mass., 1996.

[Kiczales *et al.* 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. In proceedings of the *Eleventh European Conference on Object-Oriented Programming*, Finland. Springer-Verlag LNCS 1241. June 1997.

[LaLonde et al. 86] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An Exemplar Based Smalltalk. *OOPSLA '86 Conference Proceedings*, pp. 322-330, Portland, OR, September, 1986. Published as *SIGPLAN Notices* 21(11), November, 1986.

[Leavens & Millstein 98] Gary T. Leavens and Todd D. Millstein. Multiple Dispatch as Dispatch on Tuples. *Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vancouver, British Columbia, October 1998.

[Lieberman 86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. *OOPSLA '86 Conference Proceedings*, pp. 214-223, Portland, OR, September, 1986. Published as *SIGPLAN Notices* 21(11), November, 1986.

[Millstein & Chambers 99] Todd Millstein and Craig Chambers. Modular Statically Typed Multimethods. Technical Report UW-CSE-99-03-02, Department of Computer Science and Engineering, University of Washington, March 1999.

[Mugridge *et al.* 91] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-Methods in a Statically-Typed Programming Language. *ECOOP '91 Conference Proceedings*, Geneva, Switzerland, July, 1991, volume 512 of Lecture Notes in Computer Science; Springer-Verlag, New York, 1991.

[Odersky & Wadler 97] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, pp. 146-159. ACM, New York, January 1997.

[Paepcke 93] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[Reenskaug *et al.* 92] T. Reenskaug, E. Anderson, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming*, 5(6): October 1992, pp. 27-41.

[Reynolds 80] John C. Reynolds. Using Category Theory to Design Implicit Conversions and Generic Operators. *Semantics-Directed Compiler Generation*, Aarhus, Denmark, pp. 211-258. Volume 94 of *Lecture Notes in Computer Science*, Springer-Verlag, NY, 1980.

[Shalit 97] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley, Reading, Mass., 1997.

[Smaragdakis & Batory 98] Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. *Twelfth European Conference on Object-Oriented Programming*, Brussels, Belgium, pp. 550-570, July 1998.

[Steele 90] Guy L. Steele Jr. *Common Lisp: The Language* (*second edition*). Digital Press, Bedford, MA, 1990.

[Ungar & Smith 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *OOPSLA '87 Conference Proceedings*, Orlando, Florida, volume 22, number 12, of *ACM SIGPLAN Notices*, pp. 227-241. ACM, New York, December, 1987.

[VanHilst & Notkin 96] M. VanHilst and D. Notkin. Using C++ Templates to Implement Role-Based Designs. *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, pp. 22-37.

Multi-Method Dispatch Using Multiple Row Displacement*

Candy Pang, Wade Holst, Yuri Leontiev, and Duane Szafron

University of Alberta, Edmonton AB T6G 2H1 Canada
{candy,wade,yuri,duane}@cs.ualberta.ca

Abstract. Multiple Row Displacement (MRD) is a new dispatch technique for multi-method languages. It is based on compressing an n-dimensional table using an extension of the single-receiver row displacement mechanism. This paper presents the new algorithm and provides experimental results that compare it with implementations of existing techniques: compressed n-dimensional tables, look-up automata and single-receiver projection. MRD uses comparable space to the other techniques, but has faster dispatch performance.

1 Introduction

Object-oriented languages can be separated into *single-receiver languages* and *multi-method languages*. *Single-receiver languages* use the dynamic type of a dedicated *receiver* object in conjunction with the method name to determine the method to execute at run-time. *Multi-method languages* use the dynamic types of one or more arguments[1] in conjunction with the method name to determine the method to execute. In single-receiver languages, a call-site can be viewed as a message send to the receiver object. In multi-method languages, a call-site can be viewed as the execution of a behavior on a set of arguments. The run-time determination of the method to invoke at a call-site is called *method dispatch*. Note that languages like C++ and Java that allow methods with the same name but different static argument types are not performing actual dispatch on the types of these arguments; the static types are simply encoded within the method name.

Since most of the commercial object-oriented languages are single-receiver languages, many efficient dispatch techniques have been invented for such languages [1]. However, there are some multi-method languages in use, such as Cecil [2], CLOS [3], and Dylan [4]. In such languages, multi-method dispatch is necessary.

There are two major categories of method dispatch: *cache-based* and *table-based*. *Cache-based* techniques look in either global or local caches at the time of dispatch to determine if the method for a particular call-site has already

* This research was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada under grant OGP8191

[1] In the rest of this paper, we will assume that dispatch occurs on all arguments.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 304–328, 1999.
© Springer-Verlag Berlin Heidelberg 1999

been determined. If it has been determined, that method is used. Otherwise, a *cache-miss* technique is used to compute the method, which is then cached for subsequent executions. *Table-based* techniques pre-determine the method for every possible call-site, and record these methods in a table. At dispatch-time, the method name and dynamic argument types form an index into this table. This paper focuses exclusively on table-based techniques. Table-based techniques have constant dispatch time. In addition, even when cache-based techniques are used, table-based techniques can be effectively used for cache-misses.

In this paper we present a new multi-method table-based dispatch technique. It uses a time efficient n-dimensional dispatch table that is compressed using an extension of a space efficient row displacement mechanism. Since the technique uses multiple applications of row displacement, it is called Multiple Row Displacement and will be abbreviated as MRD. MRD works for methods of arbitrary arity. Its execution speed and memory utilization are analyzed and compared to other multi-method table-based dispatch techniques.

The rest of this paper is organized as follows. Sect. 2 introduces some notation for describing multi-method dispatch. Sect. 3 presents the row displacement single-receiver dispatch technique. Sect. 4 summarizes the existing multi-method dispatch techniques. Sect. 5 describes n-dimensional table dispatch and presents the new multi-method table-based technique. Sect. 6 presents time and space results for the new technique and compares it to existing techniques. Sect. 7 discusses future work, and Sect. 8 presents our conclusions.

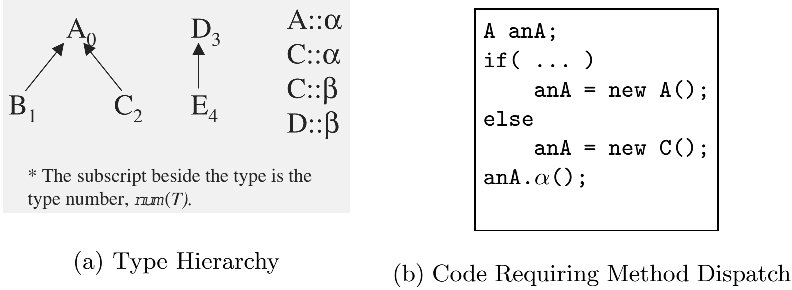
2 Terminology for Multi-Method Dispatch

2.1 Notation

Expr. 1 shows the form of a k-arity multi-method call-site. Each argument, o_i, represents an object, and has an associated *dynamic type*, $T^i = type(o_i)$. Let $\mathcal{H}$ represent a type hierarchy, and $|\mathcal{H}|$ be the number of types in the hierarchy. In $\mathcal{H}$, each type has a type number, $num(T)$. A directed *supertype edge* exists between type T_j and type T_i if T_j is a *direct subtype* of T_i, which we denote as $T_j \prec_1 T_i$. If T_i can be reached from T_j by following one or more supertype edges, T_j is a *subtype* of T_i, denoted as $T_j \prec T_i$.

$$\sigma(o_1, o_2, ..., o_k) \quad (1)$$

Method dispatch is the run-time determination of a method to invoke at a call-site. When a method is defined, each argument, o_i, has a specific static type, T^i. However, at a call-site, the dynamic type of each argument can either be the static type, T^i, or any of its subtypes, $\{T|T \preceq T^i\}$. For example, consider the type hierarchy and method definitions in Fig. 1a, and the code in Fig. 1b. The static type of `anA` is `A`, but the dynamic type of `anA` can be either `A` or `C`. In general, we do not know the dynamic type of an object at a call-site until run-time, so method dispatch is necessary.



(a) Type Hierarchy

(b) Code Requiring Method Dispatch

Fig. 1. An example hierarchy and program segment requiring method dispatch

Although multi-method languages might appear to break the conceptual model of sending a message to a receiver, we can maintain this idea by introducing the concept of a product-type. A *k-arity product-type* is an ordered list of k types denoted by $P = T^1 \times T^2 \times ... \times T^k$. The *induced k-degree product-type graph*, $k \geq 1$, denoted $\mathcal{H}^k$, is implicitly defined by the edges in $\mathcal{H}$. Nodes in $\mathcal{H}^k$ are k-arity product-types, where each type in the product-type is an element of $\mathcal{H}$. Expr. 2 describes when a directed edge exists from a child product-type $P_j = T_2^1 \times T_2^2 \times ... \times T_2^k$ to a parent product-type $P_i = T_1^1 \times T_1^2 \times ... \times T_1^k$, which is denoted $P_j \prec_1 P_i$.

$$P_j \prec_1 P_i \Leftrightarrow \exists i, 1 \leq i \leq k : (T_2^i \prec_1 T_1^i) \wedge (\forall j \neq i, T_2^j = T_1^j) \qquad (2)$$

The notation $P_j \prec P_i$ indicates that P_j is a *sub-product-type* of P_i, which implies that P_i can be reached from P_j by following edges in the product-type graph $\mathcal{H}^k$. Fig. 2 presents a sample inheritance hierarchy $\mathcal{H}$ and its induced 2-arity product-type graph, $\mathcal{H}^2$. Three 2-arity methods (γ_1 to γ_3) for behavior γ have been defined on $\mathcal{H}^2$ and associated with the appropriate product-types.[2] Note that for real inheritance hierarchies, the product-type hierarchies, $\mathcal{H}^2$, $\mathcal{H}^3$, ..., are too large to store explicitly. Therefore, it is essential to define all product-type relationships in terms of relations between the original types, as in Expr. 2. Next, we define the concept of a *behavior*. A behavior corresponds to a generic-function in CLOS and Cecil, to the set of methods that share the same signature in Java, and the set of methods that share the same message selector in Smalltalk. Behaviors are denoted by $\mathcal{B}_\sigma^k$, where k is the arity and σ is the name. The maximum arity for all behaviors in the system is denoted by K. Multiple methods can be defined for each behavior. A method for a behavior named σ is denoted by σ_j. If the static type of the ith argument of σ_j is denoted by T^i, the list of argument types can viewed as a product-type, $dom(\sigma_j) = T^1 \times T^2 \times ... \times T^k$. With multi-method dispatch, the dynamic types of all arguments are needed.[3]

[2] The method γ_4 in the dashed box is an implicit inheritance conflict definition, and will be explained later.

[3] In single-receiver languages, the first argument is called a receiver.

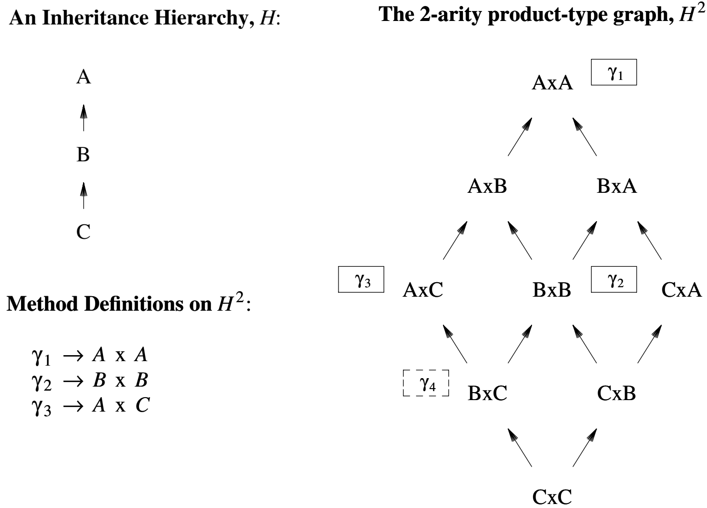


Fig. 2. An Inheritance Hierarchy, $\mathcal{H}$, and its induced Product-Type Graph $\mathcal{H}^2$

2.2 Inheritance Conflicts

In single-receiver languages with multiple inheritance, the concept of *inheritance conflict* arises. In general, an inheritance conflict occurs at a type T if two different methods of a behavior are visible (by following different paths up the type hierarchy) in supertypes T_i and T_j. Most languages relax this definition slightly. Assume that n different methods of a behavior are defined on the set of types $\mathcal{T} = \{T_1, ..., T_n\}$, where $T \preceq T_1, ..., T_n$. Then, the methods defined in two types, T_i and T_j in $\mathcal{T}$, do not cause a conflict in T, if $T_i \prec T_j$, or $T_j \prec T_i$, or $\{\exists\, T_k \in \mathcal{T} \mid T_k \prec T_i \;\&\; T_k \prec T_j\}$.

Inheritance conflicts can also occur in multi-method languages, and are defined in an analogous manner. A conflict occurs when a product-type can see two different method definitions by looking up different paths in the induced product-type graph $T^1 \times T^2 \times ... \times T^k$. Interestingly, inheritance conflicts can occur in multi-method languages even if the underlying type hierarchy, $\mathcal{H}$, has single inheritance. For example, in Fig. 2, the product-type $B \times C$ has an inheritance conflict, since it can see two different definitions for behavior γ (γ_3 in $A \times C$ and γ_2 in $B \times B$). For this reason, an implicit conflict method, γ_4, is defined in $B \times C$ as shown in Fig. 2. Similar to single-receiver languages, relaxation can be applied. Assume that n methods are defined in product-types $\mathcal{P} = \{P_1, ..., P_n\}$, and let $P \prec P_1, ..., P_n$. Then, the methods in P_i and P_j do not conflict in P if $P_i \prec P_j$, or $P_j \prec P_i$, or $\{\exists\, P_k \in \mathcal{P} \mid P_k \prec P_i \;\&\; P_k \prec P_j\}$. In multi-method languages, it is especially important to use the more relaxed definition of an

inheritance conflict. Otherwise, a large number of inheritance conflicts would be generated for almost every method definition.

2.3 Statically Typed Versus Dynamically Typed

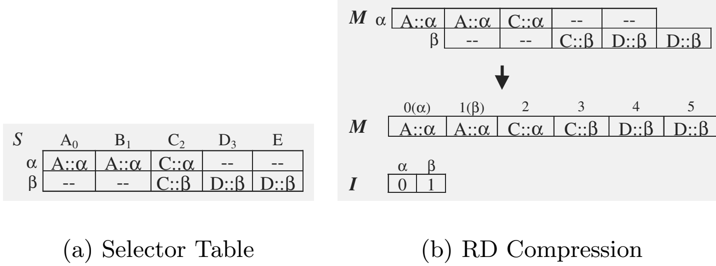
Some programming languages (C++, Java, Eiffel) require each variable to be declared with a static type. These languages are called statically typed languages. Other languages (Smalltalk, CLOS) which do not bind variables to static types, are called dynamically typed languages. In statically typed languages, a type checker can be used at compile-time to ensure that all call-sites are type-valid. A call-site is *type-valid*, if it has either a defined method for the message or an implicitly defined conflict method. In contrast, a call-site is type-invalid, if dispatching the call-site will lead to *method-not-understood*. For example, the static type of the variable anA is A in Fig. 1b. The dynamic type of anA can be either A or C (which is a subtype of A). Since the message α is defined for type A, no matter what its dynamic type is, anA can understand the message α. Therefore, the type checker can tell at compile-time that the call-site $anA.\alpha()$ is type-valid. If the static type of anA is D, neither D nor any of its supertypes understand the message α. The type checker will find at compile-time that the call-site $anA.\alpha()$ is type-invalid, and return a compile-time error.

With implicitly defined conflict methods in statically typed languages, no type-invalid call-site will be dispatched during execution. However, in dynamically typed languages, call-sites may be type-invalid. All dispatch techniques that use compression may return a method totally unrelated to the call-site. Therefore, in dynamically typed languages, a method prologue is used to ensure that the computed method is applicable for the dispatched behavior. It also ensures that each of the arguments is a subtype of the associated parameter type in the method.

The dispatch technique, Multiple Row Displacement introduced in this paper also has the problem of returning a wrong method for a type-invalid call-site in dynamically typed languages. The problem can be solved by minor changes to the data structure, see [18] for details. However, in this paper we assume the call-sites are statically typed.

3 Single-Receiver Row Displacement Dispatch (RD)

In single-receiver table dispatch, the method address can be calculated in advance for every legal class/behavior pair, and stored in a *selector table*, S. Fig. 3a shows the selector table for the type hierarchy and method definitions in Fig. 1a. An empty table entry means that the behavior cannot be applied to the type. At run-time, the behavior and the dynamic type of the receiver are used as indices into S [5]. This algorithm is known as STI in the literature [6]. Although STI provides efficient dispatch, its large memory requirements prohibit it from being used in real systems. For example, there are 961 types and 12130 different behaviors in the VisualWorks 2.5 Smalltalk hierarchy. If each method address



(a) Selector Table (b) RD Compression

Fig. 3. Compressing A Selector Table By Row Displacement

required 4 bytes, then the selector table would have more than 46.6 Mbytes ($961 \times 12130 \times 4$ *bytes*). Fortunately, 95% of the entries in the selector table for single-receiver languages are empty [7], so the table can be compressed.

Row displacement (RD) reduces the number of empty entries by compressing the two-dimensional selector table into a one-dimensional array [7,8]. As illustrated in Fig. 3b, each row in S is shifted by an offset until there is only one occupied entry in each column. Then, this structure is collapsed into a one-dimensional *master array*, M. When the rows are shifted, the shift indices (number of columns each row has been shifted) are stored in an index array, I.

At run-time, the behavior is used to find the shift index from the index array, I. In fact, each behavior has a unique index determined at compile-time, and it is this index which is used to represent the behavior in the compiled code. For simplicity, we will just use the behavior name in this paper. The shift index is added to the type number of the receiver to form an index into the master array, M. For example, to dispatch behavior β with D as the dynamic type of the receiver, the shift index for β is $I[\beta] = 1$. The type number of the receiver, D, is 3. Therefore, the final shift index is $1 + 3 = 4$, and the method to execute is at $M[4]$ which is D::β. Compared with other single-receiver table dispatch techniques, row displacement is highly space and time efficient [1]. We will show how this single-receiver technique can be generalized to multi-method languages in Sect. 5.

4 Existing Multi-Method Dispatch Techniques

This section provides a brief summary of the existing multi-method dispatch techniques.

1. *CNT: Compressed N-Dimensional Tables* [9,10,11] represents the dispatch table as a behavior-specific k-dimensional table, where k represents the arity of a particular behavior. Each dimension of the table is compressed by grouping identical dimension lines into a single line. The resulting table is indexed by *type groups* in each dimension, and mappings from type number to type group are kept in auxiliary data structures.

2. *LUA: Lookup Automata* [12,13] creates a lookup automaton for each behavior. In order to avoid backtracking, and thus exponential dispatch time, the automata must include more types than are explicitly listed in method definitions (inheritance conflicts must be implicitly defined). The automaton can then be converted to a function containing only *if-than-else* statements. At dispatch, this function is called to locate the correct method. LUA has been extended in [19].
3. *SRP: Single-Receiver Projections* [14] maintains k extended single-receiver dispatch tables and projects k-arity multi-method definitions onto these k tables. Each table maintains a bit-vector of applicable method indices, so dispatch consists of logically anding bit-vectors, finding the index of the right-most on-bit and returning the method associated with this index.
4. *Extended Cache-Based Techniques* are used in Cecil [2]. The cache-based techniques from single-receiver languages [6] are extended to work for product-types instead of just simple types.

5 Multiple Row Displacement (MRD)

5.1 N-dimensional Dispatch Table

In single-receiver method dispatch, only the dynamic type of the receiver and the behavior name are used in dispatch. However, in multi-method dispatch, the dynamic types of all arguments and the behavior name are used.

The single-receiver dispatch table can be extended to multi-method dispatch. In multi-method dispatch, each k-arity behavior, $\mathcal{B}^k_\sigma$, has a k-dimensional dispatch table, D^k_σ, with type numbers as indices for each dimension. Therefore, each k-dimensional dispatch table has $|\mathcal{H}|^k$. At a call-site, $\sigma(o_1, o_2, ..., o_k)$, the method to execute is in $D^k_\sigma[num(T^1)][num(T^2)]...[num(T^k)]$, where $T^i = type(o_i)$. For example, the 2-dimensional dispatch tables for the type hierarchy and method definitions in Fig. 4a are shown in Fig. 4b. In building an n-dimensional dispatch table, inheritance conflicts must be resolved. For example, there is an inheritance conflict at $E{\times}E$ for α, since both α_1 and α_2 are applicable for the call-site $\alpha(anE, anE)$. Therefore, we define an implicit conflict method α_3, and insert it into the table at $E \times E$.

N-dimensional table dispatch is very time efficient. However, analogous to the situation with selector tables in single-receiver languages, n-dimensional dispatch tables are impractical because of their huge memory requirements. For example, in the Cecil Vortex3 type hierarchy there are 1954 types. Therefore, a single 3-arity behavior would require 1954^3 bytes = 7.46 gigabytes.

5.2 Multiple Row Displacement by Examples

Multiple Row Displacement (MRD) is a time and space efficient dispatch technique which combines row displacement and n-dimensional dispatch tables. We will first illustrate MRD through examples, and then give the algorithm. The

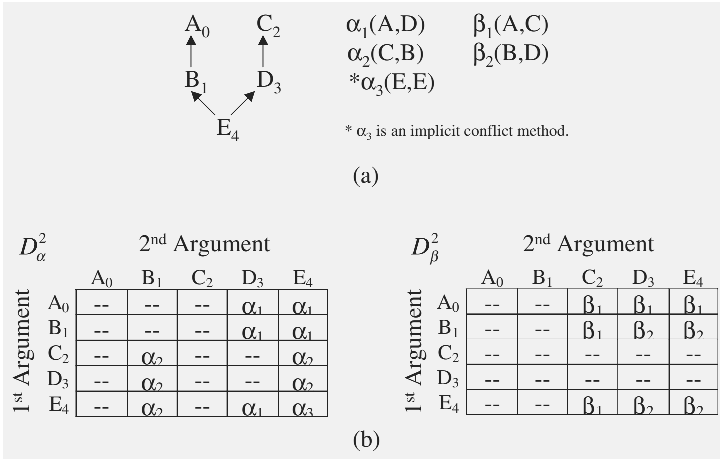


Fig. 4. N-Dimensional Dispatch Tables

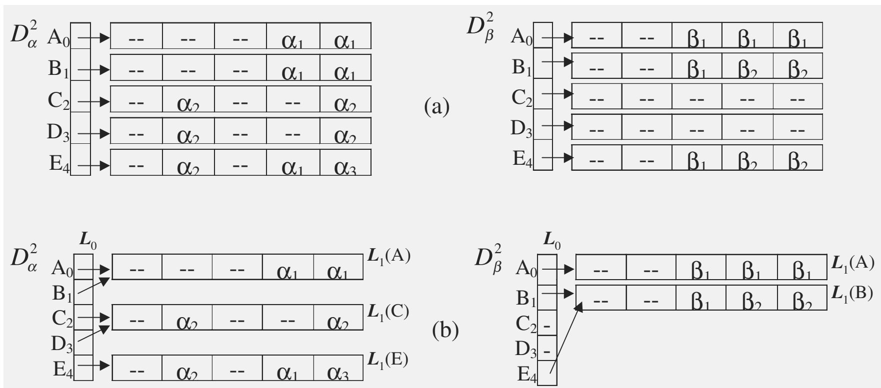


Fig. 5. Data Structure for Multiple Row Displacement

first example uses the type hierarchy and 2-arity method definitions from Fig. 4a. Instead of the single k-dimensional array shown in Fig. 4b, each table can be represented as an array of arrays as shown in Fig. 5a. The arrays indexed by the first argument are called *level*-0 arrays, L_0. There is only one *level*-0 array per behavior. The arrays indexed by the second argument are called *level*-1 arrays, $L_1(\cdot)$. If the arity of the behavior is greater than two then the arrays indexed by the third arguments are called *level*-2 arrays, $L_2(\cdot)$; and so on. The highest level arrays are *level*-$(k-1)$ arrays, $L_{k-1}(\cdot)$, for k arity behaviors.

It can be seen from Fig. 5a that some of the *level*-1 arrays are exactly the same. Those arrays are combined as shown in Fig. 5b. In general, there will be many identical rows in an n-dimensional dispatch table, and many empty rows. These observations are the basis for the CNT dispatch technique mentioned in Sect. 4, and are also one of the underlying reasons for the compression provided by MRD. It is worth noting that this sharing of rows is only possible due to the fact that we are compressing a table that uses types to index into all dimensions. In single-receiver languages, the tables being compressed have behaviors along one dimension, and types along the other. Sharing between two behavior rows would imply that both behaviors invoke the same methods for all types, and although languages like Tigukat [15] allow this to happen, such a situation would be highly unlikely to occur in practice. Sharing between two type columns is also unlikely since it occurs only when a type inherits methods from a parent and does not redefine or introduce any new methods. Such sharing of type columns is more feasible if the table is partitioned into subtables by grouping a number of rows together. This strategy was used in the single-receiver dispatch technique called Compressed Dispatch Table (CT) [16].

We have one data structure per behavior, D^k_σ, and MRD compresses these per behavior data structures by row displacement into three global data structures: a Global Master Array, M, a set of Global Index Arrays, I_j, where $j = 0, ..., (K-2)$, and a Global Behavior Array, B.

In compressing the data structure D^2_α in Fig. 5b, the *level*-1 array $L_1(A)$ is first shifted into the Global Master Array, M, by row displacement, as shown in Fig. 6a. The shift index, 0, is stored in the *level*-0 array, L_0, in place of $L_1(A)$. In the implementation, a temporary array is created to store the shift indices, but in this paper, we will put them in L_0 for simplicity of presentation. Fig. 6b shows how $L_1(C)$ and $L_1(E)$ are shifted into M by row displacement, and how they are replaced in L_0 by their shift indices. Finally, as shown in Fig. 6c, L_0 is shifted into the Global Index Array, I_0 by row displacement. The resulting shift index, 0, is stored in the Global Behavior Array at $B[\alpha]$. After D^2_α is compressed into the global data structures, the memory for its preliminary data structures can be released. Fig. 7 shows how to compress the behavior data structure, D^2_β, into the same global data structures, M, I_0 and B. The compression of the *level*-1 arrays, $L_1(A)$ and $L_1(B)$, are shown in Fig. 7a. The compression of the *level*-0 array, L_0, is shown in Fig. 7b. Note that only I_0 is used in the case of arity-2 behaviors. For arity-3 behaviors, I_1 will also be used. For arity-4 behaviors, I_2 will also be used, etc. As an example of dispatch, we will demonstrate how to

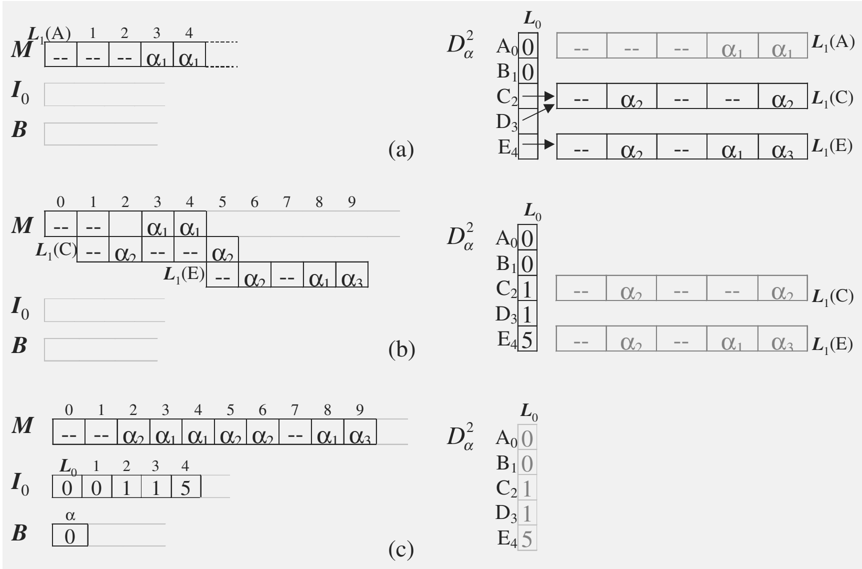


Fig. 6. Compressing The Data Structure for α

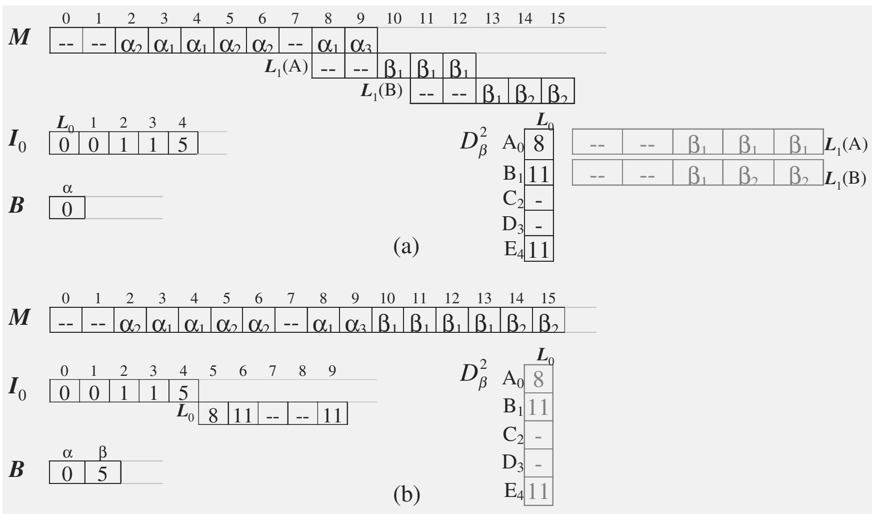


Fig. 7. Compressing The Data Structure For β

dispatch a call-site $\beta(anE, aD)$ using the data structures in Fig. 7b. The method dispatch starts by obtaining the shift index of the behavior, β, from the Global Behavior Array, B. From Fig. 7b, $B[\beta]$ is 5. The next step is to obtain the shift index for the type of the first argument, E, from the Global Index array, I_0. Since the shift index of β is 5, and the type number of E, num(E), is 4, the shift index of the first argument is $I_0[5+4] = I_0[9] = 11$. Finally, by adding the shift index of the first argument to the type number of the second argument, $num(D) = 3$, an index to M is formed, which is $11 + 3 = 14$. The method to execute can be found in $M[14] = \beta_2$, as expected. MRD can be extended to handle behaviors of any arity. Fig. 8a shows the method definitions of a 3-arity behavior, δ, and Fig. 8b shows its preliminary behavior data structure, D^3_δ. Figs. 8c to 8e show the compression of this data structure. First, the *level*-2 arrays, $L_2(B \times D)$, $L_2(D \times B)$ and $L_2(E \times E)$ are shifted into the existing M as shown in Fig. 8c. Their shift indices (15, 14, 19) are stored in $L_1(B)$, $L_1(D)$ and $L_1(E)$. In fact, every pointer in Fig. 8b that pointed to $L_2(B \times D)$ is replaced by the shift index 15. Pointers to $L_2(D \times B)$ are replaced by the shift index 14 and the single pointer to $L_2(E \times E)$ is replaced by the shift index 19. Then, the *level*-1 arrays, $L_1(B)$, $L_1(D)$ and $L_1(E)$, are shifted into the Global Index Array I_1 as shown in Fig. 8d. The shift indices (0,1,5) are stored in L_0. Finally, L_0 is shifted into the Global Index Array I_0 and its shift index (7) is stored in the Global Behavior Array at $B[\delta]$, as shown in Fig. 8e.

5.3 A Description of the Multiple Row-Displacement Algorithm

We have shown, by examples, how MRD compresses an n-dimensional dispatch table by row displacement. On the behavior level, a preliminary data structure, D^k_σ, is created for each behavior. D^k_σ is a data structure for a k-arity behavior named σ, as shown in Fig. 8b. It is actually an n-dimensional dispatch table, which is an array of pointers to arrays. Each array in D^k_σ has size $|\mathcal{H}|$. The *level*-0 array, L_0, is indexed by the type of the first argument. The *level*-1 arrays, $L_1(\cdot)$, are indexed by the type of the second argument. The *level*-$(k-1)$ arrays, $L_{k-1}(\cdot)$, always contain method addresses. All other arrays contain pointers to arrays at the next level.

After the compression has finished, there is a Global Master Dispatch Array, M, $K-1$ Global Index Arrays, I_0, ..., I_{k-2}, and a Global Behavior Array, B. The Global Master Dispatch Array, M, stores method addresses of all methods. Each Global Index Array, I_j, contains shift indexes for I_{j+1}. The Global Behavior Array, B stores the shift indices of the behaviors.

At compile-time, a D^k_σ data structure is created for each behavior. The *level*-$(k-1)$ arrays, L_{k-1}, are shifted into M by row displacement. The shifted indices are stored in L_{k-2}. Then, the *level*-$(k-2)$ arrays, L_{k-2}, are shifted into the index array, I_{k-2}. The shift indices are stored in L_{k-3}. This process is repeated until the *level*-0 array, L_0, is shifted into I_0, and the shift index is stored in $B[\sigma]$. The whole process is repeated for each behavior. The algorithm to compress all behavior data structures is shown in Sect. 5.5.

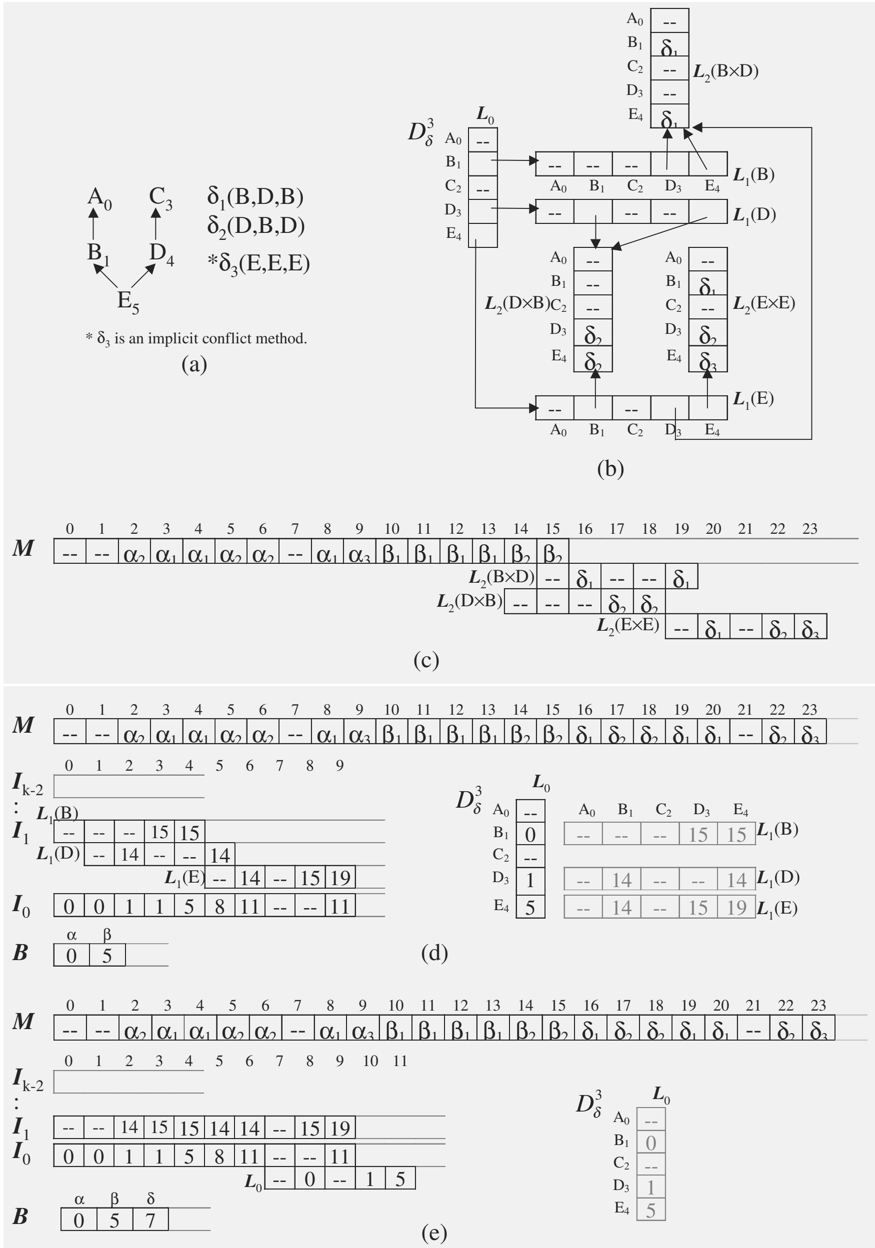


Fig. 8. Compressing The Data Structure For δ

The dispatch formula for a call-site, $\sigma(o_1, ..., o_k)$, is given by Expr. 3, where $T^i = type(o_i)$.

$$M[\ I_{k-2}[\ I_{k-3}[\ ...\ I_1[\ \ I_0[\ \ B[\ \sigma\ ] + num(T^1)\ ] \\ +\ num(T^2)\ ] + ...\ ] + num(T^{k-2})\ ] + num(T^{k-1})\ ] + num(T^k)\ ] \quad (3)$$

As an example of dispatch with Expr. 3, we will demonstrate how to dispatch a call-site $\delta(anE, aD, aB)$ using the data structures in Fig. 8e. Since δ is a 3-arity behavior, Expr. 3 becomes Expr. 4.

$$M[\ I_1[\ I_0[\ B[\ \delta\ ] + num(E)\ ] + num(D)\ ] + num(B)\ ] \quad (4)$$

Substituting the data from Fig. 8e into Expr. 4 yields the method δ_1, as shown in Expr. 5.

$$\begin{aligned} & M[\ I_1[\ I_0[\ 7 + 4\ ] + 3\ ] + 1\ ] \\ &= M[\ I_1[\ I_0[\ 11\ ] + 3\ ] + 1\ ] \\ &= M[\ I_1[\ 5 + 3\ ] + 1\ ] \\ &= M[\ I_1[\ 8] + 1\ ] \\ &= M[\ 15 + 1\ ] \\ &= M[\ 16\ ] = \delta_1 \end{aligned} \quad (5)$$

5.4 Optimizations

Single *I*. For simplicity of presentation, we defined an Index Array per arity position. Actually, we only need one Global Index Array, I, to store all *level*-0 to *level*-$(k-2)$ arrays. Using a single Index Array provides additional compression, and has no negative impact on dispatch speed. Expr. 6 shows the modified dispatch formula that accesses one Global Index Array.

$$M[\ I[\ I[\ ...\ I[\ \ I[\ \ B[\ \sigma\ ] + num(T^1)\ ] \\ +\ num(T^2)\ ] + ...\ ] + num(T^{k-2})\ ] + num(T^{k-1})\ ] + num(T^k)\ ] \quad (6)$$

Row-Matching. Note that the row-shifting mechanism used in our implementation of row displacement is not the most space-efficient technique possible. When the row-shifting algorithm is replaced by a more general algorithm called *row-matching* (based on string-matching), we get a higher compression rate. In row-matching, two table entries match if one entry is empty or if both entries are identical. For example, using row-shifting to compress rows R1 and R2 in Fig. 9a produces a master array with 9 elements as shown in Fig. 9b. However, using the improved algorithm to compress R1 and R2 produces a master array with only 6 elements as shown in Fig. 9c. Using row-matching instead of row-shifting provides an additional 10-14% compression. Our improved algorithm cannot be used in single-receiver row displacement, since different rows contain different behaviors, and thus different addresses.

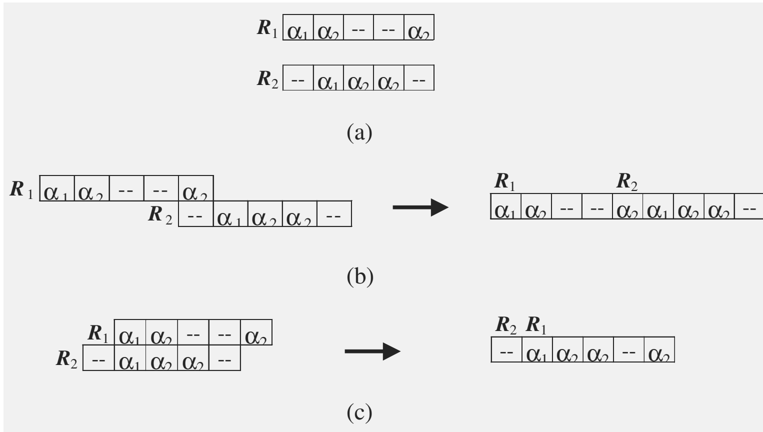


Fig. 9. Row-Shifting vs. Row-Matching

Byte vs. Word Storage. MRD stores four bytes function addresses in M. In a large hierarchy, M is the most memory consuming data structure. To reduce the size of M, a method-map, D_σ^{MRD}, is introduced per behavior. Since all methods of a behavior are stored in D_σ^{MRD}, a method can be represented by an index into D_σ^{MRD}. Since it is very unlikely that more than 256 methods are defined per behavior, only one byte is needed to store the index to the corresponding D_σ^{MRD}. If M stores this index instead of the function address, the size of M is reduced to one-forth of its original size. However, there is an extra indirection to access the method-map at dispatch time. We denote the technique which stores bytes instead of words by MRD-B.

Type Ordering. In single receiver row displacement type ordering has a significant impact on compression ratios [7]. We have investigated type ordering in multi-method row displacement and found that the impact is smaller.

5.5 The MRD Data Structure Creation Algorithm

The algorithm to build the global data structure for MRD is given below:

```
Array $M$, $I$;
createGlobalDataStructure() begin
    for(each behavior $B_\sigma^k$ ) do
        BehaviorStructure $D_\sigma^k$ = $B_\sigma^k$.createStructure();
        createRecursiveStructure( $D_\sigma^k.L_0$, 0 );
        $B_\sigma^k$.shiftIndex = $D_\sigma^k.L_0$.getShiftIndex();
    endfor
end
```

```
createRecursiveStructure( Array L, int level ) begin
    for(int i=0; i<L.size(); i++ ) do
        if( L[i] == null ) then
            continue;
        elseif( L[i].getShiftIndex() == -1 ) then
            if( level == k-2 ) then
                L[i] = M.add( L[i] );
            else
                createRecursiveStructure(L[i],level+1);
                L[i] = L[i].getShiftIndex();
            endif
        else
            L[i] = L[i].getShiftIndex();
        endif
    endfor
    I.add( L );
end
```

This algorithm uses three support routines: `Array.add(Array)`, `Array.getShiftIndex()`, and `Behavior.createStructure()`. The `Array.add(Array)` function shifts the given array into the current array by row-matching or row-shifting, and returns the shift index. The returned shift index is also stored in the given array. The `Array.getShiftIndex()` function returns the shift index of the current array, which is stored in the current array when it is added to another array. If the current array has never been added to another array, this function returns -1. The `Behavior.createStructure()` function creates an n-dimensional table for the current behavior.

5.6 Separate Compilation

With table-based dispatch, the tables must be built before they can be used. If a language does not support separate compilation, then the tables can be built at compile-time when the entire type hierarchy and all the method definitions are compiled. If a language supports separate compilation, then neither the type hierarchy nor the set of all method definitions for a particular behavior are available when a class is being compiled. In this case, the dispatch tables must be built at link-time. Fortunately, these tables only take a few seconds to build. In addition to building the dispatch tables, call-sites in compiled code must be patched with base table start addresses and global behavior shift indices. However, this is no more difficult than resolving other external references in separately compiled object files.

6 Performance Results

Here we present memory and execution results for the new technique, MRD, and three other techniques, CNT, LUA and SRP. When analyzing dispatch tech-

niques, both execution performance and memory usage need to be addressed. A technique that is extremely fast is still not viable if it uses excessive memory, and a technique that uses very little memory is not desirable if it dispatches methods very slowly. We present both timing and memory results for MRD, SRP, LUA and CNT. This is the first time a comparison of multi-method techniques has appeared in the literature.

The rest of this section is organized into three subsections. The first subsection discusses the data-structures and dispatch code required by the various techniques. The second subsection presents timing results. The third subsection presents memory results.

6.1 Data Structures and Dispatch Code

This section provides a brief description of the required data-structures for each of the four dispatch techniques in a static context. The code that needs to be generated at each call-site is also presented. In the subsections that follow, the code presented refers to the code that would be generated by the compiler upon encountering the call-site $\sigma(\ o_1,\ o_2,\ ...,\ o_k\ )$.

The notation $N(o_i)$ represents the code necessary to obtain a type number for the object at argument position i of the call-site. Naturally, different languages implement the relation between object and type in different ways, and dispatch is affected by this choice. Our timing results are based on an implementation in which every object is a pointer to a structure that contains a 'typeNumber' field (in addition to its instance data).

MRD. MRD has an M array that stores function addresses, an I array that stores level-array shift indices, and a B array that stores behavior shift indices.

The dispatch sequence is given in Expr. 7.

$$(*(M[\ I[\ ...I[\ I[\#b^\sigma + N(o_1)\ ] + N(o_2)\ ] + ...\ ] + N(o_{k-1})\ ] + N(o_k)\ ])\ )(o_1, o_2, ..., o_k) \quad (7)$$

Note that the Global Behavior Array, B, from Expr. 3, is known at compile-time, so $B[\sigma]$ is known at compile-time. Thus $\#b^\sigma$ is a literal interger obtained from $B[\sigma]$.

MRD-B. The dispatch sequence for MRD-B is given in Expr. 8.

$$(*(D_\sigma^{MRD}[\ M[\ I[\ ...I[\ I[\ \#b^\sigma + N(o_1)\ ] + N(o_2)\ ] + ...\ ] + N(o_{k-1})\ ] + N(o_k)\ ]\ ])\ )(o_1, o_2, ..., o_k) \quad (8)$$

CNT. For each behavior, CNT has a k-dimensional array, but since we are assuming a static environment, this k-dimensional array can be linearized into a

one-dimensional array. Indexing into the array requires a sequence of multiplications and additions to convert the k indices into a single index. For a particular behavior, we denote its one-dimensional dispatch table by D_σ^{CNT}.

In addition to the behavior-specific information, CNT requires arrays that map types to type-groups. In [11], these group arrays are compressed by selector coloring (SC). Our dispatch results are based on such a compression scheme, and assume that the maximum number of groups is less than 256, so that the group array can be an array of bytes. Furthermore, since the compiler knows exactly which group array to use for a particular type, it is more efficient to declare n statically allocated arrays than it is to declare an array of arrays. Thus, we assume that there are arrays G_1, ..., G_n, and that the compiler knows which group array to use for each dimension of a particular behavior.

If we assume that the compressed n-dimensional table for k-arity behavior σ has dimensions $n_1^\sigma, n_2^\sigma, ..., n_k^\sigma$, where the n_i^σ values are behavior specific, and that the group arrays for these dimensions are $G_1^\sigma, G_2^\sigma, ..., G_k^\sigma$ then the call-site dispatch code is given in Expr. 9.

$$
\begin{aligned}
(*\ (D_\sigma^{CNT}[\quad & G_1^\sigma[N(o_1)] \times \#(n_1^\sigma \times n_2^\sigma \times ... \times n_{k-1}^\sigma) \\
+\ & G_2^\sigma[N(o_2)] \times \#(n_2^\sigma \times ... \times n_{k-1}^\sigma) \\
+\ & ... \\
+\ & G_k^\sigma[N(o_k)]\ \ ]\ )\ )\ (o_1, o_2, ...o_k)
\end{aligned}
\tag{9}
$$

Note that since the n_i^σ are known constants, the products of the form: $\#(n_1^\sigma \times ... \times n_j^\sigma)$, can be precomputed. Thus, only $k-1$ multiplications are required at run-time.

Note that [11] assumes a behavior specific function-call to compute the dispatch using Expr. 9. Although this function-call reduces call-site size, it significantly increases dispatch time. We have remove the function-call by inlining to make CNT more competitive in our timings.

SRP. SRP has K selector tables, denoted $S_1, ..., S_K$ where S_i represents the applicable method sets for types in argument position i of all methods. These dispatch tables can be compressed by any single-receiver dispatch technique, such as selector coloring (SRP/SC), row displacement (SRP/RD), or compressed dispatch table (SRP/CT). The timing and space results, and the code that follows, are for SRP/RD.

In addition to the argument-specific dispatch tables, SRP has, for each behavior, an array that maps method indices to method addresses, which we denote by D_σ^{SRP}.

The dispatch code for SRP is given in Expr. 10, where FirstBit() is some macro or function that implements the operation of finding the position of the first '1' bit in a bit-vector. [14] discusses this in some detail. Our timing and space results assume that this is a hardware-supported operation with the same

performance as shift-right.

$$(*(D_{\sigma}^{SRP}[\ FirstBit(S_1[N(o_1) + \#b_1^{\sigma}]\ \& \\ S_2[N(o_2) + \#b_2^{\sigma}]\ \& \\ ...\ \& \\ S_k[N(o_k) + \#b_k^{\sigma}])\ ]\ )\ )\ (o_1, o_2, ..., o_k) \tag{10}$$

Note that $\#b_i^{\sigma}$ is the shift index assigned to behavior σ in argument-table i and is a literal integer.

LUA. LUA is, in some ways, the most difficult technique to evaluate accurately. First, there are a number of variations possible during implementation, that have vastly different space vs. time performance results. For example, in order to provide dispatch in O(k), the technique must resort to an array access in certain situations, at the expense of substantially more memory. Second, [13] does not provide any explicit description of what the code at a particular call-site would look like. They discuss the technique in terms of data structures, and do not mention that in a statically-typed environment, a collection of *if-then-else* statements would be a much more efficient implementation. It is only indicated later in [19] that method dispatch will happen as a function-call to a behavior-specific function. Given this assumption the call-site code for LUA is given in Expr. 11.

$$dispatch_{\sigma}(o_1, o_2, ..., o_k); \tag{11}$$

Although the published discussion of CNT also assumes such a behavior-specific call, we have provided a more time-efficient implementation of CNT by inlining the dispatch computation (Expr. 9), at the expense of more memory per call-site. Unfortunately, it is not feasible to inline the dispatch computation for LUA because the call-site code would grow too much.

Our timing results assume the best possible dispatch situation for LUA, in which there are only two k-arity methods from which to choose. In such a situation, LUA needs to perform at most k subtype tests. Although numerous subtype-testing implementations are possible [17,19], we have choosen one that provides a reasonable trade-off between time and space efficiency. Each type, T, maintains a bitvector, sub_T, in which the bit corresponding to every subtype of T is set to 1, and all other bits are set to 0. Assuming the bit-vector is implemented as an array of bytes, we can pack 8 bits into each array index, so determining whether T_j is a subtype of T_i consists of the expression: $sub_{T_i}[num(T_j) >> 3] \& (1 << (num(T_j)\ \&\ 0x7)\ )$. However, note that the actual subtype testing implementation does not really affect the overall dispatch time because LUA invokes a behavior-specific dispatch function, and this extra function call is, in general, much more expensive than the actual computation itself.

The size of the per behavior function to be executed depends on the number of methods defined for the behavior. In the best possible case, there are only two

methods, m_1 and m_2 defined for each behavior in a statically typed language (if there is only one method, no dispatch is necessary). We reiterate that this is a rather liberal under-estimate of the actual time a particular call-site takes to dispatch. The simplest function that a behavior can have is shown in the code:

```
dispatch_σ( o1, ..., ok ) {
        if ( sub_T1[N(o1) >> 3] & ( 1 << (N(o1) & 0x7) ))
            ...
                if ( sub_Tk[N(ok) >> 3] & ( 1 << (N(ok) & 0x7) ))
                    return call m1(o1, ..., ok);
        return call m2(o1, ..., ok);
}
```

6.2 Timing Results

In order to compare the address-computation time of the various techniques we generate technique-specific C++ programs that perform the computations listed in the previous section. Each program consists of a loop that iterates 2000 times over 500 blocks of code representing the address-computation for randomly generated call-sites, where a call-site consists of a behavior name and a list of k applicable types (for a k-arity behavior). Each block consists of two expressions. The first expression assigns to a global variable the result of an address-computation (i.e. the code described in the previous section, without the actual invocation). The second expression in each group calls a dummy function that modifies the previously assigned variable. These contortions are performed in order to stop the compiler from doing optimizations (such as only performing the last assignment in each group of 500, or in moving the code outside the 2000-iteration loop). Note that we are timing just the computation of addresses, since this is the only part of the dispatch process that varies from technique to technique (the actual invocation of the computed address is the same in all techniques). We also time a loop over 500 constant assignments interleaved with calls to the dummy function in order to time the overhead incurred (this is referred to as *noop* in the results).

Thus, each execution of one of these programs computes the time for 1,000,000 method-address computations. For each technique, such a program is generated and executed 20 times. The program is then regenerated (thus resulting in a different collection of 500 call-sites) an additional 9 times, and each such program is executed 20 times. This provides 200 timings of 1,000,000 call-sites for each of the techniques. The average time and standard-deviation of these 200 timings are reported in our results. In the graph, the histograms represent the mean, and the error-bars indicate the potential error in the results, as plus and minus twice the standard deviation.

In order to establish the effect that architecture and optimization have on the various techniques, the above timing results are performed on five different platforms using optimization levels from -O0 to -O3. All code is compiled using GNU C++ (in future work, we will obtain timings for a variety of different

compilers). In the interest of space, we present results for two platforms, and only for optimization level -O2. Furthermore, we only present results for 2-arity dispatch, since all techniques scale similarily for higher-arity dispatch sequences. In this and subsequent sections, Platform1 refers to a 299MHz Sun Microsystems Ultra 5/10 running Solaris 2.6 with 128 Mb of RAM and Platform2 refers to a 400MHz Prospec PII running Linux 2.0.34 with 256Mb of RAM. From Fig. 10, it

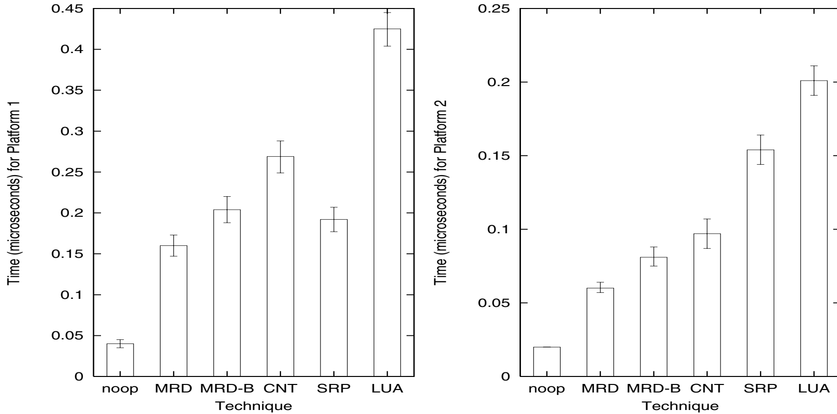


Fig. 10. Number of microseconds required to compute a method at a call-site

can be seen that MRD provides the fastest dispatch time on both platforms, and did so for all five platforms tested.[4] Furthermore, LUA has the slowest dispatch time on all platforms. However, the relative performance of MRD-B, SRP and CNT varied with platform, although MRD-B was usually fastest, followed by SRP, followed by CNT.

6.3 Memory Utilization

We can divide memory usage into two different categories: 1) data-structures, and 2) call-site code-size. The amount of space taken by each of these depends on the application, but in different ways. An application with many types and methods will naturally require larger data-structures than an application with fewer types and methods. As well, although the size of an individual call-site is independent of the application, the number of call-sites (and hence the amount of code generated) is application dependent.

In order to compare the call-site size of the various techniques, we generated another set of technique-specific C++ programs. For each technique, a program

[4] The other three platforms were: a Sun SPARCstation 10 Model 50 running SunOS 4.1.4 with 128 Mb, an 180MHz SGI O2 running IRIX 6.5 with 64 Mb, and an IBM RS6000/360 running AIX 4.1.4 with 128 Mb

was created that represented the code for 200 consecutive two-arity method invocations, including the dispatch computation. The program placed a label at the beginning and end of this code and reported the computed average call-site size based on the difference between the addressed of the labels. Note that the call-site size for a particular technique can vary slightly if the randomly generate arguments happen to be identical, or if the constants in the dispatch computation happen to be less than 256 or less than 65535, allowing them to be stored using smaller instructions. Fig. 11 shows the number of bytes required by the call-site

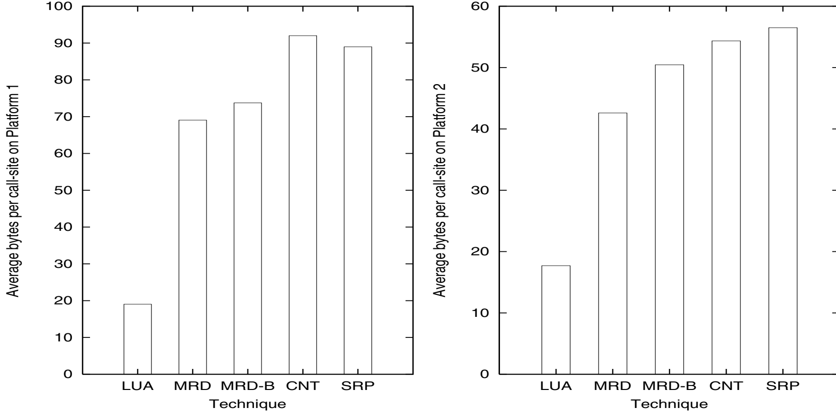


Fig. 11. Call-Site Memory Usage

dispatch code. Similar results are returned from higher arity behaviors. Since the data-structure size is dependent on an application, we chose to measure the size required to maintain information for all types and all behaviors in the Cecil Vortex3 (Cecil compiler [20]) hierarchy and the Harlequin Dylan hierarchy (a Dylan [4] GUI hierarchy called *duim*). Harlequin is a commercial implementation of Dylan. The Cecil Vortex-3.0 hierarchy contains 1954 types, 11618 behaviors and 21395 method definitions. The Dylan hierarchy contains 666 types, 2146 behaviors and 3932 method definitions.

In order to measure the amount of space required by the various techniques, we filtered the set of all possible behaviors to arrive at the set of behaviors that truly require multi-method dispatch. In particular, we do not consider any 0-arity or 1-arity behaviors, because the address for such behaviors can be identified at compile-time and with single-receiver techniques respectively. Furthermore, since our data assumes a statically-typed language, we ignore behaviors with only one method defined on them, since they too can be determined at compile-time. Finally, for each remaining behavior, we remove any arguments in which only one type participates. If there is only one type in an argument position, no dispatch is required on that argument. For example, if behavior σ

# Arity	*# Behavior*
2	203
3	22
4	11

Method Count	*# Behavior*
2	53
3	33
4	35
5-8	41
9-15	14
16-32	12
33+	4

(a) Cecil Vortex3 Type Hierarchy

# Arity	*# Behavior*
2	95
3	13

Method Count	*# Behavior*
2	21
3	11
4	32
5-8	23
9-15	12
16-32	7
33+	2

(b) Harlequin Type Hierarchy

Fig. 12. Type Hierarchy Details for Two Different Hierarchies

is defined only on $A \times A$, $B \times A$ and $C \times A$, then no dispatch on the second argument is required (because we are assuming statically typed languages). By reducing behaviors down to the set of arguments upon which dispatch is truly required, we get an accurate measure of the amount of multi-method support the language requires. After the reduction, the Cecil Vortex3 hierarchy has 1954 types, 226 behaviors and 1879 methods, and the Dylan hierarchy has 666 types, 108 behaviors and 738 methods. The method distributions of these hierarchies are shown in Fig. 12. The data-structure memory usage for each technique is shown in Fig. 13. In these reduced Cecil Vortex3 and Dylan hierachies, many

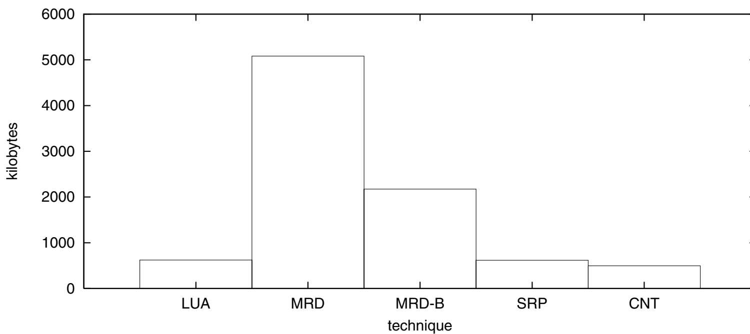


Fig. 13. Static Data Structure Memory Usage for Cecil Vortex3

of the method definitions have arguments typed as the *root-type*. Whenever an argument is typed as the *root-type*, MRD suffers. All rows on the dimension of that argument will be filled, so that, not much compression can be claimed from row-shifting or row-matching. More research is needed to find out whether it is a common practice to define many methods with arguments typed as the *root-type* in multi-method programming languages. However, if we remove all methods with *root-typed* argument(s) from the reduced Cecil Vortex3 hierarchy, the data structure size of each technique is profoundly different from those shown in Fig. 13. As multi-methods become more common, we expect that the actual distribution of methods will be somewhere between these two extremes.

After removing all methods with *root-typed* argument(s), there are 1661 types, 660 behaviors and 1299 methods remaining in the Cecil Vortex3 hierarchy. The data structure size of each technique for this *no root-type* Cecil Vortex3 hierarchy is shown in Fig. 14. The results for the Dylan hierarchy are similar.

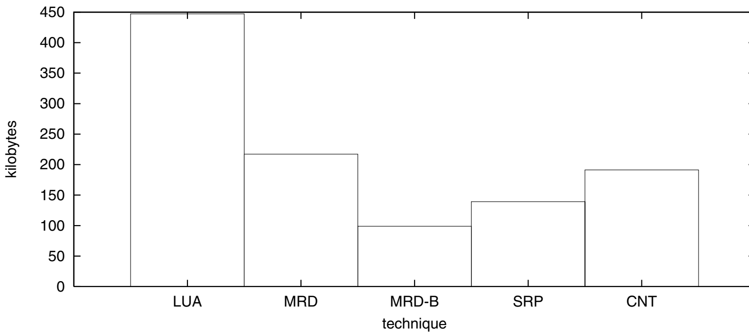


Fig. 14. Static Data Structure Memory Usage for *No Root-Typed* Cecil Vortex3

7 Future Work

The research that produced MRD is part of a larger research project analyzing various multi-method dispatch techniques. Numerous issues impact the performance results given in this paper. For example, the simple loop-based timing approach poses a problem. It reports an artificially deflated execution time due to caching effects. Since the same data is being executed 10 million times, it stays hot. This problem can be partially solved by generating large sequences of random call-sites on different behaviors with different arguments. However, this approach might actually discount caching effects that would occur in a real program, since random distributions of call-sites will have poorer cache performance than real-world applications that have locality of reference.

Furthermore, some of the techniques allow for a variety of implementations. The implementations usually trade space for time, so we can choose the implementation with the execution and memory footprint that most closely satisfies our application requirements. Also related to the issue of implementation is the impact of inlining of dispatch code. In single-receiver languages, the dispatch code is placed inline at each call-site, but some of the multi-method dispatch techniques have large call-site code chunks. For example, LUA defines a single dispatch function for each behavior. This function reduces call-site size, but significantly increases dispatch time. Rather than always calling a function, conditional inlining of a call-site is an open area of future research.

In order to obtain the best possible analysis of the various techniques, we need some indepth metrics on the distribution of behaviors in multi-method languages. In particular, the number of behaviors of each arity, and the numbers of methods defined per behavior are critical. As more and more multi-method languages are introduced, we will be able to get a better feel for realistic distributions. Note that call-site distributions are especially important for accurate analysis of LUA, since its dispatch time depends on the average number of types that need to be tested before a successful match occurs.

8 Conclusion

We have presented Multiple Row Displacement (MRD), a new multi-method dispatch technique that compresses an n-dimensional table by row displacement. It has been compared with existing table-based multi-method techniques, CNT, LUA and SRP. MRD has the fastest dispatch time and the second smallest per-call-site code size (next to LUA, which uses a function call). If the other techniques used a function call, they could reduce their call-site size at the expense of dispatch time.

In addition to presenting the new technique, we have provided the first performance comparison of the existing table-based multi-method dispatch techniques.

References

1. Holst, W., Szafron, D.: A General Framework For Inheritance Management and Method Dispatch in Object-Oriented Languages, ECOOP'97.
2. Chambers, C.: Object-Oriented Multi-Methods in Cecil, ECOOP'92 Conference Proceedings, 1992.
3. Bobrow, B., DeMichiel, D., Gabriel, R., Keene, S., Kiczales G., Moon, D.: Common Lisp Object System Specification, June 1988, X3J13 Document 88-002R.
4. Dylan Interim Reference Manual, Apple Computer, Inc., 1994.
5. Cox, B.: Object-Oriented Programming, An Evolutionary Approach, Addison-Wesley, 1987.
6. Driesen, K., Hölzle, U., Vitek, J.: Message Dispatch on Pipelined Processors, ECOOP'95 Conference Proceedings, 1995.
7. Driesen, K., Hölzle, U.: Minimizing Row Displacement Dispatch Tables, OOPSLA'95 Conference Proceedings, 1995.

8. Driesen, K.: Selector Table Indexing and Sparse Arrays, OOPSLA'93 Conference Proceedings, 1993.
9. Amiel, E., Gruber, O., Simon, E.: Optimizing Multi-Method Dispatch Using Compressed Dispatch Table, OOPSLA'94 Conference Proceedings, 1994.
10. Dujardin, E., Amiel, E., Simon, E.: Fast Algorithms for Compressed Multi-Method Dispatch Table Generation, TOPLAS'96 Conference Proceedings, 1996.
11. Dujardin, E., Amiel, E., Simon, E.: Fast Algorithms for Compressed Multi-Method Dispatch Table Generation, TOPLAS Journal, vol. 20, no. 1, Jan 1998, p116-165.
12. Chen, W., Turau, V., Klas, W.: Efficient Dynamic Look-up Strategy for Multi-methods, ECOOP'94 Conference Proceedings, 1994.
13. Chen, W.: Efficient Multiple Dispatching Based on Automata, Darmstadt, Germany, 1995.
14. Holst, W., Szafron, D., Leontiev, Y., Pang, C.: Multi-Method Dispatch Using Single-Receiver Projections, TR-98-03, University of Alberta, Edmonton, Alberta, Canada, 1998.
15. Özsu, T., Peters, J., Szafron, D., Irani, B., Lipka, A., Muñoz, A.: TIGUKAT: A Uniform Behavioral Objectbase Management System, VLDB'95 Conference Proceedings, 1995.
16. Vitek, J., Nigel Horspool, R.: Compact Dispatch Tables for Dynamically Typed Programming Languages, CC'96 Conference Proceedings, 1996.
17. Krall, A., Vitek, J., Nigel Horspool, R.: Near Optimal Hierachical Encoding of Types, ECOOP'97 Conference Proceedings, 1997.
18. Pang, C.: Multi-Method Dispatch Using Multiple Row Displacement, thesis, University of Alberta, 1999.
19. Chambers, C., Chen, W.: Efficient Predicate Dispatching, Technical Report UW-CSE-98-12-02, Department of Computing Science and Engineering, Univeristy of Washington.
20. Chambers, C.: Object-oriented multi-methods in Cecil, ECOOP'92 Conference Proceedings, 1992.

Internal Iteration Externalized

Thomas Kühne

Staffordshire University, UK
T.Kuehne@soc.staffs.ac.uk

Abstract. Although it is acknowledged that internal iterators are easier and safer to use than conventional external iterators, it is commonly assumed that they are not applicable in languages without builtin support for closures and that they are less flexible than external iterators.
We present an iteration framework that uses objects to emulate closures, separates structure exploration and data consumption, and generalizes on folding, thereby invalidating both the above statements. Our proposed "transfold" scheme allows processing one or more data structures simultaneously without exposing structure representations and without writing explicit loops.
We show that the use of two functional concepts (function parameterization and lazy evaluation) within an object-oriented language allows combining the safety and economic usage of internal iteration with the flexibility and client control of external iteration. Sample code is provided using the statically typed EIFFEL language.

1 Introduction

Collections play an important role in software design. Slightly surprisingly, the case on how to organize collection libraries and provide iteration facilities for them has not been closed yet. One reason for this are differences in the implementation languages used [22]. But also for a single language alternative designs compete with each other [30]. In the following we investigate how to design a general iteration scheme that is both flexible and easy to use. This paper compares the two main approaches to iteration, internal and external iteration, and arrives at a synthesis that virtually retains the advantages of both. The description of the problem is followed by a solution whose details and advantages are described in Sect. 4. The concluding remarks summarize and examine the implications for language design.

2 Issues in Iterator Design

Collections of data elements, such as lists, trees, graphs, etc., are without doubt very useful in the design of systems and for the implementation of algorithms. Although we often want to treat a collection as a single entity we also frequently need to individually access the contained data elements. The established mechanism for accomplishing this is an iterator, allowing us to step over all data elements until all have been visited. Whether we just want to access the elements,

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 329–350, 1999.
© Springer-Verlag Berlin Heidelberg 1999

e.g., for printing (read iterator), search for a particular element satisfying certain criteria, modify the elements, or create a new collection of elements (write iterator), depends on the intended purpose but the same basic iteration scheme underlies all these examples.

It is clearly not an option to let clients iterate over collections using knowledge about collection internals. Each client iteration would be subject to change when collection implementations change. Evidently, an iterator abstraction is necessary that allows accessing the elements of a collection independently of their respective internal representation. With the decision for a dedicated iteration abstraction, however, the following issues must be resolved:

- *How to combine iteration and action?* How do we combine an iteration algorithm (e.g., a simple loop) with a particular function or action (e.g., print an element)? The iterator client may do it by calling both, or the iterator could be subclassed for each function, or the iterator could accept a function.
- *Who knows how to iterate a collection?* Where is the best place to put the iteration logic? Will the collection explore itself or is it better to externalize such functionality?
- *Who controls the iteration?* Who is in control of advancing the iteration and who may decide to stop prematurely, i.e., avoid a full collection exploration?
- *How to support multiple iteration strategies?* Non-linear collections, such as trees and graphs support different traversal strategies like breadth-first-search and depth-first-search with variations such as pre-order, in-order, and post-order traversal. How do we allow a choice between alternatives without excessively widening the collection's interface?
- *How to allow several iterations in parallel?* In which way can we support multiple iterators using a shared collection? Two or more clients may want to process the same collection with interleaving execution. It may even be the case that a reading iterator uses the results of a write iterator that changes the elements of a structure.

Some proposed solutions can easily be dismissed:

- Combining iterator and iteration function with inheritance [19,24,20] does not scale with respect to the number of traversal alternatives and iteration functions. Further problematic issues are described in [14].
- Equipping collections with a cursor that allows iteration of the whole collection (e.g., lists [23,29]) is problematic in the presence of multiple iterations. Even if interference is prevented by providing a cursor stack, which clients use to push and pop cursors, the resulting scheme is inelegant and error prone [12]. This suggests that the state of iteration should be kept outside the iterated collection.
- Schemes relying on co-routines or specializations thereof [18,25], are not easily applicable in languages without these mechanisms.

We are left with two fundamentally different approaches:

1. External iterators place the iteration logic outside of collections and provide clients with an interface to start, advance, and inquire the actual element of an iteration:
```
from books.start until books.exhausted -- initialize & test
loop
  io.putstring(books.item.title);      -- iteration action
  books.forth;                         -- advance iterator
end
```
2. Internal iterators are typically a part of the collection's interface. When given an iteration function they autonomously perform the traversal, thus, releasing the client to provide a control structure:
```
books.do(printTitle); -- printTitle prints argument's title
```

2.1 Internal Iteration

Internal iteration corresponds to functional mapping, i.e., the parameterization of iterators with iteration functions, and there are a number of arguments in favor of it: It adheres to the maxim "Write a Loop Once" [20], i.e., code the traversal control once inside the collection and let all clients rely on it. Hence, the duplication of virtually identical code in clients for stepping over a collection is avoided.

Traversal strategies typically rely on internal collection details for stepping over collections [25] and it is easier to use a recursive method for descending a collection than to memorize an access path externally [8].

A particular expensive incident due to external iteration was caused when the Mariner space-probe was lost due to an error in a loop [26]. In the FORTRAN code of Fig. 1 the dot should have been a comma. As it happened, just the value `1.3` was assigned to `DO3I` and no iteration took place at all. An internal iteration might not have been applicable, but at least the above example demonstrates that (fatal!) errors can be introduced in even the most simple loops.

```
DO 3 I = 1.3
   Code to be executed with I=1, 2, 3.
```

Fig. 1. Code example "Goodbye Mariner"

All the above observations argue in favor of making iteration an autonomous operation of the collection, but:

> *"External iterators are more flexible than internal iterators. It's easy to compare two collections for equality with an external iterator, for example, but it's practically impossible with internal iterators. Internal iterators are especially weak in a language like* C++ *that does not provide anonymous functions, closures, or continuations like* SMALLTALK *and*

> CLOS. *But on the other hand, internal iterators are easier to use, because they define the iteration logic for you. [8].”*
>
> – GOF Group

Definition *Closure*

A closure is a function equipped with a map from variable names to values. Hence, a closure is a function that has partially or fully received its arguments and awaits its evaluation.

A number of other authors agree that either builtin closure support is needed to use internal iterators [2,12,8] or that internal iterators are inflexible to the extent of disallowing the comparison of two collections [25,12,8,5,27].

Also, it seems that clients know best when an iteration can be stopped, e.g., when an element has been found, and internal iterators do not account for this[1].

2.2 External Iteration

The Iterator pattern [8] implements external iteration and resolves a number of issues: The combination of iterator and iterator function is trivial, since the client calls both in a dedicated loop (see code in Sect. 2). Pattern Iterator gives the client full control of iteration advancement and termination. It, furthermore, allows multiple traversal strategies and multiple iterators on the same collection. It is weak on requiring clients to duplicate control structures, time and again. It also may force collections to provide a (possibly protected) interface to allow their efficient scrutinization for traversal. Finally, an external iterator has to keep track of the iteration state – which may involve book keeping of paths into tree-like collections.

2.3 Comparison of Internal and External Iteration

Diametrical to an external iterator an internal iterator is strong on localizing the control to one loop, information hiding of collection internals, and competence of collection exploration. Unfortunately, it requires closures for combining iterator and iteration function, takes termination control out of the client's hands, and allows only one iteration at a time. Furthermore, multiple traversal strategies cause the collections interface to be bloated with iteration methods. Table 1 summarizes the comparison of external with internal iteration.

The last point in Table 1 refers to the fact that external iterators typically depend on the properties of the collections they traverse (e.g., trees need different iterators than linear collections). Therefore, it is common to observe a class hierarchy of iterators paralleling a collection hierarchy [8,24].

While internal iterators seem to be more faithful to software design matters than external iterators, they apparently let clients down in terms of straightforward (active) usage and flexibility.

[1] This is not true for SMALLTALK where blocks returning a value cause control to be passed back to the iteration client.

Table 1. External versus internal iteration

	Iterator kind	
Iteration framework aspect	*external*	*internal*
combination of iteration and action is trivial		⋆
record keeping of iteration state is straightforward		
no special access interface to collection is required		
no explicit loop needed for client iteration		
client may stop iteration early		†
iterating multiple collections in lock-step is easy		
traversal alternatives do not bloat collection's interface		
multiple iterations sharing one collection		
no parallel hierarchy of iterators and collections		

⋆ Support for closures required.
† By inelegantly passing a `continue?`-flag from function to iterator.

3 Stream Based Iteration

An elegant way to resolve the forces in Table 1 is to

– provide a way to flatten collections to a stream of data and
– iterate internally over (multiple) streams.

Definition *Stream*

A stream is a possibly infinite list. Streams allow access to the first element and to the tail of the stream only. Streams, hence, may produce elements only when they are actually required. Clients can not distinguish between streams that already know all their elements, produce them as blocks in a buffered fashion, or produce them element-wise by demand. Streams can, therefore, be regarded as lazy lists.

For instance, a tree is first tranformed into a linear stream and then this stream is consumed by an internal[2] iterator. An element of the intermediate stream might contain a collection element or a lazy exploration of further parts of the collection, i.e., an iteration continuation yielding further collection elements when inspected (see Fig. 2 and Sect. 4.3, *Traversal alternatives*). The stream consumer, hence, can decide which parts are to be explored next.

While all streams share the same interface, it depends on the collection type how many iteration continuations it produces as stream elements. The stream consumer selects a traversal alternative (e.g., pre-order, in-order, post-order) by evaluating the iteration continuations in the corresponding order.

Intriguingly, this approach manages to resolve all the forces discussed above (see Sect. 4.3), apart from two remaining crucial points:

[2] Note that "internal" now only characterizes the passive role of the client rather than the location of the iteration interface with regard to the collection.

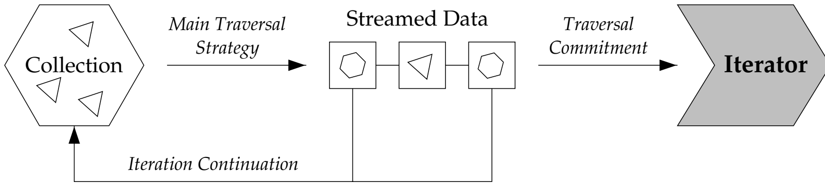


Fig. 2. Iteration topology with lazy streams

1. How to circumvent the need for closure support?
2. How to avoid the inflexibility of internal iterators?

3.1 Eliminating the need for Builtin Closures

The first problem can be immediately solved by emulating closures. In an object-oriented language it is very easy to use objects as function closures. Simply define a **Function** interface (also see Fig. 9) with an application method, e.g.,

```
deferred class Function[In, Out]   -- argument/result types
feature
  infix  "@" (a : In) : Out is deferred end; -- application
end
```

Subclasses then refine the generic type parameters and provide the implementation for various functions. For instance,

```
class Square
inherit Function[Integer,  Integer];
feature
  infix "@" (arg : Integer) : Integer is
  do
    Result := arg * arg;
  end;
end
```

If a function requires two or more arguments then the result of the first application is a further function object awaiting application to the rest of the arguments. Hence, this design nicely incorporates partial parameterization [13] (see the appendix or the full, clickable HTML source code [15] for details).

3.2 Making Internal Iteration More Flexible

The inflexibility of internal iterators, e.g., to compare two collections with simultaneous iterations, is easily fixed with a small but very effective idea. Indeed, it is practically impossible to consider a second iteration while an internal iteration focuses on its sole collection iteration. The problem, however, is easily resolved

by generalizing internal iteration from one to many collections (see Fig. 3). An internal iterator, consuming two number streams, for instance, takes a function with two parameters and applies it to the two foremost numbers. Then, both streams are advanced simultaneously and the next application will be on the two following numbers.

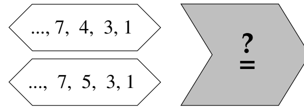


Fig. 3. Comparison of two collections in lock-step

To generalize from two to any number of input streams the processing function (e.g., equality) must process a list of arguments. Then, the arity of the function is always one (one list) but the real number of arguments is determined by the list length. Restricting all arguments to conform to the list type makes this approach statically type safe with no need for dynamic type checks (reverse assignment attempts or downcasts). All that is needed is a (genericity) mechanism that allows restricting the element type of a structure.

Transposing the list[3] of input streams yields a structure ready to be processed row by row (see Fig. 4).

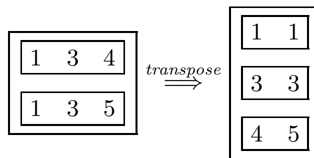


Fig. 4. Transposing lists

In case the input streams have different length then the shortest one determines the end of iteration. See the appendix for the Eiffel implementation of transpose.

Note that transposition, i.e., the processing of multiple input streams in lock-step is just one possible iteration scheme. A different operator might consume the input streams with individual speeds, for instance, to implement merge sort.

The next step towards a definition of a multi-collection iterator is to define a function that will process the rows of the transposition result. As it applies (maps) a function to each row in the transposed structure we call it transmap.

[3] We say "list" when referring to the ordered collection of input streams for clarity of presentation only. Technically, this list will be a stream too.

Definition *Fold*

A fold function processes a list with a function. Folding a list can yield any result type including a transformed version of the input list. Typically, however, a fold reduces a list to a single result by applying the function to the first element and the result of folding the rest of the list. In order to define the value of folding an empty list, an initial value is passed as an argument to fold. A transformed result list may be obtained by using a function that transforms input elements individually and builds up a list from the results.

The function argument to transmap will typically reduce (fold) one row into a single result (see the appendix for an implementation of fold). For instance, a product calculation is possible by reducing the list with multiplication. It is now straightforward to calculate the element-wise products of two integer lists (see Fig. 5).

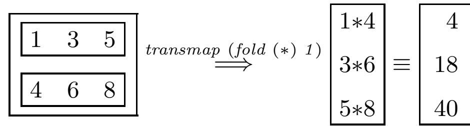


Fig. 5. Computing a list of products

Until now, function transmap allows varying the function to combine the elements of multiple collections. However, we also need to iterate over the results. For instance, it is only a matter of summing up the values in the above result list to obtain the inner product of the two argument lists. Or, returning to our original example, when comparing two collections the results of comparing corresponding elements must be reduced to a single result (see Fig. 6).

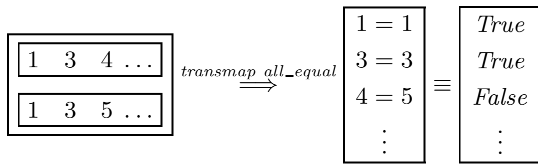


Fig. 6. Transmapping equality

Obviously, we need to reduce the result list with the logical "*and*" operator to obtain a single equality result. Likewise, the result list of Fig. 5 requires reduction with "+" to obtain the final inner product. Therefore, as the final step to define the multi-collection iterator we reduce the results of processing

the rows to a single result. Here is how transfold[4] can be expressed with Eiffel assuming `streams` to be the list of input streams (see the appendix for its full implementation):

```
Result := fold @ foldFunc @ init @
          ((map @ mapFunc) @ transpose(streams));
```

For instance, the application of transfold to "+" and (fold $*$ 1) on the argument streams of Fig. 5 will yield the result 62. Expressed with Eiffel code:

```
Result := transfold @ plus @ 0 @ (fold @ times @ 1) @ streams;
```

Functions `foldFunc`, `mapFunc`, and value `init` correspond to functions f, g, and value a of Table 2 respectively. It lists all transfold parameters with their type and meaning.

Table 2. Transfold's arguments

Parameter	*type*	*purpose*
g	$[a] \rightarrow b$	the function that is applied (mapped) to each row of the transposed argument, transforming a row to an element of the intermediate result of type b.
a	c	the initial element for producing the final result, used as the induction base for an empty list.
f	$b \rightarrow c \rightarrow c$	function that finally reduces the intermediate result of element type b to a result of type c, using the initial element.
	$[[a]] \rightarrow c$	resulting type of transfold after all arguments but the last are supplied. Transforms a matrix (list of streams) with element type a into a result of type c.

Given a function *all_equal* that checks whether all elements in its argument stream are equal, the application of transfold to *and* and *all_equal* on the argument streams of Fig. 6 will yield *False*.

Note that, for example, in the context of comparing collections, lazy transposition and reduction functions allow stopping the exploration of the (possibly infinite) argument streams when a non-equal argument pair has been found.

To illustrate the possible type changes from argument to result let us calculate the sum of all row products from a matrix[5] (see Fig. 7), using (very academically) three different number types. We use

```
innerProd := transfold @ plus @ 0.00 @ (fold @ times @ 1.0);
```

[4] Transpose and fold.
[5] A matrix shall be represented as a list of row streams.

where the input matrix contains integer elements, 1.0 denotes a real, and 0.00 denotes a double. Hence we establish the mapping $[a \mapsto \mathit{integer},\ b \mapsto \mathit{real},\ c \mapsto \mathit{double}\ ]$. See Fig. 7 for the calculation process.

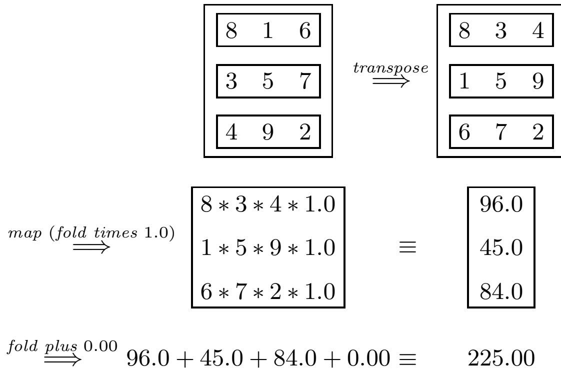


Fig. 7. Transfolding the inner product of a matrix

Folding does not have to imply reduction, though. Using functions *reverse* and *add_back* that establish the mapping $[a \mapsto \mathit{integer},\ b \mapsto [\mathit{integer}],\ c \mapsto [[\mathit{integer}]]\ ]$, we may transpose a matrix along its minor axis (see Fig. 8).

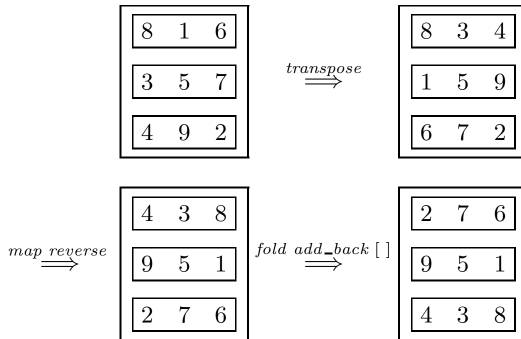


Fig. 8. Minor axis matrix transposition

We used a mixture of illustrations, functions, and sketches of EIFFEL code to demonstrate transfold and its associated functions as clearly and concisely as possible. With the help of two functional patterns (Function Object and Lazy Object [16]), it is very easy to fully implement the solution in an object-oriented language (see also the clickable HTML code [15]).

4 Stream Based Iteration Framework

The following sections describe the proposed iteration framework with a class diagram, a list of participants with their responsibilities, and the sequence of events for an iteration. Section 4.3 concludes the description with a list of framework properties.

4.1 Framework Participants

Figure 9 shows all participants in the iteration framework and their relationships. Dashed boxes at the top right hand corner of a box indicate generic classes.

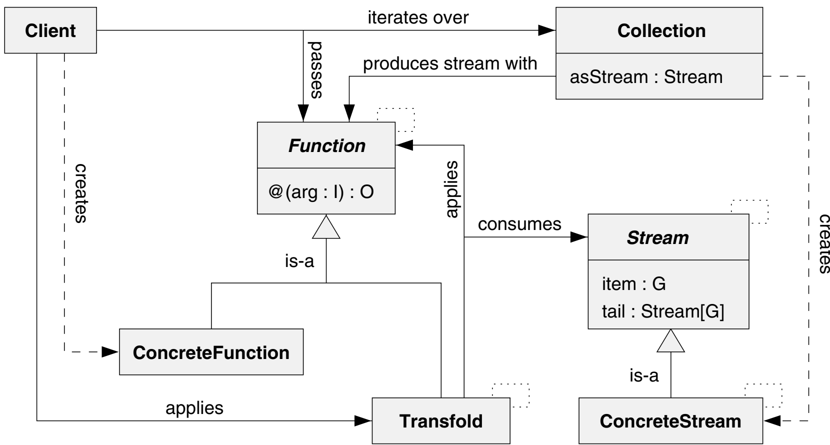


Fig. 9. Structure diagram

- **Client**
  - requests a **Collection** to provide a **Stream** of itself.
  - passes two **Function**s and a value as parameters to **Transfold**.
- **Collection**
  - provides a **ConcreteStream**, containing a flattened version of itself.
  - uses a lazy **Function** to produce a **ConcreteStream**.
- **Function**
  - provides an application interface for all functions including the stream building function, the **Transfold** parameters, and **Transfold**.
- **Stream**
  - provides an interface to access any concrete streams.
  - implements a lazy, infinite list semantics.
- **Transfold**
  - takes two **Function**s and a value as processing parameters.
  - transforms its input (a **Stream** of **Stream**s) to an arbitrary result type.

4.2 Sequence of Events

- A client requests a collection to flatten itself to a stream.
- The collection's asStream method and a lazy function mutually call each other to explore the collection lazily, while producing a stream.
- The client uses or creates two function objects, which it passes – along with an initial value – to a transfold object.
- The transfold object lazily accesses the collection stream by using the stream interface, e.g., operations "item" and "tail". While the stream is accessed, the argument functions to transfold are applied accordingly.

The purpose of the above details is to present explicitely the mechanics of the iteration framework. Real clients, however, should not have the burden of asking the collection for a stream and then of feeding it into an iterator. It is more reasonable to use a method in an abstract collection interface that takes all transfold parameters and does the stream creation and feeding behind the scenes.

4.3 Framework Properties

► *Abstraction.* Accessing the elements of a collection does not expose its internal representation.
► *Locality.* A particular operation can be performed by just passing function objects, without requiring inheritance or client control structures.
Since there is only one iteration loop, used by each client, loop-related errors are much easier to avoid and to discover. More time can be spent on the validation of a single loop[6] and any errors are removed for all clients.
► *Multiple Traversals.* When a collection is to be iterated by multiple clients in alternation it is possible to share the (read-only) collection stream for independent consumption by multiple clients. Hence, any exploration effort by the collection is beneficial to all consumers. Once explored, a subpart does not need to be traversed again due to the call-by-need semantics of streams. Whenever a read iterator needs to see the results of a write iterator we propose to use a chain of iterators where collections are not mutated but intermediate results are produced (see Fig. 10). Instead of destructively changing the contents of one collection – which can cause considerable trouble in the presence of sharing – an intermediate collection which contains the new data is produced to be subsequently consumed by the read iterator.
► *Connectivity.* Stream producing transfolds allow cascaded transformations[7] and converting one collection into another one, possibly with intermediate processing, using collection constructors with stream arguments to build collections (see Fig. 10).
Intermediate results never exist in their entirety due to lazy evaluation. The demand driven characteristics of lazy evaluation only ever creates as much of

[6] By referring to "loop" we also include the stream generation processes.
[7] Note that one transfold can perfectly just consume a single stream.

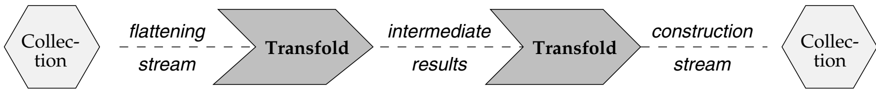


Fig. 10. Transformation chain

the intermediate results as necessary to pass the desired information through. An intuitive picture of the process is to imagine the source collection to be a wool bale being unwound, while just the thread is passed through the transfolds, to wind the result wool bale. Maintaining the internal shape of a collection during a transformation as shown above is easier with destructive updates. However, the collection constructor of the destination collection can use the order of the arriving elements and its own structure invariants to create a collection that is behaviorally equivalent to the original.

► *Termination control.* Both transfold and the client are in control of iteration advancement and termination. Through the use of lazy stream processing functions, the collection exploration is completely demand driven. When a stream processing function does not evaluate its second argument – e.g., an *And* does not need to examine the second argument, if the first is already *False* – the whole transfold process stops. This scheme is far more elegant than letting an iteration function return a `continue?`-flag, as designed in the internal version of the Iterator-pattern [8].

► *Traversal Alternatives.* Collections produce a stream containing elements and iteration continuations which are again represented by streams (see Fig. 2). For instance, graphs can produce a stream representing a forest (a stream of tree streams). Stream consumers decide in which order they consume elements and explore continuations. Hence, it is easy to support a variety of traversal strategies outside of collections (see Fig. 11).

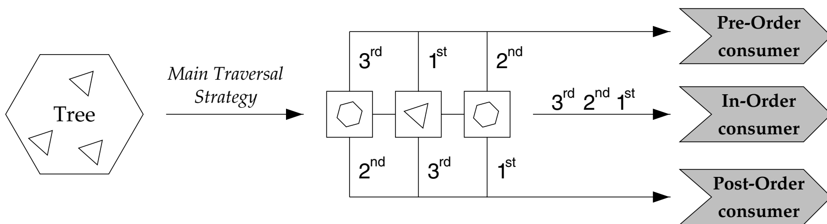


Fig. 11. Deriving traversal alternatives

One can even dynamically dispatch on traversal strategies. Through lazy exploration of the collection the stream processing functions (passed as arguments to a transfold object) are in command of the exploration order.

Not all traversal alternatives, however, can be derived from just one main traversal strategy while retaining lazy exploration. Assume, for instance, a linear structure providing the elements one by one. It is then not possible to derive a backward iteration from this main traversal strategy without fully exploring the structure to arrive at the last element. In such cases it is possible to provide alternative asStream methods in the collection interface.

► *Separation.* The best of both external and internal iteration are combined by separating the *exploration* of a collection and the subsequent *consumption* of the exploration result.

+ Since iteration (consumption) is defined outside collections, their interfaces can be kept small. The only trace of an iteration is an asStream method, which is of general interest anyway (see bullet "*Streamable Collections*" below).
+ The responsibility to explore a collection is assigned to the most competent instance, the collection itself. The collection may use all its internal knowledge and recursive calls – thereby memorizing an exploration stack – to perform its exploration. Contrast this with a much more complicated scheme implementing recursion with an explicit stack and counting (observing) node removals [38].

  It is possible to separate consumption and exploration without having the overhead of a full exploration, because the intermediate stream is lazily produced. For a general account on pattern Lazy Object see [16].
+ Streams work as a *lingua franca*[8] between collections and iterators. Both stream consumption (iteration) and generation are easy to vary. Iteration schemes, such as transfold, must be defined only once, for all streamable collections.

  Special traversal orders may depend on stream organization but not on collections, which is a useful indirection to decrease coupling.

► *Streamable Collections.* The asStream method of data collections can also be used for many other purposes, such as a persistence or net-transfer protocol mechanisms.

Collections may be transfered into each other by means of an intermediate stream. For instance, streaming a **Bag** to a **Set** is an elegant way to remove duplicates. No special mechanisms, e.g., the Serializer pattern [31], will be needed anymore to support persistence in this manner.

This also implies that there is a uniform way to construct collections, for example, from constants. Any collection type that allows manifest constants in the syntax of a language (e.g., arrays), could be used to be transformed to the desired collection type.

► *Versatility.* Folding may implement a wealth of operations, for instance for lists: sum, product, average, max, min, map, filter, reduce, reverse, append, exists, all, variance, horner, etc. According to [37], 60% of the code in the Fortran Scientific Subroutine Package fits neatly into the maps, filters, and accumulations (i.e., transfold) paradigm.

[8] Agreed language of communication.

► *High-level Mindset.* A capable, high-level operation like transfold enables approaching problems with a much more powerful decomposition strategy compared with a procedural paradigm, restricted to e.g., array indexing. Timothy Budd tells an anecdote of a FORTRAN programmer who, predetermined to think in terms of loops and array access, designs a three-level nested loop to find a pattern repetition in a DNA sequence. His algorithm's complexity turns out to be $O(M * N^2)$, where M = pattern length and N = sequence length. An APL programmer, thinking in high-level operations like vector to matrix conversion, sorting, and matrix reduction (all akin to and expressible with transfold), arrives at a solution with complexity $O(M * N \lg N)$ [5]. As the anecdote suggests, high-level operations allow problems to be approached from a different, valuable perspective.

► *Choice of Style.* In cases where no predefined iteration scheme, like transfold, seems appropriate, it is possible to consume a collection stream with an external iterator, i.e., to write a control loop which consumes the stream. The essence of the proposed iteration framework lies in stream producing collections, function objects, and lazyness. The transfold operator is just one of many possible.

► *Fixed Forms.* A view emerged from the so-called *squiggolist* school [21] aims at a formalism that does not allow users the free definition of recursive functions but supplies a limited set of functional forms, such as fold, that are well controlled and amenable to program transformations. Relying on a set of functional forms (e.g., transfold) is advantageous in many ways:

- Algorithms using the forms have a concise, readily understandable structure as well as a determined complexity in time and space. Any programmer familiar with the functional forms will understand the algorithm by just looking at the essential parameterized parts.
- Well-known laws for the functional forms may be exploited to transform programs. For instance, instead of multiplying all elements of a collection by 2 and then summing them up, they can be summed up first applying the multiplication by 2 to the sum, hence only once. The general law for this transformation is a free "fold-fusion" theorem that is derivable from the signature of fold [36]. More such laws can be found in [21].
- Algorithms written as a combination of parameterized combinators can be easily varied. For instance a *tournament-sort* combinator taking two reduction strategies as parameters can express *InsertSort* [3], *TreeSort* [7], and *ParallelTournamentSort* [33], just by using different combinations of the two reductions operators [10]. *ParallelTournamentSort*, in fact, has very desirable properties, which demonstrates that using standard combinators does not necessarily imply inefficient solutions and may, on the contrary, help to discover better solutions.

► *Contra-Indication.* This iteration framework should not be used in case of very tight memory and time constraints, where the overhead of an intermediate stream and emulation of lazy evaluation is not tolerable. Management of stream elements consumes time. Stream suspensions and lazy function closures represent a memory overhead. In most cases, however, system performance should not be a problem.

5 Related Work

Although C^{++} usually promotes external iteration, there is an example of an internal iterator (`foreach`) interface in the Borland C^{++} libraries [4]. Since it builds on passing function pointers, it must use an extra, unsafe `void` type for passing parameters.

The Standard Template Library (STL) [34], provides function objects, a variety of (forward, backward, etc.) iterator types including stream iterators. However, iterators are of the external type and collections directly return iterators instead of streams with continuations.

The generic collection library for JAVA (JGL) [28] has its roots in STL. It provides an alternative to the JAVA collections API [35]. In a readers poll [30] the usage of function objects within the JGL was found to be powerful and flexible while the collections API was conceived to be more lightweight and simpler to use.

The function fold originates from functional programming [3], but is also used in SCHEME [1] and available in the SMALLTALK library [17]. SMALLTALK users do not use it frequently [32], probably because they are unfamiliar with the nature of folding and its peculiar name (inject: into:). The SMALLTALK collection library even contains a with: do: method, allowing iteration over two collections in parallel, which represents a special case of transfolding. SMALLTALK also uses streams to implement concatenation of collections efficiently. The caching effect of streams eliminates the need to repeatedly generate the prefix of sequenced concatenations like `coll1 + coll2 + coll3 + coll4`.

APL [9] is well-known for its high-level operations on vectors, matrices, and structures of even higher dimension. Three of its four primitive extension operators[9], reduction (`f/`A), scan (`f\`A), and innerProduct (A`f.g`B), can directly be expressed with transfold. The fourth, outerProduct (A`∘.f`B), is expressible with a combination of transfold and map. Function map is just a (trans-)fold with argument functions that do not reduce the input stream but only apply a function to it.

An interesting competitive iteration approach is set out by the SATHER programming language. Language support for iterators – in the form of a restricted kind of coroutines – allows defining collection exploration within collections while still enabling flexible, external iteration style, consumption [25]. The open questions are which style (passing functions or coroutine-loops) is more expressive and understandable, and whether possible code optimizations by the SATHER compiler justify the requirement for an additional language construct. In effect, the ability of lazy evaluation to defer calculations and resume control to them whenever necessary is very similar to coroutines. With coroutines, however, the emphasis is on explicitely scheduling control whereas lazy evaluation causes a more declarative, demand driven style.

Kofler investigated how to make iteration over collections which are changed during iteration a safe and unambiguous operation [12]. Using transfold's scheme

[9] These extend an operation to a collection.

a so-called *iterator adjustment* scheme [12], is particular well implementable, since the collection controls its own exploration and, thus, may adjust an exploration process according to element removal or insertion. Consequently, no registering of active iterators [12] is necessary. In cases where updates should have no effect on the iteration process, it is possible to simply iterate on a copy of the collection.

6 Conclusions

Neither external iterators nor internal iterators operating on a single collection only provide a satisfactory general iteration framework. Functional techniques (made available by the functional patterns Function Object and Lazy Object [16] for closure and lazy semantics emulation respectively), however, make internal iteration feasible for standard object-oriented languages. Function objects enable behavior parameterization, whereas lazy evaluation makes the separation of collection exploration and data consumption feasible. Combined with the idea of simultaneously processing multiple collections during one internal iteration, the result called "transfold" provides the safety and economic efficiency of internal iteration while maintaining the flexibility and control of external iteration. Transfold, however, is just one of many possible operations. Other iteration schemes may process multiple collections without imposing a lock-step processing fashion. The presented stream based iteration framework even extends into an approach for creating data from manifest constants, transforming collections into each other, and data persistence.

Some users, who are unfamiliar with the map/fold style of functional programming, may find the general nature of transfold intimidating and awkward to use. Nevertheless, transfold can be used to build specialized iteration operations that are easier to use. With just one iteration primitive (folding) many operations can be expressed (see Sect. 4.3, *Versatility*). This is in contrast to, e.g., EIFFEL's plethora of iteration features (do_all, do_if, do_while, do_until, etc.) in its iteration classes [24]. Although EIFFEL supports only one language loop construct, its library approach to iteration apparently does not allow such a minimalistic solution. While not every computable function is expressible with folding [11], it is sufficiently general to rely on it as the basic iteration principle. Fortunately, stream based iteration easily allows the introduction of new iteration schemes whenever necessary.

In general, internal iteration is better suited for parallel execution since, unlike a custom external iteration, the iteration process is guaranteed to be encapsulated [2]. In particular, functional forms such as *map* or *parallel-reduce* allow parallel evaluation. In fact, there is a tradition in the area of parallel computations to employ fixed evaluation forms called algorithmic skeletons [6].

Several improvements to current object-oriented languages would aid the construction of the transfold framework. First, instead of manually defining function object classes that just forward arguments to implement partial parameterization, the language should do this automatically. The recently proposed mech-

anism for delayed calls in Eiffel supports partial parameterization to some extent. While it is possible to leave parameter positions open during the creation of a delayed call, it does not seem to be possible to obtain another delayed call from a delayed call by passing a subset of the open parameters only.

Second, using Eiffel it is not possible to achieve transparent lazy semantics for basic types, e.g., **Integer**, because access to them is implicit, i.e., there is no access method which could defer the calculation. Although, it is possible to use wrapper classes for basic result types, which demand using a value or item method to access values (see Eiffel's **NUMERIC_REF** classes or Java's approach to treat basic types as objects), this solution obviously makes lazy semantics visible to clients. Language support for lazy semantics could fix this and in addition eliminate the need for an extra lazy stream generating function object. Lazy collection methods could achieve lazy exploration on their own, thus implementing collection exploration at the best possible location, inside collections only.

Third, although standard parametric polymorphism is sufficient to support transfold with regard to its flexible types, type system support could be much better. For instance, the Eiffel implementation forces us to include the intermediate (row result) type in the list of generic parameters for **Transfold**. This is unavoidable because it is the only way to ensure that the result type of the row processing function (g) and the input type of the row results reduction function (f) match. Eiffel does not provide access to the actual value of the generic type parameter of one function in order to match it with the argument type of another. Hence, the introduction of dependent types would further support the definition of transfold in the presence of static typing. Interestingly, the way C^{++} treats generic type parameters implicitly, does not cause the above complication.

While the usefulness of function objects in object-oriented languages is generally acknowledged by now, the above observations also strengthen the case for integrating a seemingly exotic functional concept, i.e., lazy evaluation, into object-oriented languages. The benefits of using a complete functional pattern system and its implications on language design are described in [16].

Acknowledgments

The author would like to thank Thilo Kielmann for his feedback, John Hopkinson for proof reading, and the anonymous reviewers for their extremely knowledgeable and helpful comments.

References

1. Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA, London, 6th edition, 1987.
2. Henry G. Baker. Iterators: Signs of weakness in object-oriented languages. *ACM OOPS Messenger*, 4(3):18–25, July 1993.

3. Richard Bird and Philip Wadler. *Introduction to Functional Programming.* C.A.R. Hoare Series. Prentice Hall International, 1988.
4. Borland. *Borland C/C^{++} 4.0 Reference Manual.* Borland, Inc., 1994.
5. Timothy Budd. *Multiparadigm Programming in Leda.* Addison-Wesley, 1995.
6. John Darlington, Yi-ke Guo, Hing Wing To, and Jin Yang. Parallel skeletons for structured composition. In *PPoPP '95*, pages 19–28, St. Barbara, CA, July 1995.
7. R. W. Floyd. Treesort (algorithm 113). *CACM*, December 1964.
8. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture.* Addison-Wesley, 1994.
9. Edward Harms and Michael P. Zabinski. *Introduction to APL and computer programming.* Wiley, 1977.
10. Aaron Kershenbaum, David Musser, and Alexander Stepanov. Higher-order imperative programming. Technical Report 88–10, Rensselaer Polytechnic Institute Computer Science Department, April 1988.
11. Richard B. Kieburtz and Jeffrey Lewis. Programming with algebras. In *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 267–307. Springer, 1995.
12. Thomas Kofler. Robust iterators for ET++. *Structured Programming*, 14(2):62–85, 1993.
13. T. Kühne. The function object pattern. *C++ Report*, 9(9):32–42, October 1997.
14. Thomas Kühne. Parameterization versus inheritance. In Christine Mingins and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 15*, pages 235–245, Prentice Hall International, London, 1995.
15. Thomas Kühne. Transfold Eiffel code implementation: Classes & HTML, `http://www.soc.staffs.ac.uk/~cmttk/transfold.html`, November 1998.
16. Thomas Kühne. *A Functional Pattern System for Object-Oriented Design.* ISBN 3-86064-770-9, Kovač Verlag, Hamburg, 1999.
17. Wilf LaLonde. *Discovering Smalltalk.* Benjamin / Cummings Publishing, 1994.
18. Barbara Liskov and John Guttag. *Abstraction and Specification in Programm Development.* MIT Press, 1986.
19. O. L. Madsen, K. Nygaard, and B. Möller-Pedersen. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley and ACM Press, 1993.
20. R. Martin. Discovering patterns in existing applications. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 365–393. Addison-Wesley, 1994.
21. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Coomputer Architecture, Cambridge, Massachusetts*, LNCS 523, pages 124–144. Springer Verlag, August 1991.
22. Gisela Menger, James Leslie Keedy, Mark Evered, and Axel Schmolitzky. Collection types and implementations in object-oriented software libraries. In *The* 26th *TOOLS conference USA '98*, 1998.
23. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, 1988.
24. Bertrand Meyer. *Reusable Software.* Prentice Hall, 1994.
25. Stephan Murer, Stephen Omohundro, and Clemens Szypersky. Sather Iters: Object-oriented iteration abstraction. Technical Report TR-93-045, ICSI, Berkeley, August 1993.
26. P.G. Neumann. Risks to the public in computer systems. *ACM Software Engineering Notes 11*, pages 3–28, 1986.

27. Peter Norvig. Design patterns in dynamic programming. Presentation at Object World '96, May 1996.
28. ObjectSpace. JGL – the generic collection library for java v3.0, `http://www.objectspace.com/products/jgl/`, 1997.
29. Stephen Omohundro and Chu-Cheow Lim. The sather language and libraries. Technical Report TR-92-017, ICSI, Berkeley, March 1992.
30. Reader poll. Schould Sun scrap its collections API in favor of the more robust JGL?, `http://www.javaworld.com`. *Java World*, August 1998.
31. D. Riehle, W. Siberski, D. Bäumer, D. Megert, and H. Züllighoven. Serializer. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, Reading, Massachusetts, 1997. Addison-Wesley.
32. David N. Smith. *Dave's Smalltalk FAQ*. dnsmith@watson.ibm.com, July 1995.
33. A. Stepanov and A. Kershenbaum. Using tournament trees to sort. Technical Report 86–13, Polytechnic University, 1986.
34. A. Stepanov and M. Lee. The standard template library. ISO Programming Language C++ Project. Doc. No. X3J16/94-0095, WG21/NO482, May 1994.
35. SUN. Java collections API, `http://java.sun.com/products/jdk/1.2/docs/guide/collections/overview.html`, 1997.
36. Philip Wadler. Theorems for free! In 4th *International Symposium on Functional Programming Languages and Computer Architecture*, London, September 1989.
37. Richard C. Waters. A method for analyzing loop programs. *IEEE Transactions on Software Engineering*, 5(3):237–247, January 1979.
38. M. A. Weiss. *Algorithms, Data Structures and Problem Solving with C++*. Addison Wesley, 1996.

Appendix

The following code sketches should provide enough information concerning the implementation of transfold with Eiffel. A complete set of all required classes and a working test class, however, can be obtained from the author's homepages [15].

Section 3.2 discussed the calculation of the inner product of a matrix (see Fig. 7). We can code the corresponding inner product operation in Eiffel as:

```
local
  ip : Function [Stream [Stream [Integer]], Double]
...
  ip:=transfold @ plus @ 0.00 @ (fold @ times @ 1.0);
```

We use the same type progression (integer, real, double) for the intermediate results as in Sect. 7. Therefore, function objects `times` and `plus` must promote from integer to real and real to double respectively.

The `ip` function can be now be applied to a matrix. So, constructing integer streams with `addItem` and building the list of input streams with `addStream` –

```
vec1, vec2, vec3 : Stream [Integer];
vecs             : Stream [Stream [Integer]];
...
```

```
vec1:=addItem @ 8 @ (addItem @ 1 @ fromConst (6));
vec2:=addItem @ 3 @ (addItem @ 5 @ fromConst (7));
vec3:=addItem @ 4 @ (addItem @ 9 @ fromConst (2));
vecs:=addStream @ vec1 @
      (addStream @ vec2 @ (addStream @ vec3 @ void));
```

– the following function applications yield the results given in Fig. 12.

```
io.putdouble (ip @ vecs);
io.putdouble (ip @ vecs.tail);
```

$$225 = \texttt{ip} @ \texttt{vecs} \left\{ \begin{array}{l} 8\ 1\ 6 \\ 3\ 5\ 7 \\ 4\ 9\ 2 \end{array} \right. \left. \begin{array}{l} \\ \\ \end{array} \right\} \texttt{ip} @ \texttt{vecs.tail} = 71$$

Fig. 12. Inner product application

The row processing function **Fold**, used in the implementation of `ip` and **Transfold** is implemented (omitting the preceding argument collection classes) as:

```
class Fold2[B, C]                      -- B = stream element type
inherit Function[Stream[B], C];        -- C = fold result type
creation make
feature
  func : Function[B, Function[C,C]]; -- function argument
  init : C;                           -- initial value argument

  make(i : like init; f : like func) is -- called by Fold1,
  do                        -- whose application method "@"
    init:=i;                -- receives "f". Fold1's make is
    func:=f;                -- is called by Fold's application
  end;                      -- method, which received "i".


  infix "@" (stream : Stream[B]) : C is
  local fold : expanded Fold[B, C];
  do
    if stream=void then            -- end of input stream?
      Result:=init;                -- yes, return initial value.
    else
      Result:=func @ stream.item @   -- no, apply recursively.
              (fold @ func @ init @ stream.tail);
    end;
  end;
end
```

With the help of **Fold** and **Map**, implementing the body of **Transfold** is straightforward:

```
class   TransFold3[A, B, C]             -- A = element type
inherit Function[Stream[Stream[A]], C] -- B = reduction result
        StreamUtility[A]               -- C = transfold result
creation make
feature
  foldFunc : Function[B, Function[C, C]]; -- result processing
  init     : C;                           -- initial value
  mapFunc  : Function[Stream[A], B];      -- row processing

  make(f : like foldFunc; i : like init; m : like mapFunc) is
  do
    foldFunc:=f;
    init:=i;
    mapFunc:=m;
  end;

  infix "@" (streams : Stream[Stream[A]]) : C is
  local
    map  : expanded Map[Stream[A], B];
    fold : expanded Fold[B, C];
  do
    Result:=fold @ foldFunc @ init @
            ((map @ mapFunc) @ transpose(streams));
  end; ...
```

The use of `expanded` is a language idiom to save the otherwise necessary explicit attachment of an object to `map` and `fold`.

The `transpose` method of class **TransFold3**, nicely demonstrates a high-level, functional, internal iteration style, which is possible using streams and functional forms such as map and fold.

```
transpose(rows : Stream[Stream[A]]) : like rows is ...
do
  newRow:=mapToHeads @ head @ rows;   -- collect all row heads
  tails:=mapToTails @ tail @ rows;    -- collect all row tails

  if (fold @ oneEmpty @ False @ tails) then -- row exhausted?
    tails:=void;                    -- yes, end of result rows
  else
    tails:=transpose(tails);        -- no, transpose the rest
  end;

  Result:=addStream @ newRow @ (tails); -- build result
end
```

Type-Safe Delegation for Run-Time Component Adaptation

Günter Kniesel

Universität Bonn
Institut für Informatik III
Römerstr. 164, D-53117 Bonn, Germany
gk@cs.uni-bonn.de
http://javalab.cs.uni-bonn.de/research/darwin/

Abstract. The aim of component technology is the replacement of large monolithic applications with sets of smaller software components, whose particular functionality and interoperation can be adapted to users' needs. However, the adaptation mechanisms of component software are still limited. Most proposals concentrate on adaptations that can be achieved either at compile time or at link time. Current support for dynamic component adaptation, i.e. unanticipated, incremental modifications of a component system at run-time, is not sufficient.
This paper proposes object-based inheritance (also known as delegation) as a complement to purely forwarding-based object composition. It presents a type-safe integration of delegation into a class-based object model and shows how it overcomes the problems faced by forwarding-based component interaction, how it supports independent extensibility of components and unanticipated, dynamic component adaptation.

1 Introduction

Component-oriented programming aims at the replacement of monolithic applications with sets of smaller software components. Its ... motivation is two-fold. For software engineers, assembly of applications from existing components should increase reuse, thus allowing them to concentrate on value-added tasks and to produce high-quality software within a shorter time. For users, component oriented programming promises tailor-made functionality from the adaptation of ready-made components.

Component adaptation includes the customization of individual components as well as the customization of a whole component-based application by replacing some components with others that are better suited for a specific task.

Known approaches to component adaptation can be classified according to

- their need for preexisting "hooks" in the application as either suitable for *anticipated* or *unanticipated* changes,
- the time when adaptation is performed as either *static*, *load-time* or *run-time* (dynamic)
- their ability to adapt whole component types or individual component instances as either *global* or *selective*. Selective approaches can be further classified as either *replacing* or *preserving* , depending on whether they replace an existing component instance by its adapted version or let both be used simultaneously.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 351–366, 1999.
© Springer-Verlag Berlin Heidelberg 1999

– the applied techniques as either *code-modification-based*, *proxy-based* or *meta-level-based*.

With respect to the above classification this paper presents an approach to unanticipated, dynamic[1], selective, proxy-based, object preserving component adaptation. It leverages proxy-based techniques by introducing typed delegation as a new basic interaction primitive beyond simple message sending.

The need for *unanticipated* dynamic adaptation has been repeatedly pointed out in literature (e.g. in [MS97]) and its practical relevance has been impressively demonstrated by the recent transition from national currencies to the Euro. As opposed to the "year two thousand problem" this change of requirements could not be forseen as the currently deployed software systems had been designed. So the software departments of European banks, insurances and other companies providing 24-hours services to their customers were suddenly faced with the problem to perform unanticipated changes to their systems, without discontinuing operation. Not all of them succeeded, e.g. some banks needed more than three days to get their automatic teller machines operational with the new software. Others succeeded thanks to redundand hardware.

Detailing the ideas sketched in [Kni98b] this paper explores the feasibility of more timely and inexpensive, purely software-based solutions.

When active components can neither be directly modified nor unloaded from a running system, we are faced with the problem to change their behavior solely by adding more components. This paper explores how far delegation can help to solve this apparent contradiction and how component adaptation can be performed without deleting the "old" version of a component. Delegation enables the joint use of different versions of a component and the easy modeling of components that present different interfaces to different clients.

An overview of the state of the art is given in section 2 using the Euro example to illustrate the benefits and limitations of different component adaptation techniques. This section motivates the need for typed delegation as a new component interaction primitive. Section 3 introduces typed delegation in the framework of the DARWIN object model and the LAVA programming language. The use of DARWIN/LAVA for dynamic component adaptation is described in section 4.

2 State of the Art

Currently, the problem of component adaptation is mainly tackled from a programmer's point of view with proposals that aim at easing reuse of existing components in the development of new applications. Most proposals in this category are based on code modification ([Bos98,Har97,ACLN98,Kel97,Hol98]). The paper [KAZ98] proposes programing guidelines for the construction of components that should anticipate and ease the construction of wrappers.

A second line of work concentrates on the problem of adapting running appliations. Proposals in this category are based on wrappers (e.g. [PBJ98]) or metalevel architectures (e.g. [MS97]).

[1] The proposed approach can also be applied statically or at load-time. However, its dynamic use is the most beneficial one with respect to component adaptation

Before proceeding to the introduction of typed delegation it is instructive to review the aforementioned techniques (code modification, adapters, and metaobject protocols) in the light of their suitability for dynamic component adaptation. We do not go into the details of approaches that impose special design guidelines ([KAZ98,MS97]), making them unsuitable for unanticipated adaptation.

Code Modification. Code modification uses two inputs, a class to be modified and a specification of the modification. The result is a modified version of the initial code which is used instead of the old version. Examples in this category are Jan Bosch's superimposition technique ([Bos98]), the class composition proposal of Harrison and Ossher ([Har97]) and the recent work of Keller and Hölzle on binary component adaptation ([Kel97,Hol98]). Superimposition is a language construct within a special object model, LayOM. LayOM programs are translated to C++ and Java, making the modified source code available for further use. Class composition as proposed by Harrison and Ossher is also applied at "compile-time" and requires source-code availability. Binary component adaptation can be seen as a more general form of class composition, which can adapt *binary* components when they are *loaded*. The basic technique behind binary component adaptation can be generalized to other approaches. In particular, it can be used to produce a more dynamic (load-time) version of subject oriented programming ([HO93,HOT98]), with similar component adaptation properties.

Code modification is applicable at compile-time or load-time but has only limited applicability at run-time. Its essence is the replacement of an existing class by a new version of that class. This is very difficult in a running system, where instances of the class to be replaced already exist and are being used. This is the well-known yet still generally unsolved problem of schema evolution in database systems, transferred to the run-time environment of an object-oriented language. Even if the schema evolution problem could be takled in some ad hoc way, dynamic class replacement would be too coarse grained for many applications, globally affecting all existing instances of a component, even if selective adaptation was intended.

Component instance replacement. In the extreme case when each component has only one instance, neither the schema evolution problem, nor the granularity problem would arise. Thus a component and its sole instance could be replaced by a new version. However, component instance replacement has its own shortcomings. First of all, the component to be replaced might not be prepared to hand all its relevant private data over to the new version, if adaptation was not anticipated. Secondly, the application might require joint use of the old and the new component. For instance, for the Euro transition it is required that prices in the national currency be printed in addition to prices in Euro during the first two years.

Wrappers. When active components can neither be directly modified nor unloaded from a running system, we are faced with the problem to change their behavior solely by adding more components. This leads us to the use of wrappers ([GHJV95]). A wrapper is interposed between a component and each of its clients that should perceive some new behaviour. For each operation visible to the client the wrapper either provides an own implementation or forwards corresponding messages to the "wrapped" component. In the sequel a wrapper will also be called a *child* and the "wrapped" component will be called its *parent*.

The wrapper approach enables unanticipated, dynamic, selective adaptation, joint use of different versions of a component and easy modeling of components that present different interfaces to different clients. However, wrappers in traditional class-based object-oriented systems fall short of achieving the desired adaptation functionality. The usefulness of the wrapper approach is limited by the underlying object model, which provides message sending as the only component interaction primitive[2]. Thus forwarding of messages that should be answered by the parent on behalf of the child can only be implemented by sending the message from the child to the parent.

2.1 Adaptation and the *Self* Problem

It has been repeatedly pointed out (e.g. by [Har97,Szy98]) that message sending limits the range of component interaction and component adaptation that can be achieved. This is due to the so called "*self*-problem" [Lie86] that is inherent in message passing. In [Har97] Harrison and Ossher rephrased the *self* problem in component-based terminology:

> Robust solutions [...] require that when components of composite objects invoke operations [...], the operations need to be applied to the composite object, rather than to the component object alone.

With respect to wrappers, this means, that when a message is *sent* from a child to a parent, the value of the *self* pseudovariable is bound to the parent. Thus all subsequent messages to *self* are addressed to the parent. As a consequence, the parent "is not aware" of modified behaviour in the child. If the child provides an own implementation of a method, say print(), the parent will ignore it and continue to use its own print() method, even in the context of forwarded messages.

Consider the wrapper-based modeling of the Euro transition scenario illustrated in listing 1.

The class DM (for the German currency) represents a component that existed before the decision to adopt the Euro was taken. It provides two methods:

- `amount()` computes the curent value of an investment (deposit, stock, assurance), using some private, non-shared data structures
- `foo()` does something else but calls `self.amount()` in its implementation.

The class EuroDMWrapper represents the wrapper. It encapsulates a reference to a DM instance and also provides two methods:

- `amount()` calls amount() on its wrapped DM instance and divides the result by the fixed DM-to-Euro exchange rate.
- `foo()` calls foo() on its wrapped DM instance

[2] For the purpose of this discussion events do not add any new insight and are therefore not mentioned.

```
public class DM{
       ... some private data ...
       ... constructor ...
    int amount() { return ...
    }
    void foo() { ... self.amount() ...
    }
}

public class EuroDMWrapper{
       // the wrapped component:
    DM parent;
       // constructor:
    EuroDMWrapper(DM p) { parent = p
    }
       // redefined method:
    int amount() { return parent.amount() / 1.96
    }
       // forwarding method:
    void foo() { parent.foo()
    }
}
```

Listing 1: The Euro scenario: Traditional wrapper-based adaptation fails

When a foo() message is sent to the wrapper, the same message is sent to its parent and in the course of its evaluation, the `self.amount()` message will be sent to the parent. Thus the adapted definition of `amount()` from the child will be ignored, and a wrong result will be produced in the end.

What is required in such cases is the ability to make one component act as a specialization of another one by sharing its *self*. This is exactly what is provided by delegation or object-based inheritance.

3 Delegation

"Delegation" was originally introduced by Lieberman ([Lie86]) in the framework of a class-free (prototype-based) object model. An object, called the *child*, may have modifiable references to other objects, called its *parents*. Messages for which the message receiver has no matching method are *automatically* forwarded to its parents. When a suitable method is found in a parent object (the *method holder*) it is executed after binding its implicit *self* [3] parameter. This parameter refers to the object on whose behalf the method is executed. Automatic forwarding with binding of *self* to the message receiver is called *delegation* (figure 1). Automatic forwarding with binding of *self* to the method holder is called *consultation*.

In contrast to code modification, delegation does not require source code or abstract binaries – it works equally well on native code and can be employed at run-time rather than at compile- or load time. Operating by addition rather than replacement and at the level of objects rather than classes delegation does not incur the schema-evolution problems of run-time code modification.

[3] The implicit self parameter is also called *this* (in Simula, Java and C++) and *current* (in Eiffel).

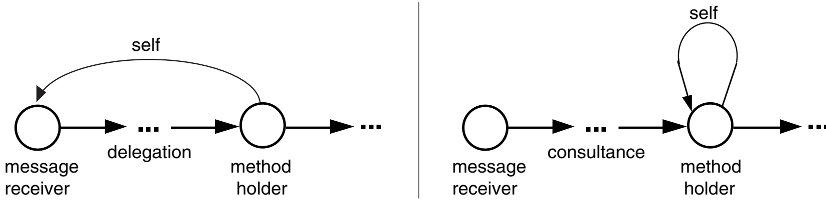


Fig. 1. Different effect of delegation and consultation on *self*

Why simulations of delegation are not enough. In spite of its advantages, no widely-used class-based object oriented language has incorporated delegation yet. Instead, various simulations of delegation have been proposed, either as language specific idioms or general "design patterns".The possible simulation techniques and their drawbacks are summarised and evaluated in [Har97] and [Kni98a]. The main disadvantages of the discussed simulations are:

- the need to anticipate the use of a piece of software as part of a larger composite and to build in "hooks" that allow the correct treatment of self in the context of the composite. Components that do not provide such hooks are not effectively composable ([Szy98]).
- the need to obey rigid coding conventions for implementing the required hooks. On one hand, the absence of a standard convention goes against any attempt to make composable software by simulating delegation. Components made according to different conventions cannot be deployed together. On the other hand, the risk of standardising immature proposals has been demonstrated by JavaSoft's "Object Aggregation and Delegation Model", which was initially contained in the Glasgow Proposal for the new JavaBeans model and has been dropped as result of public criticism of its limitations[4].
- the need to edit (or at least recompile, if a tool for automatic generation of forwarding methods is available) the "delegating" classes when the interface of the class "delegated to" changes, in order to propagate the change, e.g. the addition of a method. This introduces another variant of the "syntactic fragile base class problem" ([MS97,Szy98]).

Each of the individual simulation techniques has additional weaknesses in terms of limited applicability, limited functionality, limited reusability or excessive costs [Kni98a]:

- *Storing a reference to self in parent objects* has a very limited applicability. Sharing of one parent by multiple delegating children cannot be expressed at all and recursive delegation can only be simulated with significant run-time and software mainatenance costs.

[4] Note that the limitations of the Glasgow Proposal's rejected part have already been hardwired into the design of COM ([Box98])

– *Passing a reference to self as an argument* of forwarded messages requires to extend the interface of methods in parent objects, which might not be possible, if the parent object is part of a ready-made, black-box component.
  Furthermore the typing of the explicit *self* argument interacts in subtle ways with the construction of subclasses of parent classes. In the end, the simulation either does not reach the full functionality of delegation or it does so at the price of excessive costs for managing class hierarchy changes, rendering reuse *ad absurdum.* ([Kni98a]).

Why delegation has been (said to be) difficult. Considering the above list of limitations inherent to simulation approaches it is obviously worthwile to rethink the reasons why delegation has been considered unsuitable or unfeasible in the context of mainstream object-oriented languages. A survey of literature would reveal that many authors have acknowledged the modeling power and elegance of dynamic delegation but at the same time called for ways to harness this power and to make it more amenable to a "disciplined use" ([Don92,Szy98,Tai93,Wec97]). This is essentially a critique on the lack of a static type system for delegation-based languages. However, even some well respected authorities ([Aba96]) claimed that delegation cannot be combined with static typing and subtyping without severe restrictions of the way objects are used ([Fis95]). Therefore, the use of simulations seemed to be the only choice.

It is the main achievement of the DARWIN model ([Kni99]) to have shown that type-safe dynamic delegation with subtyping *is* possible and *can* be integrated into a class-based environment, laying the foundations for dynamic component adaptation. The DARWIN model is sketched in the next section to the degree relevant in the context of this paper, pointing out the interesting parallel between type-safety and independent extensibility of components ([Szy96]). The use of DARWIN for dynamic component adaptation is described in section 3.

4 Darwin and Lava: Combining Class-Based and Object-Based Inheritance

We assume that the reader is familiar with the notions of class, instance, and class-based inheritance ([Weg90]). For simplicity of presentation, we shall introduce DARWIN () using the syntax of LAVA ([Cos98,Sch97]), a proof of concept extension of Java that conforms to the DARWIN model [5] .

Parents and Declared Parents. In DARWIN / LAVA, objects may delegate to other objects referenced by their *delegation attributes*. Delegation attributes have to be declared in an object's class by adding the keyword `mandatory delegatee` to an instance variable declaration. Delegation attributes are *mandatory*, i.e. they must always have a non-nil value. This is automatically enforced by the compiler and suitable run-time checks. If class *C* declares a delegation attribute of type *T* we say that *C* is a *declared child class* of *T* (and of *T*'s subtypes), and *T* is a *declared parent type* of *C* (and of *C*'s subclasses).

[5] The only deviation of Lava from Darwin is the restriction to single inheritance following the design of Java. This is not a real limitation because Lava offers multiple delegation.

Dynamic Delegation. Since a delegation attribute can reference any value that conforms to its declared type, assignment to a delegation attribute can be used to change the behavior of an object at run-time by changing its parent object(s). This is called *dynamic* delegation. If desired, we can restrict delegation to be static by adding the Java keyword `final` to the delegation attribute's declaration.

Mandatory and Optional Atrributes. An attribute is called *mandatory* if it must always have a non-nil value, *optional* otherwise. This is specified by corresponding keywords that can be added to any attribute declaration (if nothing is specified *optional* is the default). The *mandatory* keyword is most relevant in connection with delegation attributes because it influences the typing relation between a child class and its parent type(s).

Types. In purely inheritance-based models, the type of an instance corresponds to the signature of the methods defined by its class (and its superclasses). Delegation has the effect of extending this interface by the interfaces defined for the *declared* types of *mandatory* delegation attributes. Thus, in LAVA delegation and inheritance are two orthogonal ways to create subtypes of a base type. Delegating objects may be used in any place where an object of their declared parent type is expected.

An Example. In LAVA, a class of text formatting objects (`Formatting`) that may use and dynamically switch between different line breaking strategies (type `LineBreaking`) can be written as shown in Listing 2

```
public class Formatting {
       // delegate line breaking requests to the object referred to by lb;
   mandatory delegatee LineBreaking lb;
       // create object with default strategy
   public Formatting () { lb = new SimpleLineBreaking();
   }
       // switch strategy
   public setLBStrategy (LineBreaking _lb) { lb = _lb;
   }
       // By how many pixels can individual text components be stretched
       // Overrides method from parent type LineBreaking.
   public int[] getStretchability() { ...
   }
}
```

Listing 2: The strategy pattern in Lava

The `Formatting` class may use all methods of the `LineBreaking` type as if they where locally defined or inherited from a superclass - with the essential difference that it may dynamically switch to a different set of method implementations simply by assigning an object of a different `LineBreaking` subtype to the variable `lb`. Like in the case of inheritance, the `Formatting` class can fine tune the "inherited" behavior via overriding. For instance, in the above example it was assumed that the `LineBreaking` type specifies that the line breaking algorithm calls the method `getStretchability()` to determine by how many pixels individual text elements can be stretched. Then providing a specialized version of this method (as shown above) might be all that is needed to adapt the "inherited" behavior to the delegator's needs. Note especially that the designer and the implementors of the `LineBreaking` type do not have to be aware of its use as a parent class and hard-code any hooks to enable this use.

4.1 Type Safety, Independent Extensibility and Overriding

Let us consider the scenario illustrated in figure 2 (Gray arrows represent delegation at instance and class level, arrows with hollow heads instantiation resp. inheritance, and solid arrows "normal" object references.) : `c`, an instance of class `Child`, delegates to `p`, an instance of class `Parent`; `Parent` is a subtype of the declared parent class of `Child`.

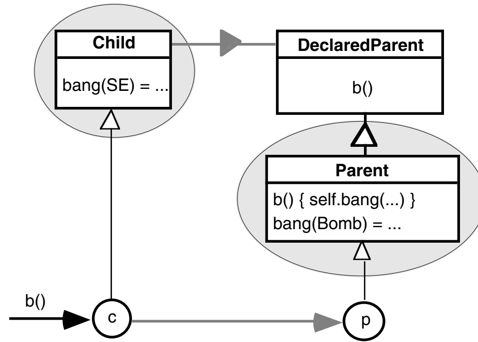


Fig. 2. What happens during evaluation of the message `c.b()`?

Typing Problem. In figure 2 the two `bang` methods have different argument types: `Child` expects an argument of type `StockExchange` whereas `Parent` provides an argument of type `Bomb`. Therefore, Fisher ([Fis95]) argues that during the evaluation of the message `c.b()` delegated from `c` to `p` the message `self.bang(aBomb)` sent back to `c` would be unsafe, because its argument would not have the expected type. However, arguing about type-safety in the above example is misleading, because the essence of the problem is not typing.

Independent Extensibility Problem. The astute reader might have noted that the classes `Child` and `Parent` might have been developed and compiled independently, knowing only `DeclaredParent` but not each other. Therefore, even if the two independently introduced `bang` methods had the same signature it would still be very unlikely that they have the same semantics. Overriding of `Parent::bang` by `Child::bang` would therefore be undesirable anyway, because it would silently change the semantics relied upon by methods from `Parent`, leading to obscure, hard to locate errors.

The importance of independent extensibility for component programming has already been described by Szyperski ([Szy96,Szy98]) in a delegation-free environment. Whereas Szyperski focused on the joint use of two independent extensions of a base type by a third party, our discussion relates to the delegation-based composition of one independent extension with another one.

Overriding. The point of the above discussion of type-safety and independent extensibility is that both seemingly unrelated problems have a common cause: the implicit

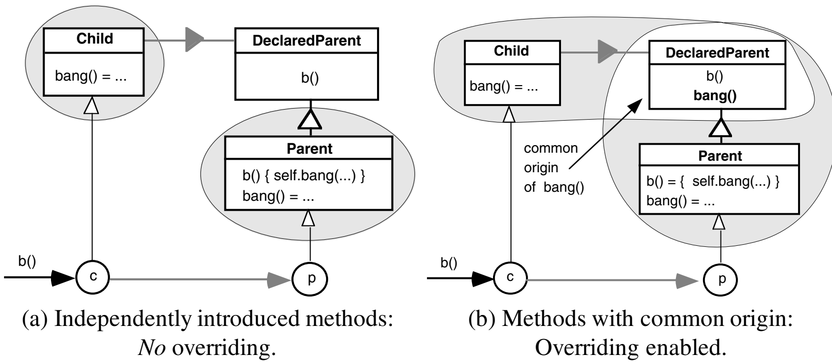
assumption that methods with the same name (and possible same signature) may override each other.

This assumption can be safely made only if it is guaranteed that the author of the overriding method is aware of the effect. This is always the case with class-based inheritance: a method implementation in a class can only override one in a known superclass. Assuming a sensible documentation of the superclass and no intentional fraud, authors of subclasses will not create semantically incompatible overriding methods. The assumption is unsafe if independent extensibility is possible.

The solution to both problems discussed above is the following adapted rule for method overriding:

For a message *recv.n(args)* a method with signature σ from type *T* *overrides* the matching [6] method from the static type of *recv*, T_{stat}, if there is some common *declared* supertype of *T* and T_{stat} that contains σ.

Thus a method from type *T* will only override methods from a declared parent type of *T*. This rule reflects the fact that the common declared supertype (`DeclaredParent` in figure 2 and figure 3) is the common semantic base on which implementors of independent extensions can rely.



(a) Independently introduced methods: *No* overriding.

(b) Methods with common origin: Overriding enabled.

Fig. 3. Overriding

The above definition of overriding influences the operational semantics of the system. A method of an object is applicable to a message only if it may override the matching method from the static type of the receiver expression. Messages with no applicable local method are delegated further to a parent object.

In the case of static delegation the type-safety of the approach follows from the fact that the sender of the message to *self* is itself among the parent objects.

For instance, in figure 3a, the message `self.bang()` sent from `p` to `c` will not find an applicable method in `c` and be delegated further up the object hierarchy, back

[6] A method with signature n(T1, ..., Tn):T matches a message recv.n(expr1, ..., exprn) if for all i = 1...n, the static type of expri is a subtype of Ti.

to p (where the search will succeed). In figure 3b the message will find an applicable method in c.

The type-safe treatment of dynamic delegation is more involved and its discussion is beyond the scope of this paper. Static delegation already sufices for most wrapper-based component adaptation tasks. A complete presentation of the DARWIN model including type-safe dynamic delegation is contained in [Kni99].

To recap, the integration of delegation into statically typed object-oriented languages offers, among others, an easy way

- to make an object appear to be part of and act on behalf of various other ones, and
- to extend existing objects in unanticipated ways, without fear of semantic conflicts.

As a general object-oriented language mechanism delegation has a multitude of possible uses. For instance, many well-known behavioural patterns ([GHJV95]) turn out to be simple applications of delegation (e.g. chain of responsibility, proxy, decorator, strategy, state, visitor). In the sequel we shall concentrate on the use of delegation for dynamic component adaptation.

5 DCA: Dynamic Component Adaptation

Dynamic component adaptation is a modification of a component's functionality at run-time that can be achieved by

- adding further components to a system and
- transferring part of the existing component's "wiring" to the new components.

The technical prerequisites for this functionality are language support for delegation and support for component "re-wiring" by the underlying component architecture. This section discusses both aspects. For simplicity components are considered to be JavaBeans. The described considerations equally apply to any other component model (e.g. COM) provided that in a future version it will support type-safe delegation according to DARWIN.

5.1 Incremental Component Assembly and Adaptation with Delegation

Delegation enables extension and modification (overriding) of a parent component's behavior. Child components can be transparently used in any place where parent components are expected. Unlike other approaches, which irrecoverably destroy the old version of a component, delegation enables two types of component modifications. Additive modifications are the product of a series of modifications, each applied to the result of a previous one. Disjunctive modifications are applied independently to the same original component.

Additive modifications are enabled by the recursive nature of delegation. They meet the requirement that the result of compositions / adaptations should itself be composable / adaptable ([Bos98,Szy98]). In the user view (figure 4a) additive composition is depicted by stacking components one on top of the other. The system view in figure 4b

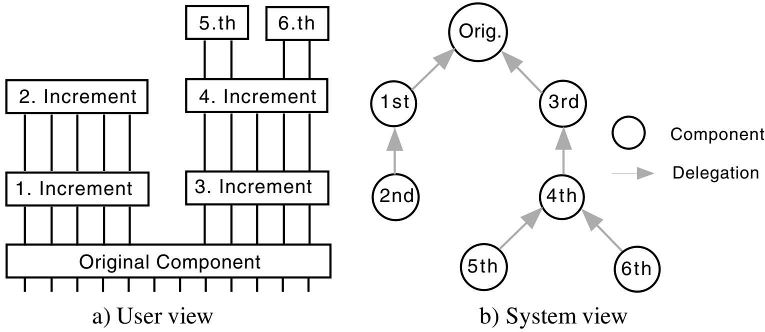


Fig. 4. Additive and disjunctive composition

shows the implementation by building chains of delegating components. E.g. the first and second increment of the original component together form an additive modification.

Disjunctive extensions are enabled by the fact that each extension is encapsulated in a separate component instance that can be addressed and reused independently. Disjunctive extensions are most useful in modeling components that need to present themselves differently to different clients. In the system view (figure 4a), disjunctive extension are visualized sitting on top of the jointly extended component. For instance component 5 and 6 represent different extensions of component 4, which itself is part of a disjunctive extension branch of the original component. At the implementation level (figure 4b), disjunctive extensions delegate to the same parent component.

5.2 Dynamic Component Assembly

The effects described so far only take effect if the most specialized increment components along each disjunctive modification branch are used as the receiver of messages or events instead of the original component. Therefore, dynamic component adaptation requires dynamic component (re)assembly, i.e.

- rerouting of all ”input connections” of a component to its increment (or to different disjunctive increments) and
- routing of the ”output connections” of all increments to the same destinations as the corresponding outputs of the original component.

In the JavaBeans model input connections correspond to the registration of a component as an event listener of other components and output connections correspond to the set of own registered event listeners.

The complete schema for dynamic component adaptation, including component reassembly (”re-wiring”), is illustrated in figure 5. Part a) shows the implementation view of the original component configuration. Part b) shows the re-wired configuration after addition of three increment components, one of which represents a disjunctive modification.

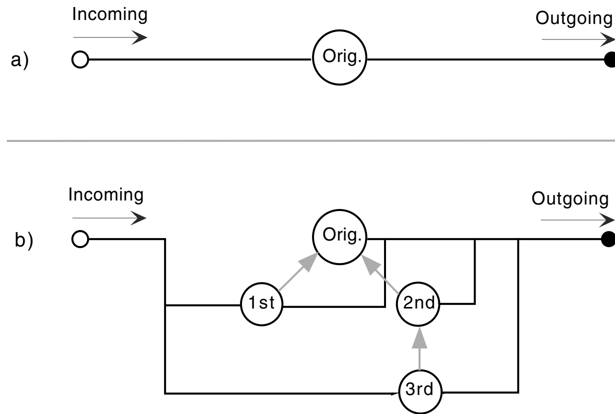


Fig. 5. Component (re)wiring: before extension (a) and after extension (b)

A component architecture that supports dynamic component (re)assembly must provide

- a run-time component directory,
- the ability to ask every component for its current input and output connections,
- the ability to ask every component to abandon all its input connections in favor of one or more other components. This must happen as an atomic operation in order to guarantee that the system is not left in an inconsistent state.

These requirements are not met by current component architectures, although dynamic component rewiring is an essential infrastructure which would benefit also simple forwarding based dynamic composition techniques, not just delegation. E.g. the ”Extensible Runtime Containment and Services Protocol” of the Glasgow model for JavaBeans

- provides no run-time component directory; every BeanContext functions as a directory of nested components but there is no directory of top-level BeanContexts,
- a JavaBean is only required to maintain a list of registered Listener objects but no list of objects to which it listens itself; thus it only knows about its outgoing connections but not about incoming connections,
- there is no atomic operation for handing a bean’s incoming connections over to another bean.

We are currently exploring suitable extensions of the JavaBeans model. Similar dynamic rewiring infrastructures have also been proposed and already implemented by other researchers, e.g. [PBJ98].

5.3 Example: Delegation-Based Euro Transition

Comming back to the example from Listing 1 this section shows how a delegation-based modelling of the Euro scenario would look like. Listing 3 shows the delegation-based variant of the EuroDMWrapper class. Note that an explicit definition of the `foo` method is not necessary. Messages for `foo` are implicitly delegated to `parent`. The `<-` operator in the amount method denotes explicit delegation to the object reffered to by `parent`. This is analogous to a `super` call in class-based inheritance.

```
public class EuroDMWrapper{
      // delegate to the object referred to by parent:
    mandatory delegatee DM parent
      // constructor:
    EuroDMWrapper(DM p) {  parent = p
    }
      // redefined method:
    int amount() { return parent<-amount() / 1.96
    }
}
```

Listing 3: The Euro scenario: Delegation-based adaptation

Now a `foo` message to the wrapper will produce correct results because the `self.amount()` message from DM (cf Listing 1) will correctly be addressed to the wrapper object thus using the adapted definition of `amount()`.

6 Conclusions

The crucial role of dynamic, object-based inheritance (delegation) as a basis of component interaction and the need for an integration of delegation into the statically typed class-based model has been repeatedly pointed out by many researchers. In this context the contributions of this paper are two-fold:

- In the first place, it introduced a general model for dynamic, type-safe delegation, DARWIN, and an implemented language, LAVA, that provide the required language support for component-oriented programming.
- Secondly, the problem of *dynamic* component adaptation was discussed and a solution based on delegation and dynamic component reassembly was presented.

There are, nevertheless, still many open questions. For instance, a high-performance implementation of LAVA is required, to help the current proposal make its way into mainstream commercial languages like Java or C++, thus providing broad language support for delegation-based component interaction. Also, components to be composed by delegation depend on the “specialization interface” of their parent components. Therefore, advanced component interface specifications and approaches to the “semantic fragile base class problem” are required ([Lam93,Ste96,MS97,Szy98,Wec97]).

References

Aba96. Abadi, Martin and Cardelli, Luca. *A Theory of Objects*. Springer, 1996.

ACLN98. P.S.C. Alencar, D. D. Cowan, C.J.P. Lucena, and L.C.M. Nova. A model for gluing components. In Wolgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proc. of Third Intern. Workshop on Component Oriented Programming (WCOP 98)*, volume 10, pages 101–108. Turku Center for Computer Science, 1998.

Bos98. Bosch, Jan. Superimposition - A Component Adaptation Technique, 1998.

Box98. Don Box. *Essential COM*. Addison Wesley, 1998.

Cos98. Costanza, Pascal. *Lava: Delegation in a Strongly Typed Programming Language – Language Design and Compiler (In German: Lava:Delegation in einer streng typisierten Programmiersprache – Sprachdesign und Compiler).* Masters thesis, University of Bonn, 1998.

Don92. Dony, Christophe and Malenfant, Jacques and Cointe, Pierre. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, 27(10):201–217, 1992.

Fis95. Fisher, Kathleen and Mitchell, John C. A Delegation-based Object Calculus with Subtyping. In *Proceedings of 10th International Conference on Fundamentals of Computation Theory (FCT '95)*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer, 1995.

GHJV95. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.

Har97. Harrison, William and Ossher, Harold and Tarr, Peter. Using Delegation for Software and Subject Composition. Research Report RC 20946 (922722), IBM Research Division, T.J. Watson Research Center, Aug 1997.

HO93. William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, 28(10):411–428, 1993.

Hol98. Holzle, Urs and Keller, Ralf. Binary Component Adaptation. In *Proceedings of ECOOP 98*, pages 307–329. Springer Verlag, 1998.

HOT98. William Harrison, Harold Ossher, and Peter Tarr. Load-time subject-oriented programming, June 1998 1998.

KAZ98. Bülent Kücük, M. Nedim Alpdemir, and Richard N. Zobel. Customizable adapters for black-box components. In Wolgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proc. of Third Intern. Workshop on Component Oriented Programming (WCOP 98)*, volume 10, pages 53–60. Turku Center for Computer Science, 1998.

Kel97. Keller, Ralph and Hölzle, Urs. Supporting the Integration and Evolution of Components Through Binary Component Adaptation. Technical Report TRCS97-15, University of California at Santa Barbara, September 1997.

Kni98a. Günter Kniesel. Delegation for java api or language extension? Technical Report IAI-TR-98-4, ISSN 0944-8535, University of Bonn, May 1998 1998.

Kni98b. Günter Kniesel. Type-safe delegation for dynamic component adaptation. In Wolgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proc. of Third Intern. Workshop on Component Oriented Programming (WCOP 98)*, volume 10, pages 9–18. Turku Center for Computer Science, 1998.

Kni99. Günter Kniesel. *Darwin - A Unified Model of Sharing for Object-Oriented Programming*. Ph.d. thesis (forthcoming), University of Bonn, 1999. Knie98.

Lam93. Lamping, John. Typing the Specialization Interface. *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, 28(10):201–214, 1993.

Lie86. Lieberman, Henry. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 21(11):214–223, 1986.

MS97. Kai-Uwe Mätzel and Peter Schnorf. Dynamic component adaptation. Technical report 97-6-1, Union Bank of Swizerland, June 1997 1997.

PBJ98. F. Plasil, D. Balek, and R. Janecek. Sofa/dcup:architecture for component trading and dynamic updating. In *ICCDS 98*, Annapolis, Maryland, USA, 1998. IEEE CS Press.

Sch97. Schickel, Matthias. *Lava: Design and Implementation of Delegation Mechanisms in the Java Runtime Environment (In German: Lava:Konzeptionierung und Implementierung von Delegationsmechanismen in der Java Laufzeitumgebung).* Masters thesis, University of Bonn, 1997.

Ste96. Steyaert, Patrick and Lucas, Carine and Mens, Kim and D'Hondt, Theo. Reuse Contracts: Managing the Evolution of Reusable Assets. *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 268–285, 1996.

Szy96. Szyperski, Clemens. Independently Extensible Systems Software Engineering Potential and Challenges . In *Proceedings of the 19th Australian Computer Science Conference, Melbourne, Australia*. Computer Science Association, 1996.

Szy98. Szyperski, Clemens. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

Tai93. Taivalsaari, Antero. Object-Oriented Programming with Modes. *Journal of Object-Oriented Programming (JOOP)*, 6(3):25–32, 1993.

Wec97. Weck, Wolfgang. Inheritance Using Contracts & Object Composition. In Weck, Wolfgang and Bosch, Jan and Szyperski, Clemens, editor, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP '97)*, pages 105–112. Turku Center for Computer Science, Turku, Finland, 1997. ISSN 1239-1905.

Weg90. Peter Wegner. Concepts and paradigms of object-oriented programming (expansion of oopsla 89 keynote talk). *ACM OOPS Messenger*, 1(1):7–87, 1990.

Towards Automatic Specialization of Java Programs*

Ulrik Pagh Schultz, Julia L. Lawall**, Charles Consel, and Gilles Muller

Compose Group, IRISA
Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France
{ups,jll,consel,muller}@irisa.fr
http://www.irisa.fr/compose

Abstract. Automatic program specialization can derive efficient implementations from generic components, thus reconciling the often opposing goals of genericity and efficiency. This technique has proved useful within the domains of imperative, functional, and logical languages, but so far has not been explored within the domain of object-oriented languages.
We present experiments in the specialization of Java programs. We demonstrate how to construct a program specializer for Java programs from an existing specializer for C programs and a Java-to-C compiler. Specialization is managed using a declarative approach that abstracts over the optimization process and masks implementation details. Our experiments show that program specialization provides a four-time speedup of an image-filtering program. Based on these experiments, we identify optimizations of object-oriented programs that can be carried out by automatic program specialization. We argue that program specialization is useful in the field of software components, allowing a generic component to be specialized to a specific configuration.

1 Introduction

Although initially designed for embedded systems and marketed as a language for web-based programming, Java is increasingly gaining acceptance as a general-purpose language. The object-oriented paradigm has well-recognized advantages for application design, and more specifically for program structuring. It makes it possible to decompose an application into well-defined, generic components, closely corresponding to the structure of the modeled problem. This structuring leads to a number of important software engineering improvements regarding maintainability and re-usability of code. However, these advantages often translate into a loss in performance.

The conflict between software generality and performance has been recognized in areas such as operating systems [6] and graphics [20]. This conflict

* Supported in part by BULL, and in part by Alcatel under the Reutel 2000 contract.
** Supported in part by an H.H. Powers grant from Oberlin College, and by NSF grant EIA-9806718. Author's current address: Department of Computer Science, Brandeis University, Waltham, MA 02254. e-mail: jll@cs.brandeis.edu

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 367–390, 1999.
© Springer-Verlag Berlin Heidelberg 1999

is being increasingly addressed, with success, using forms of *program specialization* [21]. Program specialization adapts a generic program component to a given usage context. This approach can lead to considerable performance gains, by eliminating from the general code all aspects not related to that precise context.

Specialization has been performed manually by adapting critical program components to the most common usage patterns [5,28,29]. Manual specialization improves performance, but has a limited applicability, because the process is error-prone. Recently, tools have been developed to automatically specialize programs [1,2,3,7,8,18,19]. Applications of program specialization are emerging in a number of fields, including scientific code [3,4,8], systems software [13,24], and computer graphics [15], with very promising results. However, automatic specialization of object-oriented programs remains uninvestigated. Given the existing base of experience with imperative languages, we construct a specializer to optimize object-oriented programs, specifically programs written in Java.

Developing an automatic specializer for a realistic imperative language is a long and complex task. Rather than designing a Java specializer from scratch, we are experimenting with an approach based on the Tempo [8] specializer for C programs, coupled with the Harissa optimizing Java-to-C compiler [25,26].

In this paper, we present preliminary experiments in specializing Java programs. Our contributions are as follows:

- We identify specialization transformations that are particularly useful in object-oriented languages. These transformations are implemented within a single, uniform framework.
- We illustrate our approach with a concrete example, a graphical filtering application, transforming the generic code into a form close to hand-optimized code.
- We apply the declarative approach to specialization proposed by Volanschi *et al.* [32] to specify specialization strategies in a high-level manner. In contrast to Volanschi *et al.*'s work, however, we perform specialization automatically, not manually.
- We demonstrate that the choice of C as a target language for specializing Java programs makes it possible to address optimizations at all levels from the source program to the run-time environment. We also address the issue of expressing the result of program specialization as Java source code.
- We argue that program specialization is a key approach to improving the performance of generic software components, by adapting the code of a component to its configuration.

This paper is organized as follows: first, Section 2 informally presents specialization. Then, Section 3 introduces our example program. This program is specialized in Section 4, where we identify some typical opportunities for specialization in Java programs. Afterwards, Section 5 describes the functionality of the staged program specialization process implemented by our prototype. Section 6 describes related work. Section 7 discusses future work. Finally, Section 8 concludes.

2 Background

Program specialization has been explored for a variety of languages ranging from functional to logic languages. In recent years, program specializers for real-sized languages such as Fortran [3] and C [1,8] have been developed. Realistic applications of program specializers to areas such as systems software [24] and scientific computing [3] have clearly demonstrated that automatic program specialization is an effective tool to allow programmers to write *generic* programs without loss of efficiency.

Although object-oriented languages encourage genericity, and thus offer opportunities for specialization, specialization has so far been mostly unexplored for this class of languages.

In this section, program specialization is introduced. We also present a declarative approach to specifying how a program should be specialized.

2.1 Program Specialization

Intuitively, program specialization is aimed at instantiating a program with respect to some of its parameters. By restricting a generic program to a specific usage context, one hopes to enable aggressive optimizations. For an object-oriented program, this approach can be applied to the methods of a class. A method can be specialized with respect to parts of its calling context including parameters, fields of the enclosing object, and static fields of other classes. For example, the parameters of a method may represent options to be analyzed to determine a particular task to be performed. Specialization can typically eliminate the interpretation of such contextual information.

One approach to program specialization is *partial evaluation*, which performs aggressive interprocedural constant propagation (of all data types). Based on the input values specified by the user and on constants explicit in the program, partial evaluation does constant folding, as well as optimizations such as loop unrolling and some forms of strength reduction. Computations that can be simplified based on information available during specialization are known as *static*. Other computations are known as *dynamic*, and are reconstructed literally to form the specialized program.

Let us illustrate program specialization with a simple Java class for computing the power function, displayed in Figure 1.

Assume the `calculate` method is invoked repeatedly with a specific exponent, say `3`, within a loop where only the base changes. In this situation, it is worthwhile to specialize the `calculate` method with respect to the given exponent. Specialization unrolls the loop in the `calculate` method, producing the following specialized method.

```
int calculate_3( int b ) {
  int res;
  res = b * b * b;
  return res;
}
```

```
class Power {
  private int exp;
  Power( int e ) { this.exp = e; }
  int calculate( int b ) {
    int res = 1;
    for( int i=0; i<this.exp; i++ )
      res *= b;
    return res;
  }
}
```

Fig. 1. A Java class for the power function.

The set of values to specialize over may become gradually available during compilation and execution. Thus, a program may need to be specialized both at compile time and at run time. In the `Power` class example, we could have dynamically generated a specialized version of `calculate` for any exponent supplied at run time. In fact, the partial evaluator Tempo offers both strategies in a uniform environment [9]. Run-time specialization widens the opportunities to eliminate genericity in programs.

Partial evaluation differs from ordinary optimization in that no resource limits are imposed on the computations that are performed during transformation. While this enables transformations that are out of the scope of an ordinary compiler, it does imply that the partial evaluation process must be guided by the user. Because the process of specializing parts of a large program, and then reorganizing the program to use the specialized code, is complex, we need a language for declaring specialization opportunities.

2.2 A Declarative Approach to Program Specialization

Volanschi *et al.* [32] present a declarative approach for specifying specialization opportunities in object-oriented programs. Specialization opportunities are declared separately from the program, in the form of *specialization classes*. A specialization class defines the conditions under which specialization should occur and what methods to specialize.

Let us illustrate this approach for the `Power` class example of the previous section. The associated specialization class is defined as follows.

```
specclass Cube specializes Power {
  exp == 3;
  calculate( int b );
}
```

This specialization class specifies that the `calculate` method should be specialized with respect to the value of the `exp` field. Mentioning the `exp` field declares

it as a static location, and providing a value indicates that specialization can be carried out at compile time.

If no value were supplied for the `exp` field in the specialization class, specialization would occur at run time when the `exp` field is assigned a value. In this case, the specialization class expresses the possibility to specialize `calculate` for any exponent.

Specialization classes are processed by the *Java Specialization Class Compiler*, which determines what methods should be specialized and to what values, as well as whether specialization should occur at compile time or run time. The Java Specialization Class Compiler also adds a method to the original program for switching between specialized implementations (and for generating the specialized implementations at run time if needed), and places guards that automatically invoke the method for switching implementations when a value that was used for specialization changes.

```
private void setImplementation() {
  if( this.exp == 3 )
    this.scImpl = this.getSpecImpl( "Cube" );
  else
    this.scImpl = this.getGenImpl();
}
```

An invocation of the original method is replaced by an invocation of the method currently stored in the `scImpl` field. This field can either contain the specialized method, or the original (generic) implementation.

The `Cube` specialization class specifies specialization with respect to an integer value. Specialization classes also allow invariants over object types to be expressed, by specifying the type of a field to be the name of a specialization class. These references between specialization classes allow the user to specify properties ranging across several objects. As described by Volanschi *et al.*, the dependencies between such aggregate specialization classes can be determined by the specialization class compiler, making it possible to eliminate the virtual call that is otherwise used to switch between implementations. This optimization can be essential for an efficient implementation, and is used for the specialization example of the next section.

3 Specialization Example

Object-oriented programs can be given structure and genericity by being composed from individual objects that interact through generic interfaces. As an example, we present an image filtering program. Here, multiple layers of abstraction facilitate extensibility and maintenance of a wide range of functionalities.

We use the example to illustrate specialization opportunities in object-oriented programs. In particular, the example exploits well-known abstraction mechanisms such as the strategy and abstract factory design patterns [14]. In general,

the performance benefits of specialization depend on both the amount of overhead that is eliminated by specialization, and on the specialization opportunities presented by the algorithm itself.

This section first introduces image filtering, then details the structure of the example, and finally discusses the opportunities that we find for specialization.

3.1 Image Filtering

We consider image filtering based on a matrix, known as a *mask*. Filtering is performed by moving the mask across an image, computing the filtered pixel in the position of the center of the mask by performing operations on the pixels covered by the mask. Such filters can be used to obtain a variety of effects, including blurring, edge detection, and noise elimination [30]. For cache efficiency, the image is decomposed into rectangular *tiles*, and filtering is performed a tile at a time.

There are many possible representations for the data that defines an image, each representation having specific advantages. For this reason, all image data are manipulated abstractly by the filtering process, making it possible to choose the most efficient representation for an image independently of the choice of the image filter.

3.2 Structure of the Implementation

The structure of the example is shown with an object diagram in Figure 2. Only those classes that are critical to the presentation are shown. The central class `ImageFilter` manipulates data stored in a `Tile` using a pixel processing strategy defined as an `ImageOperator`, implemented by `ConvolutionOperator` and `MedianOperator`. The `ConvolutionOperator` is parameterized by a `Kernel` that defines its mask. A `Tile` stores the offsets of the tile within the image and a `DataBuffer` for holding image data. The image data processed by the operators is accessed from the `DataBuffer` through `Pixel` objects, via the data representation strategy defined by `SampleModel`. `Pixel` objects are created using `SampleModel` as an abstract factory.

We now describe the implementation in more detail, showing the parts of the source code that will later be specialized into more efficient implementations.

The `ImageFilter` class (Figure 3) processes a tile of the image. The `getTile` method produces a filtered tile using a double `for`-loop that applies the `computePoint` method of the filtering operator to each pixel within the source tile.

The `ConvolutionOperator` (Figure 4) and `MedianOperator` classes, which implement `ImageOperator`, each define a `computePoint` method which computes a filtered pixel.[1] In the `computePoint` method of `ConvolutionOperator`, the pixels

[1] The convolution operator computes a linear combination of the points covered by the mask, while the median operator selects the median of the pixels covered by the mask.

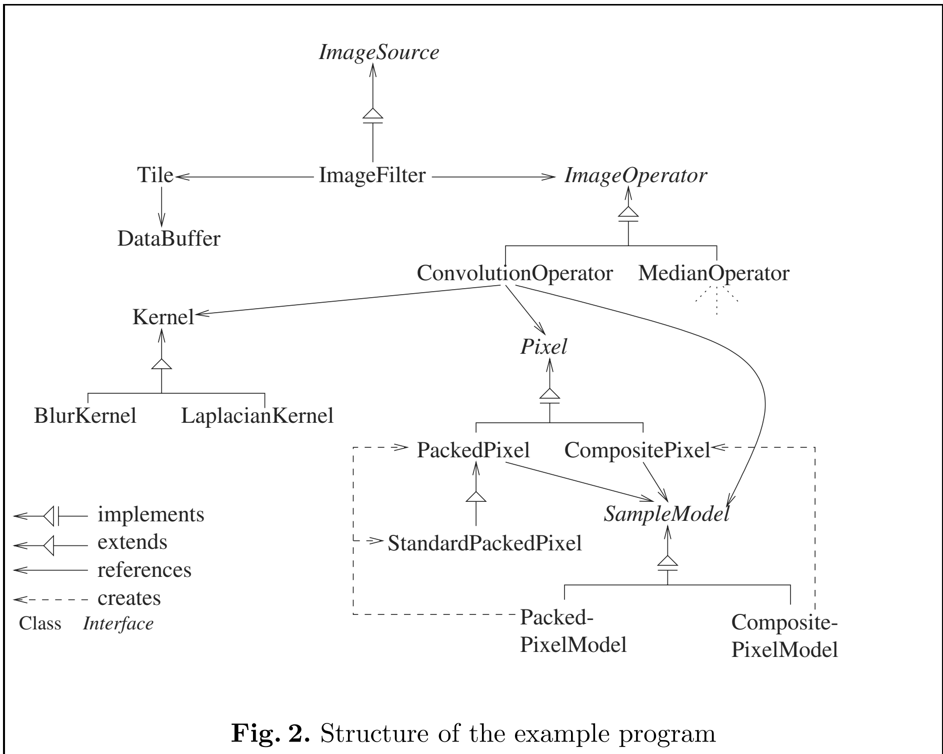


Fig. 2. Structure of the example program

covered by the mask[2] are traversed by a double `for`-loop, and extracted into the pixel `p`, using the `getPixel` method of the class `Tile` (Figure 5). This method uses the `SampleModel` associated with the `Pixel` to look up the data in the `DataBuffer` of the `Tile`. Finally, the pixel is scaled according to the mask, and the result is accumulated in the pixel `acc`.

The `PackedPixelModel` class is an implementation of `SampleModel`, where multiple samples (color components, for example) are packed into a single integer value. The methods for reading a pixel (from data stored in a `Tile`) using this model are shown in Figure 5. The `PackedPixel` class is an implementation of `Pixel` that is instantiated using the `PackedPixelModel` as a factory. For a more efficient representation, the subclass `StandardPackedPixel` (Figure 6) of `PackedPixel` is instantiated when the pixel consists of three 8-bit samples. To avoid unnecessary extraction of samples from a packed pixel value, this implementation switches between a packed representation and a directly accessible representation, as indicated by the `usingSamples` field.

[2] The specialization class compiler must guard the values stored in the mask, so the special class `ArrayOfInt` is used to represent the mask, rather than an ordinary integer array.

```
public class ImageFilter implements ImageSource {
  ImageSource src;
  ImageOperator op;
...
  public Tile getTile( int tx, int ty, int tw, int th ) {
    int b = op.getBorder();
    Tile in = src.getTile( tx-b, ty-b, tw+2*b, th+2*b );
    Tile out = in.createOutTile( tx, ty, tw, th );
    for( int y=0; y<th; y++ )
      for( int x=0; x<tw; x++ )
        out.setPixel( x, y, op.computePoint( x+b, y+b, in ) );
    return out;
  }
}
```

Fig. 3. The `ImageFilter` class. Applies an operator to each pixel of an image.

While the separation of the image processing algorithm into distinct classes simplifies the implementation and facilitates future extensibility, it induces a heavy performance penalty. Even when limited to a small blurring mask, our implementation performs 50 virtual calls to filter a single pixel! A hand-optimized implementation, where a dedicated filter is programmed independently for each kind of operator and data representation, would perform no virtual calls. Indeed, image processing applications are rarely structured with as much genericity as we have chosen for our application.[3] Instead, efficiency is enhanced at the price of genericity and ease of maintenance. As an alternative, program specialization can be applied to our program to enhance performance. We start by identifying the critical points that offer opportunities for optimization by specialization.

3.3 Opportunities for Specialization

The image filtering program is structured to allow flexibility in specifying the filter to be applied to the image, and the concrete representation of the pixels of the image. Nevertheless, once we begin applying a particular filter to a particular image, both the filter and the pixel representation remain invariant. Thus, the flexible program structure we have chosen presents significant opportunities for specialization.

The `ImageFilter.getTile` method (of Figure 3) can be specialized to a specific filtering operator, removing the separation between the image traversal and the action to perform for each pixel. This specialization is captured by the `ImageFilterForConvolution` specialization class of Figure 7, which specifies that a blurring operator should be used.

[3] For example, in the Java 2D API, it is recommended to type cast to each specific sample model, having dedicated code for each kind of representation.

```
public class ConvolutionOperator implements ImageOperator {

  int mask_width, mask_height, mask_center_x, mask_center_y;
  double mask_divisor;
  ArrayOfInt mask; // implements a simple integer array
  SampleModel model;

  ...

  Pixel acc_pixel, p_pixel; // initialized using model.getNewPixel()

  public Pixel computePoint( int x, int y, Tile inTile ) {
    int mx, my, xmax, ymax;
    int mask_element;
    ymax = mask_height - mask_center_y;
    xmax = mask_width - mask_center_x;
    Pixel acc = acc_pixel, p = p_pixel; // For memory efficiency
    acc.reset();                  // Reset pixel to neutral color

    for( my=-mask_center_y; my<ymax; my++ )
      for( mx=-mask_center_x; mx<xmax; mx++ ) {
          inTile.getPixel( x + mx, y + my, p ); // Covered by mask
          mask_element =
            mask.get(   (my+mask_center_y) * mask_width
                      + (mx+mask_center_x) );
          p.scale( mask_element ); // Scale pixel colors
          acc.add( p );            // Add pixel colors
        }

    acc.normalize( mask_divisor ); // Normalize to normal range
    return acc;
  }
}
```

Fig. 4. The `ConvolutionOperator` class. Combines pixels covered by the kernel.

```
public class Tile {
  int x, y, width, height; // Coordinates of Tile in image
  public DataBuffer data;
  ...
  void getPixel( int x, int y, Pixel p ) {
    p.getModel().getPixel( data, x+y*width, p );
  }
  ...
}

public class PackedPixelModel implements SampleModel {
  ...
  public void getPixel( DataBuffer d, int idx, Pixel p ) {
    ((PackedPixel)p).setValue( d.intData[idx] );
  }
  ...
}
```

Fig. 5. The `Tile` and `PackedPixelModel` classes. Storage of image data and access to image data.

The `ConvolutionOperator.computePoint` method (of Figure 4) can be specialized to apply a specific convolution kernel to the image. The specialization class `BlurConvolutionOperator` of Figure 7 specifies a three-by-three kernel that takes the average of the nine surrounding pixels. The pixels manipulated by this operator are specified to be represented by a specific sample model.

An operator manipulates image data through methods in `Tile` and the concrete `Pixel` implementation. These methods make use of a concrete `SampleModel` class to manipulate the raw data stored in the `DataBuffer` of the `Tile`. Specializing the operator to a concrete `SampleModel` class exposes the concrete type of pixels, and allows the data in the `DataBuffer` to be directly manipulated. The specialization class `RGB8bitPixelModel` of Figure 7 captures the common case of a `PackedPixelModel` with three 8-bit samples used to represent a pixel (i.e., red, green, and blue). This specialization class implies that the `PackedPixelModel` always instantiates pixels of type `StandardPackedPixel` (by the implementation of the concrete `Pixel` factory, not shown).

The specialization classes of Figure 7 are linked together so that when the `getTile` method is called, the concrete `ImageFilter` object is in a specialized state only if the aggregate objects described by the aggregate specialization classes are also in a specialized state. Thus, a virtual call to make the choice of implementation is only necessary when calling the `getTile` method.

We next describe how we realize these specialization opportunities.

```
public class StandardPackedPixel extends PackedPixel {
  int value;
  boolean usingSamples;          // When false, 'value' is used
  int sample1, sample2, sample3; // Three 8-bit samples
  public void setValue( int v ) {
    value = v; usingSamples = false;
  }
  void initializeSamples() {
    int pixel = value;
    sample1  = (pixel & 0xff0000) >> 16;
    sample2  = (pixel &   0xff00) >>  8;
    sample3  = (pixel &     0xff);
    usingSamples = true;
  }
  public void scale( int s ) {
    if( !usingSamples ) initializeSamples();
    sample1 *= s; sample2 *= s; sample3 *= s;
  }
... // Other methods implementing the Pixel interface
}
```

Fig. 6. The `StandardPackedPixel` class. Representation dedicated to 8-bit pixels with three samples.

```
specclass ImageFilterForConvolution specializes ImageFilter {
  BlurConvolutionOperator op;
  void getTile( int tx, int ty, int tw, int th );
}
specclass BlurConvolutionOperator specializes ConvolutionOperator {
  mask_width == 3;
  mask_height == 3;
  mask_center_x == 1;
  mask_center_y == 1;
  mask_divisor == 9.0;
  mask == {1,1,1,1,1,1,1,1,1};
  RGB8bitPixelModel model;
  void computePoint( int x, int y, Tile inTile );
}
specclass RGB8bitPixelModel specializes PackedPixelModel {
  numberOfSamples == 3;
  bitsPerSample == 8;
}
```

Fig. 7. Declaration of specialization opportunities.

4 Specializing Java Programs

Traditionally, program specialization optimizes a program based on input values provided by the user, and on constants explicit in the program. In the case of Java, specialization can exploit additional static information, such as the type of each object, and can simplify operations implicit to the virtual machine.

Throughout this section, for readability, we illustrate the result of specialization using Java code, rather than the equivalent C code actually produced automatically by our implementation.

4.1 Specializing Data Encapsulation

In an object-oriented language such as Java, values are systematically encapsulated. As a consequence, accessing a value is implemented in terms of a sequence of pointer dereferences whose cost depends on the embedding depth of the object structure. Specialization replaces a reference to a field containing a static value by the value itself. This transformation eliminates memory references and improves performance.

Example. The specialization class `BlurConvolutionOperator` (Figure 7) indicates that `computePoint` (Figure 4) should be specialized with respect to the dimensions and weights of the mask. Specialization propagates these constants, eliminating references to these fields, and triggering other optimizations.

Because the dimensions of the mask control the double `for`-loop of `computePoint`, it can be unrolled during specialization. The resulting code consists of nine (i.e., `mask_width * mask_height`) blocks of code, each specialized according to the current loop indices. These known indices allow the specializer to evaluate the references to the weights of the mask, thus replacing these array references by constants. The result of this specialization is illustrated in Figure 8.

4.2 Specializing Object Types

In an object-oriented language such as Java, control flow depends on object types as well as program values. The choice of which method is invoked by a Java method call depends on the type of the receiver object. When the definition of the invoked method cannot be determined at compile time, the method call is said to be *virtual* and is typically implemented using a table and a pointer dereference. If the type of the receiver object can be determined based on extra information available during specialization, then specialization replaces a virtual method call by an ordinary procedure call. Such a procedure call can be inlined, leading to further optimizations.

Example. The `getTile` method of the `ImageFilter` class (Figure 3) contains a virtual call `op.computePoint` in a doubly nested loop. The `ImageFilterForConvolution` specialization class of Figure 7 specifies that `op` has type `BlurConvolutionOperator`. Specializing `ImageFilter` to the type of `op` replaces the

```
public Pixel computePoint( int x, int y, Tile inTile )
{
  int mask_element;
  Pixel acc = acc_pixel, p = p_pixel;
  acc.reset();
  {
    inTile.getPixel( x + (-1), y + (-1), p );
    p.scale( 1 );
    acc.add( p );
  }
  // Repeats nine times, with different arguments to getPixel
...
  acc.normalize( 9.0 );
  return acc;
}
```

Fig. 8. The `computePoint` method specialized for data values.

virtual call `op.computePoint` by a direct call to the `computePoint` method of the specialized `ConvolutionOperator` class, thus enabling subsequent inlining. (To save space, this optimization is not illustrated in the figures.)

The use of the sample model in the `computePoint` method provides a more dramatic example. The specialization class `BlurConvolutionOperator` specifies that pixels are represented according to the specialization class `RGB8bitPixelModel`. This specialization class implies that the sample model is a `PackedPixelModel` class that creates pixels using the optimized `StandardPackedPixel` representation. Thus, specialization replaces virtual calls performing pixel operations with direct calls to the methods of a `StandardPackedPixel` object. Furthermore, the implementation of `StandardPackedPixel` is optimized to choose between two representations, depending on how the pixel is used. Because specialization can determine the control flow through these methods, it can eliminate the overhead (specifically, the `usingSamples` field) associated with maintaining two possible representations. The transformed method, which can be viewed as a more specialized version of Figure 8, is illustrated in Figure 9. The C code actually produced by specialization can be compiled using an optimizing C compiler, which (for example) eliminates the multiplications by 1.

4.3 Specializing the Virtual Machine

Java requires run-time checking to be performed by the virtual machine to ensure some safety properties. In particular, casts and array references are systematically checked. An object can only be cast to a type that is a supertype of the actual object type. An array reference must access an element within the array bounds. In general, these tests must be carried out at run time. When the specializer can determine either the type of the object, or the size of the array and

```
public Pixel computePoint( int x, int y, Tile inTile )
{
  Pixel acc = acc_pixel, p = p_pixel;
  ((StandardPackedPixel)acc).value = 0;
  ((StandardPackedPixel)acc).usingSamples = false;

  {
    // inTile.getPixel( x + (-1), y + (-1), p )
    ((StandardPackedPixel)p).value =
      inTile.data.intData[(x-1)+(y-1)*intTile.width];

    // p.scale( 1 )
    {
      int pixel = ((StandardPackedPixel)p).value;
      ((StandardPackedPixel)p).sample1  = (pixel & 0xff0000) >> 16;
      ((StandardPackedPixel)p).sample2  = (pixel &   0xff00) >>  8;
      ((StandardPackedPixel)p).sample3  = (pixel &     0xff);
      ((StandardPackedPixel)p).usingSamples = true;
    }
    ((StandardPackedPixel)p).sample1 *= 1;
    ((StandardPackedPixel)p).sample2 *= 1;
    ((StandardPackedPixel)p).sample3 *= 1;

    // The specialized code for acc.add( p )
    ...
  }

  // This block repeats nine times, with different array indices
  ...
  // Afterwards, the specialized code for acc.normalize( 9.0 );
  ...

  return acc;
}
```

Fig. 9. The `computePoint` method specialized for types.

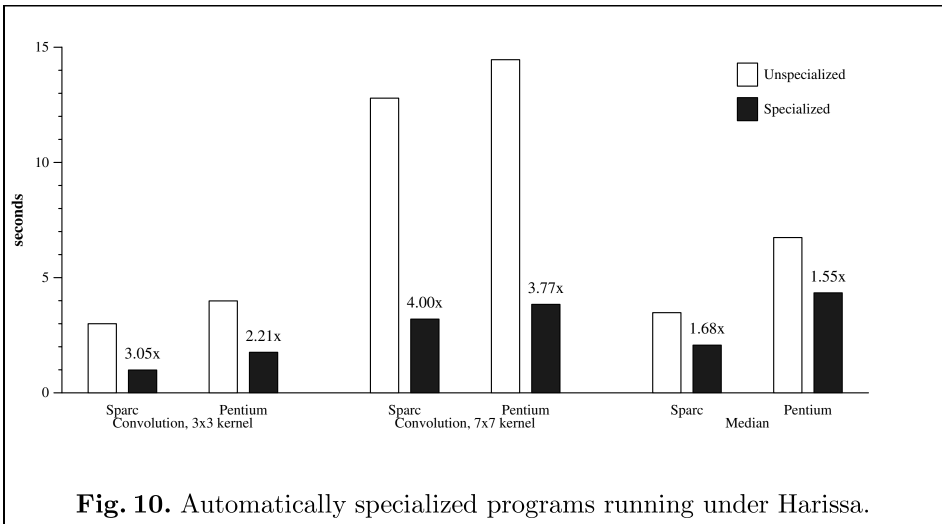


Fig. 10. Automatically specialized programs running under Harissa.

the index of the accessed element, the tests can be eliminated during specialization. Unfortunately, by definition of Java bytecode, these optimizations cannot be expressed at the bytecode level. Since we directly execute the specialized C code using the Harissa environment, our approach does not suffer from this problem. Concretely, the specialized program is as efficient as an equivalent C program, but with the safety of the original Java program, at the price of Java portability.

Example. After having specialized the `computePoint` method with respect to both values and types, a number of type casts remain for the pixels `acc` and `p`. Since the types of both pixels are static, these type casts can be checked and eliminated during specialization. This is not shown in Figure 9, since we cannot show this at the Java level.

4.4 Result of Specialization

The original image filtering program is essentially constructed from a collection of generic, interacting objects. Program specialization automatically adapts each object to its context, obtaining an implementation very similar to a hand-optimized version. In the case of our example, specialization produces an optimized implementation adapted to a specific filtering strategy.

Experiments with Harissa code. The performance of the specialized code is significantly better than that of the generic, unspecialized code. We have tested the automatically specialized code using the Harissa environment. Experiments were conducted both on a Sun workstation (300MHz Ultra SPARC) and on a

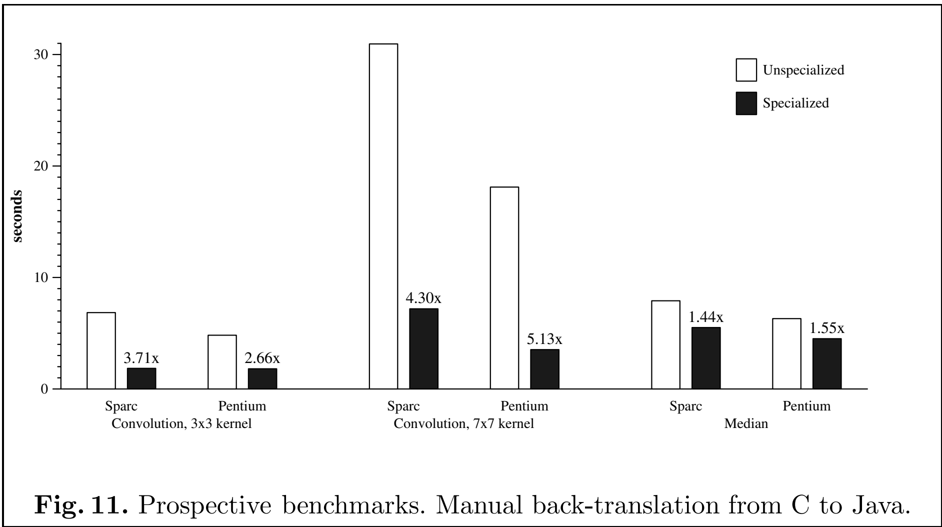


Fig. 11. Prospective benchmarks. Manual back-translation from C to Java.

Dell PC (200MHz Pentium Pro). The results, including both the execution time and the speedup of the specialized code, are shown in Figure 10.

We conducted experiments on a three-by-three blurring convolution filter, a seven-by-seven blurring convolution filter, and a median filter. The speedups are between 1.55 and 4.00, the highest speedup being obtained for the large convolution filter. The relatively minor speedup for the median filter is derived primarily from structural simplifications, the running time being dominated by the median computation. For the convolution filters, the code is simplified to such a degree that memory access becomes the major bottleneck.

Experiments with specialized Java source code. To estimate the gains that can be expected from a program specializer that outputs Java bytecode programs, we have manually translated the specialized C code into Java. The benefits due to virtual machine specific optimizations have been retained where possible, mimicking an optimized translation process rather than a direct, naive one. The tests were executed using JDK 1.2b4 (reference version) on the same machines as were used in the previous experiment. Figure 11 shows the result of our experiments (note that the scale has changed).

The speedups are between 1.44 times and 5.13 times, resembling the speedups of the automatically specialized programs. The speedups due to specialization are a bit higher than those that were observed using Harissa. Since the JIT executes programs more slowly than the Harissa environment, memory access is not as dominating a factor as with the Harissa tests, making the benefits of the code simplifications performed by the specializer more apparent.

5 Description of the Prototype

Rather than writing a program specializer from scratch, we have chosen to construct a prototype from existing tools, minimizing the amount of work needed to perform realistic initial experiments. The prototype is implemented as a staged process. First, the Java program and the specialization classes are processed to prepare for specialization and produce the corresponding run-time environment. Then, the Java code is translated into C. Finally, the C code is specialized using the Tempo specializer.

We have seen that taking the approach of translating Java programs into C allows aspects of the semantics of Java to be made explicit, which exposes specialization opportunities. Examples include virtual calls, casts, and array references (see Section 4). Thus, following our approach, the kinds of specialization that can be expressed using this approach are not limited by the syntax and semantics of Java, but by the expressiveness of C. However, there is a possible loss of precision when analyses are performed in C as opposed to Java. For example, alias analysis in Java need not consider incorrectly-typed aliases. Should this loss of precision pose a problem, we could perform the alias analysis before translation to C, and generate annotated C code specifically for Tempo.

5.1 Structure of the Prototype

The overall structure of the prototype is presented in Figure 12. The input to the specialization process is a Java program and a set of specialization classes. The Java Specialization Class Compiler (`JSCC`) [32] prepares the program for specialization. For each class to be specialized, it generates a “dispatcher” to select the appropriate (specialized or generic) implementation to invoke with respect to the execution context; this is called an enclosing object (X-enclosing.java). Additionally, `JSCC` produces stub methods to (native) specialized implementations of this object (X-specializable.java) that will be generated by Tempo in a later phase.

Next, the extended Java program is compiled into bytecode using `javac` and translated into C via the Harissa compiler.

Tempo takes the C-translated program together with the specialization parameters and generates the specialized versions. The C-translated program and the specialized code can either be translated back into Java, at the expense of losing some optimizations, or be run directly using the Harissa environment.[4] The latter option is the only one currently implemented. Note that Java programs can be specialized at both compile time and at run time.

5.2 Harissa

Harissa is an off-line compilation system that supports dynamic loading of bytecode. It compiles Java bytecode into C code that can be further compiled by

[4] Specialization simplifies the program without adding new code, so the specialized program is still compatible with the Harissa environment.

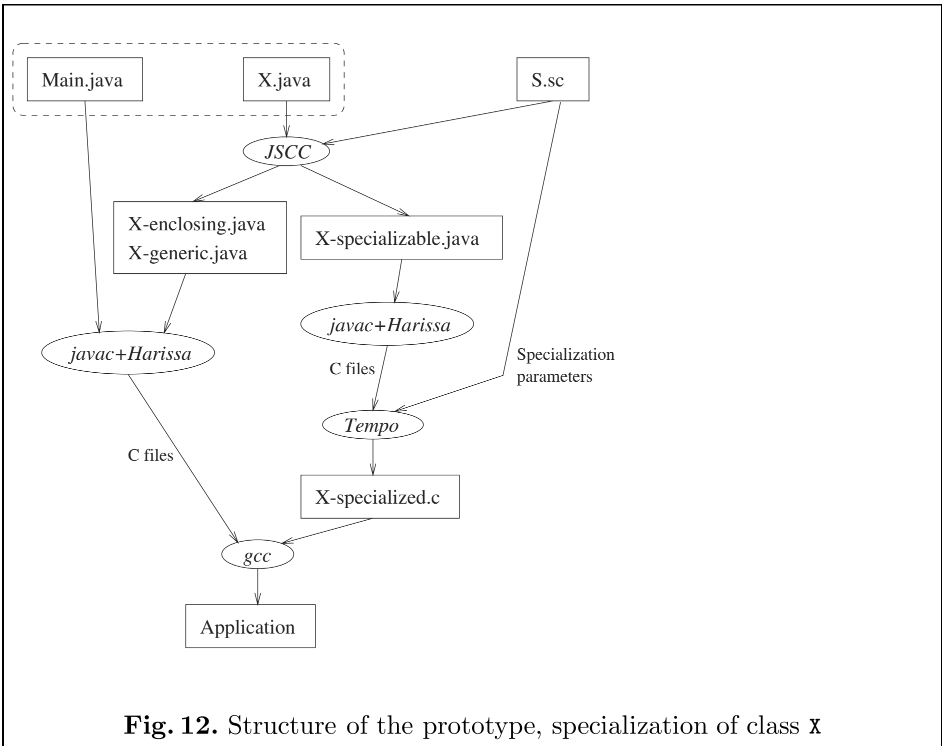


Fig. 12. Structure of the prototype, specialization of class `X`

standard C compilers. Harissa includes aggressive optimizations, such as method inlining, driven by a Class Hierarchy Analysis and an intra-procedural static class analysis [11]. These optimizations are complemented by the C compiler. Harissa's run-time system integrates a Java bytecode interpreter to execute dynamically loaded bytecode.

Harissa is designed to compile a Java program into C code while preserving as much of the original structure as possible. Preserving the structure of the program makes Harissa suitable for generating programs that are to be processed by a back-end phase such as Tempo. An object is represented using an ordinary C structure, and a virtual call is expressed as an indirect C function call through a pointer stored in a structure field, allowing it to be precisely treated by Tempo, and replaced by a direct call when the pointer is static. Harissa has been extended to generate extra information useful to Tempo. In particular, Harissa generates a context for the specialization containing information about those parts of the program that are not being specialized.

Currently, Harissa implements exceptions using the C `longjmp` function, which is ignored by Tempo. Specializing in the presence of exceptions can thus produce incorrect results. In the future, we plan to extend Tempo with explicit treatment of exceptions, extending the C input language to include a Java-like exception mechanism, and then to have Harissa generate C code with such exceptions.

5.3 Tempo

Tempo is an off-line program specializer for the C language [8], that supports both compile-time and run-time specialization. The analyses of Tempo are sufficient to effectively specialize a wide range of C programs. We found it necessary, however, to improve the precision of some of the analyses to better specialize Java programs.

The C code generated from a Java program is very different from human-written C programs. In particular, because objects are represented as C structures, structures are used more heavily in Harissa generated code than in ordinary C programs. Also, memory management in C is different from memory management in Java, where all objects are dynamically allocated. Because of this situation, we extended Tempo with a more precise treatment of structures.

Upward type casts are implicit in Java, and type casts between objects are common. In contrast, they rarely occur in C and are often considered as a bad programming style. To address this need, Tempo has been extended to handle type casts between structures. This capability is geared towards the programs generated by Harissa, in which the layout of structures is guaranteed to be compatible.

6 Related Work

To our knowledge, there have been only preliminary investigations of specialization of object-oriented programs. We apply the approach of translation to an intermediate language for which a specializer exists; this approach has been investigated earlier to implement program specialization for a simple imperative language. Also, optimizing compilers have addressed specialization of both data representation and control flow.

Specialization of object-oriented languages. Marquard and Steensgaard developed a partial evaluator for a small object-oriented language based on Emerald [22]. They focused primarily on implementation issues such as ensuring termination and the representation of unknown values during the specialization process. In contrast, we have investigated the applicability of program specialization to the object-oriented paradigm.

Khoo and Sundaresh investigate partial evaluation as a means for eliminating virtual calls in simple object-oriented language [17]. Their work focuses on formalizing the analysis and transformations realized by performing program specialization on programs with virtual calls.

The C++ language incorporates templates that allow a single class to be statically compiled to different types, in effect performing specialization. Templates can also be used to perform some computations at compile time, as demonstrated by Veldhuizen [31], albeit not on object types, and with a significant compilation overhead. Special syntax must be used to write programs so that they can be optimized using templates, and having both specialized and generic variants of a method requires writing two different implementations.

Specialization via translation. Moura has also investigated the approach of using translation to extend the applicability of an existing program specializer to new program constructs [23]. She designed an approach to specializing imperative programs by translation into a functional subset of Scheme, followed by specialization using an existing specializer for Scheme, named Schism [7].

Optimization of data representation. Object inlining is a technique for storing temporary objects on the stack [12]. An object that is used only locally in a method can be stack-allocated rather than heap-allocated, thus improving the performance of the program.

The control flow simplification performed by specialization enables further optimizations on data structures. Object inlining is an instance of such optimizations that we will consider in the future.

Optimization of control flow. The Vortex compiler [10] implements optimizations similar to those performed by program specialization. Vortex is based on static, global analyses, complemented by an automated profiling system. Profiling information guides aggressive optimizations. These optimizations eliminate virtual calls and specialize methods according to the types of their arguments. In fact, the optimizations offered by Vortex depend on the accuracy of its analyses and profiling system. As a result, the level of optimization is difficult to predict. By contrast, specialization is parameterized by information provided by the programmer (or component user).

Furthermore, Vortex's optimizations only propagate and exploit type information to remove virtual calls. Specialization goes beyond types, also propagating values. As a result, more optimizations can be performed with these values (*e.g.,* loop unrolling and array bounds checking).

With Vortex's approach, all compilation is performed before the program is run. Rather than considering compilation, execution, and profiling as separate phases, the execution environment can include all these tasks, and perform compilation as a continuous process. This approach allows aggressive optimizations similar to those employed by program specialization, since information in the form of usage patterns is available at run time [16]. However, the analyses that can be performed are inherently limited by the amount of available time and space. Also, checks must be retained to verify invariants that could have otherwise been detected by a specializer.

7 Future Work

We have provided an outline of how automatic program specialization of an object-oriented language such as Java can be achieved. Program specialization of Java as presented in this paper has many applications and raises many issues. This section outlines possible extensions to our work.

Software components. A trend in software engineering is the development of systems from software components. This emerging software architecture is illustrated by Java Beans and its rapidly growing selection of components. This trend stresses the need for genericity to address a class of solutions by a specific software component. To overcome the expected performance penalty, specialization will likely become a critical tool.

In this context, our next step aims at developing a methodology for designing and developing highly-generic components, that, once integrated, specialize in a predictable way into efficient implementations.

Java as target language. For portability, it can be beneficial to produce specialized code in Java. One option is to translate the specialized C program back into Java. Nevertheless, such an approach would eliminate some optimizations, mainly those related to the virtual machine (see Section 4.3).

Translation back into Java is possible if the C specialized code has enough information and structure to enable Java constructions to be recovered. Given our specialization process, this depends on the translations performed by the Harissa compiler and the transformations done by Tempo. Motivated by the encouraging benchmark results in Section 4.4, future work includes the development of such a back translator.

Run-time specialization. Although the current implementation of the specialization class compiler does not support run-time specialization, Tempo does support specialization at both compile-time and run-time. For run-time specialization, executable code is assembled directly from binary templates generated by a C compiler. This approach can be used directly in the Harissa environment. Specializing at run time has the obvious advantage of exploiting values that are only available when running the program.

8 Conclusion

Component-based software technology is a growing trend in software development. It makes genericity a central issue. In this paper, we have demonstrated that specialization is a key tool to overcome the performance penalty incurred by genericity.

Specialization exploits global information available when software components are integrated into an application. Performing transformations such as virtual-call elimination, inlining, constant propagation and constant folding turns a modular component-based application into a monolithic optimized program.

We have developed a specializer for Java based on existing tools, namely, a Java-to-C compiler (Harissa) and a C specializer (Tempo). We have used it to specialize an image-filtering application. This application is structured in a modular way to support a variety of data representations and image treatments.

Specialization has been shown to eliminate the overhead incurred by this structuring strategy. In practice, the specialized program is up to four times as fast as the original one.

Several lessons can be drawn from this work.

- Specialization of modular Java programs can drastically improve performance. Besides the usual optimizations offered by imperative specializers, we have found that object-oriented programs offer other opportunities for improvement, opportunities traditionally studied as advanced optimizing compilation techniques.
- Re-using existing tools to develop our Java specializer has proved successful in terms of development time. This result has been obtained largely without compromising the quality of the specialized program.
- Because our approach involves specializing C programs, it enables a class of optimizations that is out of reach of ordinary Java source-to-bytecode compilers.

We believe that a specializer is a key tool for object-oriented software development environments. In particular, in the context of Java Beans, a specializer should allow modular applications to be mapped into efficient implementations.

Availability. Tempo and the Java Specialization Class Compiler are freely available. Harissa is available under the GNU Public license (with full source code). More information can be found at the Compose home page:
`http://www.irisa.fr/compose`

Acknowledgements. Thanks to the anonymous referees for their valuable comments, to Renaud Marlet for helping to streamline the paper, and to Aino Rasmussen for her encouraging comments.

References

1. L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.
2. J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In PLDI'96 [27], pages 149–159.
3. R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Orlando, FL, USA, June 1994. Technical Report 94/9, University of Melbourne, Australia.
4. A.A. Berlin. Partial evaluation applied to numerical computation. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, Nice, France, 1990. ACM Press.
5. B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

6. W.H. Cheung and A. Loong. Exploring issues of operating systems structuring: from microkernel to extensible systems. *ACM Operating Systems Reviews*, 29(4):4–16, October 1995.
7. C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.
8. C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.
9. C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the* 23rd *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996. ACM Press.
10. J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: an optimizing compiler for object-oriented languages. In *OOPSLA' 96 Conference*, pages 93–100, San Jose (CA), October 1996.
11. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.
12. J. Dolby and A. A. Chien. An evaluation of automatic object inline allocation techniques. In *Proceedings OOPSLA '98 Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices. ACM, 1998.
13. D.R. Engler and M.F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 26–30, Stanford University, CA, August 1996. ACM Press.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1994.
15. B. Guenter, T.B. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.
16. Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 326–336, New York, NY, USA, June 1994. ACM Press.
17. S. C. Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 211–222, Yale University, 17–19 June 1991.
18. P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. In U. Meyer and G. Snelting, editors, *Workshop Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*, pages 45–54. Justus-Liebig-Universität, Giessen, Germany, 1994. Report No. 9402.
19. P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI'96 [27], pages 137–148.
20. B.N. Locanthi. Fast bitblt() with asm() and cpp. In *European UNIX Systems User Group Conference Proceedings*, pages 243–259, AT&T Bell Laboratories, Murray Hill, September 1987. EUUG.
21. R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering*, pages 183–192, Lake Tahoe, Nevada, November 1997. IEEE Computer Society.

22. M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, University of Copenhagen, Department of Computer Science, Universitetsparken 1, 2100 Copenhagen O., Denmark, April 1992.
23. B. Moura. *Bridging the Gap between Functional and Imperative Languages*. PhD thesis, University of Rennes I, April 1997.
24. G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.
25. G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Portland (Oregon), USA, June 1997. Usenix.
26. G. Muller and U. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, pages 44–51, March 1999.
27. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996. ACM SIGPLAN Notices, 31(5).
28. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.
29. C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.
30. J.C. Russ. *The Image Processing Handbook*. CRC Press, Inc., second edition, 1995.
31. T. L. Veldhuizen. C++ templates as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18, San Antonio, TX, USA, January 1999. ACM Press.
32. E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, USA, October 1997. ACM Press.

Wide Classes

Manuel Serrano

Université de Nice Sophia-Antipolis
650, route des Colles, B.P. 145 F-06903 Sophia-Antipolis, CEDEX
Manuel.Serrano@unice.fr
http://kaolin.unice.fr/~serrano

Abstract. This paper introduces the concepts of *wide classes* and *widening* as extensions to the object model of class-based languages such as Java and Smalltalk. Widening allows an object to be temporarily *widened*, that is transformed into an instance of a subclass, a *wide class*, and, later on, to be *shrunk*, that is reshaped to its original class. Wide classes share the main properties of plain classes: they have a name, a superclass, they may be instantiated, they have an associated class predicate and an associated type that may be used to override function definitions.
Widening is also useful to implement transient data storage for long-lasting computations. In particular, it helps reducing *software data retention*. This phenomenon arises when the actual data structures used in a program fail to reflect time-dependent properties of values and can cause excessive memory consumption during the execution.
Wide classes may be implemented for any dynamically-typed class-based programming language with very few modifications to the existing runtime system. We describe the simple and efficient implementation strategy used in the Bigloo runtime system.
Keywords: Language implementation, dynamic inheritance, dynamic type checking, instance modification.

1 Introduction

Few object-oriented programming languages permit objects to change their nature during execution. If an object `obj` is an instance of a class $\mathcal{C}$ at some point of the execution, then `obj` remains an instance of $\mathcal{C}$ for the whole computation. In other words, the *type* of an object may not evolve over time. Most Object-oriented languages only provide a mechanism to associate behaviors to types by the means of *methods*. In consequence, object-oriented programming languages do not allow the behavior to evolve over time. For instance, consider window objects in an object-oriented window manager. We may imagine several kind of windows such as windows with a menu bar and windows without menu bar. This can be easily implemented in a class-based object-oriented language. We may build a bare window class and a subclass for windows with menu bar. This is easy because if a window has a menu bar, it will keep it for its entire life time. The trouble comes when one wants to express that a window is either visible or

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 391–415, 1999.
© Springer-Verlag Berlin Heidelberg 1999

iconified. A window may be visible at some point and iconified at another point. The property of being visible or iconified is dynamic; it changes over time. In consequence, the behavior associated to the window should also change over time. For instance, refreshing a visible window or refreshing an iconified window requires different operations. In general, object-oriented languages do not provide mechanisms that allow a straightforward representation of this phenomenon.

Wide classes extend the standard object-oriented programming model with a disciplined type conversion operation called *widening*. This allows an object to be temporarily *widened* (that is transformed into an object of a subclass, a *wide class*) and then to be *shrunk* (that is reshaped to its original class). Widening is a runtime operation. During the execution, a program explicitly widens and shrinks objects. Widening and shrinking are tools of choice to implement transient states. In our window manager example, we may widen a visible window to an instance of the iconified window class when the program requests iconification and we may shrink the iconified window to a plain visible window when the program requests de-iconification. Widening and shrinking change the type of the window from visible to iconified, so we may write methods that will apply to visible windows and methods that will apply to iconified windows. Furthermore widening may add data attributes needed by the new behavior while shrinking sheds data that is not needed anymore.

Both Smalltalk and Clos propose a construction that allows unrestricted changes to the runtime type of objects. These type-change operations may turn any object into an instance of any class. Widening is more disciplined as type-changes are restricted to (wide) subclasses. In other words, widening preserves subtyping relationships (after widening the new runtime type of an object is a subtype of its former runtime type). This limitation is important for program correctness and essential for obtaining an efficient implementation of wide classes. We show in this paper that wide classes have a simple and fast implementation with two important properties:

- Wide classes have no impact on the implementation of the rest of the runtime system.
- A program that does make use of widening does not pay any extra cost for that feature.

Neither of these properties hold for the equivalent mechanisms offered in Smalltalk and Clos.

From a programmer's point of view, widening is very useful to implement transient data storage and thus helps facing the problem of *software data retention*.

1.1 Software Data Retention

During the execution of a program, it might happen that some data items have a limited life time but that the actual data structure in which they are contained is too rigid and will waste memory because it will retain space longer than

necessary. Softwares that use a *pipelined* architecture, that is softwares that cascade several distinct stages, must frequently face these retention problems.

As an example, let's suppose that in a compiler one would like to introduce a stage that removes useless variables. For the purpose of that stage, the number of occurrences of each variable is needed and we suppose that this information is never needed elsewhere. The problem is "where to store the number of occurrences in the abstract syntax tree". A data structure allocating an extra field holding the number of occurrences for each node that implements variables wastes memory as soon as the stage is completed. (Of course, many stages of a compiler use more than one temporary information.) We say that a program making use of such data structure faces *data retention*.

1.2 Fighting Software Data Retention

Current programming languages do not propose mechanisms to reduce software data retention and thus programmers are used to one of the following *ad-hoc* strategies.

1. *Global data*: this strategy consists in allocating a global data structure that provides room for all the data of all the stages of the execution. This solution is inefficient in space because the structure is much larger than the actual data that it contains. Many fields of the data structure will be empty at any given time. Moreover with a programming language that enables automatic memory management, the program must take provisions to explicitly free the unused fields; otherwise the algorithm to reclaim memory will fail at re-using the memory that has been allocated for a completed stage. In other words, the program must explicitly free the dead data structures; this greatly minimizes the pros of automatic memory management.
   Further, the *global data* approach fails to enforce strict barriers between stages, and since the information from the previous stages is still available there is a temptation of using it. This is in contradiction to the principles of clear separation between stages of the pipelined architecture.
2. *Copy on entrance*: when entering a new stage the global data structure is copied into a new one that provides rooms for the local data that belongs to that stage. This solution is safe to avoid confusion between distinct stages. It is efficient in space because the size of the largest structure live at a time is the size of the largest structure allocated by each stage. Unfortunately, this framework is inefficient in execution time as it implies a large amount of memory allocation/deallocation. Thus, it is likely that the time spent in the memory manager will be significant. Moreover the time required to copy the structure is important because sharing properties must be preserved from one copy to another.
3. *User field:* with this strategy, a so-called *user field* is added to each structure of the global data type and then each stage hooks its own local structures using this extra field. This solution is frequently used: for example, the C structure `fieldnode` of the `form` library declares a field `usrptr` that can be

used by the clients of that library. If this technique has the advantage of being space efficient (only one word is wasted by structure to hold the extra field) it has two severe disadvantages:

- It forfeits static type checking. The type of the user field has to be the type of the most generic user type (e.g., `void *` in C). As a consequence, the *user field* framework leads to a very common error. The stage S_i reads the user field that is still holding a local data of the stage S_{i-1}. Then an error (or an exception for languages with higher level runtime support) is raised and the execution stops.
- More pragmatically, it is simply inconvenient to use, as each access to the user slot requires an extra indirection (and possibly an extra coercion for statically type checked languages). This all makes the code dealing with user fields unnecessarily verbose.

Wide classes and widening represent a solution to software data retention. Objects may be widened to hold transient data and shrunk when that data appears to be useless. Wide classes use inheritance. That is, widening affects the dynamic binding of methods. Widening is used extensively in the Bigloo compiler for the Scheme programming language. Bigloo compilation passes are implemented using the *widening/shrinking* mechanism. Upon entrance to a pass, objects are widened with pass-specific fields and, on pass exit, they are shrunk back in order to forget the information related to that pass. Our experience has been that using widening has greatly simplified the organization of the code of the compiler; from a software engineering point of view this has been beneficial, and from an efficiency point of view the current release of the compiler uses memory more sparingly than its predecessors.

1.3 Organization

Section 2 presents a general overview of the Bigloo object model. Readers familiar with the Clos-like style may skip that section. Section 3 introduces the design of wide classes. The new language constructions are presented here. Examples of small programs using widening are given. This section concludes with a discussion of the limitations of wide classes. Section 4 presents widening used in the context of a large program, our Scheme compiler, Bigloo. Section 5 discusses the implementation of wide classes and widening operations. Section 6 compares wide classes with other object oriented features. Section 7 presents some possible extensions to widening and wide classes.

2 Bigloo

Bigloo is an open implementation of the Scheme language. From the beginning, the aim was to propose a realistic and pragmatic alternative to the strict Scheme implementation defined by [7]. Bigloo does not implement "all" of Scheme: for example, the execution of tail-recursions may allocate memory and very few

arithmetics are implemented (Bigloo only supports integers and reals). However, Bigloo implements numerous extensions: support for lexical and syntactic analysis, pattern matching, an exception mechanism, a foreign interface and a module language. The recent Bigloo versions offers a new extension with an object layer, to a great extent inspired by MEROON [8] of C. Queinnec.

The term "object-oriented" is a rather loose fitting description that has been prefixed to a number of very different languages and systems over the years. We will restrict our attention to two object models: the Smalltalk model [5] where methods are associated to classes and the Clos model [1] where methods are associated to generic functions. Each of those models has many incarnations: with static or dynamic type checking, with single or multiple inheritance, and single or multiple dispatch. In this paper we assume a Clos-like object model with single inheritance and single dispatch.

To introduce the Clos object model, let us present a small example. Consider a class *point* implementing 2 dimensional points and a class *point-3d* implementing 3-d points and suppose that we want to implement a function that computes the norm of these points. Instead of placing methods in the declaration of *point* and *point-3d* that implement this functionality as we would do in Smalltalk or Java, we define a *generic function* that will apply to the *point* and *point-3d* classes:

```
(define-generic (norm::real p::point))
```

In a first step a generic function may be considered as a declaration. No user code is associated to it. A generic function is a placeholder for method definitions. The above declaration of `norm` specifies that methods called `norm` can be defined, taking exactly one argument which is a subtype of *point*. Methods are said to *override* the generic function.

```
(define-method (norm::real p::point)
   (sqrt (sqr (point-x p)) (sqr (point-y p))))

(define-method (norm::real p::point-3d)
   (sqrt (sqr (point-3d-x p)) (sqr (point-3d-y p)) (sqr (point-3d-z p))))
```

The functions `point-x`, `point-y`, `point-3d-x`, ... are accessors, automatically generated by the compiler. Methods can be used as:

```
(let ((p   (instantiate::point (x 3.3) (y 4.1)))
      (p3d (instantiate::point (x 8.9) (y 4) (z 2))))
   (print (+ (norm p) (norm p3d))))
```

The Clos model fits well with the traditional functional programming style. A function can be thought as a generic function overridden with exactly one method. Another reason to choose the Clos model is the nature and the architecture of the Bigloo compiler that we wanted to rewrite with objects.

Former versions of the Bigloo compiler were written in plain Scheme. The compiler is a program of 40,000 code lines. It reads a program to be compiled and builds the abstract syntax tree used to represent the program. This tree contains a structure of 23 different node types. There is a node for constants, variable

assignments, conditionals, function calls, etc. The compiler is made up of stages, each of which can be seen as a process that modifies the abstract syntax tree. The driver is a Scheme function that looks like the following one:

```
(define (compiler src)
   (let ((ast (build-ast src)))
      (macro-expand! ast)        ;; 1st stage
      (function-inline! ast)     ;; 2nd stage
      ...
      (code-generate! ast)))     ;; 20th stage
```

Each stage is implemented in a single file, except for the most complex ones that can be split over various files. For the new releases making use of object-oriented programming, we wanted to keep this decomposition as we felt it was the most natural way to write pipelined systems.

This encouraged us to adopt a generic function object model rather than the traditional Smalltalk model as this last model would have forced us to give up the stage-driven structure in favor of a structure driven by the abstract syntax tree. Each compiler stage would have been split in all classes implementing the tree nodes. A file, implementing the `function-inline!` stage for example, would not exist anymore and its code would have been split among classes.

Adding new nodes to the syntax tree, however, would have been easier for our compiler with the Smalltalk model. Using this model, adding a stage requires to modify and to recompile all the existing code (because we need to add the methods defining the new stage for each class). But, adding a node does not require changing the existing code. Adding a node means that we are adding a feature to the target language (which is uncommon), while adding a stage usually means that we are adding a compiler optimization (very common for compiler writers). Our experience has shown that it is important to ease the task of writing new stages as these have a local impact, while language additions are difficult anyway because they may impact many features.

2.1 Class Declarations

Classes can be declared static or exported. It is then possible to make a declaration accessible from another module or to limit its scope to one module. The abbreviated syntax of a class declaration is:

```
(class class-id::super-class-id <field>*)

<field> ::= field-id::type-id
  | (field-id::type-id <option>*)
  | (* field-id::type-id <option>*)

<option> ::= read-only
  | (default value)
```

A class can inherit from a single super class. Classes with no specified super class inherit from the `object` class. The type associated with a subclass is a subtype of the type of the super class.

A field is typed (with the annotation `::type-id`), might be immutable (`read-only`), and may have a default value (`default`). Fields with clauses that begin with `*` are said to be indexed: those fields do not contain a single value but a sequence of values. Indexed fields (directly inspired by Meroon) can be used, for example, to implement strings of characters. Here are possible declarations for the traditionals `point` and `point-3d`:

```
(module small-points
   (export (class point
              (x::double read-only (default 0.0))
              (y::double read-only (default 0.0)))
           (class point-3d::point
              (z::double read-only (default 0.0)))))
```

2.2 Generic Functions

Generic function declarations are function declarations annotated by the `generic` keyword. They can be exported: this means that they can be used from within other modules and that methods can be added to those functions from other modules. They can be static, that is not accessible from within other modules. In this case, no method can be added to the ones introduced by the module that defines the generic function. The syntax to define a generic function is very similar to an usual function definition:

```
(define-generic (fun-id::type-id arg-id::class-id ...) optional-body)
```

Generic functions should have at least one argument as it is the first argument that is used to solve the *dynamic dispatch* of methods. This argument is of type T that must be associated to a class. It is impossible to override generic functions with methods whose first argument is not of a subtype of T. Generic functions can have a body. This body will be evaluated if, when calling a generic function, no method is defined for the type of the first argument. If this situation occurs and the generic function has no body, an error is raised. Here is an example of a generic function definition whose distinguishing argument must be a subtype of `point`:

```
(define-generic (display-point::obj p::point)
   (print "<???>"))
```

2.3 Methods

Methods are declared by the following syntactic form:

```
(define-method (fun-id::type-id arg-id::class-id ...) body)
```

Defining a method if there is no defined generic function with a compatible prototype (arguments and result types in a sub-typing relation with those of the generic function) is an error. A method can be written as follows:

```
(define-method (display-point::obj p::point)
   (with-access::point p (x y)
      (print "<point:" x ", " y ">")))
```

Methods override generic function definitions. When executing the following code:

```
(let ((p (instantiate::point)))
   (display-point p))
```

because there is a method defined over the class *point* the code of that method will be executed instead of the default code of the generic function.

2.4 Instances

When declaring a class *cla*, Bigloo automatically generates the predicate `cla?`, an allocator `instantiate::cla`, a cloner `duplicate::cla`, accessors (e.g., `cla-x` for a field `x`), modifiers (e.g., `cla-x-set!` for a field `x`), and, an abbreviated access form, `with-access::cla`, to allow accessing and writing fields by referencing them by their name. Here is how to allocate and access an instance:

```
(let ((p (instantiate::point-3d (y -3.4) (x 1.0))))
   ;; The initialization value of a field can be omitted from the arguments list if the
   ;; field has a default value; this is the case for the field z of class point-3d.
   (with-access::point-3d p (x y z)
      (sqrt (+ (sqr x) (sqr y) (sqr z)))))
```

3 Wide Classes

The key point to *wide classes* is that objects may become (may be widened to) instances of wide classes and thus that the type of an object may change dynamically over the execution.

3.1 Wide Class Declarations

The declarations of *wide classes* are similar to the declarations of plain classes:

```
(wide-class class-id::super-class-id <field>*)
```

The super class of a wide class may be a plain class or a wide class. By contrast the super class of a plain class cannot be a wide class. This point will be discussed in Section 3.6. As an example, here are the declarations of “named points” and “named 3-d points”.

```
(module named-point
   (export (wide-class named-point::point
              name::string)
           (wide-class named-point-3d::named-point
              (z::double (default 0.0)))))
```

3.2 Wide Instances

Wide objects (i.e. objects that are instances of wide classes) may be used in the exact same way as plain objects. Wide objects are instantiated from wide classes. Wide objects are used to solve generic function run time dispatches and fields of wide objects are fetched from the instances. The types of the wide objects are checked by automatically generated class predicates. Thus, we may write:

```
(define-method (display-point::obj p::named-point)
   (with-access::named-point p (x y name)
      (print "<point(" name "): x:" x ", y:" y ">")))

(define (display-instance x)
   (if (point? x)
       (display-point x)
       (display x)))

(let ((p (instantiate::named-point (name "org") (x 0) (y 0))))
   (display-instance x))
```

3.3 Widening

The new feature introduced by wide classes is that during its life-time, an instance of a given class may be *widened* to an instance of one of its wide subclasses. Later on, the same instance may be *shrunk*, that is reshaped to its original class. The mandatory point here is that the widening and shrinking operations do not introduce copies. A widened instance stays the same, being merely extended with new slots. In addition, if an object `obj` satisfies a class predicate $\mathcal{P}$, then after widening, `obj` *still satisfies* $\mathcal{P}$.

The syntax of the widening operator resembles the syntax of the instance allocator:

```
(widen!::class-id instance (field-id0 val0) (field-id1 val1) ...)
```

Class-id is the name of a wide class, the name of the wide class the *instance* is widened to. It is an error if *class-id* is the class identifier of a plain class. The *field-id*$_i$ identifiers are the name of the fields of the wide class *class-id*. The values *val*$_i$ are initial expressions for the fields of the wide class. As for object instantiation, fields declared with default values may be omitted within a `widen!` expression. Here is an example of widening from the class *point* to the class *named-point*.

```
(let ((pt (instantiate::point (x 0.0) (y 5.3))))
   (widen!::named-point pt (name "dummy"))
   (display-point pt))
```

Instantiation: Instantiating a wide object is semantically equivalent to creating a plain object and then applying successive widening until the desired wide class is obtained. That is

```
(instantiate::named-point-3d (x 0) (y 0) (z 0) (name "gee"))
```

is a shorten for:

```
(let ((p (instantiate::point (x 0) (y 0))))
   (widen!::named-point p (name "gee"))
   (widen!::named-point-3d p (z 0))
   p)
```

Identity: The semantics of widening preserves object identity, in Scheme terminology this means that a widened instance is still `eq?` to what it was before widening. Thus,

```
(let* ((p     (instantiate::point))
       (new-p (widen!::named-point p (name "new"))))
   (eq? new-p p))
```

evaluates to true. This feature is very important because it allows some parts of a whole data structure to be widened while preserving sharing relationships. We have tried hard to design widening so as to limit the runtime cost. Section 5 presents an implementation strategy which allows us to get widening at no extra implementation cost.

3.4 Shrinking

Shrinking is simpler. Instances get shrunk one class at a time by the means of the form:

```
(shrink! instance)
```

It shrinks an *instance* by one level from its widest level. Thus,

```
(let ((p (intantiate::point)))
   (widen!::named-point p (name "dummy2"))
   (widen!::named-point-3d p (z 3.0))
   (display-point p)
   (shrink! p)
   (display-point p)
   (shrink! p)
   (display-point p))
```

when executed, prints out:

```
<point(dummy2): x: 0.0 y: 0.0 z: 3.0>
<point(dummy2): x: 0.0 y: 0.0>
<point: x: 0.0 y: 0.0>
```

It is an error to shrink an object that is not a wide object. Wide objects may be tested with the predicate:

```
(wide-object? obj)
```

As an example, here is the code of a polymorphic function that takes any object and returns the same object with all wide layers removed (if there were any):

```
(define (shrink-all! obj)
   (if (wide-object? obj)
       (begin (shrink! obj) (shrink-all! obj))
       obj))
```

In presence of multiple widening, it could be useful to specify a destination class when shrinking. The `shrink!` operator accepts an optional argument that denotes that class. The syntax of this form is:

```
(shrink!::class-id instance)
```

Shrinking with a destination class enables several shrink operations to be automatically performed. Its implementation is close to `shrink-all!`. A restriction exists for *`class-id`*. It must denote a super class of the actual class of *instance* that is a wide class or the first plain class that *instance* belongs to. The following example is a legal shrink:

```
(let ((p (intantiate::named-point-3d (name "dummy2") (z 3.0))))
   (shrink!::point p)
   (display-point p))
```

3.5 Example

Widening may be used to implement transient object properties. For instance, suppose we would like to model marital status. The class implementing persons would be:

```
(class person
   name::string
   fname::string
   (sex::symbol read-only))
```

As we can see, people are allowed to change their name but not their sex. The identity of a person can be printed as follows:

```
(define-method (object-display p::person)
   (with-access::person p (name fname sex)
      (print "firstname : " fname)
      (print "name      : " name)
      (print "sex       : " sex)
      p))
```

Married men and women are implemented by the means of two wide classes:

```
(wide-class married-man::person
   mate::person)
(wide-class married-woman::person
   maiden-name::string
   mate::person)
```

A married woman's identity is printed by the following method (we suppose an equivalent method definition for married-man):

```
(define-method (object-display p::married-woman)
  (with-access::married-woman p (name fname sex mate)
     (call-next-method)
     (print "married to: " (person-fname mate) " " (person-name mate))
     p))
```

The form `(call-next-method)` invokes the method that would have been used if no method for *married-woman* was overriding the generic function. Here, the `(call-next-method)` expression will act as a call to the method `object-display` defined over the *person* class. A wedding is celebrated using the `get-married!` function:

```
(define (get-married! woman::person man::person)
   (if (not (and (eq? (person-sex woman) 'female)
                 (eq? (person-sex man) 'male)))
       (error "get-married" "Illegal wedding" (cons woman man))
       (let* ((mname (person-name woman))
              (wife  (widen!::married-woman woman
                        (maiden-name mname)
                        (mate man))))
          (person-name-set! wife (person-name man))
          (widen!::married-man man (mate woman)))))
```

We can check if two persons are married by:

```
(define (married? woman::person man::person)
   (and (married-woman? woman)
        (married-man? man)
        (eq? (married-woman-mate woman) man)
        (eq? (married-man-mate man) woman)))
```

We may now study what happens during a wedding:

```
(define *junior* (instantiate::person
                    (name "Jones") (fname "Junior") (sex 'male)))
(define *pamela* (instantiate::person
                    (name "Smith") (fname "Pamela") (sex 'female)))

(define *old-boy-junior* *junior*)
(define *old-girl-pamela* *pamela*)
(get-married! *pamela* *junior*)
```

We may check the wedding:

```
(married? *pamela* *junior*)
   ⇒ #t
(display-object *pamela*)
   ⇒ name       : Jones
      firstname  : Pamela
      sex        : FEMALE
      married to : Junior Jones
```

Note that both `*pamela*` and `*junior*` are still the same persons:

```
(eq? *old-boy-junior* *junior*)  ⇒ #t
(eq? *old-girl-pamela* *pamela*) ⇒ #t
```

Finally, divorce is implemented as:

```
(define (divorce! woman::person man::person)
   (if (not (couple? woman man))
       (error "divorce!" "Illegal divorce" (cons woman man))
       (let ((mname (married-woman-maiden-name woman)))
          (begin
             (shrink! man)
             (shrink! woman)
             (person-name-set! woman mname))
```

And thus:

```
(divorce! *pamela* *junior*)
(display-object *pamela*)
   → name       : Smith
     firstname  : Pamela
     sex        : FEMALE
(eq? *old-boy-junior* *junior*)  ⇒ #t
(eq? *old-girl-pamela* *pamela*) ⇒ #t
```

3.6 Restrictions

We present here several rules that restrict widening and class declarations. We conclude with a description of the features a language must provide to be able to host wide classes.

Widening and Plain Classes: An object cannot be widened to a plain class. This is not possible because as described in Section 5 accessing fields of wide classes requires different operations than accessing fields of plain classes. We enforce this separation between plain and wide classes for efficiency reasons: we want to be able to generate efficient code for programs which do not use wide classes; this would not be the case if arbitrary widening along the inheritance tree would be allowed.

In consequence, wide classes may inherit from plain classes or from wide classes. But plain classes cannot inherit from wide classes. There is no technical reason for that. The only reason is that it is useless to declare a class that inherits from a wide class because no object may be widened to a plain class!

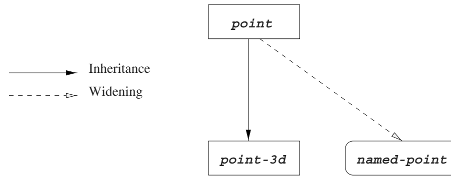
Single Subtyping and Widening: As we mentioned in Section 2, the object model of the Bigloo's source language is based on single subtyping. That is a class has one single superclass. To be consistent with this choice we have chosen to restrict widening accordingly. Thus, an object can only be widened to a wide class that directly inherits from the current class of the object. In other terms,

it is illegal to widen an instance whose actual class is $\mathcal{C}_1$ into an instance of class $\mathcal{C}_2$ if $\mathcal{C}_2$ does not inherit from $\mathcal{C}_1$. Thus

```
(module named-point-3d
   (export
    (class point ...)
    (class point-3d::point ...)
    (wide-class named-point::point ...)))

(let ((p (intantiate::point-3d)))
   (widen!::named-point p ...))
```

is illegal because *named-point* does not inherit from *point-3d* (the actual class of p). If simultaneous widening were permitted for different branches of inheritance, then we could build a tree like the following one:

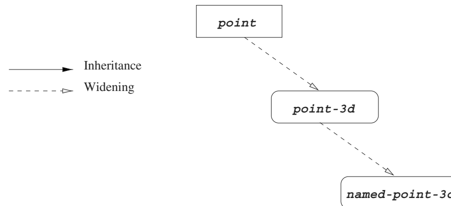


This obviously breaks the single subtyping rule. An object is simultaneously a *point-3d* and a *named-point* and neither the rule *point-3d* < *named-point* nor *point-3d* > *named-point* stands. On the other hand:

```
(module named-point-3d
   (export
    (class point ...)
    (wide-class point-3d::point ...)
    (wide-class named-point-3d::point-3d ...)))

(let ((p (intantiate::point)))
   (widen!::point-3d p ...)
   (widen!::named-point-3d p ...))
```

is legal because *point-3d* inherits from *point* (the actual class of p) and because at the time of the widening to *named-point-3d*, p is an instance of *point-3d* and *named-point-3d* inherits from *point-3d*. The restriction that applies to widening is justified to avoid multiple simultaneous widening. A plain object may be widened only once at a time. It builds the following inheritance tree:



which preserves single subtyping property.

Type Checking: Widening may be implemented for any language that supports dynamic type checking. Languages belonging to the Lisp family such as Scheme or Smalltalk fulfill that criteria. For statically typed languages wide classes cannot be used as such. The problem comes from the nature of shrinking. Shrinking, like widening, preserves the sharing structure of programs (that is, it respects `eq?`). So for example, consider the case where `obj` is shrunk from class `B` to `A`. The semantics of wide classes and shrinking guarantees that `B` is a subtype of `A`. The problem is that if any other object in the system keeps a reference to `obj` as an instance of `B`, then there is a risk of accessing fields that have been erased. That situation is very frequent for languages using "`self`" pseudo-variables. Widening is able to change the type of "`self`"! The only solution is to enforce a dynamic type test when accessing *any* wide object. This solution could be implemented within Eiffel or Java runtime system because they provide enough run-time type informations. Obviously this solution is equivalent to (partially) giving up on static type checking.

Concurrent Programming: Because they operate physical updates, it is mandatory that in presence of multi-threading, widening and shrinking be atomic operations. Otherwise, it might be possible that a thread accesses the field of one wide object before that field is created. Such a situation may arise when two threads share an object and that one of the threads accesses the object while the other thread is widening it.

3.7 Possible Extensions

It may be useful to propose a `shrink-then-widen!` operator. This would first shrink a wide object to the lowest common ancestor class of the current and desired wide class, then widen. This new form would allow wide classes to be defined as subclasses of an abstract class $\mathcal{C}$ because the `shrink-then-widen!` operator eliminates the need for explicit constructions of instances of the class $\mathcal{C}$.

4 Widening in a Real Context

As we mentioned in section 2 the compiler for the Scheme programming language Bigloo has been re-implemented with wide classes. Bigloo tries hard at optimizing the source code and thus contains many different analyses and optimization phases. For instance, Bigloo *inlines* user functions, reduces the number of instructions executed with data flow analyses such as *common sub-expression elimination* and *copy propagation*, minimizes the amount of memory needed to execute the program (*closure analysis* that improves the compilation of first class closures, *control flow analysis* that isolates polymorphic part of the programs that can be compiled has monomorphic programs), etc. All these analyses and optimizations are implemented in separate stages; in total there are twenty stages, that comes one after the other.

The whole compilation uses one unique global abstract syntax tree (AST henceforth). This data structure contains the minimal common information required by every stage. For instance, during the whole compilation nodes that implement variables are required to have a field holding the name of the variable. The AST changes from one stage to the next. Some nodes are removed while new nodes are introduced (for instance, *dead code elimination* optimization obviously prunes the AST and *loop unrolling* introduces code duplications). Nevertheless, the global structure of the AST barely changes during the compilation. Each compilation stage accepts as input the same data structure.

The only communication channel between stages is the AST. This strict policy makes each stage acts as a stand alone program. This property is very useful for a compiler because it helps the debugging process, it makes possible to change the order in which the stages are applied, and multiple applications of one stage (for instance, *copy propagation* enlarges the set of candidates to *common sub-expression elimination* and *common sub-expression elimination* enlarges the number of candidates to *copy propagation*: thus it is a convenient solution to apply the *copy propagation*, the *common sub-expression elimination* and then to re-apply the *copy propagation*).

Despite the global sharing of the AST some stages need local informations. For instance, the stage that makes the closures allocation (the stage that turns Scheme closures into C functions) needs to access the closure free variables. Once closures are allocated there are no more free variables. Freeness property is thus strictly local to that stage. Another example is the control flow analysis stage. This compilation pass computes static approximations to actual dynamic values. Once these approximations are computed the optimizer searches for polymorphic definitions that are used with only one single type of argument. These definitions may be compiled using an efficient monomorphic framework. Once this optimization is completed there is no further need for approximations. Approximations are then local to the control flow analysis. The common framework for a stage is:

1. Widen the nodes of the abstract syntax tree that are of some use for the stage.
2. Process the computation of the stage on the wide tree.
3. Shrink the nodes of the abstract syntax tree.

This framework reduces memory consumption for the AST because only the informations relevant to one pass are live at a time. Moreover this framework helps the development of the compiler because it strictly enforces the separation between stages. If a piece of information is needed by more than one pass, it has to be held by the global tree making apparent that the information is global.

Former Bigloo versions (prior to our implementation of wide classes) relied on the *user field* strategy. With this strategy, the AST used to be extended with one “free” field acting as a hook for each stage that needed local informations. The new version of Bigloo has eliminated all the *user fields* in favor of wide instances. Widening helps at the design and the implementation of programs that conform to the compiler’s framework. The improvement is fivefold:

- The source code is easier to write because we benefit from the object library and object facilities. For instance, with traditional languages access to a value held by a user field looks like:

```
(define (add-approx! funcall ap)
   (let* ((uf (funcall-user-field funcall))
          (approx (cfa-approx uf)))
      (cfa-approx-set! uf (cons ap approx))))
```

  Using widening and the `with-access` construction the same code is turned into:

```
(define (add-approx! funcall ap)
   (with-access::wide-funcall funcall (approx)
      (set! approx (cons ap approx))))
```

  The second version is shorter to write and we think easier to read. It is still possible to write sets of macros or functions that simulate the look and feel of the wide instance but it is in charge of the programmer. Wide classes mechanisms help the programmer with convenient ways to access objects and we think that it is the role of a programming language to provide with such help.
- In addition to accesses, wide instances benefit from the full features of the object-oriented model. Wide classes use inheritance and subtyping effectively: methods may override generic function definition for wide classes.
- The code of the compiler can be easily extended. The extensions may be implemented by the means of supplementary widening. Adding local informations to the AST or adding new compiler stages do not require a change in the whole program because the common AST can be left unchanged.
- Wide classes are safer because they benefit from the type checking of the compiler and the runtime system. As we discussed in the introduction, *user fields* may not be of another type than the most generic type (`void *` for C). In the context of a programming language that supports full polymorphism as Scheme does, this gives no opportunities for the compiler to produce good code by eliminating dynamic type tests and to help the programmer by signaling, at compile time, type errors. Wide classes is of great help in this context. For instance, in t

  The consequence of that extra safety is that within the context of our compiler, wide classes help enforcing strict separation between compiler passes. Pass separation is implemented as type tests and these tests are *automatically* introduced by the compiler.
- Wide instances implementation is more efficient than *user fields*. This is discussed in the following Section 5.

5 Wide Classes Implementation

This section describes an implementation strategy for wide classes. It is very close to the implementation of the current Bigloo release.

5.1 Instances

Bigloo compiles Scheme modules into C programs. The natural mapping for a Bigloo object is a C data structure. In addition to the fields found in the class declaration, two extra fields are added to the C structure. The first one, `class_num`, holds the runtime type information used for type predicates and generic functions. The second one is used as a `mark` for object marshaling. As an example, the class:

```
(class a-class x::long y::char)
```

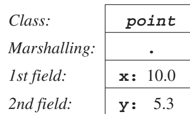
is compiled into:

```
typedef struct a_class_125 {
   class_t  class;   // The class pointer.
   void    *mark;    // The object marshaling mark.
   long     x;       // The 1st field
   char     y;       // The 2nd field
} *a_class_125_t;
```

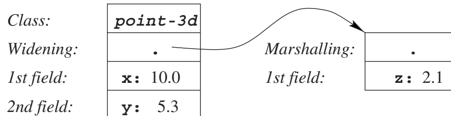
Accessing a field of an instance is thus implemented as accessing a C structure slot.

5.2 Widening and Shrinking

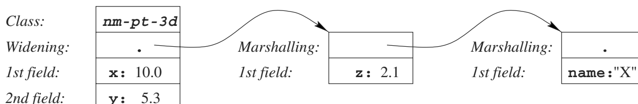
Widening implementation is straightforward: widening means allocating rooms for the new fields of the wide class; establishing a link between the instance and that new memory chunk (henceforth *wide chunk*); changing the header field of the object to reflect the operation. The wide chunk contains one extra field to implement the instance marshaling. The hooking between the instance and wide chunk uses the slot previously devoted to marshaling. Let us consider the memory implementation of an instance of class *`point`*.



After widening to the *`point-3d`* class, the instance becomes



Multiple widening repeat that framework and thus the memory layout for the widening of the instance of *`point-3d`* to the *`named-point-3d`* wide class is:



With our implementation framework, shrinking is pruning the last *widening* pointer and restoring class pointers. There is no need to store in the wide chunk the former class of the instance because, when shrinking, that class is statically known.

5.3 Accessing Wide Instances

Accessing the field of a plain instance is implemented by accessing the slot of a C structure. Accessing the field of a wide class requires several accesses to several C structures. For instance, the C sequence for fetching the `z` field of an instance `p` of the class *`point-3d`* is:

```
(point_3d_t)(p->mark)->z;
```

Fetching the `name` field of an instance of class *`named-point-3d`* is:

```
(named_point_3d_t)((point_3d_t)(p->mark)->mark)->name;
```

The wider the object is, the more expensive is fetching a slot of its wider chunk. Fetching a wide instance slot has the same performances as the *user field* framework. On the other hand, because of the dynamic type checking nature of Scheme, even a smart compiler is not always able to prove at compile time that the access implied in a fetch is of the correct type. In consequence, dynamic type checks have to be executed at runtime. In the case of wide instances, this requires exactly one check: is the object an instance of a given class? This operation is constant in execution time. With the *user field* approach, this does not apply anymore. The object has to be checked for its class and each accessed user field has to be checked itself. Accessing the `name` field of an instance of the wide class *`named-point-3d`* costs exactly one test. Implemented without wide classes the same operation would have required 3 tests.

6 Related Work

6.1 Smalltalk `become:` and Clos `change-class`

The `become:` mechanism of Smalltalk-80™ allows the values of two objects to be swapped. The Clos library function `change-class` allows an object of a class $\mathcal{C}_1$ to be reshaped into an instance of any other class $\mathcal{C}_2$. Both Smalltalk's `become:` and Clos' `change-class` are some kind of unrestricted widening+shrinking operations. Using the Smalltalk construction two objects of unrelated classes may be swapped. Using the Clos generic function an object may be reshaped into an instance of any other class. This is not possible with widening. Widening is restricted to subclasses. When an object is widened, its actual type is turned into a subtype of its previous actual type. Such a property does not hold for `become:` or `change-class`.

We believe that preserving subtyping relationships is very important. It helps the system at statically detecting type errors and it makes the semantics of widening more intuitive and simpler.

The other important advantage of widening is that it is easy to implement. Widening may be efficiently implemented because it does not change the memory layout of an object (see Section 5). This is not true for `become:` [13] nor `change-class`. The CommonLisp `change-class`, as the widening operator, preserves `eq` equivalence. If this has no consequence for widening it has a consequence for `change-class` where a kind of double indirection is required to access object slots. If an object stays `eq` to what it used to be before a `change-class` and if `change-class` may have totally changed the shape of that object, it means that the object slots are not encoded inside the object but outside the object. This is why a double indirection is needed to access one object field.

Another important property of widening is that a program that does not use widening does not pay any extra cost for that construction. This is not true for `change-class` nor `become:` as they impact the whole run-time design.

6.2 Class Predicates and Instance Classification

The programming language Cecil [2] extends traditional object-oriented languages with *Predicate classes* [3]. Wide classes may look like predicate classes because a predicate class is also designed to reify transient states or behavior modes of objects. A predicate class is a class extended with one predicate. An object belongs to that class if the predicate applied to that object is satisfied. The promotion from a class to a predicate class is automatic as soon as the predicate is satisfied. This is the main difference with wide classes. An object is explicitly widen and shrunk in the program.

Traditional languages cannot dispatch on general object property such as “is a window iconified?”. Predicate classes are designed to deal with such situations. Wide classes could be used too because the iconification action is explicitly required and then a widening operation may take place at that time. Wide classes cannot be used when there is no location in the source file associated to the object state modification. It is not easy to model changes in object states on some global property such as “more than 10000 instances”.

Predicate classes are not designed to help at reducing software retention which is the main goal of wide classes. “Cecil predicate objects (instances of predicate classes) reverse space for any fields that might be inherited from a predicate object, i.e., those fields inherited by an object assuming all predicate expressions evaluate to true. The value stored in a field of an object persists even when the controlling predicate evaluates to false and the field is inaccessible”.

Moreover, it is likely that predicate classes do not enable efficient implementation. Method lookup is augmented with additional evaluations of predicate expressions for methods attached to predicate objects. Type checking might be expensive because it requires user code evaluation and the time complexity of the type checking depends on the number of predicate classes.

The Kea programming language is a functional object-oriented language [6] that proposes *classifiers*. These are similar to Cecil's predicate classes. Both support property-based classification. Being a strict functional language Kea is not designed to help at implementing object which types vary over the execution time. Objects in Kea can't be modified.

6.3 Dynamic Slots

The paper [11] presents dynamic slots that enable runtime extensions of objects. A dynamic slot is an attribute that exists in all objects, an for which storage is only allocated if the dynamic slot is used, i.e., if a value is assigned to it. Dynamic slots are statically declared in the source code, hence statically typed. The main difference between dynamic slots and wide classes is that dynamic slots are available on all objects, i.e., they are orthogonal to the type system. When an instance is widen, its type changes and thus, its behavior may change. Because dynamic slots are not related to the type system, they cannot be used to change the behavior of an object. Contrarily to widening, defining dynamic slots for an object does not allow that object to react to new methods.

Dynamic slots are statically declared but the way objects can be dynamically extended is left undeclared. Thus, dynamic slots are implemented by the means of linked lists. Fetching the value of a dynamic slot requires a lookup in an association table. It is linear in time. Has it has been presented in Section 5, because widening restricts the way objects may evolve over the execution, accessing the slot of a wide object is constant in execution time, i.e., the offsets associated to the wide object slots are statically computed.

Conceptually dynamic slots exist for all object. Thus, it is unclear if dynamic slots can be erased from an object. In consequence, we do not think that dynamic slots can help at solving the software data retention problem.

The *adoption* presented in [10] is close to dynamic slots. In that model inspired from Smalltalk's `become:`, an instance may be *adopted* by another class. The adoption may enlarge the number of fields implemented by an object. Fields are thus held by lists as for dynamic slots. In opposition to dynamic slots, an instance may be adopted several times. Thus it is possible to simulate widening and shrinking with class adoptions.

6.4 Object-Oriented Programming with Modes

The paper [12] presents an extension to the class-based object-oriented model in order to support explicit definition of logical states, named *modes*. Modes can be added to class by allowing the definition of multiple sets of operations within class definitions and modifying the late binding algorithm to automatically select between these sets of operations on the basis of the current mode of the current receiver object. In addition, a notion of transition is introduced to enable mode determination to be performed without explicit mode changing statements.

Modes helps at implementing specific behaviors with respect to some object states. For instance, if we consider the example of the window of the introduction:

a window has two states, open or iconified. The behavior of a window differs if it is open or iconified. For instance, clicking on an icon opens a window but clicking on an open window simply selects it. Modes proposes an elegant way to implement such situations. Two different implementations for the `click` message will be given in the window `class`: one for each states. The transition function for windows simply records if the window is open or iconified.

The primary goal for widening is not to express states for objects. It is to help the implementation of transient data storage. Modes does not address this problem. Modes can be simulated with wide classes. To change the behavior of an instance `o` of a class $\mathcal{C}$, it is sufficient to add a wide class $\mathcal{C}'$ with no additional slot. Widening `o` from $\mathcal{C}$ to $\mathcal{C}'$ or shrinking from $\mathcal{C}'$ to $\mathcal{C}$ may change the behavior of `o` because methods can be added in $\mathcal{C}'$. However, because widening uses inheritance, it is restricted by subtyping relationships. Modes implement type unions rather that subtypes. Type unions are not easily mapped to inheritance trees, specially with our system that makes use of single inheritance.

7 Extensions

7.1 Multiple Widening

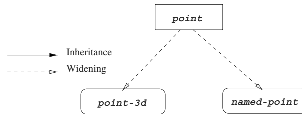
A possible extension to wide classes could be to allow multiple widening of a same object. This could be useful when unrelated extra informations have to be added to an object. To get back to the point example, it could be a useful feature to allow a plain `point` to be widened into a `point-3d` but also to be widened next into a `named-point`. The example below illustrates that possibility:

```
1: (module points
2:    (export (class point ...)
3:            (wide-class point-3d::point ...)
4:            (wide-class named-point::point ...)))
```

The same component of an instance may be widened more than once:

```
1: (let ((p (instantiate::point)))
2:    (widen!::point-3d p)
3:    (widen!::named-point p))
```

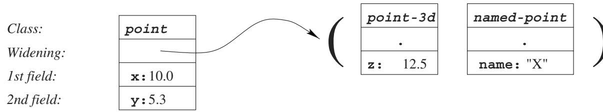
The inheritance tree is:



This is forbidden with the model previously presented because at line *3* `named-point` is not a subtype of the actual type of `p` (i.e. `point-3d`). The multiple widening would require to change the `shrink!` operation. It is now mandatory to decide which component of a wide object must be shrunk.

We have decided not to implement the multiple widening because we have found no way to implement it as efficiently as simple widening. In the presence

of multiple widening an object could contain several wide chunks (see Section 5). The new memory layout with multiple widening may look like:

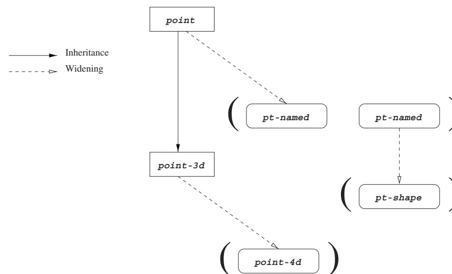


Testing for a typing property (e.g. is the object p an instance of the class $\mathcal{C}$?) is harder with multiple widening because an object may have several simultaneous actual types that are not in a subtyping relationship. It is likely that there is no way to perform type checking without checking in sequence all actual types of a wide object.

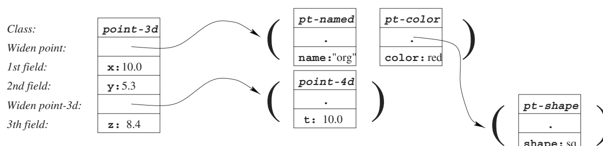
Accessing the slots of a wide object in presence of multiple widening is a more complex operation than with single widening because it is first necessary to fetch the correct wide chunk. The complexity of the worst case of that operation is $\mathcal{O}(c)$ where c is the number of classes.

7.2 Multiple Widening and Multiple Inheritance

The second step to widening is to meet the multiple inheritance paradigm. With that programming model it would be coherent to allow a plain object to be widened more than once for each of the class it inherits from. Thus, it would be also possible to widen several times the same component of an instance. That is, it would be possible to first widen the `point` part into, for instance, a `point-named` and then to widen one more time the `point` part into a `point-colored`, etc. An instance must be provided with as many slots for widening as the number of classes it inherits from. For our example, a `point-3d` instance must be provisioned with 2 slots for widening, one for the `point` component, one for the `point-3d` component. This is not enough: a slot for widening must contain the list of wide chunks instead of pointer to a single wide chunk. The inheritance tree would look like:



The data layout for such an object would look like:



With that naive framework accessing the slot of a wide chunk and checking for a wide type is expensive. Most likely the clever representations that are used within implementation for languages that support multiple inheritance would applied here.

8 Conclusion

Wide classes extend traditional object-oriented programming languages with a construction that models transient properties. Such a construction does not exist in regular languages. Thanks to wide classes and wide inheritance the shape of an object may change during its life-time. An object can be, at a certain point of the execution, *widen* to another class, that is, an object may be extended with additional fields. Later on, this same object can be *shrunk* to its original shape. We think these features may help the programming task because we think programs using data structures that vary over execution time are frequent.

Wide classes are efficient at solving software retention. This problem arises when an actual data structure is not able to reflect time properties in order to minimize the memory consumption of an execution. We have shown how wide classes apply to the construction of our Scheme compiler and how they succeed in improving the programming style.

Wide classes may also be used to model object states. If an object goes from one state to another this transformation can be implemented by the means of wide classes. Wide classes provide a mechanism to change the type of an object and thus to change the way generic functions will react when applied to it.

Wide classes could probably be easily implemented in other object-oriented programming languages although dynamic type checking or, at least runtime type informations, are likely to be required. We don't think languages that make use of type inference like O'Caml [9] can benefit from wide classes. W [4] based type inference algorithms are not able to detect errors in programs like the one shown below where the last `point-3d-z` access is illegal:

```
(let ((p (instantiate::point)))
   (widen!::point-3d p (z 45))
   (+ (point-3d-z p) (begin (shrink! p) (point-3d-z p))))
```

The reason is that current type inference algorithms of that trend cannot associate several types to a same object. An object has to be either a *point* or a *point-3d* but it can't be at some point in time a *point*, becomes a widen *point-3d* and returns back to a plain *point*. On the other hand, languages that propose coercion operations like Java could probably be extended with wide classes although at the cost of extra dynamic type checks.

Acknowledgments

Many thanks to Christian Queinnec, Jan Vitek and to Céline for their helpful feedbacks on this work.

References

1. D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common lisp object system specification. In *special issue*, number 23 in SIGPLAN Notices, September 1988.
2. C. Chambers. The Cecil Language: Specification and Rationale. Technical Report 93-03-05, University of Washington, Department of computer Science and Engineering, March 1993.
3. C. Chambers. Predicate classes. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.
4. L. Damas and R. Milner. Principle Type Inference for Functional Programs (extended abstract). In *9th ACM Symposium on Principle of Programming Langugages*, pages 207–212, 1982.
5. A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
6. J. Hamer. Un-Mixing Inheritance with Classifiers. In *Multiple Inheritance and Multiple Subtyping: Position papers of the ECOOP'92 Workshop*, LNCS 707, pages 6–9, Utrect, Netherlands, July 1992. Springer-Verlag.
7. IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.
8. C. Queinnec. Designing MEROON v3. In *Workshop on Object-Oriented Programming in Lisp*, 1993.
9. D. Rémy and J. Vouillon. Obective ML: A simple object-oriented extension of ML. In *Symposium on Principles of Programming Languages*, pages 40–53, 1997.
10. F. Rivard. *Évolution du comportement des objets dans les langages à classes réflexifs*. 98-1-info, École des Mines de Nantes, 1998.
11. R. Schmidt. Dynamically Extensible Objects in a Class-Based Language. In *TOOLS USA*, July 1997.
12. A. Taivalsaari. Object-oriented programming with modes. *Journal of Object-oriented programming*, pages 25–32, June 1993.
13. D. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, Cambridge, Mass., USA, 1986.

An Approach to Classify Semi-Structured Objects*

Elisa Bertino[1], Giovanna Guerrini[2], Isabella Merlo[2], and Marco Mesiti[3]

[1] Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
Via Comelico 39/41 - I20135 Milano, Italy
bertino@dsi.unimi.it
[2] Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Dodecaneso 35 - I16146 Genova, Italy
{guerrini,merloisa}@disi.unige.it
[3] Bell Communications Research
445, South Street - Morristown (NJ), U.S.A.
marco@research.bellcore.com

Abstract. Several advanced applications, such as those dealing with the Web, need to handle data whose structure is not known a-priori. Such requirement severely limits the applicability of traditional database techniques, that are based on the fact that the structure of data (e.g. the database schema) is known before data are entered into the database. Moreover, in traditional database systems, whenever a data item (e.g. a tuple, an object, and so on) is entered, the application specifies the collection (e.g. relation, class, and so on) the data item belongs to. Collections are the basis for handling queries and indexing and therefore a proper classification of data items in collections is crucial. In this paper, we address this issue in the context of an extended object-oriented data model. We propose an approach to classify objects, created without specifying the class they belong to, in the most appropriate class of the schema, that is, the class closest to the object state. In particular, we introduce the notion of weak membership of an object in a class, and define two measures, the conformity and the heterogeneity degrees, exploited by our classification algorithm to identify the most appropriate class in which an object can be classified, among the ones of which it is a weak member.

1 Introduction

In the last few years, there has been in the database community a growing interest in the management of *semi-structured* data [1]. Semi-structured data are data whose structure is not regular, is heterogeneous, is partial, has not a fixed format and quickly evolves. Moreover, the distinction between the data described

* Work partially supported by the Italian MURST under the Interdata Project.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 416–440, 1999.
© Springer-Verlag Berlin Heidelberg 1999

by the structure and the structure itself is blurred. Those characteristics are typical of data available on the Web [5], of data coming from heterogeneous information sources [24] and so on. The lack of a fixed a-priori schema and of information on the data structures makes it difficult handling semi-structured data through conventional database technology.

Currently, the research activity concerning the management of semi-structured data is moving along three directions [14]: (1) techniques for gathering all kinds of information (HTML pages, images, multimedia documents and so on) from various information sources (like the Web) and for extracting structural information from them [3,19]; (2) development of data models able to represent such kinds of information and extension of traditional database techniques to manage them; (3) development of query execution techniques able to exploit the structural information extracted from data and of techniques to export data on the Web [2].

In the data model area the research community has proposed two main approaches to model semi-structured data [10,14]. The first one is a more traditional approach and consists of adapting existing data models to deal with semi-structured data. In particular, extensions to the object-oriented data model have been proposed with less restrictive type systems [13,26]. The second approach, by contrast, does not have any notion of type and schema to avoid any restriction on the structure of the data to be stored in the database. The basic idea of this approach [4,12] is to use a labeled graph to store structural information together with data they refer to. An advantage of the first approach over the second one is the existence of a structure containing information on the type of data separated from the data themselves. This is important for efficiently querying data and for developing adequate storage structures and indexing techniques. To overcome the drawbacks of the lack of "schema" information, proposals following the second approach have been recently extended with the introduction of some flexible "schema mechanism" [11,17], able to represent information on the data, and yet leaving a high degree of freedom with respect to the data entered into the database.

An important issue, quite independent from the modeling approach adopted, is to capture the existence of some regularity in the data, i.e., typing data or, as in our approach, classifying them. Automatic typing or classification is crucial in order to achieve effective storage and retrieval. However, limited work has been carried out in the context of semi-structured data. In this paper, we address such issue by defining a classification approach for data, whose structure is not known, with respect to classes in an inheritance hierarchy of an object-oriented database. Therefore we assume the existence of an a-priori defined schema and allow one to create objects, whose class is not known, which are automatically classified in the existing schema. Once an object has been classified, it can be effectively considered part of the database. Applications can, therefore, access and modify such an object exploiting all the database features.

It is important to remark that the problem of automatically classifying information has also been investigated in other areas. However, semi-structured data

have features requiring the development of specific classification techniques. In particular, the problem of automatic classification has been dealt with in the context of frame-based terminological languages [25], which use automatic classification techniques both at the terminological and assertional levels (schema and data, respectively), for correctly positioning a concept in the taxonomy and for determining the most specific concept for an instance. Classification approaches, typical of such languages, rely on determining subsumption relations among concepts. Subsumption, however, takes into account also attribute values, rather than only considering the similarity of the structure, as in our approach, which then results in simpler and more efficient algorithms. The problem has also been investigated in the software engineering area. In [7] a CASE tool is proposed that starting from a set of object examples derives a schema suited for handling those objects.

In the context of semi-structured data the problem of automatic typing has been addressed in [23]. However, the goal of that work substantially differs from ours, since their main aim is to extract schema information from data, that is, to extract structure from raw data. They deal with the problem of how to avoid the proliferation of types by defining a distance among types, but they do not address how their framework could exploit some a-priori knowledge on the data schema. We remark that this knowledge, that we assume in our approach, often occurs in practice, for instance when integrating semi-structured data, discovered on the Web, with data having a known structure or when the semi-structured data have associated some kind of structural information (for example the Document Type Definition associated with an XML page [21]). Moreover, in [23] it is not specified whether the insertion of new objects, once the schema is set, can result in schema modifications and attribute domains are not kept into account.

Our classification approach has been proposed in the context of a reference data model [8]. The reference data model includes some new types ensuring a highly flexible type system. In particular, its modeling power is comparable to that of the best known data models for semi-structured data, such as [4,13,26], in that it captures all the kinds of data heterogeneity that can be represented in those models. Moreover, we remark that, though tailored to a given data model, our approach to automatic classification is highly independent from the particular type system and it can be easily adapted to other object-oriented data models and type systems supporting union types. In fact, union types represent a common extension to a traditional object type system to meet the flexibility requirements for managing semi-structured data.

In our model, a *semi-structured object* is an object that has been created without specifying the class it belongs to. To this purpose, our model supports a new operation in which the class to which the object belongs to may not be specified.

In the context of semi-structured data, the assumption that for each object there is a type exactly describing it is too strong. Thus, in our model we do not make such assumption and we rely on a notion of *weak membership*. Such notion is weaker than the classical notion of class membership, since we only

require the components[1] in the object state be a subset of the components of the *structural type* of the class,[2] rather than requiring the components of the object state be exactly all and only those appearing in the structural type of the class, as in traditional object-oriented data models. According to our notion of weak membership, an object can be a weak member of no class, of just one class or of several classes, even not related by inheritance hierarchies. To determine the *most appropriate* class for an object, among the ones of which the object is a weak member, we use two measures: the *conformity degree*, measuring the similarity degree between the type of the semi-structured object and the structural type of the class, and the *heterogeneity degree* of the class, measuring how much the extension of the class is heterogeneous. If an object is a weak member of no class, it is inserted in a repository of unclassified objects. As the schema evolves the repository is periodically examined, trying to classify objects contained in it.

As it is outlined in [23], addressing the problem of extracting structure from semi-structured data leads to approximate typing or classification, since heuristic techniques are exploited. The conformity and heterogeneity degrees are measures that allow one to classify a given object in the schema, inserting it into a class which is as close as possible to the actual object structure.

Among the possible applications of our classification technique, we would like to mention its use in supporting content-based search on the Web. The idea is to record the content of HTML pages in an existing database. Through an information extraction tool one can delimit a (semi-structured) object representing the relevant information of the page. Then, our automatic classification tool determines the most appropriate class to insert the object in. Once the object, corresponding to the HTML page, is inserted in the database, such information can be used to support content-based data retrieval through a query language. We believe that such an approach could represent a relevant improvement to the well-known techniques, based on pattern matching, adopted by the most popular Web search engines.

The remainder of the paper is organized as follows. In Section 2 we review the concepts of the reference data model that are relevant to this work. In Section 3 we introduce the notion of weak membership, whereas in Section 4 we discuss the proposed classification approach and we present an algorithm to automatically classify objects according to our notion of weak membership. Finally, Section 5 concludes the paper and discusses future work. Appendix A presents the formal definitions of some concepts introduced throughout the paper.

2 Reference Data Model

The *reference data model*, defined as extension of the *basic object-oriented data model* [18], is based on a type system which consists of three kinds of types: *value types*, *object types*, and the `spring` *type*. **Value types** are classical types such

[1] A *component* of a record value or of a record type is one of the slots composing it.

[2] The structural type of a class is the record type containing the attributes of the class and their respective domains.

as *basic value types* (`integer`, `bool`, `real`, etc.) and *structured types* (built by means of `record`, `set` and `list` constructors). The reference data model adds to this set of types the *union type*, that we will discuss in more details below. **Object types** are types corresponding to classes (class names). Finally, the `spring` **type** is a new type, not present in the basic object-oriented data model, allowing one to specify that an attribute does not have any specific domain. Because of the relevance of this type in handling semi-structured data, we will also discuss it in more details below. It is important to remark that the reference data model, as the basic object-oriented data model, supports all the common features of object-oriented data models such as object identity, user-defined operations, classes, inheritance (we refer the reader to [8] for details on the reference data model). The `spring` and union types enrich the original object-oriented data model with the flexibility required to manage semi-structured data, and make the type system of our model more flexible than those of existing data models for semi-structured data [13,26]. In order to provide a safe object-oriented data model, in [8] subtyping relationship and class refinement are addressed.

In the remainder of this section we first discuss the new types added to the basic data model and then we introduce the notions of class and object as supported by the model.

2.1 Union Types

A **union type** consists of a set of types belonging to the basic type system each one associated with a distinct label. Let $T_1, \ldots, T_n$ be value types of the basic object-oriented data model or object types and $a_1, \ldots, a_n$ be distinct labels, then the type *union-of(*$a_1 : T_1, \ldots, a_n : T_n$*)* is a union type. Our union type definition is similar to the one proposed in [13], but it is not identical since we impose the restriction that the types of the union type components be neither the `spring` type nor union types. This restriction ensures efficiency in terms of space allocation and type safety, and simplifies classification of semi-structured objects. Subtyping rules for union types are similar to those proposed in [13]. Legal values for a union type are pairs $l : v$, where l is the label of a union type component, and v is a legal value for the type associated with l.

As a consequence of the introduction of union types, we have modified the record type definition of the basic object-oriented data model to allow one to omit the label associated with a component whose type is a union type. In this way, in order to access that component, we only need to use the label appearing in the union type definition.

Example 1. Let `person` be a class name. Let `record-of(a:integer, union-of (b:string, c:person))` be a record type. Let X be a variable of this type. In order to access component `b`, we simply write X.`b`. ◇

To avoid ambiguities in accessing a component of a record type, we impose that the labels of record type components and the labels of union type components be all distinct. That is, we disallow record types such as `record-of(a:integer, union-of(a:string,c:person))`.

A legal value, for a record type, has the form $(a_1 : v_1, \ldots, a_n : v_n)$, where a_i is the label of a record type component or the label of a union type component appearing in the record type definition, and v_i is a legal value for the type associated with a_i in the corresponding record type definition. For example, let i_p be the identifier of an object belonging to the class `person`, then $(a : 5, b :$'*rose*'$)$ and $(a : 8, c : i_p)$ are legal values for the type of the previous example.

2.2 Spring Type

The `spring type` is the common supertype of value types and object types. The introduction of this type allows us to manage data without knowing their actual type. Each legal value of each type of the model is a legal value for the `spring` type. Note that our notion of `spring` type is different from the notion of `Object` type, supported by some systems like GemStone [9]. The first difference is that in our model we have both value types and object types, whereas those systems only support object types. The `spring` type, in our model, is not an object type and is not a value type, rather it is a common supertype of all (value and object) types of the model. Another relevant difference is that in our model the `spring` type cannot be directly instantiated, that is, no objects or values can be proper instances of the `spring` type. In other systems, by contrast, objects can be proper instances of the `Object` type.

2.3 Classes, Objects and Semi-Structured Objects

Our model supports a quite standard notion of class, with some differences arising from the introduction of the union and `spring` types. Each class, moreover, has a structural type, which is a record type describing the state of the class instances, formally defined as follows.

Definition 1. (Structural type of a class). *Given a class c, defined as*

$$\mathit{class}\ c\ \{a_1 : T_1, \ldots, a_m : T_m,\\ \mathit{union\text{-}of}(a_1^1 : T_1^1, \ldots, a_1^p : T_1^p), \ldots, \mathit{union\text{-}of}(a_n^1 : T_n^1, \ldots, a_n^p : T_n^p)\ \}$$

the record type $\mathit{record\text{-}of}(a_1 : T_1, \ldots, a_m : T_m, T_{m+1}, \ldots, T_{m+n})$, *where, for* $k = 1, \ldots, n$: $T_{m+k} = \mathit{union\text{-}of}(a_k^1 : T_k^1, \ldots, a_k^p : T_k^p)$, *is the structural type of class c, denoted by* $\mathit{stype}(c)$. □

Note that, as specified in the definition above, the class contains some fixed attributes ($a_1, \ldots, a_m$), and some other components for which one out of some possible alternatives, specified through a union type, can be chosen (component $m+1$ to $m+n$).

The notion of object supported by the model, formalized by the following definition, is also quite standard.

Definition 2. (Object). *An object is a triple $o = (i, v, c)$ where i is an object identifier, v is a record value (the object state) and c is the most specific class to which o belongs.* □

Finally, the following definition states the conditions for an object to be an instance of a class.

Definition 3. (Instance). *An object o is an* instance *of a class c if $o.v$ is a legal value for stype(c).* □

Definition 3 above requires that the following conditions hold:

(1) for each component $a : v$ of the object state, a component $a : T$ exists in *stype*(c) such that v is a legal value for T or a component *union-of*$(a_1 : T_1, \ldots, a_p : T_p)$ exists such that $a : v$ is a legal value for that component, that is, $\exists\, i, 1 \leq i \leq p$, such that $a = a_i$, and v is a legal value for T_i;
(2) for each component $a : T$ in *stype*(c), a component $a : v$ exists in the object state such that v is a legal value for T, and for each component *union-of*$(a_1 : T_1, \ldots, a_p : T_p)$ in *stype*(c) a component $a : v$ exists in the object state such that $a : v$ is a legal value for that component, that is, $\exists\, i, 1 \leq i \leq p$, such that $a = a_i$ and v is a legal value for T_i.

Condition (1) above requires that each component in the object state corresponds either to an attribute of the class (and in this case the component value must be a legal value for the attribute domain) or to one of the components of a union type in the structural type of the class (and in this case the component value must be a legal value for the union type component domain). Condition (2) above, by contrast, requires that the object state contains a component for each attribute of the class and a component for each union type in the structural type of the class (corresponding to one of the components of the union type).

The following is an example of classes and objects in our model.

Example 2. Suppose we wish to model information about people, and in particular name, age, birthday and love, where name may be a `string`, or a `record` with two components, first name (`f-name`), surname (`s-name`), and love may assume any value (a person may love another person or an animal or anything else). Let `date` be a class of the database schema and i_d be the identifier of an object instance of class `date`. We may define a class `person` whose structural type is:

```
record-of(union-of(nameS:string, nameR:record-of(f-name:string,
        s-name:string)), age:integer, birthday:date, love:spring).
```

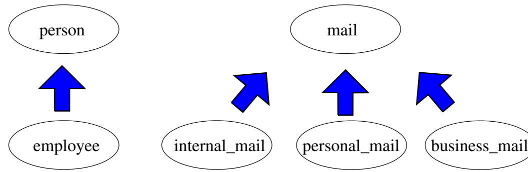


Fig. 1. A chunk of a database schema

The following objects are instances of class `person`: $o_1 = (i_1, v_1, \texttt{person})$, where $v_1 = (\texttt{nameS} :\text{'}X\text{'}, \texttt{age} : 25, \texttt{birthday} : i_d, \texttt{love} :\text{'}rose\text{'})$, and $o_2 = (i_2, v_2, \texttt{person})$, where $v_2 = (\texttt{nameR}:(\texttt{f-name} :\text{'}Max\text{'}, \texttt{s-name} : \text{'}X\text{'}), \texttt{age} : 25, \texttt{birthday} : i_d, \texttt{love} : i_p)$. Note that, even if their states are legal values for the structural type of class *person*, they have different structures. ◇

In our model, finally, we denote by **semi-structured object** an object created without specifying the class it belongs to. The object is called semi-structured because it is inserted in the database without any a-priori information about its class. The object can be an instance of several classes of the schema, or of no class. In other words, the object state may be a legal value for the structural type of more than one class, or it may be a legal value for no structural type associated with any class in the schema.

Example 3. Suppose we want to classify e-mails in a database in order to make their retrieval easier. Suppose we have a mailer that allows to associate some structural information with e-mails and that we have an extraction information tool able to take that structural information out from them. Now we want to create a database to store e-mails using our model. The idea is that when a new e-mail is sent or received through the mailer, the extraction information tool takes out information (with a structure) from the e-mail and tries to insert this semi-structured object in the database. Suppose we have created the following classes in the database:[3]

- *stype*(`mail`) =
  `record-of(union-of(receiverS:string, receiverP:person), body:spring);`
- *stype*(`internal_mail`) =
  `record-of(subject:spring, sender:person, union-of(receiverS: string, receiverP:employee), body:string);`
- *stype*(`personal_mail`) =
  `record-of(subject:person, sender:person, union-of(receiverS: string, receiverP:person), body:string);`
- *stype*(`business_mail`) =

[3] We do not present classes `person` and `employee` and the other classes of the schema because they are irrelevant for the example.

```
record-of(logo:string, sender:person, union-of(receiverS:
string, receiverP:employee), body:string).
```

The meaning of the previous classes is intuitive. The idea is to have a class for e-mails and to refine the personal, internal and business e-mails in distinct subclasses. Figure 1 shows a chunk of the database schema, showing the inheritance relationships among classes. Suppose the extraction information tool generates objects o_1, ..., o_4 from some e-mails arrived in the mailbox, whose states are, respectively:

1. (`receiverS`:'*Elena F* ...', `body`:'*Dear Monica* ...'),
2. (`subject`:i_p, `sender`:i_p),
3. (`sender`:i_p, `receiverP`: i_e, `body`:'*Hello Ugo* ...'),
4. (`subject`:'*Summer in* ...', `attachment`:'*photo.jpg*')

where i_p is an identifier of an object of class `person` and i_e is an identifier of an object of class `employee`. The first object is an instance of class `mail`, whereas the others have less attributes or, as in the last case, has an attribute not in the schema. In the following we will see how our classification approach handles these situations. ◇

3 Weak Membership

In the management of semi-structured objects we want to emphasize the role of the class as a repository that contains objects whose states have the same type,[4] rather than as a template for creating objects. In this context, we allow applications to create objects without specifying the class they belong to. Then, it is the system that automatically classifies those objects in an appropriate class. In order to classify a semi-structured object, we need a criterion to bind such object to a class. The notion of instance could be used for this purpose, but its definition is too restrictive to be used for semi-structured objects. In order to achieve the flexibility needed to classify semi-structured objects, we propose a weaker notion, referred to as *weak membership*, only requiring condition (1), stated after Definition 3. Thus, the structural type of a class may have more components than those appearing in the object state. In such a case, we need some exception-handling mechanism to manage accesses to components not present in the classified object. The idea of classifying semi-structured data in an existing a-priori database schema could seem too restrictive. By contrast we believe that our automatic classification, based on the notion of weak membership, represents a compromise between the flexibility of semi-structured data and the rigidity of object-oriented schemas and allows one to benefit from all the features of object-oriented database systems to manage this kind of information.

[4] Note that, in our model, this condition does not mean that all objects instances of a class have the same structure (cfr. Example 2).

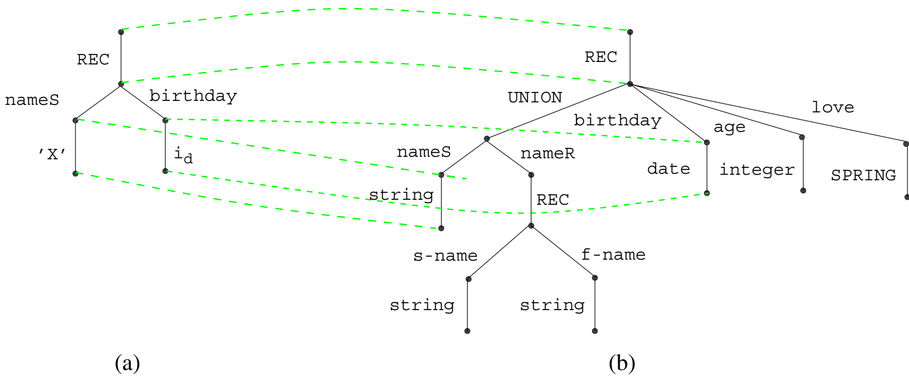


Fig. 2. (a) Object value expression, (b) class structural expression and their simulation relation

In order to formally define the notion of weak membership and to define a method to check whether an object is a weak member of a class, we extend a well-known theoretical notion, the simulation relation [22]. First, we provide an abstract representation of the structural type of a class, the *class structural expression*, and an abstract representation of the object state, the *object value expression*. Then, to verify whether the object is a weak member of the class, we check whether a particular simulation exists between those two expressions. Intuitively, the class structural expression is a tree labeled with symbols representing the attributes of the class and their types, whereas the object value expression is a tree labeled with symbols representing the attributes of the object and their values. In the remainder of this section, we first present the formal definitions concerning class and object expressions (Subsection 3.1) and then the weak membership notion is formally defined (Subsection 3.2).

3.1 Class and Object Expressions

In the following the set $\mathcal{PRED}$ denotes a set of predicates where each predicate represents the set of legal values for basic value types and object types. A predicate $p \in \mathcal{PRED}$ applied to a value v holds if and only if v belongs to the set of instances associated with the type p, where the type p may be a basic value type or an object type. Moreover, given the set $\mathcal{AN}$ of attribute names, $\mathcal{LT}$ denotes the set of tree labels, that is $\mathcal{LT}$ = {LIST, REC, SET, UNION, SPRING} $\cup\, \mathcal{AN} \cup \mathcal{PRED}$. The following definition states the notion of *class structural expression*.

Definition 4. (Class structural expression). *Given a class c, the class structural expression of c (denoted by $\varepsilon_t(c)$) is a tree (V_t, E_t, φ_t), labeled on $\mathcal{LT}$, where V_t is a set of vertices, $E_t \subseteq V_t \times V_t$ is a set of edges, and $\varphi_t : E_t \rightarrow \mathcal{LT}$ is the edge labeling function.* □

Since the class structural expression is a tree associated with a type of the model (the structural type of a class), we have developed an inductive system to map any type of the model into a labeled tree [8]. Figure 2(b) shows the class structural expression associated with class `person` of Example 2. Note that `string`, `date`, and `integer` symbols are predicates which represent the set of legal values for the corresponding types. Note also that we have not generated the structural expression associated with the object type `date` since we are interested in shallow[5] comparison among objects and classes.

In the following definition, stating the notion of *object value expression*, $\mathcal{LV}$ denotes the set of labels of object value expressions, that is, $\mathcal{LV} = \{$LIST, REC, SET, UNION, NULL$\} \cup \mathcal{AN} \cup \mathcal{V}$, where $\mathcal{V}$ denotes the set of legal values for basic value types and object identifiers.

Definition 5. (Object value expression). *Given an object o, the object value expression of o (denoted by $\varepsilon_v(o)$) is a tree (V_v, E_v, φ_v), labeled on $\mathcal{LV}$, where V_v is a set of vertices, $E_v \subseteq V_v \times V_v$ is a set of edges and, $\varphi_v : E_v \rightarrow \mathcal{LV}$ is the edge labeling function.* □

Similarly to what has been done for the class structural expression, an inductive system has been defined in [8] to map values of the model into labeled trees. Figure 2(a) shows the object value expression associated with a semi-structured object whose state is (`nameS` :'X', `birthday` : i_d). According to our shallow approach, we have not generated the object value expression associated with the state of the object identified by i_d.

We also introduce the notion of refinement among structural expressions, which is used by our classification algorithm. Intuitively, a structural expression (that is, a tree whose edges are labeled in $\mathcal{LT}$) ε' is a refinement of a structural expression ε if the two trees are isomorphic, but some of the labels of ε' are class names corresponding to subclasses of the corresponding labels in ε. Let $\mathcal{PRED}^o$ denote the subset of $\mathcal{PRED}$ corresponding to object types, and $\leq_{ISA}$ denote the inheritance relationship on object types, then, the notion of refinement among structural expressions is defined as follows.

Definition 6. (Structural expression refinement). *Let $\varepsilon = (V_t, E_t, \varphi_t)$ and $\varepsilon' = (V_t, E_t, \varphi'_t)$ be two structural expressions. ε' is a refinement of ε if*

$$\forall e \in E_t : \varphi_t(e) = \varphi'_t(e) \vee (\varphi_t(e) \in \mathcal{PRED}^o \wedge \varphi'_t(e) \in \mathcal{PRED}^o \wedge \varphi'_t(e) \leq_{ISA} \varphi_t(e)). \ \square$$

The following example illustrates the notion of structural expression refinement.

Example 4. The structural expression presented in Figure 3(b) is a refinement of the one presented in Figure 3(a) because the two trees have the same structure and the labels are all equals except `person` and `employee` which are in the $\leq_{ISA}$ relation. ◇

[5] *Shallow* is used here with the same meaning as in shallow equality [16].

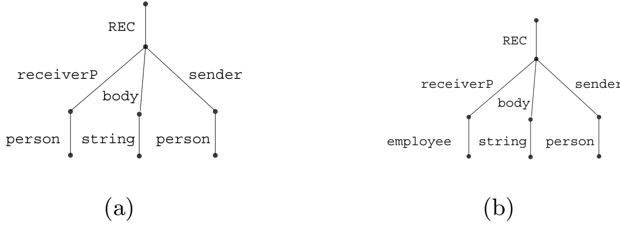


Fig. 3. Structural expression refinement

3.2 Simulation Relation

Before defining the relation between the class structural expression and the object value expression we introduce a mapping between labels in set $\mathcal{LV}$ and labels in set $\mathcal{LT}$, that is used to identify a set of cases to be managed in the same way.

Definition 7. (Relation $\approx_{\mathcal{L}}$ between labels). *A relation $\approx_{\mathcal{L}}$ holds between a label $l_v \in \mathcal{LV}$ and a label $l_t \in \mathcal{LT}$ (denoted by $l_v \approx_{\mathcal{L}} l_t$), if and only if one of the following conditions holds: (1) $l_v = NULL$ and $l_t \neq SPRING$; (2) $l_v, l_t \in \{LIST, REC, SET, UNION\} \cup \mathcal{AN}$ and $l_v = l_t$; (3) $l_t \in \mathcal{PRED}$ and l_t holds on l_v.* □

We are now able, using relation $\approx_{\mathcal{L}}$, to introduce the notion of *simulation*. The simulation is a particular relation among the vertices of the object value expression and the vertices of the class structural expression that takes into account the symbols used to label the edges of these expressions. The idea of simulation is used in several research areas [11,20] and it has a solid theoretical foundation. We will use it to formally define the notion of weak membership. Our definition of simulation in an extension of the "classical" one, thus it preserves its good properties [20].

Informally, a relation $\mathcal{R}$ between the vertices of an object value expression (V_v, E_v, φ_v) and the vertices of a class structural expression (V_t, E_t, φ_t) is a simulation if the following conditions hold:

- If the label $l_v \in \mathcal{LV}$ associated with the edge $(u_1, u_1') \in E_v$, outgoing from vertex u_1, identifies a particular type (structural, basic, object), then an edge $(u_2, u_2') \in E_t$ must outgo from u_2 labeled with a symbol $l_t \in \mathcal{LT}$, for which relation $\approx_{\mathcal{L}}$ holds between l_v and l_t. Moreover, relation $\mathcal{R}$ must hold between u_1' and u_2'.
- If the label $l_v \in \mathcal{LV}$ associated with the edge $(u_1, u_1') \in E_v$, outgoing from vertex u_1, is an attribute name ($l_v \in \mathcal{AN}$) and the label associated with the edge $(u_2, u_2') \in E_t$, outgoing from vertex u_2, is $UNION$, then an edge $(u_2', u_2'') \in E_t$ must exist, outgoing from vertex u_2', with the same label of the edge (u_1, u_1'). Moreover, relation $\mathcal{R}$ must hold between u_1' and u_2''.

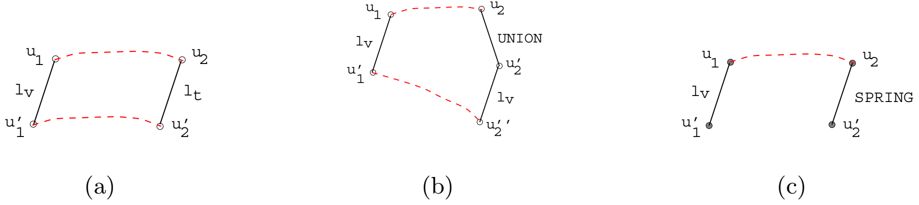


Fig. 4. Visual representation of relation among vertices of item (2) of Definition 8

– If the label $l_t \in \mathcal{LT}$ associated with the edge $(u_2, u'_2) \in E_t$, outgoing from vertex u_2, is $SPRING$, then there is no condition to verify. In this situation we do not need to check other pairs in the relation whose first component is a vertex belonging to the subtree rooted at u_1, since the value associated with the subtree rooted at u_1 surely is a legal value for the type associated with the subtree rooted at u_2 (that is, the `spring` type).

The following definition formally states our notion of simulation. In the definition *root(A)* denotes the root of tree A and $u \xrightarrow{l} u'$ denotes an edge (u, u') such that $\varphi((u, u')) = l$.

Definition 8. (Simulation). *A binary relation* $\mathcal{R}$ *from the vertices of* $A_v = (V_v, E_v, \varphi_v)$ *labeled on* $\mathcal{LV}$ *to the vertices of* $A_t = (V_t, E_t, \varphi_t)$ *labeled on* $\mathcal{LT}$*, is a simulation if and only if the following conditions hold:*

1. $root(A_v)\ \mathcal{R}\ root(A_t)$;
2. *if* $u_1\ \mathcal{R}\ u_2$, *then* $\forall u_1 \xrightarrow{l_v} u'_1$ *in* E_v, $\exists\ u_2 \xrightarrow{l_t} u'_2$ *in* E_t, *such that one and only one of the following conditions holds:*
   (a) $l_v \approx_{\mathcal{L}} l_t$ *and* $u'_1\ \mathcal{R}\ u'_2$,
   (b) $l_t = UNION$, $\exists u'_2 \xrightarrow{l'_t} u''_2$ *in* E_t *such that* $l_v = l'_t$ *and* $u'_1\ \mathcal{R}\ u''_2$,
   (c) $l_t = SPRING$. □

In Figure 2 the dashed lines represent the simulation between the object value expression associated with the semi-structured object, that we have introduced previously, and the structural expression associated with the structural type of class `person` of Example 2. A visual representation of relation among vertices of item (2) of Definition 8 is shown in Figure 4. The dashed lines identify the relation that must hold between the vertices of the two trees. Note that, as you can see in Figure 4(c), we do not require the relation to hold between vertices u'_1 and u'_2.

For determining weak membership, we do not consider every simulation. Consider the following example.

Example 5. Consider the object value expression associated with the object state `(a`: 5, `b`:'*rose*'`)` and the class structural expression associated with the structural type `record-of(union-of(a:integer, b:string))`. According to Definition 8 a simulation exists between them. ◇

The simulation in the above example, however, does not capture our notion of the set of legal values for the record type in the example. The idea of the union type is, instead, that of choosing *one* out of some possible alternatives. Thus, in the definition of weak membership, we leave out this kind of simulations, as formally stated by the following definition.

Definition 9. (Weak membership). *An object o is a weak member of a class c if a simulation $\mathcal{R}$ exists between the object value expression associated with o ($\varepsilon_v(o)$) and the class structural expression associated with c ($\varepsilon_t(c)$), such that $\forall u_2 \overset{UNION}{\longrightarrow} u_2'$ labeled edge of $\varepsilon_t(c)$ at most one pair $(u, u') \in \mathcal{R}$ exists such that $u' \in \{\bar{u} \mid (u_2', \bar{u})$ is an edge of $\varepsilon_t(c)\}$.* □

The above definition of membership is more flexible than the notion of instantiation. According to such definition, an object state can contain less components than those present in the structural type of a class. Such definition, however, does not allow one to identify only one class to which the object belongs. In the next section we propose an approach to establish the most appropriate class to which the object belongs.

4 Automatic Classification Approach

In the previous section we have proposed an approach to determine whether a semi-structured object is a weak member of a class. An object may be a weak member of several classes.

Example 6. Consider an object whose state consists only of the component `(age`: 25`)`. Such object is a weak member of all the subclasses of class `person` in the schema of Example 2. ◇

When an object is a weak member of several classes, we need some measures to determine the most appropriate class in which we can classify the object. If no class exists of which the object is a weak member, we insert it into a repository of unclassified objects. As the schema evolves the repository is periodically examined, trying to classify objects contained in it.

In the remainder of this section we propose two measures to select the most appropriate class where we can classify a given object, among those of which the object is a weak member. We also outline an algorithm using those measures to automatically classify semi-structured objects. Finally, we compute the algorithm complexity and present some examples of automatic classification.

4.1 Conformity Degree

With the first measure, referred to as *conformity degree*, we want to check how much the type of the semi-structured object is close to the structural type of a given class. In other words, we check how many components the class has in addition to those of the object. In case an object is a weak member of more than one class, we select the classes that have the minimal number of additional components with respect to the components in the object state. For example, if an object is a weak member of a class and it is a weak member of some subclasses of that class, we are not interested in classifying the object in the most specific class of the inheritance hierarchy if this class has several attributes which are not part of the object. To formally define the conformity degree, we introduce an additional data structure, referred to as *object structural expression*, representing the actual type of the object. This data structure, intuitively, is a subtree of the tree associated with the structural type of a class of which the object is a weak member. It is associated with a legal type of our type system and allows the actual type of the object to be compared with the structural type of the class, since the object structural expression is built starting from the class structural expression. Informally, to generate this structure we start from the existing simulation between the object value expression and the class structural expression and extract the vertices of the class structural expression that appear in the second component of the simulation. Then, we add to this set of vertices other vertices to handle two particular cases: the presence of null values in the object state and the presence of `spring` types in the structural type of the class. The edges and the labeling function of this tree are created accordingly. For further details on the formal definition of the object structural expression, that will be denoted by $\varepsilon(o, c)$, we refer the reader to Appendix A. Figure 5(a) shows the object structural expression associated with the object value expression shown in Figure 2(a). As we can see, this object structural expression represents the type `record-of(nameS:string, birthday:date)`. The value associated with the object value expression shown in Figure 2(a) is a legal value for that type. Moreover, to formally define the conformity degree, we must take into account that when there is a union type in the structural type definition of a class only one of its components may appear in the object state. Thus, we consider the *real paths* of a class structural expression. Real paths, formally defined in Appendix A, are paths that do not contain any edge labeled by $UNION$ followed by an edge labeled by l ($l \in \mathcal{AN}$) where l is an attribute not appearing in the object state. Figure 5(b) shows the tree only containing the real paths of the class structural expression shown in Figure 2(b). The following definition formalizes the notion of conformity degree.

Definition 10. *(*Conformity degree*). Let o be a semi-structured object and c be a class such that o is a weak member of c. We define the conformity degree of o with respect to c (denoted by $C^{\circ}(o, c)$), as the ratio of the number of paths of the object structural expression and the number of real paths of the class structural*

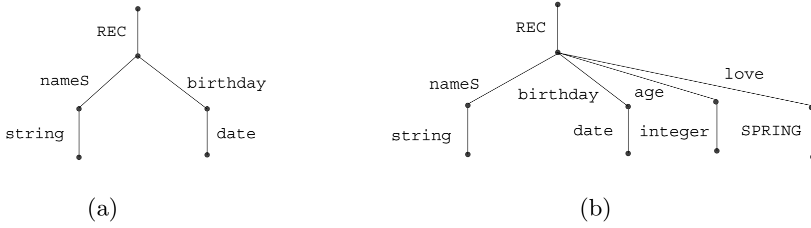


Fig. 5. (a) Object structural expression, and (b) the part of class `person` structural expression containing only the union components that appear in the object state

expression. Formally:

$$C^{\circ}(o, c) = \frac{\#(path(\varepsilon(o, c)))}{\#(real\text{-}path(\varepsilon_t(c)))}$$

□

In the previous example, the number of paths of the object structural expression is 2, the number of real paths of the class structural expression is 4, thus the conformity degree is 0.5.

The following proposition (proved in [8]) holds.

Proposition 1. *The following results on conformity degree and weak membership hold:*

- *The conformity degree is always a number between 0 (low conformity) and 1 (high conformity).*
- *If a semi-structured object is an instance of a class, the conformity degree is 1.*
- *If a semi-structured object is a weak member of a class and the conformity degree is 1, then the object is an instance of the class.* ○

4.2 Heterogeneity Degree

With the second measure, referred to as *heterogeneity degree*, we want to check how much the extension of a class is heterogeneous. By using the heterogeneity degree, we can insert a given object in the class with the most homogeneous extension. The advantage of having classes with a homogeneous extension is that more efficient query execution strategies and storage organizations are possible. In Section 2 we have seen that, because of the presence of union and `spring` types in the type system, several structures may correspond to the same type. In Section 2 we have also seen that for each class in the schema the set of objects

belonging to a class may have different structures. For example, if in the structural type of a class there is only a union type with two components, and there is no component of `spring` type, then the extension of this class consists of a set with two kinds of objects: the ones having the first component of the union type, and the ones having the second component of the union type. Thus the heterogeneity degree is 2.[6] By contrast, if we have only a component of `spring` type in the structural type of the class, this component may assume any legal value of any type in the schema. Thus, the structure of objects belonging to this class may be highly heterogeneous. In such case, the heterogeneity degree is evaluated as the number of all value and object types introduced in the schema (these sets are denoted by $\mathcal{VT}$ and $\mathcal{CI}$, respectively). The heterogeneity degree of a record type is the product of the heterogeneity degree of its components, while the heterogeneity degree of a set type (list type) is the heterogeneity degree of its component types. The heterogeneity degree of other types (belonging to the basic type system) is 1 since they do not generate heterogeneous extensions. In computing the heterogeneity degree we take into account that we perform a shallow comparison among the class structural expression and the object value expression. That is, if the type of an attribute in a class c is an object type $\bar{c}$, in calculating the heterogeneity degree we do not take into account the heterogeneity degree associated with the class $\bar{c}$, rather we state that its heterogeneity degree is 1. The following definition states how the heterogeneity degree of a class is computed.

Definition 11. *(*Heterogeneity degree*). Let $T = stype(c)$ be the structural type of class c, then the heterogeneity degree associated with c is the value returned by the following function applied to T.*

$$H^\circ(T) = \begin{cases} 1 & \textit{if } T \textit{ is a basic value type or object type} \\ n & \textit{if } T = \textit{union-of}(a_1 : T_1, \ldots, a_n : T_n) \\ \#\mathcal{VT} + \#\mathcal{CI} & \textit{if } T = \texttt{spring} \\ \prod_{i=1}^{m+n} H^\circ(T_i) & \textit{if } T = \textit{record-of}(a_1 : T_1, .., a_m : T_m, T_{m+1}, .., T_{m+n}) \\ H^\circ(T') & \textit{if } T = \textit{list-of}(T') \textit{ or } T = \textit{set-of}(T') \end{cases}$$

□

Note that the heterogeneity degree, like the class structural expression, does not depend on the database instances but only on the schema. Thus, the heterogeneity degree and the class structural expression may be computed at schema definition time. This is important in order to define efficient algorithms to classify objects in the schema.

4.3 Classification Algorithm

In our classification approach we look for a class such that: the semi-structured object is a weak member of the class with the highest conformity degree; the

[6] Note that, since the types of union type components are constrained to belong to the basic type system, their heterogeneity degree is always 1.

class has the lowest heterogeneity degree. In addition, for classes with the same conformity and heterogeneity degrees, we take into account the inheritance hierarchy, by choosing the most specific class in the hierarchy.

The classification algorithm takes as input a semi-structured object and executes the following steps:

1. The set of classes of which the object is a weak member is computed; such set is denoted as WMS. If $WMS = \emptyset$ then the object cannot be classified and it is simply inserted in the repository of unclassified objects. Otherwise,
2. The set of classes $WMS_{C\text{-}max}$ is extracted from the set WMS by choosing the classes with respect to which the object has the highest conformity degree. If this set is a singleton, the most appropriate class has been found and the object is inserted in the class extension. Otherwise,
3. The set of classes $WMS_{H\text{-}min}$ is extracted from the set $WMS_{C\text{-}max}$ by choosing the classes with the lowest heterogeneity degree. If this set is a singleton, the most appropriate class has been found and the object is inserted in the class extension. Otherwise,
4. We delete from $WMS_{H\text{-}min}$ all the classes having a subclass in that set, and any class c such that $WMS_{H\text{-}min}$ contains a class c' whose object structural expression[7] is a refinement of the object structural expression of c, according to Definition 6. If the resulting set is a singleton, then the most appropriate class has been found and the object is inserted in the class extension. Otherwise, an arbitrary class is selected in which the object is inserted.

In the previous algorithm, first of all we find out the set of classes having the highest conformity degree from the classes which the object is a weak member of. We use the conformity degree as the main measure in the classification approach because it allows one to identify the classes with the smallest number of attributes not present in the object state. At this point we try to minimize the heterogeneity degree. To select a class among the remaining classes, we choose those classes which are most specific in the inheritance hierarchy as well as those classes whose attribute domains most closely matches the attribute values of the object.

Note that, if the resulting set of the algorithm is not a singleton then an arbitrary class is selected in which the object is inserted. An alternative approach, which however is left for future investigation, would be to classify the object in all the classes in that set.

4.4 Complexity of the Classification Algorithm

In this section we present the complexity of our algorithm. The following notation is used:

[7] We recall that the object structural expression (formally defined in Appendix A) is the structural expression representing the portion of the class structural expression actually present in the object.

- C is the set of classes of the schema with respect to the object is being classified;
- k the number of classes in C;
- $dim(o)$ is the dimension of the object value expression associated with the object being to classified, that is, the number of vertices of the tree;
- $dim(c)$ is the dimension of the class structural expression associated with a class $c \in C$, that is, the number of vertices of the tree.[8]

The first step of the algorithm is the computation of the set of classes of which the object is a weak member. According to Definition 9, this is equivalent to determine whether a simulation exists between the object value expression and the class structural expression. In the computation of the simulation relationship (Definition 8), at each step, for each edge $u_1 \xrightarrow{l_v} u'_1$ of the object value expression we check whether an edge $u_2 \xrightarrow{l_t} u'_2$ exists such that one of the conditions of Definition 8 holds. Note that, since the time required to check whether one of the conditions of Definition 8 holds is constant, we have to compute how many times this step is iterated. Since, for both the object value expression and the class structural expression, the outgoing edges from a given vertex having an attribute name as label are distinct, no backtracking is needed during the iteration process, which, at most, repeats the check of the properties as many times as the number of edges of the object value expression. Since the number of edges in a tree is equal to the number of the vertices minus one, we can conclude that the first step has a cost in $O(dim(o))$ for each class, that is, in $O(k*dim(o))$ for all the classes of the schema. In the second step, first the conformity degree is computed for each class of which the object is a weak member. Because the number of such classes is at most k, the third step has a cost in $O(k * max_{c \in C}\{dim(c)\})$. After that, the set of classes with respect to which the object has the highest conformity degree, $WMS_{C\text{-}max}$, is computed. This step has a cost in $O(k)$. In the third step, the set of classes with respect to which the object has the lowest heterogeneity degree, among the ones in $WMS_{C\text{-}max}$, is computed. Such step has a cost in $O(k)$. In fact, the heterogeneity degree is an information which can be associated with a class when it is created, without overhead for the algorithm. Finally, in the fourth step, we compare all the remaining classes testing for inheritance and refinement. Supposing testing for subclassing constant [6], the step can be executed in $O(k * max_{c \in C}\{dim(c)\})$.

Therefore, $O(k * max\{dim(o),\ max_{c \in C}\{dim(c)\}\})$ is the total cost of the algorithm. Thus, the classification algorithm solves the problem in a time that is linear in the dimension of the entities involved in the classification process.

4.5 An Example of Classification

In this section we present some examples of the application of our classification algorithm. We classify the semi-structured objects that we have presented in

[8] Note that given an object o and a class c the object structural expression ($\varepsilon(o,c)$) is a subtree of the tree associated with the structural type of a class of which the object is a weak member, thus $dim(c)$ is an upper-bound of the dimension of $\varepsilon(o,c)$.

Example 3. Note that the other components of the schema are not relevant for the example, we only need to know that the number of different (value or object) types defined in the schema is 20. Based on the number of different types in the schema, we can compute the heterogeneity degree of the classes presented in Example 3.

- $H^\circ(stype(\texttt{mail})) = 20 * 2 * 1 = 40$,
- $H^\circ(stype(\texttt{internal_mail})) = 20 * 1 * 2 * 1 = 40$,
- $H^\circ(stype(\texttt{personal_mail})) = 1 * 1 * 2 * 1 = 2$,
- $H^\circ(stype(\texttt{business_mail})) = 1 * 1 * 2 * 1 = 2$.

When the algorithm presented in Section 4.3 is applied to object o_1, it computes, in step 1, the set $WMS = \{$`mail`, `internal_mail`, `personal_mail`, `business_mail`$\}$. Each attribute in the object state, indeed, is an attribute in the structural type of each class of the set WMS and the values are of the correct types. Since $WMS \neq \emptyset$ the second step of the algorithm is applied and the set $WMS_{C\text{-}max} = \{$`mail`$\}$ is determined. The classes `internal_mail` and `personal_mail` have been removed from the set $WMS_{C\text{-}max}$ because they have an attribute, `sender`, not in the object state, whereas the class `business_mail` has been deleted since it has two attributes, `sender` and `logo`, not in the object state. At this point, since the set WMS_{C-max} is a singleton, the object is classified in class `mail`.

When the algorithm is applied to classify object o_2, it computes, in step 1, the set $WMS = \{$`internal_mail`, `personal_mail`$\}$. It does not consider classes `mail` and `business_mail` because they do not contain the `subject` attribute. Then, the set of classes with the highest conformity degree ($WMS_{C\text{-}max}$) is determined, but this set is equal to the previous one (the two classes have the same attributes). Therefore, the set of classes with the lowest heterogeneity degree $WMS_{H\text{-}min} = \{$`personal_mail`$\}$ is computed. Thus, the object is classified in class `personal_mail`.

When the algorithm is applied to classify object o_3, it computes, in step 1, the set $WMS = \{$`internal_mail`, `personal_mail`, `business_mail`$\}$. It does not consider class `mail` because it does not contain the `sender` attribute. Then, the set of classes with the highest conformity degree ($WMS_{C\text{-}max}$) is determined, but this set is equal to the previous one (the three classes have the same number of additional attributes). Therefore, the set of classes with the lowest heterogeneity degree $WMS_{H\text{-}min} = \{$`personal_mail`, `business_mail`$\}$ is computed. Class `personal_mail` is not a subclass of `business_mail`, but if we consider the object structural expressions associated with object o_3[9] we find out that `business_mail` is a refinement of `personal_mail`, thus the object is classified in class `business_mail`.

When the algorithm is applied to classify object o_4, it determines that the object is not weak member of any class. Thus, the object is put in the repository of unclassified objects.

[9] The object structural expression associated with o_3 with respect to `personal_mail` and `business_mail` is shown in Figure 3.

5 Conclusions

In this paper we have proposed an approach to classify objects whose structure is not necessarily a type present in the database schema. The proposed technique, which is currently being implemented, is based on the notion of weak membership and on conformity and heterogeneity degrees, and allows one to automatically classify an object in the class whose structural type best fits the object state.

Our classification approach is totally based on the object structure. An alternative approach could be to classify objects according to their response to messages that they receive. We did not investigate such an approach because it is more related to object-oriented programming languages than to databases. In object-oriented databases the structure, rather than the behavior, is regarded as the most relevant information associated with objects. However, such an approach could still represent an interesting research direction.

Another interesting research direction is the development of suitable information extraction tools. Two approaches are possible. The first one is to define "*prototype documents*" with respect to which the documents are compared. The comparison of a given document against a prototype document allows one to infer structural information from the document. The second approach is to use decision trees with rules that specify conditions on the attribute types. A path in the decision tree may thus represent a particular type to which a set of objects may belongs to. We plan to investigate those approaches as future work.

We are extending the work presented in this paper along several other directions. First, we would like to consider the possibility of classifying an object in more than one class, rather than always forcing the selection of a single class. This could be useful when there are several classes of which the object is a weak member, with the same values for conformity and heterogeneity degrees. Moreover, our current notion of weak membership is based on the fact that the object state contains less components than those of the class. Such notion can be extended to the case of objects whose state contains additional components with respect to those specified in the class, in the same spirit of the O_2 exceptional instances [15]. In this way we can achieve a more accurate classification. Another possible extension could be that of allowing components to be dynamically added, or deleted, to the state of objects in the database. This could require a re-classification of the object, that is, a migration of the object in a more appropriate class. We would like to consider the possibility that the schema evolves, as a consequence of object classification. The applicability of the classification approach to Web search engines, to perform content-based queries will also be investigated. The idea is to define, starting from the query, the value to be searched on the Web, to associate a structural expression with HTML pages, and then to verify whether a simulation exists between the tree associated with the query and the tree associated with the HTML page. If the simulation exists then the HTML page is a possible answer for the query. Finally, we plan to investigate how semi-structured objects can be handled by application programs and queried, in the context of semi-structured data, by revisiting type checking notions.

References

1. S. Abiteboul. Querying Semi-Structured Data. In F. Afrati and P. Kolaitis, editors, *Database Theory - ICDT'97*, pages 1–18, 1997.
2. S. Abiteboul, S. Cluet, and T. Milo. Correspondence and Traslation for Heterogeneous Data. In F. Afrati and P. Kolaitis, editors, *Database Theory - ICDT'97*, pages 351–363, 1997.
3. S. Abiteboul, R. Motwani, and S. Nestorov. Inferring Structure in Semistructured Data. *In Proc. Workshop on Management of Semistructured Data, SIGMOD Record*, 26(4):39–43, 1997.
4. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Journal of Digital Libraries*, 1(1):68–88, 1996.
5. S. Abiteboul and V. Vianu. Queries and Computation on the Web. In F. Afrati and P. Kolaitis, editors, *Database Theory - ICDT'97*, pages 262–275, 1997.
6. R. Agrawal, A. Borgida, and H. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 253–262, 1989.
7. P. L. Bergstein and K. J. Lieberherr. Incremental Class Dictionary Learning and Optimization. In P. America, editor, *Proc. Fifth European Conference on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, pages 377–396, 1991.
8. E. Bertino, G. Guerrini, I. Merlo, and M. Mesiti. An Object-Oriented Data Model for Semi-Structured Data. Technical Report DISI-TR-99-06, University of Genova, Department of Computer Science (DISI), 1998.
9. R. Breitl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 283–308. Addison-Wesley, 1989.
10. P. Buneman. Semistructured Data. In *Proc. of 6th ACM SIGACT-SIGMOD-SIGART Symposium on PODS*, pages 117–121, 1997. Tutorial.
11. P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In F. Afrati and P. Kolaitis, editors, *Database Theory - ICDT'97*, pages 336–350, 1997.
12. P. Buneman, S. Davidson, D. Suciu, and G. Hillebrand. A Query Language and Optimization Techniques for Unstructured Data. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 505–516, 1996.
13. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 313–324, 1994.
14. S. Cluet. Modeling and Querying Semi-Structured Data. In M. T. Pazienza, editor, *Information Extraction. LNAI 1299*, pages 192–213, 1997.
15. O. Deux et al. The Story of o_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.
16. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.
17. R. Goldman and J. Widom. Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. Twentythird Int'l Conf. on Very Large Data Bases*, pages 436–445, 1997.

18. G. Guerrini, E. Bertino, and R. Bal. A Formal Definition of the Chimera Object-Oriented Data Model. *Journal of Intelligent Information Systems*, 11(1):5–40, 1998.
19. J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting Semistructured Information from the Web, 1997. Available via anonymous ftp at ftp://db.stanford.edu/pub/paper/extract.ps.
20. M. Henzinger, T. Henzinger, and P. Kopke. Computing Simulation on Finite and Infinite Graphs. In *Proc. of 20th Symposium on Foundations on Computer Science*, pages 453–462, 1995.
21. S. Holzner. *XML Complete*. McGraw-Hill, 1998.
22. R. Milner. An Algebraic Definition of Simulation between Programs. In *Proc. of the 2nd IJCAI*, pages 481–489, London, UK, 1971.
23. S. Nestorov, S. Abiteboul, and R. Motwani. Extracting Schema from Semistructured Data. In L. M. Haas and A. Tiwary, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 295–306, 1998.
24. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proc. of the 11th Int'l Conf. on Data Engineering*, pages 251–260, 1995.
25. C. Peltason, A. Schmiedel, C. Kindermann, and J. Quantz. The BACK System Revisited. Technical Report KIT - Report 75, Technische Universitat Berlin, 1989.
26. F. Rabitti. *The Multos Document Model*, volume Human Factors in Information Technology of *6*, chapter 3, pages 17–52. North-Holland, 1990.

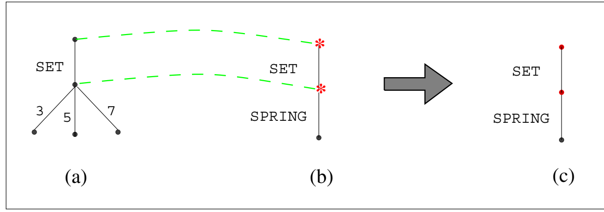


Fig. 6. Presence of *SPRING* label in a part of class structural expression

A Additional Formal Definitions

In this section we present the formal definition of object structural expression and two figures that illustrate two particular cases, that is, the `spring` type and the null value, discussed in Section 4. The figures illustrate how to obtain, starting from the object value expression (a) and the class structural expression (b), the object structural expression (c). In the following definition, given a tree A and a vertex u, we denote by $vertex(A)$ the set of vertices of A and with $tree(u, A)$ the subtree of A rooted at u.

Definition 12. *(*Object structural expression*). Let o be a semi-structured object and c be a class in the schema such that o is a weak member of c, that is, a simulation $\mathcal{R}$ exists between $\varepsilon_v(o) = (V_v, E_v, \varphi_v)$ and $\varepsilon_t(c) = (V_t, E_t, \varphi_t)$, then we define* object structural expression*, denoted by $\varepsilon(o, c)$, the following tree:*

$$(V, E, \varphi)$$

where:

- $V = \overline{V} \cup \overline{\overline{V}} \cup \overline{\overline{\overline{V}}}$
  - $\overline{V} = \{u \mid (u_1, u) \in \mathcal{R}\}$,
  - $\overline{\overline{V}} = \{u \mid (u_1, u_2) \in \mathcal{R},\ (u_2, u) \in E_t \text{ and } \varphi_t((u_2, u)) = SPRING\}$
  - $\overline{\overline{\overline{V}}} = \bigcup_{(u_1, u_1') \in E_v \text{ s.t. } \varphi_v((u_1, u_1')) = NULL} \{u \mid (u_1', u_2') \in \mathcal{R} \text{ and } u \in vertex(tree(u_2', \varepsilon_t(c)))\}$
- $E = \overline{E} \cup \overline{\overline{E}}$
  - $\overline{E} = \{(u_1, u_2) \mid u_1, u_2 \in V, (u_1, u_2) \in E_t\}$,
  - $\overline{\overline{E}} = \{(u_1, u_2) \mid u_1, u_2 \in V \text{ and } \exists u \text{ s.t. } (u_1, u), (u, u_2) \in E_t, \varphi_t((u_1, u)) = UNION)\}$
  - $\varphi((u_1, u_2)) = \begin{cases} \varphi_t((u_1, u_2)) \text{ if } (u_1, u_2) \in \overline{E} \\ \varphi_t((u, u_2)) \text{ if } (u_1, u_2) \in \overline{\overline{E}} \text{ and } (u, u_2) \in E_t \end{cases}$

□

We introduce the following definitions to formally state the concept of real path.

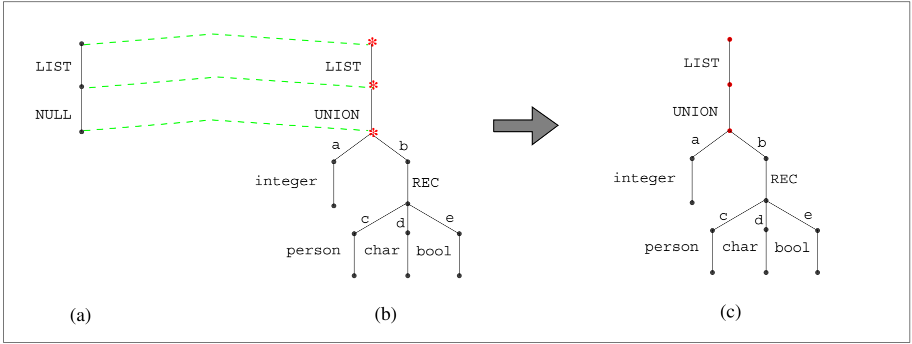


Fig. 7. Presence of $NULL$ label in a part of object value expression

Definition 13. (Labeled path). *Let $A = (V, E, \varphi)$ be a labeled tree on $\mathcal{LT}$, the sequence:*

$$u_1.l_1.u_2.l_2.\cdots.l_{n-1}.u_n$$

is a labeled path, where:

- $u_1, \ldots, u_n \in V$,
- $\forall i,\ 1 \leq i \leq n-1,\ (u_i, u_{i+1}) \in E$ *and* $\varphi((u_i, u_{i+1})) = l_i$. □

Definition 14. (Maximal labeled path). *Let $A = (V, E, \varphi)$ be a labeled tree on $\mathcal{LT}$, a labeled path*

$$u_1.l_1.u_2.l_2.\cdots.l_{n-1}.u_n$$

is called maximal if $u_1 = root(A)$ and u_n is a leaf of A. □

Now we are able to define the notion of real paths of a tree. In the following definition we denote by $path(A)$ the set of maximal labeled paths of the tree A and with $\pi_2(\mathcal{R}) = \{u \mid (\bar{u}, u) \in \mathcal{R}\}$.

Definition 15. (Real paths). *Let o be a semi-structured object and c be a class in the schema such that o is a weak member of c, that is, a simulation $\mathcal{R}$ exists between $\varepsilon_v(o)$ and $\varepsilon_t(c)$, then we define real-path the following set:*

$$\begin{aligned}&real\text{-}path(\varepsilon_t(c)) = \\ &\quad \{\omega \mid \omega \in path(\varepsilon_t(c)) \text{ and } (\omega = \ldots REC.u_1.UNION.u_2.l.u_3 \ldots \Rightarrow \\ &\quad (u_3 \in \pi_2(\mathcal{R})) \text{ or } (\{u \mid (u_2, u) \text{ is an edge of } \varepsilon_t(c) \text{ and } u \in \pi_2(\mathcal{R})\} = \emptyset))\}\end{aligned}$$

□

Object-Oriented Programming on the Network

Jim Waldo

Sun Microsystems, Inc., 1 Network Drive, Burlington, MA, 01803,U.S.A.
jim.waldo@sun.com

Abstract. Object-oriented programming techniques have been used with great success for some time. But the techniques of object-oriented programming have been largely confined to the single address space, and have not been applicable to distributed systems. Recent advances in language technology have allowed a change in the way distributed systems are constructed that does allow real object-oriented programming on the network. But these advances also change some of our most basic conceptions about the relationship between processor and code, and what it is that constitutes a computer. We will argue that a new computing architecture, based around the ideas of the network and full object-orientation, will soon become the dominant computing architecture, allowing us to tie together large numbers of devices but requiring that we think and design in entirely new ways.

1 Object-Oriented Systems

Over the past fifteen years, object-oriented programming has moved from the research lab to the mainstream. The use of object-oriented techniques is rarely questioned these days; organizing a program as a group of interacting components, each of which is treated as an object, is now the default way of designing any program of any size. To do otherwise is sometimes necessary, but that necessity must be justified; the default way of organizing a program or system of any size is to use object-oriented techniques.

Of course, different people have different notions of what it is for a program or sysem to be constructed along object-oriented lines. To program in an object-oriented fashion is more than using a language that labels itself as an object-oriented language. All of us have seen many C programs with little or no object-orientation written in either C++ or Java. Some of the better object-oriented programs I have had the pleasure of reading have been written in languages that are not themselves considered object-oriented. Real object-orientation is a design philosophy, not something that is automatically gained by choosing a language.

Given the wide variety of what is meant by „object-oriented programming", perhaps I should start by heeding the advice of Humpty Dumpty in *Through the Looking Glass*[1] and say what I mean when I use the term. The object-oriented approach to programming rests, I believe, on a small number of principle.: The first of these is the principle of abstraction- that the actual representation of an object is an accidental property of the object, with the essence of the object being shown by the operations that can be performed on that object.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 441-448, 1999.
© Springer-Verlag Berlin Heidelberg 1999

This first principle gets stated in a number of different ways. Sometimes this principle is stated as the independence of an object's interface from the implementation of the object. Other times it is stated as a principle of data hiding. By whatever name, it is at the base of object-oriented programming. It is by following this principle that we insulate ourselves from changes in the underlying data representation of an object, allow alternate implementations of the same interface, and compartmentalize our code to make it more manageable and, we hope, more stable.

The second principle underlying the power of object-oriented programming is the binding of behavior with data. This may seem like a variant of the abstraction principle described above, but in fact it is quite different. The ability to associate code with an object (or a class of objects) is necessary for abstraction, but goes far beyond what is needed for abstraction. Since an object can exhibit arbitrarily complex behavior, and since that behavior can be changed over time, the binding of object and behavior does more than just abstract the information contained by the object. In a real sense, what this association allows is a distinction between what an object does and how it goes about doing it. All a user of an object needs to know is the what of an object; the how is a private matter to the object itself, which can change without the client changing.

The third principle underlying the power of object-oriented programming is what I will refer to as polymorphism. This is often described as the ability of an object to have multiple forms. I take it to be a somewhat more interesting principle, centering around the ability to describe an object in terms of the necessary conditions on the object. When a client of an object wishes to describe the kind of object in which it is interested, it needs only say what the necessary conditions are for the object. Beyond those conditions, the object can vary in any way at all. This is more than an object having many forms. It is also the ability of users of an object to only need to specify one of the forms of the object, and allowing variation beyond that particular form.

These three principles are, I believe, all that there is to the core of object-oriented programming. There are lots of other characterizations of the subject (having to do, for example, with reuse, or design patterns, or whatever), but they all come down, in the end, to some variant or combination of these three.

While these principles and their statement may be somewhat idiosyncratic (and are also, no doubt, imprecise and perhaps incomplete), they are not new. These kinds of points have been made about object-oriented programming for years. So why do I bring them up?

I bring them up because my work has been in the combined fields of object-oriented programming and distributed systems. And in distributed systems, most of what gives power to object-oriented techniques has, until quite recently, been unavailable. While there has been lots of talk about object-oriented distributed systems, in fact such systems have been impossible under the usual assumptions that have governed distributed computing.

2 Distributed Systems

Distributed systems would seem to be a natural area for the application of object-oriented techniques. A distributed system, roughly speaking, is any system containing multiple computers that communicate over a network and share state[2]. The processes that run on these computers can generally be thought of as objects, each of

which has an interface that is used by other objects on other machines to make requests. Since the processes are separated by the network, there is already a hard and enforceable abstraction boundary.

Indeed, distributed systems often describe themselves as object-oriented, pointing out that they require interfaces, do not allow access to internal state, and must be constructed in a way that insures that the implementation of a distributed object can be changed without the client using that distributed object through the interface ever knowing. When viewed from enough distance, this claim seems reasonable. The objects tend to be large, and they tend to be rather static, but they are objects none the less, defined by an interface and accessed through that interface.

However, if we look at the kinds of interfaces that are presented by the objects in most distributed object systems, they have a decidedly non-object-oriented flavor. The information passed between these distributed objects is limited in rather sever ways. In most common distributed object frameworks, the information that can be passed from one object to another is limited to a small number of primitive data types, or references to other distributed objects, or structures made up of these types and references. One doesn't see objects being passed from one place in a distributed system to some other place in that system. And this seems strange, because if you look at examples of object-oriented interfaces in non-distributed systems, the vast majority of the information passed from one object to another object as a parameter or a return value in a method call is another object.

The reason for this oddity is both simple and deep. When doing distributed computing, there are a certain set of assumptions that have been made that define the environment in which distributed computing takes place. Because of these assumptions, the information that can be passed in such systems is severely limited.

The assumptions that I am talking about revolve around the presumption that a distributed system is made up of heterogeneous parts. The processors in a distributed system may be of different types, with different instruction sets and different data layouts. The operating systems on the machines built around those processors may be different. The languages that are used in implementing the objects that are communicating may well be different, and have been assumed to be compiled languages, that produce object code that is specialized for the instruction set of the processor and, perhaps, operating system in which the program is going to run.

Given this set of assumptions, it is rather remarkable that any distributed system has ever run at all. In an environment this hostile, getting any information from one process to another on some other machine is nothing short of miraculous. And the miracle has occurred because, no matter what sort of object-oriented veneer we have been able to put on our distributed systems, those systems have been able to work because at the most fundamental level they have not been object-oriented. Indeed, at the most fundamental level, those systems have been ways of defining the lowest level wire protocols imaginable.

To see this, all we need to do is to reflect on what really goes on in a system like CORBA[3] or DCOM[4]. Interfaces are defined in an interface definition language, where method calls are defined in terms of the primitive data types, object references, and structures of these entities that are passed back and forth for the interface. These interfaces are then compiled, for any given implementation language, with the result being stub and skeleton files. Once these are augmented with code that needs to be provided by a human programmer, they can be compiled for the target operating system and processor.

The result is object code for the particular processor and operating system that will take a call from a client, translate that call into the wire protocol, and sent those bits across the wire. Once received, the skeleton will translate the known bits into something that can be understood in the context of the receiver, make a call to the server, and then translate any results into the wire protocol that can be sent back to the original client. These known bits on the wire can be reformulated into something known by the client, and the call if finished.

While this is a great convenience over forming the bits on the wire by hand, it is hardly object-oriented. The information passed between the participants cannot change without both the client and server being updated simultaneously, since each must know exactly what is transmitted between the two parties. There is no notion of knowing a minimal set of conditions about what is transmitted; what is needed is exact knowledge. There is isolation, but the isolation is bought at the price of a static interface, hard to update or change. There is certainly no notion of behavior being sent from one participant in the distribute system to another.

Given the assumptions for distributed computing, it can hardly be otherwise. Since there is no single implementation language that can be assumed, all that can be passed from one distributed object to another are things that are common to all programming languages. Given the assumption of compiled code and different processors, only data can be handed from one distributed object to another, since code must be assumed to be local to a particular machine. While we can model the network as a whole as a set of cooperating objects, the way in which those objects cooperate and communicate is decidedly non-object-oriented.

3 Adding Objects to the Network

The advent and wide adoption of the Java(tm) programming language[5] and environment[6] has provided an opportunity to examine what can be done when the basic assumptions of distributed computing no longer hold. By assuming that the Java environment is available on every member of a distributed system, we get a very different network environment in which to develop our object-oriented techniques.

To begin with, the Java environment provides a homogeneous layer on top of the heterogeneity of the distributed system. Rather than thinking in terms of particular processors and operating systems, the Java environment provides a virtual machine that is (more or less) the same everywhere. While this property has generally been understood as giving the ability to install the same code on different processors, in the distributed system context this should be understood as offering the same system everywhere.

This homogeneity allows the second great advantage of the Java environment as a distributed computing environment- the ability to move code from one machine to another. The portable binary code that Java provides means by which behavior can be moved around the network, which in turn allows real objects (both code and data) to become network citizens.

Of course, there are other ways of providing an environment that is both homogeneous and has portable binary code- if all of the machines on a network are of a single processor type, running the same (or compatible) operating systems, then code can be moved around. The additional aspect of the Java environment that is as important as the other two is that the language is safe and the code that is moved (and dynamically

loaded) can be verified to follow the rules imposed by the language. This notion of safety is something that must be a part of the general distributed computing environment, and cannot be provided by simply having the same machine type and operating system everywhere.

The combination of a homogeneous environment, portable object code, and verifiable safety enabled Applets, which in turn brought the Java language to the attention of most programmers. By supplying objects that implemented a well-known interface, and making sure that browsers for the World-Wide Web knew how to recognize such objects and invoke the appropriate methods in that well known interface, Java allowed the introduction of active content to the world wide web.

Applets were the first application of object-oriented principles to a large scale network. By allowing different implementations to be offered for the same (known) interface, and moving that implementation into the browser that made the calls to that interface, Java Applets allowed alternate implementations of a known interface to be used to extend existing programs in interesting ways.

By taking one more step, the application of object-oriented techniques to the network could be made more general. By rejecting the usual assumption that the programming language in which a distributed object doesn't matter and instead assuming that all of the objects in a distributed system are written in the Java programming language (and running in the Java environment), the Applet approach could be generalized to any distributed program.

This ability was introduced into Java at JDK 1.1 with the introduction of the Java Remote Method Invocation system (RMI)[7]. RMI allows Java objects to be passed from one Java virtual machine to another, even when those virtual machines are located on different physical machines. Further, RMI will pass an object by its true type, not the type that is declared in the method signature of the remote method. This allows the passing of subtypes to methods declared to use a supertype. If the receiving virtual machine does not have the code associated with the actual class of the object that it receives, the code for that class is downloaded, verified, and dynamically loaded into the receiving virtual machine.

This is done for both local and remote objects. For a local object, this means that any calls made to a new subtype of thedeclared type that is passed in to the address space of the receiving machine will invoke the new behavior (if any) associated with the subtype. This allows many of the usual object-oriented patterns, previously confined to a single address space, to be used in a distributed system. This also means that a program can be updated dynamically, by receiving new implementations of classes as objects associated with those new implementations become available.

For remote objects, the case is somewhat different but just as important. The code that is received for a remote object is, effectively, the stub for that object. This stub will reflect all of the remote interfaces supported by the remote object. Using the standard Java type mechanisms and reflection, these interfaces can be discovered and used. Further, since the stub code can be downloaded, there is no need for the client to ever have to worry, either directly or indirectly, about the actual wire protocol used to communicate with the remote object. This protocol becomes a private matter between the remote object and the stub that is provided by that object. If the protocol changes, the client never needs to know--it simply makes the same calls to a local object (the downloaded stub) that has a different implementation.

This is the base functionality that has allowed the third stage of object-oriented network programming based on the Java environment, the Jini(tm) system[8]. The Jini architecture is based on the ability to move code from one machine to another, and an

exploitation of the Java type system as a way of identifying services that can be used by other members of the distributed system.

The Jini infrastructure extends the basic Java platform (including RMI) by adding a component, called the Lookup Service, that allows services to advertise themselves, and a simple protocol that allows these services and clients wanting to find a service to first find a Lookup Service. By using this protocol, joining a Jini network can be virtually automatic. A client or service wishing to find a Lookup Service sends out (via multicast) a known packet. Any Lookup Service receiving this packet will reply (to an address contained in the packet) with an implementation of the interface to the Lookup Service itself.

Once this object has been received (with, perhaps, the code for the particular implementation being downloaded), an object wanting to offer a service registers with the Lookup Service. This registration consists of handing the Lookup Service an object that can be used to access the service, along with an optional set of attributes that can be used to identify the service in ways other than its Java type.

A client wanting to use a service finds a Lookup Service using the same discovery mechanism, and also receives an object that implements the Lookup interface. Rather than registering a service, however, a client will request a service. It does this by specifying the Java type of the service that it wishes to use, along with any possible attributes that might make it want to choose between different services that implement the same interface.

On receiving such a request, the Lookup Service will return a copy of the object placed in it by the service (assuming that the object implements the interface that was requested by the client). This object may actually implement a subtype of the interface requested, or other interfaces in addition to the one requested. This can be discovered by the usual Java type and reflection mechanisms. The important thing is that, when the client receives the object, any code that might be needed by the client for the object implementation will be downloaded into the client if it is not there already.

The effect this has is that the client contacts the service using code supplied by the service. How this contact is done is up to the original service; all the client needs to know is the interface (indeed, the minimal interface) needed to talk to the service. Different implementations of the same service can use different communication mechanisms, which are encapsulated in the implementation of the object obtained from the Lookup Service.

The Jini system does not require that there be a single Lookup Service. Many Lookup Services can co-exist on the same network, perhaps differentiated by the name of the group of which they are a part. Services can register in one, many, or all of the Lookup Services that they find, depending on the local policy of the service.

4 A Network Centric Computing Architecture

While each step from standard distributed computing to the Java environment as a platform for the World Wide Web to RMI as a general platform for distributed computing and to the Jini system as a way of organizing services and clients on a network seems obvious in retrospect, the impact of the changes that have gone on is fairly revolutionary. The kind of environment allowed by the Jini system is a radical break with the assumptions that have been held constant in computer architectures for

the past 50 years.

This standard computer architecture was based on the tight connection between a processor and a mass storage device. The software that would be run on the processor was assumed to be located on the mass storage device, and therefore that software could be specialized for the kind of processor on which it would run. Low level languages (such as C) could be translated by compilers into the appropriate set of instructions for a particular processor. Such binary code would be installed on the appropriate computer, and installing a program on a processor that was not the same as the one for which the program was compiled was a way of insuring that, at best, the program would not run.

This tight connection between the processor and the storage device from which the processor would obtain all of the code run on the processor even had its effect on what we took to be computers. A computer had to have both a processor and some form of storage for its programs. Other devices might be attached as well, but these were peripheral to the core function of the computer. This tie is apparent in the very boot sequence of our computers--after running a set of built-in diagnostics, the first thing a computer will do is look for its local disk, from whence it will load the programs that continue the boot process.

Defining a computer in this way actually excludes a large number of the devices that are currently being produced that contain microprocessors. These devices, from personal digital assistants and cellular phones to automobiles and kitchen appliances, often contain considerable computational power, often more than was available in personal computers of less than a decade ago. But they do not contain much mass storage. What little code is used to run these devices resides in a persistent store that is limited in size and expensive.

However, many of these devices are either capable of being connected to a network or are already connected to some network. A cellular phone is an indication of what, I believe, is soon to be the common case. When such a device boots, it runs through some basic diagnostics. But it then looks not for local mass storage, but for a network. It is from the network connection that the device is able to offer value and give service. But it can also be from the network that the device can locate the code that is needed to expand the services offered by the device, or offer code to other devices to allow those devices to make use of its service.

Using techniques similar to those in the Jini architecture, we can move code into these devices to allow them access to new services over the network, and move code out of these devices to allow them to offer services to others on the network. This in turn greatly expands the notion of what a network citizen can be. Rather than thinking of networks as the domain of the desktop system and the large server, providing services in business and engineering, the network becomes a substrate for electronic communities that span all of our different areas of life, connecting our homes, cars, offices, and selves to communications infrastructure of untold size and diversity.

This change could be a true paradigm shift, in the original sense described by Kuhn[9] as opposed to the more popular sense of „the next new thing." As a paradigm shift, it is a change the way we view the world, and the very questions that we ask about that world. Issues of privacy and security, ignored for so long because of their difficulty, cannot be ignored if the internet reaches into our home appliances and our methods of transportation. Issues of resource management and resource discovery will need to be addressed that can, and have, been ignored by desktop and server systems that can assume a large local disk and virtual memory. The questions of how one tests and verifies a system that receives its code as needed rather than through

explicit installation will need to be addressed.

These are not the questions that concern us now, since we have been living under a different paradigm. But the world is changing. We have been living in a world of machines that have formed stand-alone islands of functionality, perhaps being tied to a network for some communication to other machines. We are moving to a world in which the network is necessary for the functioning of most systems, where the stand-alone machine will be a minority member of the computing world. Code and data will move around the network, allowing us to update parts of the network in a dynamic fashion, with clients getting the updates transparently when they are needed.

We are only beginning to understand the scope of this new world, and are only starting to ask the questions that need to be answered to make this world a reality. There will still be much to do in the context of the old world, and work in that paradigm will continue for some time. But the new paradigm will, I believe, begin to drive both the market and the technology. The world of the new paradigm is more varied and rich than the world that we now inhabit, and the possibilities are both enticing and challenging. Our users (and the market) have shown us that the network world is where they wish to be. It is up to us to define that world, and make it as rich as possible.

References

1. Carroll, L., Through the Looking Glass, in The Complete Works of Lewis Carroll, The Modern Library, (1954)
2. Schroeder, M., A State-of-the-Art Distributed System: Computing with BOB, in Mullender, S. (ed.), Distributed Systems, Addison-Wesley (1993) 1-16
3. The Object Management Group, Common Object Request Broker: Architecture and Specification, OMG Document Number 91.12.1 (1991)
4. Rogerson, D., Inside COM, Microsoft Press (1997)
5. Gosling, J., Joy, B., and Steele, G., The Java Language Specification, Addison-Wesley (1996)
6. Lindholm, T. and Yellin, F., The Java Virtual Machine Specification, Addison-Wesley (1997)
7. Wollrath, A., Riggs, R., and Waldo, J., A Distributed Object Model for the Java System, Computing Systems, Volume 9 Number 4, (1996) 265-290
8. Sun Microsystems, Jini Technology Architecture Overview, http://www.sun.com/jini/whitepapers/architecture.html
9. Kuhn, T., The Structure of Scientific Revolutions, The University of Chicago Press, (1962)

Providing Fine-Grained Access Control for Java Programs*

Raju Pandey and Brant Hashii

Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis, CA 95616
{pandey, hashii}@cs.ucdavis.edu
http://pdclab.cs.ucdavis.edu/

Abstract. There is considerable interest in programs that can migrate from one host to another and execute. Mobile programs are appealing because they support efficient utilization of network resources and extensibility of information servers. However, since they cross administrative domains, they have the ability to access and possibly misuse a host's protected resources. In this paper, we present a novel approach for controlling and protecting a site's resources. In this approach, a site uses a declarative policy language to specify a set of constraints on accesses to resources. A set of code transformation tools enforces these constraints on mobile programs by integrating the access constraint checking code directly into the mobile program and resource definitions. Because our approach does not require resources to make explicit calls to a reference monitor, it does not depend upon a specific runtime system implementation.

1 Introduction

There is increasing interest in computing models that support migration of programs. In these models, a program migrates to a remote host, executes there, and accesses the site's resources. For instance, Java [2] programs are increasingly being used to add dynamic content to a web page. When a user accesses the web page through a browser, the browser migrates Java programs associated with the page and executes them at the user's site. There are many other computing models that support mobility of programs. For example, the remote evaluation [34] model supports program migration by allowing one to upload a

* This work is supported by the Defense Advanced Research Project Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0221. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Project Agency (DARPA), Rome Laboratory, or the U.S. Government.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 449–473, 1999.
© Springer-Verlag Berlin Heidelberg 1999

program to a remote site. The mobile programming model [3, 35] supports general purpose mobility that also allows programs to migrate to other sites during their executions. The common element in all of these models is the ability of a runtime system to load externally defined user programs and execute them within the local name space of the runtime system.

Although appealing [4] from both system design and extensibility points of view, mobile programs have serious security implications. Mobile programs have the ability to maliciously disrupt the execution of programs at a site by reading and writing into their name spaces, by using unauthorized resources, by overusing resources, and by denying resources to other programs. For instance, the "Ghost of Zealand" Java applet misuses the ability to write to the screen: It turns areas of the desktop white, making the machine practically useless until it is rebooted.[1] Another example is Hamburg's Chaos Computer Club[2] demonstration of the dangers of using ActiveX [6]. ActiveX is Microsoft's mobile program technology which allows components to be dynamically installed on a user's desktop. The victim uses Internet Explorer to visit a web page that downloads an ActiveX control. The ActiveX control checks to see if Quicken, a financial management software, is installed. If it is, the control adds a monetary transfer order to Quicken's batch of transfer orders. When the victim next pays the bills, the additional transfer order is performed. All of this goes unnoticed by the victim, until she receives her statement.

In this paper, we focus primarily on a specific security problem associated with mobile programs, namely the access control problem. The access control problem involves allowing a site to control a mobile program's ability to access local resources. Many operating systems [17] implement a notion of access control by limiting accesses to specific resources that the operating systems administer. For instance, in the UNIX operating system, the owners of files can control the accessibility of their files.

The access control problem in the mobile programming domain differs from the traditional access control models in many ways. First, there is no fixed set of resources that a site can administer; different sites may define different resources. An access control mechanism cannot be based on controlling accesses to specific resources. The mechanism should be applicable to any resource that a host may define. Second, the access control model should allow the customization of access control policies from one site to another, one mobile program to another, and one resource to another. Third, the access control model should support a fine-grained access control specification. In many access control models, access control involves either allowing an access or completely denying it. In the mobile programming domain, we argue for a *conditional access control* model where accesses to resources can be based on a boolean expression [26]. In other words, a site may allow a mobile program to access resources if certain conditions are met.

[1] For full details see http://www.finjan.com/applet_alert.cfm or http://www.internetworld.com/print/1998/05/11/webdev/19980511-hostile.html.

[2] For full details see http://www.iks-jena.de/mitarb/lutz/security/activex.hip97.html or http://www.iks-jena.de/mitarb/lutz/security/activex.en.html.

These conditions may depend on the state of mobile programs, state of resources, runtime system state and/or security state. For instance, a database vendor may specify that if there are more than 20 mobile programs in the system, each mobile program can only access its database up to ten times. In this example, a mobile program's ability to access the database depends on a runtime system state, such as the number of mobile programs running, and a security state, i.e. the number of times mobile programs access the database.

Access control specification and enforcement have been studied in great detail. The different approaches can be broadly classified into three categories: *operating system-based*, *runtime system-based*, and *language-based*. In the operating system-based approaches [17, 1], an operating system implements a specific access control model which specifies how system-wide resources such as the network, files, and displays can be accessed. The operating system enforces the security policy by checking whether the type of access is allowed. In runtime system-based approaches [10, 13], a runtime system enforces specific controls over accesses to various objects. Each method first calls a reference monitor which checks to ensure that the method call is permitted. In language-based techniques [11, 37, 31, 21] access control policies are specified along with a program specification. A compiler not only generates code for the program but also code to enforce security policies.

In this paper, we present an *alternate* approach for specifying and enforcing access control over mobile programs written in Java. Specifically, the paper describes the following:

- We present an access control model for specifying how accesses to resources can be controlled. In this model, a site defines a set of access constraints, each specifying the condition under which a specific resource can be accessed.
- We present a novel access constraint enforcement mechanism in which access constraints are enforced by integrating access constraint checks directly into mobile program code and resource code before they are loaded into the runtime system.

Separating the specification of access constraints from the specification of Java programs and resources has the following implications:

- Resource developers do not need to manually insert calls to security checking code inside each resource that a host may want to protect. Further, the access control mechanism can be used to define and enforce access constraints to systems that were not designed with security in mind, such as legacy systems.
- Both resource definitions and access constraints can be modified independently without affecting each other's implementation.

We have implemented a version of this mechanism for programs represented using Java bytecode [25]. The performance results show that the overhead of this approach is moderate. Further, it performs better than the approach implemented in the Java runtime system in many cases.

This paper is organized as follows: Section 2 contains a description of our resource access model and how accesses to various resources can be specified.

Section 3 describes an implementation of this model. Section 4 contains an analysis of the approach, including its performance behavior. Section 5 contains a brief survey of related work. Section 6 contains a summary of the approach and discussion of future work.

2 Access Control Model

The access control model contains two parts: a resource model for representing resources and an access constraint specification language. We describe the two in detail below.

2.1 Resource Model

A site provides many resources to a mobile program. These resources include classes for utility libraries, accessing files, networks, and interfaces to other resources such as a proprietary database. For instance, a site providing access to a weather database exports a set of interfaces that specify how the database can be accessed. In our security model, each Java class or method represents a resource and, thus, is a unit of protection. Our access control mechanism does not differentiate between system classes and user-defined classes, or between locally defined classes and classes down-loaded from remote hosts. The model also allows the definition of class-subclass relationships among resources using the Java's inheritance model.

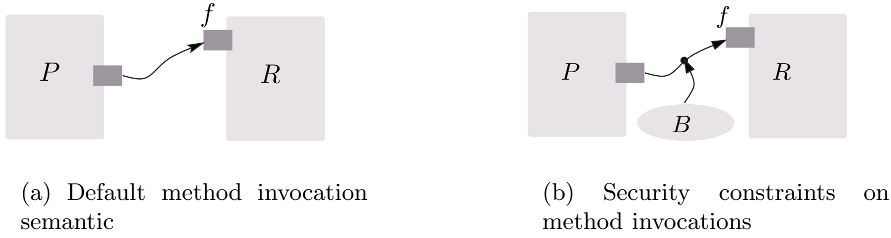
2.2 Access Constraint Specification Language

The access constraint specification language contains two parts: a notation for specifying constraints over accesses to resources and an inheritance model for access constraints.

Access Constraints: We first describe the motivation behind our access control language. A Java program uses a resource by invoking its methods. In Fig. 1(a), we show that program P invokes a method f to access resource R. During an execution of P, the control jumps to f, executes f, and returns back to P upon termination. The Java compiler implements a simple access semantics in which there are no constraints on accesses to R through f.

Our approach is to allow a host to make the access relationship between P and R *conditional* by adding a constraint, B (see Fig. 1(b)). The access constraint is specified *separately* from both P and R and has the effect of imposing the constraint that P can invoke f on R only if condition B is true. A site, thus, restricts accesses to specific resources by enumerating a set of access constraints, which forms a site's access control policy.

Below, we present only the core aspects of the language. For brevity we have omitted the details regarding specification of security constraints over groups of classes, methods and objects. The following EBNF shows how a site can specify access constraints:



(a) Default method invocation semantic

(b) Security constraints on method invocations

Fig. 1. Method invocation semantics

```
Constraints            ::= { AccessConstraint }
AccessConstraint       ::= deny '(' [Entity] Relationship Entity ')'
                                [when Condition]
Relationship           ::=  ↦  |  ⊣
Entity                 ::= ClassIdentifier | MethodIdentifier
Condition              ::= BooleanExpression
```

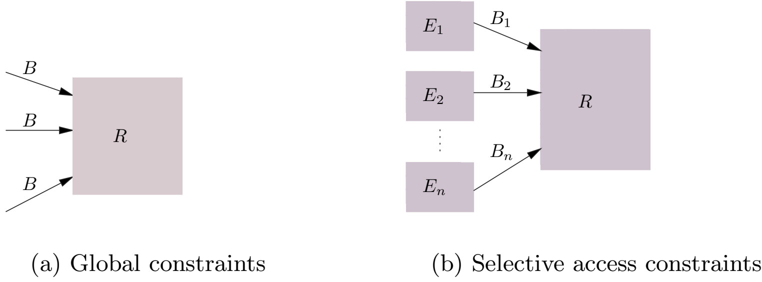
A site controls accesses to different resources (Java objects) by defining a set of AccessConstraints. We describe the various terms in the grammar informally below:

- Entity: An entity denotes objects and method invocations of Java programs. A ClassIdentifier, thus, identifies the set of objects to which a given access relationship applies. Similarly, a MethodIdentifier denotes a set of invocations of a method. The current implementation defines an entity based on its name. However, this can be extended to define an entity on the basis of its source, signature, or behavior pattern.
- Relationship: The composition mechanisms of a programming language allow one to define various relationships (data composition through aggregation and inheritance, and program composition through method invocations) among the entities of a program. We are primarily interested in the following two access relationships here:
  1. Instantiate ($\dashv$): A relation $E \dashv R$ exists if an entity E creates an instance of class R.
  2. Invoke ($\mapsto$): A relation $E \mapsto R$ exists if an entity E invokes an entity R.
- Condition: The term Condition denotes a boolean expression that can be defined in terms of object states, program state (global state), runtime system state, security state, and parameters of methods.

Semantics: An access constraint of the form

deny (E σ R) when Condition

specifies that entity E cannot access R through relationship σ if Condition is true. E is optional. Hence, there are two kinds of access constraints: *all access*



(a) Global constraints (b) Selective access constraints

Fig. 2. Category of access constraints

constraints and *selective access constraints*. Global constraints denote those constraints that do not depend on the initiator of the access relationship. For instance, as shown in Fig. 2(a), no program can access R when B is true. A host may specify the constraint that no Java applet can access a set of proprietary files.

Selective access constraints denote those constraints that depend on the initiator of the access relationship. For instance, as shown Fig. 2(b), each entity E_i's access to R is constrained by a separate and possibly different B_i. A site can use selective access constraints to associate different security policies with different Java programs that come from different sites.

Examples of all access constraints are:

Constraint	Semantics
deny ($\dashv$ C_2) when B	No instances of C_2 can be created if B is true
deny ($\mapsto$ $C_2.M_2$) when B	Method M_2 of class C_2 cannot be invoked if B is true.

Examples of selective access constraints are:

Constraint	Semantics
deny ($C_1.M$ $\dashv$ C_2) when B	Method M of class C_1 cannot create an object of C_2 if B is true.
deny ($C_1.M_1$ $\mapsto$ $C_2.M_2$) when B	Method M_1 of class C_1 cannot invoke M_2 of C_2 if B is true.

In our approach, the default is to allow all accesses unless a site specifically denies them. We call this model the *active denial model*. This is unlike most approaches in which the default is to deny all requests unless a site specifically allows them. We call this model the *active permission model*. The active permission model provides better guarantees about system security in cases when a site makes mistakes about specifying access control policy, the reasoning being that it is better to deny legitimate accesses than allow illegitimate accesses [33].

We chose to use the active denial model because we want to construct a unified access control framework for all method invocations. In other words, every action (every method call, object creation, deletion, etc.) is conceivably a security relevant event which a site may want to control. For instance, we want to be able to specify constraints such as users can invoke a function, say sqrt, only 10 times. Implementation of this access control model using the active permission model would require that a site define permissions for every method call, which can be quite cumbersome. Runtime system-based approaches [25] deal with this problem by embedding calls to an access controller checker within all methods that the site might want to control The checker enforces an active permission model over these calls. All resources that do not embed calls are not checked and hence can be accessed by anyone. Such models, thus, differentiate between resources that must be protected, through embedded calls, and those that need not. Our approach uses a single mechanism for handling both. The active denial model can be used to implement the active permission model by representing the permission conditions through the negation of denial conditions. We are, therefore, looking at ways of integrating the active permission model in our language.

Examples: We now present three examples. The first example implements a simple file access control mechanism. The second example shows how we can use the state of the runtime system to control accesses to resources. Finally, the last example shows how we can associate specific security states with program components and use these states to specify access control.

Example 1. File access control. In this example, we specify access constraints for controlling the file resources that mobile programs can access. Assume that the file resource is defined using the following Java class:

```
class File {
    public File(String Name);
    public char Read();
    public void Write(char data);
    public final String GetFileName();
}
```

The following constraint specifies that no mobile program can read "/etc/passwd" file:

```
deny ( ↦  File.Read) when (#2.GetFileName() == "/etc/passwd")
```

Here we introduce a new notation within the boolean expression. The terms #1 and #2 refer to the entities before and after the relationship, respectively. Thus, in the above expression the term #2.GetFileName() can be read File.GetFileName().

The access constraint that mobile programs can only read files A and B can be specified by expressions of the form:

```
deny ( ↦  File.Read) when
    ((#2.GetFileName() != "A") && (#2.GetFileName() != "B"))
```

The constraint that mobile programs cannot write to the local disk is specified by the following constraint:

```
deny ( ↦  File.Write)
```

As we can see from the above example, an access constraint can control executions of methods on the basis of program states. In certain cases, a site may wish to impose constraints on the basis of the state associated with the runtime system or the underlying operating system. The policy language allows specification of such constraints. We show this through an example:

Example 2. Network access control. Assume that the following defines the socket resource for making network connections:

```
Class Socket {
    Socket();
    void Open(Host hostId, int SocketId);
    void Write(Bytes data);
    Bytes Read();
}
```

Also, assume that the runtime system keeps track of the number of network connections that have already been opened. This forms the state associated with the runtime system. Let the method RuntimeSystem.Network.NumConnections() return the number of open connections. A constraint that limits the number of network connections to a specific upper-bound can be specified in the following manner:

```
deny ( ⊣  Socket) when
    (RuntimeSystem.Network.NumConnections() == UPPERBOUND)
```

In addition to runtime system state, a site may wish to store additional information for implementing access control. We call this kind of information *security state*. A site may associate a security state with a method, object, or a group of objects, and may define constraints over accesses to methods on the basis of the security state. We present an example below that illustrates this:

Example 3. Control over number of accesses. Assume that we want to implement the constraint that a program p can invoke a method, say f, on a resource R at most ten times.

This can be implemented by associating an object, say SecurityState, with p. The object keeps track of the number of times p calls f. Let method SecurityState.CheckCount(int x) be defined in the following manner:

```
public boolean CheckCount(int x) {
    if (count < x) {
        UpdateCount(); // increment the counter
        return(false);
    } else return(true);
}
```

The policy statements

```
add SecState SecurityState to R
deny (p ↦ R.f) when R.SecurityState.CheckCount(10)
```

adds the new object to R and specifies that p can invoke f at most 10 times.

Inheritance of access constraints: We now present an inheritance model for access constraints. The inheritance model describes what denials to resource accesses mean in terms of denials of accesses to subclasses of resources.

Assume that a site defines two resources, R_c and R_s:

```
class Rc {
    public void f();
    public void g();
    public void h();
}
class Rs extends Rc {
}
```

R_s is a subclass of R_c: R_s inherits methods f, g, and h from R_c. Assume that the site defines the following constraints on the resources:

deny ($E \mapsto R_c.f$) when B_{cf}
deny ($E \mapsto R_c.g$) when B_{cg}
deny ($E \mapsto R_s.f$) when B_{sf}
deny ($E \mapsto R_s.h$) when B_{sh}

There are two components to the inheritance model:

- Inheritance of access constraints: A subclass inherits all access constraints from its superclasses. Hence, the resulting access constraint on invocations of g on an instance of R_s is defined by the following expression:

  deny ($E \mapsto R_s.g$) when B_{cg}

  Access constraints are not inherited from subclasses to superclasses. Hence, although the access constraint on h in R_s is B_{sh}, there are no access constraints on h in R_c.
- Strengthening of access constraints: A subclass cannot override its inherited constraints. Specification of additional constraints in the subclass only strengthen the constraints defined in its superclasses. Hence, the resulting access constraint on invocations of f on an instance of R_s is:

  deny ($E \mapsto R_s.f$) when $B_{cf} \vee B_{sf}$

  In other words, method $R_s.f$ cannot be invoked from E if either B_{cf} or B_{sf} is true.

This model of inheritance ensures that a mobile program cannot override access constraints on methods by defining a subclass and by weakening the access constraints. Also, the above inheritance model applies for access constraints on $\dashv$ as well. That is, if a class R_c cannot be instantiated, none of its subclasses can be instantiated.

3 Access Constraint Enforcement

An enforcement of access constraints on a resource involves placing interposition code between the resource access code and resource definition code. The interposition code checks if a specific resource access is allowed. It can be inserted *manually* by site managers, generated by the compiler, or defined by the runtime systems or operating systems through special system calls. For instance, in the Java runtime system [12, 13], resource developers manually insert calls to a reference monitor in the resources they want to protect. The reference monitor consults access control policies to check if a specific resource access is allowed.

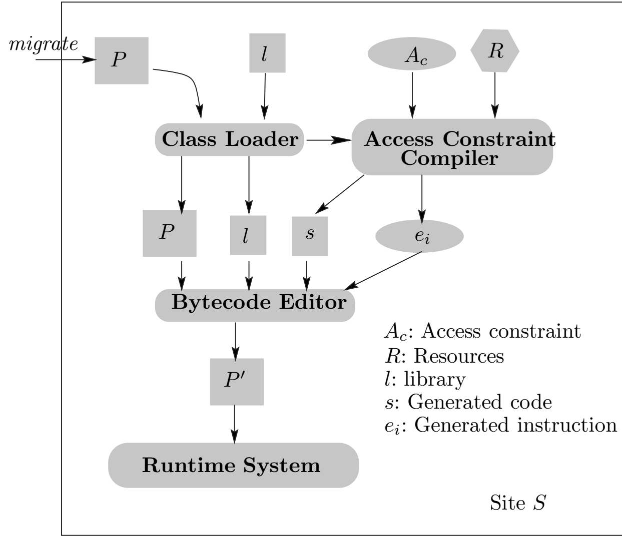


Fig. 3. Security policy enforcement of mobile programs

We use an alternate approach for generating interposition code. In this approach, a set of tools generates the interposition code and integrates them within mobile programs and resources before they are loaded in the JVM. In this approach, there are no reference monitors. In essence, the approach generates reference monitors on the fly and integrates them within the relevant Java programs and resources. The approach, thus, eliminates the need to manually include calls to reference monitors in resource definitions.

In Fig. 3, we describe our implementation for enforcing access control policies on Java programs. We show a Java program P that migrates to a site S. R denotes resources that the site makes available to mobile programs; and l denotes local libraries linked into P.

During class name resolution and dynamic linking, the Java class loader [24] retrieves R and l and passes them to a tool, called the *access constraint compiler*. The access constraint compiler examines P, R, and l to determine the resource access relationships that must be constrained in order to implement the access constraint A_c. It then generates interposition code s that implement the specific access constraints. It also generates a set of editing instructions e_i for the bytecode editor. The bytecode editor uses e_i to integrate s within P, R and l. The transformed programs and resources are then loaded into the JVM and executed.

We now describe in detail how we determine access relationships in Java programs, generate code, and edit Java class files.

3.1 Type Extraction

Type extraction involves examining Java class files to determine type definitions declared in the class files. Type definitions are used for automatically constructing a resource model from class files as well as for determining how Java classes should be modified. Type extraction can be done easily since Java class files maintain complete symbolic information about a class. Our type extraction technique makes use of two entities within the Java class file: the *constant pool* and the *method definition* sections. The constant pool is similar to a symbol table in that it contains all of the information needed to dynamically link classes. It is an index to the symbolic references of fields, classes, interfaces and methods, as well as their names. It also contains all literals, both string and numeric, used throughout a class. For example, a methodref entry in the constant pool includes all the symbolic information associated with a method. It contains two constant pool indexes: one for the class name and one for the name and type of the method. The method definitions section defines each method and identifies them by name and signature.

3.2 Extraction of Access Relationships

The extraction of access relationships involves searching the bodies of the methods for method invocation instructions. In the JVM, four opcodes (invokevirtual, invokespecial, invokestatic, and invokeinterface) are used for method invocation. Each method invocation instruction has an operand which indexes into the constant pool. Since this index is either a methodref entry or an interfaceref entry, the class name, method name, and signature of the method being invoked is immediately available. Both instantiate and invoke relationships are, thus, determined by searching the method bodies for one of the four invoke opcodes and matching it with the object's class name, method name, and signature. Note that this information may not be entirely valid due to the dynamic binding of methods. This problem is discussed in detail in the following sections.

3.3 Code Generation and Binary Editing

We now describe the nature of the code that is generated and its integration within mobile programs. Our code generation and editing involves modifying class definitions in order to add runtime state to classes and to insert runtime checks into methods.

An access constraint of the form

`deny` $(E\ \sigma\ R)$ `when` B

is implemented by generating the following code:

```
if (B)
    then error(); // raise exception
else
    access R
```

and patching it into classes and methods. The nature of the editing depends on the nature of the access constraints. Global constraints of the form

```
deny (σ R) when B
```

specify constraints on accesses to R without any regard to objects or methods that may access R. The generated code is, thus, integrated into the methods of R. On the other hand, selective access constraints of the form

```
deny (E σ R) when B
```

imposes conditions on accesses to R from E. The generated code is, thus, integrated into the methods of E which explicitly access R.

We also support addition of security states to specific Java classes in order to monitor site-specific behavior. This mechanism allows a site to customize its security policies, especially if the policies cannot be represented directly by the policy language. Security state objects are added to a class definition by using the statement:

```
add SecurityStateType SecurityStateObject to R
```

The constraint compiler generates code for initializing external objects. Example 3 shows how such objects can be used to specify access control policies.

3.4 Implementation Details

In this section we describe the code generation and code editing process for different instances of access constraints. For the purposes of explanation we restrict access to R when the first parameter is 5. Note that the boolean condition only affects the nature of code that is generated for B; it does not affect the general pattern of the access check code or the method of editing. Also, the following technique is independent of the action that should be taken in the event that an access is denied. Our implementation throws a security exception. Alternatively, one could take any conceivable programmable action, such as writing to an audit log, ending the mobile program, or even moving the mobile program to another site.

Implementation of Global Constraints: The first set of cases involve performing editing within the definition of a called method. We first consider a constraint of the form

```
deny ( ↦ R.f(I)V) when (#2.(1) == 5)
```

Recall that the term #2 refers to the entity being invoked. The term #2.(1) refers to the first parameter of that method. Also note that (I)V following R.f is the Java bytecode representation of the signature of that method. The above access constraint is enforced by generating code of the form shown in Fig. 4 and patching the code into the body of f.

In Fig. 4, the number to the left of an instruction indicates the byte offset for the instruction from the beginning of the method body. Further, a term #i in Fig. 4 and Fig. 5 indicates the ith entry in the constant pool. In code segment A of Fig. 4, #67 indexes the integer constant 5, whereas #65 in code segment B indexes the entry for a security exception class and #66 indexes the entry for its constructor.

```
0 iload 1
2 ldc #67                    A
4 if_icmpeq 10
7 goto 21

10 new #65
13 dup                       B
14 invokespecial #66
17 athrow

original code for
method R.f(I)V
```

Fig. 4. The modified method R.f(I)V

Code segment A (Fig. 4) contains the code for checking the conditional, whereas code segment B contains code for throwing an exception if the boolean condition is true. This code is inserted into the beginning of the method. Care must be taken to ensure that the security exception object and its constructors are defined in the constant pool. If they are not, then these entries are added.

Constraints of the form

deny (⊣ R) when B

specify that an instance of R cannot be created if B is true. They are implemented by putting constraints on invocations of all constructors of R, which, in the JVM, are given a special name <init>. This case is, thus, implemented by adding code similar to that shown in Fig. 4 to all methods of R with the name <init>.

Implementation of Selective Access Constraints: We now consider the cases in which methods are modified within the calling method. The most specific case involves denying access to a method from a specific method:

deny (E.g()V ↦ R.f(I)V) when (#2.(1) == 5)

Binary editing here involves first searching for all invocations of R.f(I)V within the body of E.g(). This involves examining the operands of all the invoke opcodes. Since the operand references a methodref entry in the constant pool, we can read the signature, method name, and class name of the method being called. If these match R.f(I)V, then the generated code is inserted before the invoke opcode.

The access relationship determined in this manner may only be partially correct due to the dynamic binding of methods. Assume the inheritance hierarchy of Sect. 2.2. Also, assume that method f is invoked on an object O of type R_C:

```
O.f();
```

If entity O references an object of type R_C or type R_S, and constraint B is defined for the method of class R_C, the above approach works because the constraint is inherited in the subclass as well. The problem arises when the constraint is defined over invocations to method f of R_S and object O may reference objects of type R_C or type R_S. Note that if it references objects of type R_C, the generated code should not be added because constraints are inherited from superclasses to subclasses, and not vice-versa. However, if O references an object of type R_S, the generated code should be added in order to implement the constraint. Since the reference type cannot be determined statically, additional code must be generated that checks for the type of object at runtime and performs access constraint checks on the basis of the type of the object. Thus, in cases where dynamic binding may play a role, an instanceof instruction is added to dynamically check the type of the object. The generated code for this case is shown in Fig. 5.

Fig. 5. The modified method E.g()V

The first step (code segment A) is to access the object reference by popping the operand stack, which contains method parameters and the object reference, into local variables. The method parameters and object reference are then pushed back on the stack in case the method is called later. This also need to be done if the constraint refers to the parameters of the called method. The second step (B) involves pushing the object reference onto the stack, performing an instanceof operation, and jumping to the method call if the object is not of type R. Term #3 is an index into the constant pool that refers to the class R. As in the first case, code segment C performs the conditional check, and section D throws the security exception. Section E contains the original invoke command. Term #10 is a constant pool index that refers to the method f with signature (I)V and class R. Other instances of access constraints can be implemented using the above technique.

Implementation of Inheritance Model: An implementation of the inheritance model requires care because of the possible conflicts between the Java

language mechanism for controlling extensibility and our inheritance model. We illustrate the problem with a simple example.

Assume that class R_s is a subclass of R_c. Class R_c defines a method f:

```
public void f();
```

Assume that R_s inherits f. Also, assume that the site specifies the following access constraint:

deny ($\mapsto$ $R_s.f$) when B

Since R_s inherits f, f needs to be modified in order to impose the above access constraint. However, since policies are inherited down and not up, the method body of f in R_c cannot be modified. A possible solution is, then, to redefine f in R_s:

```
public void f() {
    <interposition code for checking access>
    super.f();
}
```

The above solution works if f is not declared final in R_c. However, if f is declared to be final, we cannot redefine f in R_s as the Java bytecode verifier will reject the redefinition of a final method. Although we can edit the class file for R_c to remove the 'final' constraint, such a change may lead to security holes.

Our solution, therefore, relies on modifying class R_c as follows:

```
class Rc {
    final public void f() {
        _F_CheckMethod();
        <code for f>
    }
    private void _F_CheckMethod() { ; }
}
```

We now redefine _F_CheckMethod() in R_s in order to implement access constraint checks that are specific to R_s:

```
class Rs extends Rc {
            :
    private void _F_CheckMethod() {
        <interposition code for checking F>
    }
}
```

4 Discussions

In this section, we analyze the proposed technique for its suitability as an access constraint enforcement mechanism and for its performance behavior.

4.1 Characteristics of the Approach

In our approach, a site specifies access constraints separately from mobile programs, resources, and other class definitions. Further, the access constraint enforcement mechanism is not part of either the Java runtime system or the compiler. This impacts how access control code is managed and enforced at a site:

- Both access constraints and resource definitions can be modified independently. This makes it easy for a site to specify different access constraints for different mobile programs for the same resource. For instance, a site may specify that mobile program P can access R under condition B_p whereas mobile program Q can access R under condition B_q.
- The same set of access constraints can be applied to different resources without requiring one to copy it from one resource to another. For example, if a single access constraint B applies to multiple resources, it can be defined once and used for all resources.
- An important advantage of the separation is that our approach can be used for enforcing security on resources that were not designed with security in the first place. In other words, the security component can be added to a resource after it has been designed and implemented. Thus, it frees a library or resource designer from worrying about security concerns when designing and implementing the library.

A limitation of our approach is that it may end up building data structures that mirror some of the data structures that runtime systems build. This limitation arises because of the static nature of code enforcement. In many cases, access control policies depend on the history of execution as well as the dynamic state of an executing program. For instance, an access control policy may require that a program access a resource only if all methods currently on the stack are permitted to access the resource. This means that the interposition code must check the permission of all methods currently on the execution stack. Since our enforcement mechanism is completely separated from the runtime system, it needs to build and maintain a separate runtime infra-structure, involving an execution stack, in order to implement such policies. In runtime systems which exports the necessary state, these policies can be easily implemented.

4.2 Performance Analysis

In this section, we describe the performance behavior of the access constraint enforcement mechanism. Specifically, we analyze the following:

- What are the time and space overheads associated with our approach?
- How does our approach perform with respect to the Java runtime system's approach for enforcing access control?

We performed our experiments on a 266 MHz Pentium II running Red Hat Linux 5.0. The results show that both the time and space overheads of the approach are moderate. Further, the approach performs better than the Java runtime system in certain cases.

Overhead Measurements: We measured both the time and space costs of modifying resources.

There are four factors that affect the execution time associated with access constraint check code generation and editing:

- the cost associated with reading a method
- the number of access constraints
- the types of constraints
- the number of occurrences of restricted methods in a program

We do not consider the cost of reading class files in our measurements since the run-time system must perform this operation anyway.

In the first experiment, we looked at how the size of the method being modified affects the cost of editing. In this experiment, only a single method invocation must be wrapped. The cost of editing here is minimally affected by the size of the method. The cost varied between 0.08 and 0.16 seconds for methods ranging from 0 to 3200 instructions. In the second experiment, we looked at how the cost of editing changes when the number of method calls that needs to be wrapped changes. We found the cost to be proportional to number of methods that are wrapped.

We have also calculated the increase in size caused by adding code to class definitions. While the amount of code that is added to a class is independent of the size of the class, it depends on the number of method invocations that need to be wrapped and the complexity of the boolean portion of the constraint. For one wrapper, the minimum addition size (for a true boolean constraint), is 56 bytes. For two simple boolean expressions, it is about 206 bytes.

Performance Comparison: We now compare the performance behavior of our approach with the runtime system approach, as implemented in the JDK 1.1.3.

For this experiment we created a small program to test the performance of implementing security checks around one method invocation. Since the actual amount of work a particular site must perform depends on both the complexity of the access control policy and the number of restricted method invocations in a program, implementing a single policy statement once forms a good basis for comparison. We based our comparisons on the access control policy and classes from Example 3. The complete code for our approach is shown in Fig. 6. We implemented the same policy using Java's security manager as shown in Fig. 7. The test program calls the constrained method variable number of times. The access policy is that the method cannot be called more than 1000000 times.

Figure 8 shows the execution times of our approach and the Java's runtime system approach. In our approach, there is an initial overhead of about 0.08 seconds for code editing, which does not occur in the Java runtime system. However, after about 100000 method calls, our approach performs better than the Java runtime system. This is because our approach inlines the access control check code, whereas in case of the Java runtime system approach, each access

```
class SecState {
  public SecState() {count = 0;}
  public int check()
    { count++; return count; }
  private int count;
}
```

(a) Security object

```
add SecState SecurityState to R
deny ↦ R.f()V when
  #1.SecurityState.check() > 1000000
```

(b) Control access constraints

Fig. 6. The binary editing approach

```
class newSecMan
  extends SecurityManager {
  public newSecMan() {count = 0;}
  public void checkf()
      throws SecurityException {
    count++;
    if (count > 1000000)
      throw new SecurityException();
  }
  int count;
}
```

(a) Security Manager

```
class R {
  public void f() {
    newSecMan security;
    security =
      System.getSecurityManager();
    if (security != null)
      security.checkf();
  }
}
```

(b) Resource definition

Fig. 7. The Java Runtime System-based approach

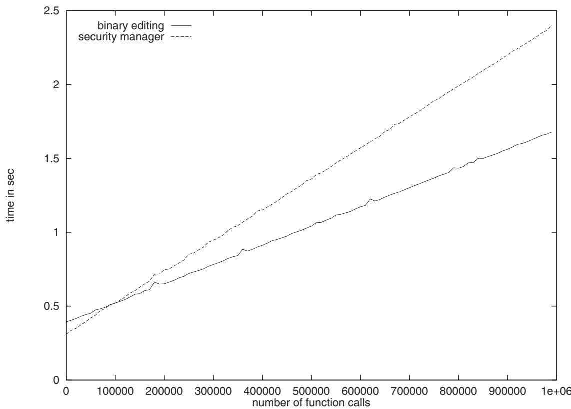


Fig. 8. Comparison of execution times with a policy

constraint check involves making two method calls: one to the system, to get the security manager, and another to the security manager itself. We can reduce our cost even further by pre-editing the methods if we know that only a single access constraint will be applied to the method, as is the case in the Java runtime system approach. Our approach, in this case, will then always outperform the Java runtime system approach.

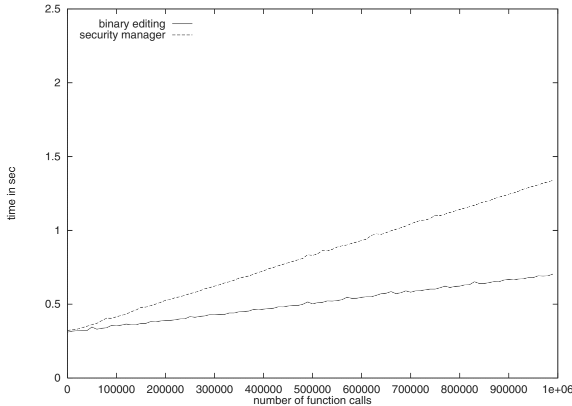


Fig. 9. Comparison of execution times without a policy

In the second experiment, we ran the same program with no policy implemented. As shown in Fig. 9, the Java runtime system is always less efficient that our approach. This is because in the Java runtime system approach, a method must always call the runtime system to check if there is a security manager installed, incurring the overhead of this call. Our approach does not incur any overhead since it does not add any code to methods that do not need to be constrained.

5 Related Work

In this section, we look at techniques that provide resource level access control. Much of the work on mobile program security has dealt with supporting different levels of security for Java programs. Therefore, we first consider Java's security model and various extensions to the model. We then turn to Safe-Tcl, an interpreter based security model. Finally, we discuss Proof Carrying Code, a language based approach.

Java: The initial security model [10, 22, 12] proposed by Sun for Java implements access control policies using a security manager. An access control policy is created by subclassing the SecurityManager class and setting this as the system's security manager. A site then ensures that all protectable resources make

an explicit call to the security manager to check if access is allowed. If the check is not allowed, the security manager throws a security exception. Otherwise, the control returns to the calling method. This decision is based on whether the code is trusted, i.e. from the local file system, or untrusted, i.e. an applet downloaded from the net.

The primary difference between our approach and this approach is that the JVM specifies policies in a procedural form. This allows the use of the full range of Java's language to specify any type of policy. In our approach policies are specified in a declarative form. This allows for easier expression and analysis of policies. We also allow policies to include procedural aspects with the security state object.

However, the extensibility of the security manager is limited. Suppose there are other services that the system is providing which needs to be restricted. While it is possible to add methods to a subclass of the SecurityManager class that will do the necessary checks, adding the code to call these checks might not be easy, especially if the programmer did not design these services to do so. This problem is further exacerbated if the software is proprietary code provided by a third party. In contrast, our approach allows us to add security information to mobile programs that might not been designed with security in mind. Further, the security models can be customized on the basis of program, security and runtime states, and method parameters.

The approach in [20] extends the Java security model to implement a domain-based access model. In this model, Java programs are given an unforgeable SecurityToken used to identify their domain. An AppletSecurity object plays the role of the Security Manager. It uses the SecurityToken of the applet to determine the capabilities of that applet, throwing a security exception if the needed capability is not there. Other capability systems have been proposed by JavaSoft, Electric Communities, and [16]. Similarly, the approach in [28] provides a more flexible mechanisms for controlling accesses to resources. Our approach differs from these works in that we propose a framework for implementing various security models and policies, including the ones implemented in [20] and [28].

Sun redesigned their security model [13] in order to provide the security infrastructure for supporting fine-grained access control and configurable security policies. The new model augments the SecurityManager with an AccessController that checks if mobile programs have permission to access specific resources. Permissions are stated in a policy language that allows users to define protection domains based on what URL they came from and on who has signed them. Each protection domain is associated with a set of actions that they are allowed to do. Unfortunately, for old resources to take advantage of the new model, these resources must be re-implemented.

The J-Kernel project [19] extends the JVM security model by implementing multiple protection domains within a single Java virtual machine. It provides access to resources by passing capabilities for them to a system-wide repository. Domains can then look up capabilities from this repository. Capabilities are im-

plemented as wrappers which provide the bookkeeping associated with changing protection domains.

Type hiding [36] modifies the dynamic linking process in Java to hide or replace classes seen by an applet. It allows a class to be replaced by a proxy class that checks the arguments of the invoked method and conditionally throws an exception or call their original methods.

Naccio [9] provides a framework for specifying resource hooks, state maintenance code, and safety policies. State maintenance and access checks are performed by adding wrappers. Programs are transformed to use these wrappers instead of the original library code.

Grimm and Bershad [14] describe an access control mechanism consisting of an enforcement manager and a security policy manager. The system is divided into protection domains. The mechanism examines the system and redirects invocations to access control checks. The security model is based on DTE.

Interpreter-Based Approaches: Safe-Tcl [23, 32, 15] requires at least two interpreters: a regular (or master) for trusted code and a limited (or safe) one for untrusted code. The designers of Safe-Tcl classified a set of instructions as being unsafe and then disabled those instructions in the safe interpreter. When untrusted code needs to access a system resource, the safe interpreter traps into the master one. The regular interpreter then decides whether or not to allow the access. A security policy is specified by aliasing the disabled instructions in the safe interpreter to procedures in the master interpreter. These procedures can then check arguments and, if the security policy allows, call the the masked instruction in the master interpreter. Furthermore, Safe-Tcl allows a program to request a policy which the interpreter can grant to the program as appropriate.

Language-Based Approach: The approach taken in Proof-Carrying Code (PCC) [30, 29] is to associate a site specific security policy with a program by constructing a compiler that takes user programs and site specific policies and generates both the binary code and proof of the program's safety with respect to the specified policies. As an external program is migrated for execution at the kernel, the proof is validated, within the context of the site specific safety policy, at the kernel site. One advantage of this approach is that it is tamper proof. If either the program or the proof has been modified in transit, then there will either be a validation error, or the resulting PCC binary will still validate the policy. Also, since PCC makes the decision on whether a program is secure on properties of the code rather than properties of the code's origin, cryptography is not needed. Further, PCC proof checks are similar to type checkers. They are simple to implement, easy to trust, and very efficient. Unfortunately, this approach is not practical for enforcing host dependent policies. In this case, the host must communicate its policy to the site manufacturing the program and the manufacturing site must create separate proofs for each host. This is especially server for mobile programs which may visit many different sites each with a different security policy.

Security Policy Languages: The area of security policy languages has also focused on mechanisms for specifying and enforcing security. Security policy languages have been considered as the basis for verifying designs of secure systems. Various considerations have been given to policy languages for doing general enforcement.

Access control matrices (ACMs) [1] are a traditional means for specifying what is and is not allowed on a system. With ACMs, a two-dimensional matrix is given with the active entities, called subjects, in the rows and all the entities, or objects, in the columns. A list of access rights that a subject has over an object is given in the corresponding matrix cell. The language described in this paper can be used to describe an access control matrix, as well as the conditional state transitions described in [18].

Miller and Baldwin [27] describe a method of access control based on boolean expression evaluation. The idea is that each subject and object is given a set of attributes. In addition, there is also a set of rules which link a subject, an object, and an action. These rules can be based on any number of attributes. Since these attributes can be anything, including security level, group membership or time of day, it can be used to implement most security policies. Our approach is similar in that we capture the various attributes in terms of boolean expressions.

Goguen and Meseguer [11] use an algebraic specification approach to specify security policies. Their particular approach expresses security policies as a set of non-interference assertions about a system. Cuppens, Saurel, and Cholvy [7, 5] use a form of deontic logic to express policies. In addition to specifying what actions an agent is permitted or forbidden to perform, it also allows statements that say what actions an agent is obliged to perform. They use deontic logic to find consistency problems between several policies. These policy languages are much more expressive than the one proposed in this paper. We plan to close this gap in the future. Our initial focus has been to develop a simple language for access control which can be implemented easily and efficiently.

The DIAMOND [31] security model provides an alternative model for inheriting security policies in object-oriented systems. This extends the MLS security model described by Denning [8] to object oriented databases. The innovation is that security levels, and hence policies, are not inherited from a class's superclass. Instead, they are derived from its instances. This allows a particular instance of a subclass to have a higher security level than its superclass.

6 Summary

We have described a mechanism for implementing general security policies on mobile programs. There are two components of our approach. The first is a simple declarative access constraint language that allows a site to restrict accesses to the objects and methods of the system. The declarative nature of the language makes it easy to specify policies while still allowing a hook to express procedural policies if necessary. The second is a set of tools that enforce the specified constraints by editing mobile programs and resources. Our approach's appeal is that a site can

specify access constraints separately from both mobile program definitions and resource definitions. This separation of concerns has a number of benefits. Both access constraints and resource definitions can be modified independently. Sites can easily specify different access constraints for different mobile programs for the same resource. Finally, our approach can enforce security on systems that were not originally designed with security in mind.

Our future work first involves generalizing our access control model to implement well-known security policies and constraints. We are developing mechanisms for facilitating the process of building security models using our approach. As part of our research in system software extensibility, we are considering various approaches for integrating our technique within the existing operating system and runtime system framework. Integration within the Java class loader is currently underway.

Acknowledgments

We thank Jeff Gragg and Raja Mukhopadhyay for help and support in implementing the system. We also thank Fritz Barnes, Earl Barr, Matt Bishop, Prem Devanbu, David Evans, Karl Levitt, Scott Malabarba, Ron Olsson, and the anonymous reviewers for their excellent comments and help in writing this paper.

References

[1] E. Amoroso. *Fundamentals of Computer Security Technology.* P T R Prentice Hall, 1994.

[2] K. Arnold and J. Gosling. *The Java Programming Language.* Addison Wesley, 1996.

[3] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, pages 34–49, October 1995.

[4] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems. Towards the Programmable Internet. Second International Workshop, MOS '96*, number 1222 in Lecture Notes in Computer Science, pages 25–47, Linz, Austria, July 1997. Springer-Verlag. Also available at http://www.research.ibm.com/massdist/mobag.ps.

[5] Laurence Cholvy and Frédéric Cuppens. Analyzing consistency of security policies. In *1997 IEEE Symposium on Security and Privacy*, pages 103–112, Oakland, California, 1997. IEEE.

[6] T. Coombs, J. Coombs, and D. Brewer. *ActiveX Sourcebook: Build an ActiveX-Based Web Site.* John Wiley & Sons, Inc., 1996.

[7] Frédéric Cuppens and Claire Saurel. Specifying a security policy: A case study. In *9th IEEE Compuer Security Foundations Workshop*, pages 123–134, Kenmare, Ireland, June 1996. IEEE, IEEE Comput. Soc. Press.

[8] D. Denning and P.J. Denning. Certification of Programs for Secure Information Flow. In *Communcations of the ACM*, volume 20(7), pages 504–513. ACM, 1977.

[9] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 1999.
[10] J.S. Fritzinger and M. Mueller. Java Security. JavaSoft White Paper, 1996. http://www.javasoft.com/security/whitepaper.ps.
[11] J.A. Goguen and J. Meseguer. Security policies and security models. In *In Proceedings of the 1982 Symposium on Security and Privacy*, pages 11–20, 1982.
[12] L. Gong. Java security: Present and near future. *IEEE Micro*, pages 14–19, May/June 1997.
[13] L. Gong, M. Mueller., H. Prafullchandra, and R. Schemers. Going beyong the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.
[14] R. Grimm and B.N. Bershad. Providing policy-neutral and transparent access control in extensible systems. Technical Report UW-CSE-98-02-02, Dept. of Computer Science and Engineering, University of Washington, 1998.
[15] S. Gritzalis and G. Aggelis. Security issues surrounding programming languages for mobile code: Java vs. Safe-Tcl. *Operating Systems Review*, 32(2):16–32, April 1998.
[16] D. Hagimont and L. Ismail. A protection scheme for mobile agents on Java. In *Mobicom '97*, pages 215–222, Budapest, Hungary, 1997. ACM.
[17] M.A. Harrison, W.L. R., and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.
[18] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.
[19] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. Technical Report 97-1160, Cornell University, 1997.
[20] N. Islam, R. Anand, T. Jaeger, and J.R. Rao. A flexible security model for using internet content. *IEEE Software*, 14(5):52–59, Sept.-Oct. 1997.
[21] S. Jajodia, S. Pierangela, and V.S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 Symposium on Security and Privacy*, pages 31–42, 1997.
[22] JavaSoft. *JDK 1.1.1 Documentation.*
[23] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July-August 1997.
[24] S. Liang and G. Brach. Dynamic Class Loading in the Java Virtual Machine. In C. Chambers, editor, *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices*, number 10, Vancouver, October 1998. ACM.
[25] J. Meyer and T. Downing. *Java Virtual Machine.* O'Reilly, 1997.
[26] D.V. Miller and R.W. Baldwin. Access control by boolean expression evaluation. In *Fifth Annual Computer Security Applications Conference*, pages 131–139, Tucson, AZ, 1990. IEEE, IEEE Comput. Soc. Press.
[27] D.V. Miller and R.W. Baldwin. Access control by boolean expression evaluation. In *Fifth Annual Computer Security Applications Conference*, pages 131–139, Tucson, AZ, 1990. IEEE, IEEE Comput. Soc. Press.
[28] N. Nagaratnam and S.B. Byrne. Resource access control for an internet user agent. In *Third USENIX Conference on Object-Oriented Technologies and Systems.* USENIX, June 1997.

[29] G.C. Necula. Proof-carrying code. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages*. ACM SIGPLAN-SIGACT, Jan. 1997.

[30] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating System Design and Implementations*. Usenix, Oct. 1996.

[31] L.M. Null and J. Wong. The DIAMOND security policy for object-oriented databases. In *1992 ACM Computer Science Conference. Communications Proceedings*, pages 49–56, Kansas City, MO, 1992.

[32] J.K. Ousterhout, J.Y. Levy, and B.B. Welch. The Safe-Tcl security model. Technical Report TR-97-60, Sun Microsystem Laboratories, 1997. Available at http://research.sun.com/technical-reports/1997/abstract-60.html.

[33] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[34] J.W. Stamos and D.K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[35] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.

[36] D.S. Wallach, D. Balfanz, D. Dean, and E.W. Felten. Extensible security architecture for Java. Technical report, Department of Computer Science, Princeton University, 1997.

[37] T.Y.C. Woo and S.S. Lam. Authorization in distributed systems: A formal approach. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 33–50, 1992.

Formal Specification and Prototyping of CORBA Systems

Rémi Bastide, Ousmane Sy, and Philippe Palanque

LIHS-FROGIS, Université Toulouse I, Place Anatole France,
F-31042 Toulouse CEDEX, France
{bastide, sy, palanque}@univ-tlse1.fr

Abstract. We propose to extend the CORBA interface definition of distributed objects by a behavioral specification based on high level Petri nets. This technique allows specifying in an abstract, concise and precise way the behavior of CORBA servers, including internal concurrency and synchronization. As the behavioral specification is fully executable, this approach also enables to early prototyping and testing of a distributed object system as soon as the behaviors of individual objects have been defined. The paper discusses several implementation issues of the multithreaded, distributed interpreter built for that purpose. The high level of formality of the chosen formalism allows for mathematical analysis of behavioral specifications.

Keywords. Formal methods, distributed object-oriented systems, CORBA, Petri nets, behavioral specification.

1 Introduction

CORBA [22], [32] (Common Object Request Broker Architecture) is a standard proposed by the Object Management Group (OMG) in order to promote interoperability between distributed object systems. The appearance of an industrial standard acknowledges the fact that the field of object-oriented distributed computing has moved, in the past few years, from experimental research projects to mainstream commercial products.

CORBA reaches several operational goals: standardize the best understood and best supported features of distributed object systems, promote interoperability between heterogeneous software systems, and provide a seamless integration with popular object-oriented and non-object-oriented programming languages such a C, C++, COBOL or Java.

CORBA defines an object model offering the following features:

- **Client/Server relationship**: a distributed object system consists of a set of objects that interact by invoking services to one another. The invocation is asymmetrical: one of the objects involved acts as a client, actively requesting

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 474-494, 1999.
© Springer-Verlag Berlin Heidelberg 1999

the service, while the other acts as a server, passively waiting for requests and executing them as they come. Note that a given object usually acts both as a client and a server at different moments of its activity.

- **Access through references**: Before a client object can request a service from a server object, the client must acquire a reference to the server through some mechanism. Moreover, the invocation is performed through a well-known, strongly typed interface: the set of services that can be invoked and the signature of these services are known at compile time, and depend on the type of the reference held by the client. This is the basic type of interaction, although CORBA also supports the dynamic construction of invocations, to allow for contacting objects whose class is not known at compile time.
- **Synchronous invocation**: the invocation of a service results in synchronization between the client and the server: an invocation is a two-ways question/response interaction, where parameters are sent and results returned. The call is blocking on the client's side: it has to wait for the results before continuing its work. This is the basic invocation mode, although CORBA has provision for unidirectional message sending (the so-called *oneway* services).
- **Dynamic topology**: The fact that an object *A* holds a reference to an object *B* introduces a *reference relationship* between A and B. The topology of the reference relationship is dynamic. An object may transfer a reference it holds to another object; new objects can be dynamically introduced in the system.

This basic set of features comes as no surprise: it closely mimics the principles of mainstream sequential object-oriented languages such as C++, and allows extending such languages to the realm of distributed and concurrent systems.

CORBA has standardized the features described above by specifying an Interface Definition Language (IDL). CORBA-IDL is independent from any programming language (although closely patterned after C++) and object-oriented, supporting specialization of interfaces through inheritance. A CORBA-IDL interface specifies at a syntactic level the services that a client object can request from a server object that implements this interface. The interface details the services supported and their signature: a list of parameters with their IDL type and parameter-passing mode, the IDL type of the return value, the exceptions that may possibly be raised during the processing of the service.

1.1 CORBA and Behavioral Specification

A recognized limitation of CORBA is that it defines remote object classes in terms of their interface only. CORBA IDL covers only the syntactic aspects of the possible use of a remote object. IDL does not cover any semantic or behavioral description of the remote object, while this information is obviously of prime importance for the clients. By behavioral aspect, we mean:

- The constraints on the order of invocation of the services described in the interface.

- The concurrency constraints of the remote object: is it able to support concurrent access to its services, or does it force a serialization on the concurrent invocations?
- The conditions under which an exception might be raised during the processing of a service.

What CORBA lacks is an abstract way to specify the semantics of an IDL interface without constraining its implementation, much in the same way that an Abstract Data Type [11] (ADT) specification can be used for specifying the semantics of a sequential data type.

This limitation becomes evident when one considers the standardization of CORBA services (COS) [23] that is underway at the OMG. CORBA services are a standardization of the various basic services that any large-scale distributed application is expected to require. Example of these services are the naming service, allowing to retrieve a remote object reference by providing a symbolic name, or the event service, allowing to define one-to-many communications that go beyond the client-server paradigm that is basically supported by CORBA.

The specification of the CORBA services is provided by the OMG in the form of a mixture of IDL (for the definition of the interfaces) and English text (for the specification of the behavior). The specification document counts no less than one thousand pages of such “semi-formal” specification.

To illustrate the dire need for some form of behavioral specification, we may quote the OMG specification document for the event service: “Clearly, an implementation of an event channel that discards all events is *not a useful* implementation” (emphasis is from the original document). However, the rest of the document defines only informally what actually constitutes a useful implementation.

The present paper aims at providing a suitable solution to the problem of behavioral specification of distributed objects, in the context of CORBA. The paper is organized as follows: We first detail the requirements for behavioral specification formalism suited to CORBA. We then present how the Cooperative Objects formalism needs to be adapted in order to support fully the CORBA model. Section 4 presents a significant case study of specification using our approach. Section 5 explains how the formalism can be used to enable rapid prototyping of distributed systems.

2 Requirements for a Behavioral Specification Formalism

Our goal is provide a notation suited to the behavioral specification of CORBA systems: we want to be able to describe the behavior of a collection of interacting objects, and not merely the behavior of a single object in isolation. A formalism aimed at serving this goal has to comply with several requirements:

- Cope with data flow as well as with control flow. The formalism needs to be able to deal with typed values, and not only with pure causal relationships. It

will often be the case that the behavior of a CORBA system depends not only on the previous history of invocations between objects, but also on the values exchanged during these invocations. For a given state of an object, an invocation may succeed or fail according to the values of parameters of the invocation.

- Allow specifying internal concurrency for objects. This point is especially important for CORBA since a CORBA server object will often be a "large-grained", entity shared by a lot of clients, providing services whose processing will take some time. It is therefore unrealistic to enforce each service to be atomic, so that at most one service will be active at any time at the server. Actually all of current CORBA ORBs allow for "multi-threaded" server implementations, where a server object can serve several services at the same time.
- Serve the needs of the implementers of the server class, as well as those of the designers of systems that will be clients for this server. On the one hand, the behavioral specification must be complete and precise enough that the programmer implementing the server in some programming language knows in a precise and non-ambiguous way what behavior to implement; and it must be abstract enough not to constraint the implementation choices of the programmer. On the other hand, the potential clients of the class will use the specification to gain a non-ambiguous understanding of the semantics of each service.

In the above set of requirements for a CORBA specification formalism, we have purposefully not included the provision for object-oriented features such as inheritance, or encapsulation. Actually, the fact that CORBA defines mapping to non object-oriented languages such as C or COBOL is a demonstration that object-orientedness is conceptually not a requirement. Actually, the implementation of a CORBA interface in some programming language can be considered as a formal behavioral specification (albeit at a very low level of abstraction). However, object-oriented specification formalisms map more naturally to the object-oriented foundations of CORBA, which is beneficial for the overall usability of the specification documents.

2.1 Related Work

The behavioral specification of object systems is a field of research per se [15], [25]. The importance of providing behavioral specifications suited to the OMG object model as been recognized by several researchers. Sankar [30] argued that the introduction of formal methods could help maintaining the current level of software quality while the complexity of software increases due to the presence of distributed object. He also noted that formal methods are more likely to be accepted in the field of distributed object systems, where their overhead is considered acceptable.

It is tempting to try to adapt the "design by contract" paradigm popularized by the Eiffel language to CORBA systems. However, due to its roots in Abstract Data Types, such approaches often make the implicit assumption that the execution of a

single operation is always atomic. As stated in [20]. "*Any client accessing an object through [an operation]*[1] *must be guaranteed exclusive access to the object throughout the duration of the call*". *The smallest permissible level of granularity for exclusive access to an object is the execution of a call to an exported [operation]*). Although this limitation can be viable in the domain of *concurrent* systems, in our opinion it presents a serious drawback in the field of *distributed* systems such as CORBA. Most approaches based on pre and post-conditions implicitly consider an object as a monitor, allowing only one service to be active at a time, thus cannot be extended to the modeling of distributed CORBA servers with internal concurrency.

The work presented in [8] falls in this category: Bryan proposes the use of a language called ASL (Architecture Specification Language) which includes a behavioral specification part, as well as other extensions supporting the description of software architectures. However, the behavioral specification (based on pre-and post-conditions for operations) does not permit do describe internal concurrency for objects. The work of Sankar [30], who uses the textual language Borneo, also falls in the same category. Several proposals have been made to adapt the LARCH specification language [12] to CORBA IDL, e.g. [18], [33]. The former does not explicitly address intra-object concurrency, while the latter addresses it with the introduction of atomic and non-atomic operations. The Z language has also been proposed as a possible behavioral specification formalism [9], but it is not clear to what extend it supports the concurrency requirements of CORBA systems. Other work, such as [26], are more theoretical in nature, and do not meet the operational requirements of CORBA-based software engineering.

Among the various formalisms proposed, StateCharts [13] comply with the above requirements, and is actually the behavioral formalism for the popular Unified Modeling Language (UML) notation [28], [29].

We propose the use of high-level Petri nets as the behavioral specification medium for CORBA systems. The usefulness of combining Petri nets and objects has been recognized by several authors, as witnessed by the workshops held at previous editions of the ATPN conference [1], [2]. However, several proposals do not fit well with CORBA: A behavioral specification formalism for CORBA has to support its fundamental object model, and more precisely to allow remote objects to be designated by references, and these references to be transmitted as invocation parameters. Several object-oriented Petri net formalisms [34], [17], aimed at providing a solid theoretical basis to concurrent object-oriented computation, refrain from using references, and prefer considering tokens as objects, and not references to objects. Such formalism prevent a same object to be referenced by different tokens in different places, and thus would make the specification of a reference-based system such as CORBA less straightforward.

The approach proposed here uses an object-oriented dialect of high-level Petri nets called Cooperative Objects. This formalism complies with the requirement listed

[1] Meyer uses the Eiffel vocabulary of "feature" instead of operation.

above, and its Petri net roots provide is with a powerful basis for modeling and analyzing concurrent behaviors.

3 The Cooperative Object Formalism

Cooperative Objects [6] (CO) are a dialect of object-structured, high-level Petri nets. Their lengthy formal definition has been provided in previous publications [3], [31], and we will only recall informally their main features, through examples.

Petri nets basics. Petri nets [24] have been studied for a long time as a mathematical formalism for the modeling of concurrent systems. A Petri net models a system as a set of state variables called *places* (represented as ellipses) and a set of state-changing operators called *transitions* (represented as rectangles). The state of the system is described as a distribution of information elements (called *tokens*) in the net's places; this distribution is called the *marking* of the net. In basic Petri nets, tokens are dimensionless entities modeling only conditions. Several dialects of Petri nets (called high-level Petri nets) allow tokens to carry information and to manipulate this information at the occurrence of transitions. Our own dialect is close to well-known high-level net models such as colored Petri nets [14] or Predicate-Transition nets [10]. They differ from these mainly by the nature of the inscriptions attached to the net elements, and by their object-oriented structure.

In Petri nets, the causality structure of the systems (stating under which condition a change of state may occur, and what will be its resulting state) is described by *arcs*, connecting places and transitions.

The input arcs of a transition (coming from places of the net) describe the preconditions of an occurrence of this transition. The transition may occur if the input places hold enough tokens. Conversely, the output arcs of a transition (going to places of the net) describe the postconditions of an occurrence of the transition, in terms of changes in the net's marking. After an occurrence, tokens are removed from the input places of the transition, and new tokens are set in its output places.

Petri nets allow for modeling very naturally systems with distributed state, and concurrent activities. Two transitions that do not share any input place may occur concurrently (marking permitting) and the resulting state will be the same whether they occur concurrently, or sequentially in any order.

A huge amount of work has been devoted to the study and analysis of Petri nets. Several important safety and liveness properties, as well as invariants in the dynamic behavior can be proved by mathematical means.

Structure of a CO class. A CO class specifies a class of objects by providing their interface (the set of services offered, along with their signature) and their dynamic behavior. The behavior of a CO class is called its *Object Control Structure* (ObCS), and is defined with a dialect of high-level Petri nets. More specifically:

- Tokens are tuples of typed values. The arity of a token is the number of values it holds, and tokens of zero-arity are thus the “basic” tokens used in conventional Petri nets. We will call *Token-type* a tuple of types, describing the individual types of the values held by a token. Token-types will be noted <Type_1,… Type_n> or just <> to denote the Token-type of zero-arity tokens.
- Places are defined to hold tokens of a certain Token-type, thus all tokens stored in one place have the same Token-type and arity. A place holds a multiset of tokens, thus a given token may be present several times in the same place.
- Each arc is inscribed by a tuple of variables, with a given multiplicity. The arity of an arc is the number of variables associated to it. The arity of an arc is necessarily the same as the arity of the Token-type of the place it is connected to, and the type of each variable is deduced from this Token-type. The multiplicity of an arc is the number of identical tokens that will be processed by the firing of a transition associated to this arc. The general form of an arc inscription is multiplicity*<v1,… vn>. A multiplicity of 1 can be omitted (thus 1*<v1,… vn> can be abbreviated as <v1,… vn>) and an empty list of variables can also be omitted (thus 2*<> can be abbreviated as 2).
- Transitions have a precondition (a boolean expression of their input variables) and an action, which may use any service allowed for the types of their input or output variables. The scope and type of each variable of an arc is local to the transition the arc connects to.

A transition is enabled when:

- A substitution of its input variables to values stored in the tokens of its input places can be found
- The multiplicity of each substituted token in the input places is superior or equal to the multiplicity of the input arc,
- The precondition of the transition evaluates to true for the substitution.

The firing of a transition will execute the transition’s action, remove tokens from input places, compute new tokens and store them in the output places of the transition.

The formalism also supports two arc extensions [16]: test arcs (allowing to test for the presence of tokens in input places of a transition, without removing them at the occurrence of the transition) and inhibitor arcs (allowing to test for the absence of certain values in input places of a transition).

4 A Case Study in CORBA Behavioral Specification

Cooperative Objects were initially defined independently of CORBA, [4] but CO and CORBA happen to share the same object model, as described in §1. CO and CORBA complement each other nicely: The initial description of CO used an idiosyncratic language to describe interfaces, while CORBA provides an attractive, language

neutral IDL. On the other hand, CO can be used to provide the behavioral information that is lacking in CORBA-IDL.
This section presents the necessary syntactic adaptation that CO require to match more closely CORBA-IDL, and some additions to the formalism supporting CORBA-specific constructs such as exceptions and parameter-passing modes.

CORBA-IDL will be used as the data description language used by Cooperative Objects. Cooperative Objects will use IDL to describe:

- The system of data types used by the ObCS nets of Cooperative Objects. The Token-type of the places will thus be described in terms of CORBA IDL, and the variables on the arc will be of an IDL-defined type.
- The interface of a Cooperative Object class itself.

To achieve the integration of Cooperative Objects and CORBA IDL, we need to define a mapping from the constructs of CORBA IDL to those of Cooperative Objects, much in the same way that CORBA defines mappings from IDL to conventional programming languages such as C++, Smalltalk or Java.

4.1 Presentation of the Case Study

To illustrate our modeling approach, we now detail an example dealing with the banking business. This example is typical of the intended use of the CORBA technology (although much shorter than real-life examples) and is in fact used as a tutorial example for several commercial ORBs. Despite its small size, it illustrates all the main aspects of our technique, including exceptions, interface inheritance and realistic behavioral specification featuring intra-object concurrency.

The banking system is composed of two different kinds of entities: *bank accounts*, which keep a (positive or null) amount of money, and *banks*, which keep a set of bank accounts. This system is first described in terms of CORBA IDL. Cooperative Object class specifications are then provided, to specify the behavior of the system.

```
module Banking {
 exception noSuchAccount;
 exception nameAlreadyExists;
 exception insufficientFunds;
 exception accountIsClosed;

 interface Bank {
      Account newAccount(in string name,
            in float initialAmount )
            raises(nameAlreadyExists);
      Account findAccount(in string name )
            raises(noSuchAccount);
      void closeAccount(in Account account)
            raises(noSuchAccount);
 }
```

```
 interface Account {
      void transfer( in float amount )
            raises(accountIsClosed, insufficientFunds);
      float balance( );
}

 interface BankAccount : Account {
      void open();
      void close();
 }
}
```

Fig. 1. the IDL text for the banking system.

Fig. 1 shows the interface of the banking system, expressed in CORBA IDL. This IDL first defines data types for the exceptions that might be thrown during processing (*noSuchAccount, nameAlreadyExists, insufficientFunds, accountIsClosed*), and defines three interfaces:

- *Bank*, which the customers will use to create, close, and access their accounts. A bank is merely a repository for bank accounts, and allows associating an *Account* reference to a human-readable name. The service *findAccount* retrieves an *Account* reference for which only the name is known;
- *Account*, offering a service to obtain the current balance (*balance*) and a single service to credit or debit the account (*transfer*).
- *BankAccount*, which will be used internally by the bank to perform *open* and *close* operations on accounts. *BankAccount* is a specialization of the *Account* interface, meaning that is offers all the services of *Account*, in addition to the new ones it introduces (*open* and *close*).

The provision for two different interfaces (*Account* and *BankAccount*) to describe the same concept of account enables to describe different access rights (or different views) of the same object, tailored for the needs of different clients. The signature of the services provided by the *Bank* interface only deal with *Account* references, which means that clients of a *Bank* object will only receive references of this type, and thus will be able to access only the services defined in *Account*.

4.2 CO-based Behavioral Specification

Obviously a lot more needs to be said, in addition to these interfaces, to have a complete specification of the banking system. The behavioral specification of the banking system is provided below, in terms of two Cooperative Object classes, *BankSpec* and *BankAccountSpec*.

IDL interfaces and CO classes. A CO class may specify one or several IDL interfaces, in the same way that a Java class, for example, may implement several Java interfaces. This is convenient since, very often, several interfaces are given for

the same entity to allow providing different views of the same object, tailored for the needs of different clients. The CO class of Fig. 2 specifies only one interface, namely *Bank*. The keyword **specifies** is followed by the list of CORBA-IDL interfaces specified by the CO class.

Mapping for services. Each service *op* defined in an IDL interface is mapped to three places in the ObCS net: a Service Input Port (SIP, labeled op), a Service Output Port (SOP, labeled op) and a Service Exception port (SEP, labeled ~~op~~). These three places are derived from the IDL, as follows:

- The Token-type of the SIP is the concatenation of the IDL types of all *in* and *inout* parameters of the service;
- The Token-type of the SOP is the concatenation of:
  - the IDL type of the result returned by the service (if any),
  - the list of the IDL types of all *out* and *inout* parameters of the service.
- The Token-type of the SEP is `<Exception>`, where Exception is the super-type of all IDL exception types. The SEP is only used if the service is defined to raise an exception.

```
class BankSpec
specifies Bank {
      place Accounts <string, BankAccount>;
      place newAccount <string, BankAccount, float>;
      place openAccount <BankAccount, float >;
      transition createAccount {
            action {a = new bankAccountSpec();}
      }
      transition open {
            action { a.open(); }
      }
      transition initialDeposit {
            action { a.transfer(initialAmount); }
      }
      transition close {
            action { a.close(); }
      }
```

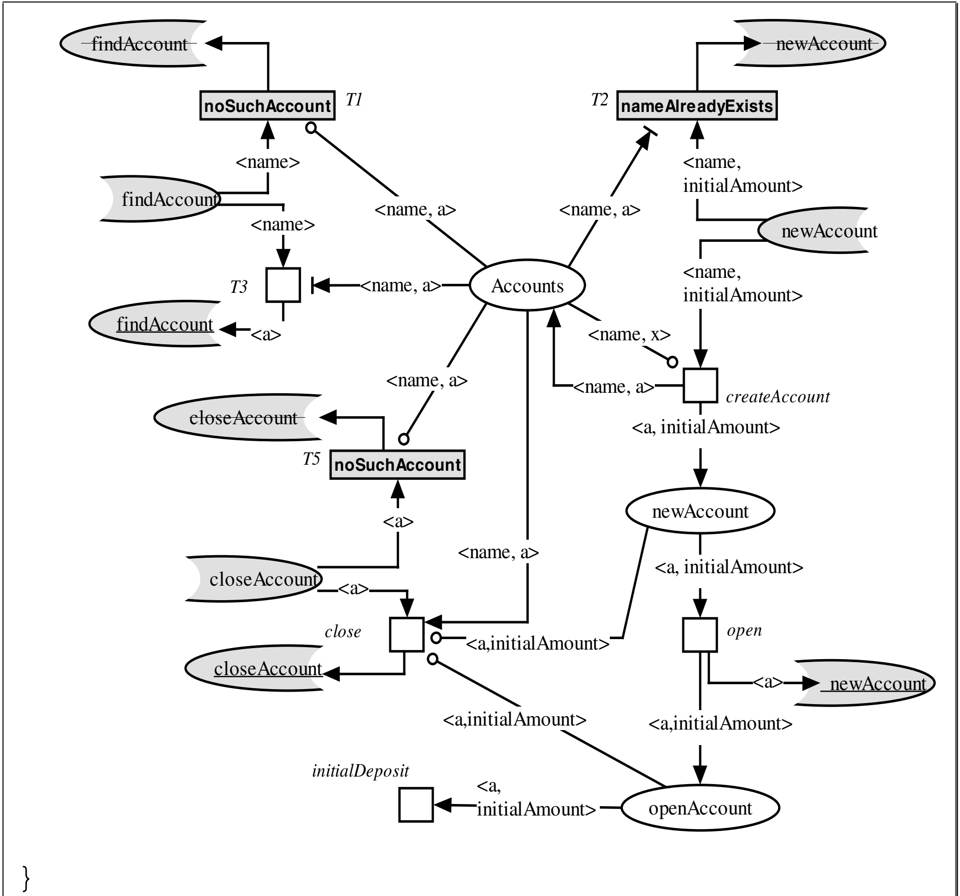


Fig. 2. Cooperative Object class specifying the *Bank* interface.

For instance, according to the IDL of Fig. 1, the service *findAccount* is mapped onto three places: findAccount for the SIP, findAccount for the SOP and ~~findAccount~~ for the SEP.

The invocation of one service results in one token holding all *in* and *inout* parameters being deposited in its SIP. The role of the ObCS net is to process this parameter token in some way, and eventually deposit a result token (holding the result of the service, plus all *out* or *inout* parameters) in the SOP, thus completing the processing of the invocation. An invocation that raises an exception at some point will instead result in an exception token being deposited in the SEP.

Fig. 2 illustrates both the graphic syntax of a Cooperative Object class and the textual annotations that are necessary to complete its description. These textual annotations are:

- The list of the interfaces that the CO class specifies (keyword **specifies**). In this case, the *BankSpec* class specifies the *Bank* interface.

- The description of the places' type: for instance, the *Accounts* place holds tuples of the form <string, BankAccount>, i.e. the name of a bank account, and the associated reference to a *BankAccount* object. The type for the input, output and exception ports needs not be stated, since it is deduced from the signature of the associated service. For example, the type of the *newAccount* input port is <string, float>, and the type of its output port is <Account>.
- The description of the transitions' preconditions and actions: Empty actions default to no action, and empty preconditions default to true. Only the transitions with non-default precondition or action need to be stated in the textual part.

The picture also illustrate the notation for inhibitor arcs (ended with a circle instead of an arrow) and for test arcs (ended with a barred arrow).

The BankSpec class describes a sensible behavior for the bank entity. For example, it states that an Account reference can be retrieved (through the findAccount service) only after an account with the same name has been created (through the newAccount service) and before it has been closed (through the closeAccount service). It also specifies under which conditions exception might be raised (for instance a *nameAlreadyExists* exception is raised when the *newAccount* service is called with a name parameter that matches a previously registered name). Such behavioral constraints could have been modeled just as well using a conventional ADT-based specification. However, the description of more subtle behaviors requires a formalism expressive enough to encompass concurrency and synchronization constraints.

In order for the invocation of services to proceed in a sound way, the ObCS structure must respect a set of constraints. Informally, an object will provide either a result or an exception for each invocation, and will only provide results if it has been previously invoked. With respect to the ObCS, the arrival of one token in the SIP will eventually result later in exactly one token being deposited either in the SOP or in the SEP. These structural constraint on the ObCS are easily expressed in terms of Petri nets theory, and can be checked automatically. The formal description of these structural constraints can be found in [7], where the notion of Operation Control Structure (OpCS) is also defined. Informally, the OpCS of a service is a subnet encompassing it's SIP, SEP and SOP.

The OpCS of the *newAccount* service is especially noticeable: unlike the other two services (*findAccount* and *closeAccount*), it is not made of atomic transitions, but of a subnet encompassing the *createAccount* and *open* transitions. This feature allows for internal concurrency within a *BankSpec* instance: While an invocation of *newAccount* is being processed (i.e. when the *newAccount* place holds one token), other incoming invocations of *findAccount* or *closeAccount* can be serviced (if the marking of the *Account* place permits it). Moreover, several invocations of *newAccount* can be serviced concurrently (this will result in the *newAccount* place holding several tokens). Finally, when the service invocation returns (by setting the reference to the newly created *BankAccountSpec* object in the *newAccount* return port), the *BankSpec*

object continues an internal processing, namely to initialize the *BankAccountSpec* object with its initial balance (transition *initialDeposit*). This faithfully models the way actual banks proceed in the creation of new customer accounts (at least in France): If a customer goes to a bank to create an account with an initial deposit, the account will be created immediately, but it will take some time before the account is actually credited with the initial deposit. If the account's balance is accessed in the meantime, it will be zero. The BankSpec also takes care that an account is not closed before the initial deposit is performed (inhibitor arc between *openAccount* and *close*).

Mapping for parameter-passing modes. The semantics for the three parameter-passing modes of CORBA IDL is clear.

- *in* parameters are values provided by the caller, that the service may use at its own will;
- *out* parameters are values computed by the service, and returned to the caller;
- *inout* parameters are values transformed by the service, i.e. provided by the caller and returned to it after being processed.

Inout parameters require a special treatment in the OpCS structure: in order to ensure that any inout parameter is properly transmitted from the Service Input Port to the Service Output Port, a sufficient condition is to check that the parameter name appears on every arc connected to each place of the OpCS.

Inter-object invocations in transitions. We want to be able to model CORBA systems, and not only isolated CORBA servers. Actually, the behavior of one class in isolation will often be of little interest, and will only become meaningful when we can describe how an instance of the class may interact with other instances of other classes in the system.

CORBA objects interact with one another by invoking services defined by the IDL interfaces they support. The Cooperative Object formalism supports this form of cooperation by allowing the action of a transition to be the invocation of a service. The operational semantics of such an invocation transition is a client / server protocol that can be formally defined in terms of Petri nets.

This client server protocol we use has been first presented in [27] for basic Petri nets, extended to object-oriented high-level Petri nets in [3], and is presented in [31]. The fact that not only the internal behavior of objects, but also their communication protocol is defined in terms of Petri nets allow us to reason about systems of cooperating objects, and not only on isolated instances. The theoretical details of how this client-server protocol is adapted to the semantics of CORBA inter-object communication are given in [7].

Mapping for exceptions. CORBA IDL allows specifying exceptions that may be raised during the processing of an invocation. An exception is an object of a specific data-type, and can hold information on the causes of its occurrence or other useful data.

When an exception is raised, the normal processing of the service is cancelled, the result, out and inout parameters of the service are undefined, an exception object is instantiated and only this object is transmitted to the client of the invocation.

In order to specify properly the behavior of a CORBA system, our formalism needs to address two concerns:

- Define under which conditions an exception may be raised during the processing of an invocation, and what corrective actions are eventually needed in the server object to restore a consistent state.
- Define what action a client object needs to take if a service invocation results in an exception instead of providing the expected result.

The first point is tackled by *exception transitions*: Exception transitions are labeled by the name of the exception data-type that is raised. They can have input and output arcs from any place of the OpCS of one service, but necessarily have exactly one output arc connected to the SEP of this service. The occurrence of an exception transition models the fact that an exceptional condition has occurred during the processing of an invocation, and that this processing cannot be carried any further. The *T5* transition in Fig. 2 is an exception transition. It models the fact that an account needs to be known by the bank before being closed.

The second point is tackled by:

- a simple syntactic extension of the graphic syntax of invocation transitions (called emission rules), not illustrated in our case study,
- a straightforward extension of the client-server protocol described in [7], acknowledging the fact that an invocation can have two different outcomes: a normal outcome, providing the expected results, or an exception outcome, providing no result other than the exception raised by the server.

```
class BankAccountSpec
specifies BankAccount {
      place closed <float> = { <0> };
      place open <float>;
      transition transferFunds {
            precondition {
                  (amount + balance) >= 0;
            }
            action {
                  newBalance = amount + balance;
            }
      }
      transition t5 {
            precondition {
                  (amount + balance) < 0;
            }
```

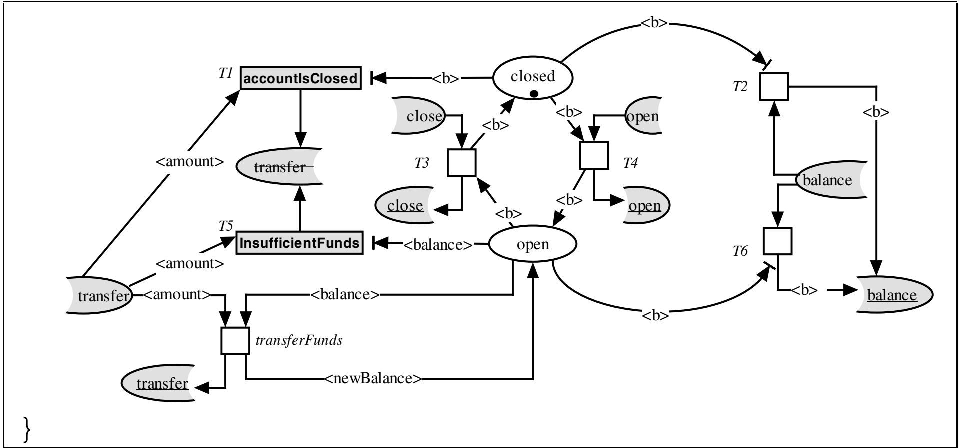


Fig. 3. CO class specifying the *BankAccount* interface.

The behavioral specification of bank accounts is shown in Fig. 3. This CO class specifies the *BankAccount* interface and implicitly specifies the *Account* interface since the two are related through inheritance. Several new syntactic constructs are illustrated in this class:

- Places can be provided with an initial marking, stating their content right after an instance of this class is created. In this case, an account is created in the closed state, with an initial balance of 0, as stated by the definition of the *closed* place.
- The *transferFunds* and *T5* transitions are in structural conflict. This conflict is deterministically solved by the preconditions of these two transitions, which are mutually exclusive. Thus, the *InsufficientFunds* exception is raised when the current balance is lower than the amount that one tries to withdraw.

Note that (for the sake of brevity), the specification detailed in the *BankSpec* and *BankAccountSpec* classes offers no provision for reopening a closed account. This could be achieved, for instance, by providing a subclass of *BankSpec* with additional services to transfer the balance from a closed account to a newly created account.

5 Prototyping CORBA Systems

A well-known advantage of Petri nets is their executability. This is highly beneficial to our approach, since as soon as a behavioral specification is provided in terms of CO classes, this specification can be interpreted to provide additional insights on the possible evolutions of the system.

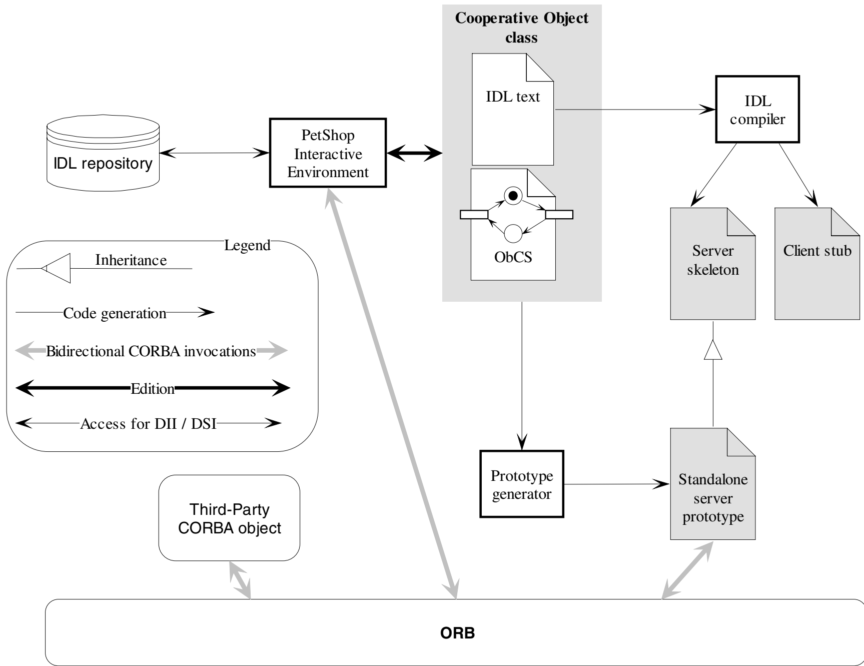


Fig. 4. Architecture of the PetShop environment

Our approach is supported by a tool called PetShop (Fig. 4), which includes a distributed implementation of the high-level Petri net interpreter described in [5], rewritten in Java. The implementation is such that:

- An interpreted ObCS can invoke (from an invocation transition) a service of a "third party" CORBA server running outside of the environment;
- Conversely, an external CORBA client can call a service of a Cooperative Object, regardless of the fact that this service invocation will be performed by interpretation of the CO ObCS.

This offers complete interoperability between the CORBA world and our formalism, and enables us to work in realistic settings, where we possess the complete behavioral specification of some objects only, but where we can nonetheless access objects provided by other sources, of which we know only the CORBA IDL.

In this section, we describe how the PetShop CASE tool fits into the CORBA program development cycle. As a whole, PetShop completely supports the following phases

- Editing of the behavioral specification as a Cooperative Object class;
- IDL generation;
- Cooperative object class analysis;
- Interactive interpretation of models;

- Prototype generation.

Cooperative object class editing. The Petri net editor provides the graphical interface for both the editing of the textual part and the ObCS part of each CO class.

As the ObCS can specify several IDL interfaces, the tool also provides access to existing IDL interfaces and allows for the creation of new IDL interfaces.

IDL generation. The original goal of our approach is to add behavior to IDL. A frequent design activity is to start from pre-defined IDLs (such as the ones provided in the definition of the CorbaServices [23]) and to write a behavioral specification for it. The designer can also start from scratch and generate CORBA IDL matching a given behavioral specification.

Cooperative object class analysis. As Cooperative Objects are a subclass of Petri nets, the analysis module allows proving some interesting properties of the nets such as invariants, conflicts (structural properties), liveness, boundedness, home state (behavioral properties).

We wish to use Petri net analysis techniques not only to prove properties on an isolated object, but also to analyze constructs specific to object-oriented systems. For example, when two IDL interfaces are related through inheritance, some form of behavioral inheritance needs to be respected for CO classes that specify these interfaces [35]. We also want to analyze the co-operation between several CO instances, to check properties of a system of interacting objects. This is possible since the client-server protocol presented in [7] is described in terms of Petri nets, which allows generating a single static Petri net from the ObCS of classes in a system [3], [31]. This global net can be used to prove properties of the system as a whole.

Interactive interpretation of models. One of the well-known advantages of Petri nets is their executability. This feature is exploited in the PetShop tool, that offers the well known advantages of interpreted environments in terms of flexibility, interactivity and ease of use ([19], [21])[21]. PetShop allows an interpreted object to seamlessly access "real" third-party CORBA servers, or to be accessed as a server from other CORBA objects developed independently and maybe in a different programming language.

In our implementation, each edited net is executed using the ObCS interpreter described in [5]: the editor itself acts as a client/server program. As a result, as soon as a net is edited, an instance of the ObCS starts executing. The interpreted instance is accessed using the dynamic CORBA mechanisms of Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI). This allows us to test the behavior in a CORBA runtime environment and make real distributed experiments between CORBA objects hosted by the same or different computers.

The environment allows for graphic debugging of the interpreted net, by examining or modifying its marking, or even modifying its control structure dynamically.

Prototyping Distributed System. In the last phase of development, the programmer is interested in shipping a finished stand-alone CORBA object that can be run without the support of the CASE tool. For that reason PetShop can generate a prototype and

launch instances of the prototypes. Those instances are "almost functional" implementations, in the sense that only behavioral requirements are dealt with. Other "quality of service" requirements, such as performance, persistence, replication or fault-tolerance, have to be taken care of in a completely functional implementation. The generated prototype is still an interpreted high-level Petri net, but does not have graphical features, like graphic debugging or live editing capabilities, which allows it to perform faster, and to function on machines that lack graphic abilities.

6 Conclusion and Future Work

The approach presented here is motivated by the momentum gained by CORBA as a standard for distributed object systems, and by the evidence that some form of abstract behavioral modeling supporting the CORBA object model can be of great help in the development life-cycle of such systems.

Several aspects of the CORBA/Cooperative Objects integration are not presented here, for space reasons: among these are the support for unidirectional service invocation (oneway services in IDL), or the provision for IDL constructs such as constants and context information.

Among the works in progress on our approach are the following:

- Use of Petri net analysis techniques: Our approach has a strong operational bias: we accept to trade full analysis possibilities in favor of modeling power. However, we do not give up the potential for analysis potential of Petri nets altogether. Currently, our tool includes the usual Petri net analysis algorithms (such as P and T-Invariants, liveness, boundedness, siphons and traps) that operate on the ObCS *underlying net*, i.e. the ObCS where all variables and data types are removed. We are currently in the process of assessing which of the analysis result for basic Petri net are preserved in the high-level ObCS. We are also investigating analysis techniques especially devised for high-level nets.
- Analysis of object-oriented features: we wish to use Petri net analysis techniques not only prove properties on an isolated object, but also to analyze constructs specific to object-oriented systems. For example, when two IDL interfaces are related through inheritance, some form of behavioral inheritance needs to be respected for CO classes that specify these interfaces [35]. We also want to analyze the cooperation between several CO instances, to check properties of a system of interacting objects.
- Test generation: Another ongoing research is the ability to generate test suites from the Cooperative Object class definition. The tool supporting our can generate a prototypes that are "almost functional" implementations or CORBA-IDL interfaces, in the sense that only behavioral requirements are dealt with. Other "quality of service" requirements, such as performance, persistence, replication or fault-tolerance, have to be taken care of in a fully functional implementation. The idea is that the CO class can serve as the formal specification of the class, and that programmers will implement the

specification in some programming language, implementing the other quality-of service requirements. This implementation needs to be tested for conformance against the original CO-based specification and the CO specification can be used to generate a test suite for the implementation.

Acknowledgements

The work presented here is funded by CNET, the National Research Center for the French telecommunications agency (France Telecom) under grant number 98 1B 059. The final goal of the project is to provide a fully integrated CASE tool supporting our approach of behavioral specification of distributed object systems using high-level Petri nets.

References

1. Agha, Gul, and De Cindio, Fiorella. "Workshop on Object-Oriented Programming and Models of Concurrency." *16th International Conference on Applications and Theory of Petri Nets, ATPN'95*, Torino, Italy, June 26-30, 1995. Gul Agha, and Fiorella De Cindio, organizers. (1995)
2. Agha, Gul, De Cindio, Fiorella, and Yonezawa, Akinori. "2nd International Workshop on Object-Oriented Programming and Models of Concurrency." *17th International Conference on Applications and Theory of Petri Nets, ATPN'96*, Osaka, Japan, June 24, 1996. Gul Agha, Fiorella De Cindio, and Akinori Yonezawa, editors. (1996)
3. Bastide, Rémi. "Objets Coopératifs : Un Formalisme Pour La Modélisation Des Systèmes Concurrents." Ph.D. thesis, Université Toulouse III (1992).
4. Bastide, Rémi, and Palanque, Philippe. "Cooperative Objects: a Concurrent, Petri-Net Based Object-Oriented Language." *Systems Engineering in the Service of Humans, IEEE-SMC'93*, Le Touquet, France, October 15-20, 1993. IEEE Press (1993)
5. Bastide, Rémi, and Palanque, Philippe. "A Petri-Net Based Environment for the Design of Event-Driven Interfaces." *16th International Conference on Applications and Theory of Petri Nets, ATPN'95*, Torino, Italy, June 1995 . Giorgio De Michelis, and Michel Diaz, Volume editors. Lecture Notes in Computer Science, no. 935. Springer (1995) 66-83.
6. Bastide, Rémi, and Palanque, Philippe. "Modeling a Groupware Editing Tool With Cooperative Objects." *Concurrent Object-Oriented Programming and Petri Nets.* Gul A. Agha, and Fiorella De Cindio, editors. Wien: Springer-Verlag (1998)
7. Bastide, Rémi, Palanque, Philippe, Sy, Ousmane, Le, Duc-Hoa, and Navarre, David. "Petri-Net Based Behavioural Specification of CORBA Systems." *20th International Conference on Applications and Theory of Petri Nets, ATPN'99*, Williamsburg, VA, USA, June 21-25, 1999. (1999)

8. Bryan, Doug. "Exactness and Clarity in a Component-Based Specification Language." *Object-Oriented Behavioral Specifications.* Haim Kilov, and William Harvey, editors. New-York: Kluwer Academic Publishers (1996) 1-15.
9. Bryant, Antony, and Evans, Andy. "A Formal Basis for Specifying Object Behaviour." *Object-Oriented Behavioral Specifications.* Haim Kilov, and William Harvey, editors. New-York: Kluwer Academic Publishers (1996) 17-30.
10. Genrich, H. J., and Lautenbach, K. "System Modelling With High-Level Petri Nets." *Theoretical Computer Science.* Vol. 13. North-Holland (1981) 109-36.
11. Goguen, J. A., Thatcher, J. W., and Wagner, E. G. "Current Trends in Programming Methodology." *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types.*Prentice-Hall (1978) 80-149.
12. Guttag, John V., James J. Horning, S. J. Garland, A. Jones, and J. M. Wing. *Larch: Languages and Tools for Formal Specification* Springer-Verlag, New-York (1993).
13. Harel, David, and Gery, Eran. "Executable Object Modeling With Statecharts." *IEEE Computer* 30, no. 7 (1997) 31-42.
14. Jensen, Kurt. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* 2nd edition ed., Vol. 2 Springer-Verlag (1996).
15. Kilov, Haïm, and William Harvey, editors. *Object-Oriented Behavioral Specifications* Kluwer Academic Publishers (1996).
16. Lakos, Charles. "A General Systematic Approach to Arc Extensions for Coloured Petri Nets." *15th International Conference on Applications and Theory of Petri Nets, ATPN'94*, June 1995. Lecture Notes in Computer Science, no. 815. Springer (1995) 338-57.
17. Lakos, Charles, and Keen, C. D. "LOOPN++: a New Language for Object-Oriented Petri Nets." *European Simulation Multiconference*, Barcelona, Spain, June 1994. (1994)
18. Leavens, Gary T., and Yoonsik Cheon. *Extending CORBA IDL to Specify Behavior With LARCH*, TR #93-20. Iowa State University, Department of Computer Science, 1995.
19. Merle, Philippe, Gransart, Christophe, Geib, Jean-Marc, and Laukien, Marc. "The CorbaScript Language." *ORBOS OMG Meeting on Scripting Languages*, Helsinki, Finland, July 27-31 1998. (1998)
20. Meyer, Bertrand. "Systematic Concurrent Object-Oriented Programming." *Communications of the ACM* 36, no. 9 (1993) 56-80.
21. Netscape Communications Inc. *CORBA Component Scripting OMG TC Revised Joint Submission. Orbos/98-07-02*, Framingham, MA (1998).
22. Object Management Group. *The Common Object Request Broker: Architecture and Specification. CORBA IIOP 2.2 /98-02-01*, Framingham, MA (1998).
23. ———. *Common Object Services Specification /98-07-05*, Framingham, MA (1998).
24. Peterson, James Lyle. *Petri Net Theory and the Modeling of Systems* Prentice-Hall (1981).
25. Prinz, Andreas. "Describing Behaviour in Interfaces." *Formal Methods for Open Object-Based Distributed Systems.* Elie Najm, and Jean Bernard Stefani, editors. Chapman & Hall (1997) 36-43.

26. Puntigam, F. "Types for Active Objects Based on Trace Semantics." *Formal Methods for Open Object-Based Distributed Systems.* Elie Najm, and Jean Bernard Stefani, editors. Chapman & Hall (1997) 4-19.
27. Ramamoorthy, C. V., and Ho, G. S. "Performance Evaluation of Asynchronous Concurrent Systems." *IEEE Transactions of Software Enginnering* 6, no. 5 (1980) 440-449.
28. Rational Software Corporation. *UML Notation Guide*. 1.1 ed.1997.
29. ———. *UML Semantics*. 1.1 ed.1997.
30. Sankar, Sriram. "Introducing Formal Methods to Software Engineers Through OMG's CORBA Environment and Interface Definition Language." *Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96*, Munich, Germany, July 1-5, 1996. Martin Wirsing, and Maurice Nivat, Editors. ed. Lecture Notes In Computer Science, no. 1101. Springer (1996) 52-61.
31. Sibertin-Blanc, Christophe. "Cooperative Nets." *15th International Conference on Applications and Theory of Petri Nets, ATPN'94*, June 1995. Lecture Notes in Computer Science, no. 815. Springer (1995) 471-90.
32. Siegel, Jon. "OMG Overview: CORBA and the OMA in Enterprise Computing." *Communications of the ACM* 41, no. 10 (1998) 37-43.
33. Sivaprasad, Gowri Sankar. *Larch/CORBA: Specifying the Behavior of CORBA-IDL Interfaces*, TR #95-27a. Iowa State University, Department of Computer Science, 1995.
34. Valk, Rüdiger. "Petri Nets As Token Objects: an Introduction to Elementary Object Nets." *19th International Conference on Applications and Theory of Petri Nets, ATPN'98*, Lissabon, Portugal, June 1998. Springer (1998)
35. van der Aalst, W. M. P., and Basten, T. "Life-Cycle Inheritance, a Petri-Net Based Approach." *18th International Conference on Applications and Theory of Petri Nets, ATPN'97*, Toulouse, France, June 1997. Pierre Azéma, and Gianfranco Balbo, editors. Lecture Notes in Computer Science, no. 1248. Springer (1997) 62-81.

A Process Algebraic Specification of the New Asynchronous CORBA Messaging Service*

Mauro Gaspari and Gianluigi Zavattaro

Department of Computer Science, University of Bologna
Mura Anteo Zamboni 7, 40127 Bologna, Italy
{gaspari,zavattar}@cs.unibo.it

Abstract. CORBA (The Common Object Request Broker Architecture) has to continually evolve in order to cope with the changes of requirement of applications which become larger and more distributed. For this reason new features are being added to the CORBA specification, for instance the last proposal for a revised CORBA Messaging Service includes two new asynchronous models of request invocation. Since these new features will be added in the next CORBA implementations a relevant issue is to study their operational behaviour from different perspectives in order to facilitate the task of implementors. This paper addresses this issue providing an analysis of the CORBA Messaging Service which includes the new asynchronous features. In particular we illustrate how CORBA models for request invocation can be mapped into a message passing architecture based on the actor model. For this purpose we exploit an algebra of actors which supports some of the main features of the abstract Object Model of the Object Management Group, such as object identity and an explicit notion of state. This approach allows us to discuss and compare different models of request invocation in a standard process algebraic perspective for instance we show how different notions of equivalence, such as standard and asynchronous bisimulation, can be adapted to reason about CORBA.

1 Introduction

CORBA (The Common Object Request Broker Architecture) is the object-oriented standard for integrating applications running in heterogeneous distributed environments developed by the Object Management Group (OMG). One of the main limitations in the CORBA specification concerning the ability to cope with large scale distributed systems has been the lack of support for asynchronous models of request invocation. This hole has been recognized by OMG in the Messaging Service RFP [13] and in the next version of CORBA [32] new features will be provided in order to overcome this limitation. The proposal for a revised CORBA messaging service was submitted one year ago by a group of companies which are members of the OMG consortium [14]. This new proposal

* This paper has been partially supported by the Italian Ministry of Universities (MURST).

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 495–518, 1999.
© Springer-Verlag Berlin Heidelberg 1999

extends the synchronous models of request invocation which were supported in CORBA 2.2 [16] with two new asynchronous models: callback and polling.

Since these new features will be added in the next CORBA implementations a relevant issue is to study their operational behaviour from different perspectives in order to facilitate the task of implementors. This paper addresses this issue providing an analysis of the CORBA Messaging Service which includes the new asynchronous features. In particular we illustrate how CORBA models for request invocation can be mapped into a message passing architecture based on the actor model. For this purpose we exploit an algebra of actors [10,11,9] which supports some of the main features of the abstract Object Model of the Object Management Group, such as object identity and an explicit notion of state.

This approach allows us to discuss and compare different models of request invocation in a standard process algebraic perspective where a nice, easy-to-understand specification style is supported. Thus we can reuse standard results of the theory of concurrency for reasoning on CORBA models of request invocation. For instance we show how different notions of equivalence, such as standard and asynchronous bisimulation, can be adapted to reason about CORBA.

The paper is organized as follows. Section 2 briefly describes the CORBA messaging facilities and the underlying object model. In Section 3 we introduce our algebra of actors. In Section 4 we specify the CORBA models for request invocation in the algebra of actors and we introduce a framework for reasoning on CORBA requests. In Section 5 we illustrate our approach for reasoning on a simple CORBA program. We conclude the paper by discussing related works and drawing conclusions.

2 The New CORBA Messaging Service

CORBA has been designed to support the integration of a wide variety of object systems providing solutions to the many technical problems that arise in this context. For this reason the CORBA architecture is quite complex and includes several components to deal with open systems and interoperability, for instance, the Interface Definition Language (IDL), the Dynamic Invocation Interface (DII), and, the Object Adapters [32].

CORBA clients can make requests exploiting an OMG IDL typed stub (a procedure which executes a single operation - the request - depending on the interface of the target object) or the dynamic invocation interface (an interface which is independent from the target object interface). Requests are sent through the Objects Request Broker (ORB) which is responsible for all of the mechanisms needed for transmitting them to the right destination. In particular the Object Adapter mediates between CORBA objects and the programming language used for the server implementation.

Although the CORBA architecture deals with all the interoperability issues presented above, a client exploits an interface which is completely independent concerning these details. Clients making their requests have only to consider the address of the server, the interface of the server (the operation supported by the

server object) and the selection of an adequate protocol for request invocation. Moreover the semantics of the dynamic or the stub interface for invoking a request is the same and the server receiving a message cannot tell how the request was invoked.

All these considerations suggest that a more abstract model can be used for reasoning on CORBA messaging facilities where all the low level aspects are hidden. This is the role of the CORBA concrete object model [16] based on the abstract OMG object model [12]. The abstract OMG object model defines basic concepts such as objects, operations, types and interfaces and it is common to all the OMG technologies. The CORBA concrete object model is summarized below.

2.1 The CORBA Object Model

The concrete object model of CORBA [16] distinguishes between clients (requestors of services) and servers (providers of services). In the following we adopt this concrete model for reasoning on messaging facilities of CORBA with the main restriction of considering an untyped world. This implies that we will not deal with subtyping and inheritance issues but we will only concentrate on dynamic issues concerning the execution of requests.

We summarize here some of the main aspects of this model considering an untyped scenario where we also abstract away from the concepts of exceptions and request context. The reader interested in complete specification of the CORBA concrete object model can refer to [12].

- Each object has an identifier (its name) that provides a means to refer to the object and it is used for dispatching requests. The name (identity) is immutable and persists as long as the object exists.
- Clients (the requestors of services) are conceptually distinguished from servers (the providers of services).
- Server objects have a local memory and a behaviour that defines the meaningful operations. They are encapsulated entities accessible through a set of operations: the object interface.
- Clients send requests to server objects.
- Requests include an operation, a target object, a request identifier and zero or more parameters. One possible outcome for a request is returning to the client the result.
- When a client issues a request a method of the target object is called. A method specifies how a server executes the requested service. The input parameters are those of the request and the result is sent back to the requestor. The execution of a method may change the state of a server.
- From a client's point of view objects can only be created and destroyed as an outcome of issuing requests.

2.2 CORBA Models of Request Invocation

According to the new proposal for the CORBA Messaging Service, clients perform requests exploiting synchronous or asynchronous operations. CORBA requests are objects that include all the information that should be transmitted to the server. In the CORBA DII requests must be created explicitly before being transmitted to the server object. In the following we do not consider all the details concerning the creation of new requests and we provide an abstract representation of them. We represent requests as records and we extract some of the most significant parameters, making them directly available in the operation.

Usually a request contains additional flags specifying different semantics for the associated operation. For instance a flag specifies if an operation is blocking or not blocking. For the limited scope of this paper we will not consider all the possible combinations of requests and flags and we will restrict our study to the operational behaviour described below. We will use a notation based on the Dynamic Invocation Interface which is more adequate for our operational purposes.

Models of Request Invocation of CORBA 2.2 The models of request invocation of the current version of CORBA are:

– **Synchronous:** *INVOKE(Server,Request,Result)*
  A client performing this request waits until the server sends the answer. The result is placed in the *Result* argument. This is similar to a remote procedure call.
– **Oneway:** *ONEWAY(Server,Request)*
  This invocation returns the control to the caller object, the request is transmitted and there is no response.
– **Deferred Synchronous:** *SEND(Server,Request)*
  This invocation transmits the request to the server and returns the control to the caller object without waiting for the operation to finish. To get the result the calling object can perform the *get_response* operation having the same request identifier as an argument.
  GET_RESPONSE(Server,Request,Result) this operation gets the first answer of the request, if it is ready otherwise it blocks waiting for the result. The result is placed in the *Result* argument. CORBA also includes a GET_NEXT_RESPONSE primitive to retrieve other possible results; this feature will not be modelled in this paper.

New Asynchronous Models of Request Invocation The proposal for a new CORBA messaging facility extends CORBA 2.2 with two new asynchronous models of request invocation.

– **Callback** *SENDC(Server,Request,Callback)*
  When a client invokes this operation it sends the request to a server object and it creates a reply handler object. The answer will be processed by this object and *Callback* represents its behaviour.

– **Polling** *SENDP(Server,Request,Poller)*
  When a client invokes this operation the address of a local poller object is returned in *Poller*. This object will be used to retrieve the answer exploiting the *POLL_GET_RESPONSE* operation. A poller object is also able to execute an *IS_READY* operation which returns true if and only if the answer is available in the poller object.

2.3 A Process Algebraic Framework for CORBA

We provide a formal framework for all the above models of request invocation following a process algebraic approach. Process algebras, like CCS [22] and CSP [18], have been developed as formalisms for the study of concurrent systems. Initially, process algebras allowed interprocess communication via a static structure of channels between processes. Mobility, one of the basic features of modern object oriented systems (where new objects can be created at run time and/or moved in different locations), was not easily representable in these formalisms. Note that the CORBA callback and polling models of request invocation need a more dynamic structure of channels.

The π-calculus [25] can be considered as the main attempt in order to overcome these limitations. In fact, it has been introduced as a calculus for *mobile* processes, *i.e.,* processes with a dynamically changing linkage structure. The π–calculus has been developed taking into account a synchronous handshake communication mechanism between processes. More recently, in [19,7] also an asynchronous fragment of the π–calculus has been studied in order to analyze also the asynchronous communication mechanism and its similarities/differences with the synchronous one [27].

There have been several attempts to adopt the π-calculus and its asynchronous version, for modelling interaction in the context of concurrent object oriented programming languages [23,33,28], but these approaches do not provide the concepts of state of a process or of object identity as a first class entities.

On the other hand, the actor model [17,2] directly deals with many features of the OMG Core Object Model such as object identity, state and operations associated to objects that characterize their behaviour; an actor has the same structural and interaction properties as a CORBA object if we restrict to an untyped world.

For this reason we exploit a new formalism for our modelling purposes, an algebra of actors [11,10,9], which has been designed as a compromise between the standard process algebras, such as the π-calculus [24], and the actor model. This process algebra supports an high level specification style which allows the user to describe how CORBA models of request invocation can be mapped into sequences of message-passing primitives.

2.4 The Actor Model

The actor model was introduced by Carl Hewitt about 20 years ago [17]. Actors are self-contained agents with a state and a behaviour which is a function of

incoming communications. Each actor has a unique name (mail address) determined at the time of its creation. This name is used to specify the recipient of a message supporting object identity [20], a property of an object which distinguishes this object from all the others. Object identity is a typical feature of object-oriented programming languages and it is used as a basic dispatching mechanism in message passing. This property is not easily embeddable in formalisms such as CCS [22] (or asynchronous π-calculus [19,7]), where message dispatching is performed by means of channels. In these formalisms the association address-process is not unique: a process may have several ports (channels) from which it receives messages and the same channel can be accessed by different processes.

Actors communicate by asynchronous and reliable message passing, *i.e.,* whenever a message is sent it must eventually be received by the target actor. Actors exploit an implicit receive mechanism. A receive operation is explicit when it appears in programs, while it is implicit when it does not correspond to an operation in the programming language and it is performed implicitly at certain points of the computation. An implicit receive mechanism is common in object-oriented programming where objects can be seen as passive entities which react to messages or to method invocation.

Actors make use of three basic primitives which are asynchronous and non-blocking: *create*, to create new actors; *send*, to send messages to other actors; and *become*, to change the behaviour of an actor [2].

3 An Algebra of Actors

Let $\mathcal{A}$ be a countable set of *actor names*: a, b, c, a_i, b_i,... will range over $\mathcal{A}$ and L, L', L'',... will range over its (finite) power set $\mathcal{P}_{fin}(\mathcal{A})$ (*i.e.,* L, L', $L'' \subseteq_{fin} \mathcal{A}$). Let $\mathcal{V}$ be a set of values (with $\mathcal{A} \subset \mathcal{V}$) containing, e.g., *true*, *false*, and let $\mathcal{X}$, ranged over by x, y, z,..., be a set of value variables that are bound to values at run-time. We assume value expressions e built from actor names, value constants, value variables, the expressions *self*, *state*, and *message*, and any operator symbol we wish. In the examples we will use standard operators on sequences: $1st$, $2nd$, *rest*, *empty*. We will denote values by v, v', v'',... when they appear as contents of a message and with s, s', s'',... when they represent the state of an actor. $[\![e]\!]^a_s$ gives the value of e in $\mathcal{V}$ assuming that a and s are substituted for *self* and *state* inside e; e.g. $[\![self]\!]^a_s = a$ and $[\![state]\!]^a_s = s$. The special expression *message* represents the contents of the last received message. Whenever a message is received, its contents is substituted for each occurrence of the expression *message* in the receiving actor.

Let $\mathcal{C}$ be a set of *actor behaviours* identifiers: C, C', ... will range over $\mathcal{C}$. We suppose that every identifier C is equipped with a corresponding behaviour definition $C \stackrel{def}{=} P$ where P is a program, that is a term defined by the following abstract syntax:

$$P ::= become(C, e).P \mid send(e_1, e_2).P \mid create(b, C, e).P \mid \\ e_1{:}P_1 + \ldots + e_n{:}P_n \mid \surd$$

Observe that we allow recursive behaviours to be defined, for example we could have $C \stackrel{def}{=} become(C, state).\surd$.

Actor terms are defined by the following abstract syntax:

$$A ::= {}^{\mathbf{a}}\mathbf{C}_{\mathbf{s}} \mid {}^{\mathbf{a}}[\mathbf{P}]_{\mathbf{s}} \mid \langle \mathbf{a}, \mathbf{v} \rangle \mid \mathbf{A}|\mathbf{A} \mid \mathbf{A}\backslash\mathbf{a} \mid \mathbf{0}$$

An actor can be idle or active. An idle actor ${}^{\mathbf{a}}\mathbf{C}_{\mathbf{s}}$ (composed by a behaviour $\mathbf{C}$, a name $\mathbf{a}$, and a state $\mathbf{s}$) is ready to receive a message. When a message is received the actor becomes active. Active actors are denoted by ${}^{\mathbf{a}}[\mathbf{P}]_{\mathbf{s}}$ where $\mathbf{P}$ is the program that is executed. The actor $\mathbf{a}$ will not receive new messages until it becomes idle (by performing a **become** primitive). Sometimes the state $\mathbf{s}$ is omitted when empty (i.e. $\mathbf{s} = \emptyset$). A program $\mathbf{P}$ is a sequence of actor primitives (**become**, **send** and **create**) and guarded choices $\mathbf{e}_1{:}\mathbf{P}_1 + \ldots + \mathbf{e}_\mathbf{n}{:}\mathbf{P}_\mathbf{n}$ terminating in the null program $\surd$ (which is usually omitted). An actor term is the parallel composition of (active and idle) actors and messages. A message is denoted by a term $\langle \mathbf{a}, \mathbf{v} \rangle$ where $\mathbf{v}$ is the contents and $\mathbf{a}$ the name of the actor the message is sent to. Also a restriction operator $\mathbf{A}\backslash\mathbf{a}$ is used in order to allow the definition of local actor names ($\mathbf{A}\backslash\mathbf{L}$ is used as a shorthand for $\mathbf{A}\backslash\mathbf{a}_1\backslash\ldots\backslash\mathbf{a}_\mathbf{n}$ if $\mathbf{L} = \{\mathbf{a}_1, \ldots, \mathbf{a}_\mathbf{n}\}$) while $\mathbf{0}$ is the usual empty term.

The actor primitives and the guarded choice are described below.

- **send**:
  The program $\mathbf{send}(\mathbf{e}_1, \mathbf{e}_2).\mathbf{P}$ sends a message with contents $\mathbf{e}_2$ to the actor indicated by $\mathbf{e}_1$:

  $${}^{\mathbf{a}}[\mathbf{send}(\mathbf{e}_1, \mathbf{e}_2).\mathbf{P}]_{\mathbf{s}} \xrightarrow{\tau} {}^{\mathbf{a}}[\mathbf{P}]_{\mathbf{s}} \mid \langle [\![\mathbf{e}_1]\!]^{\mathbf{a}}_{\mathbf{s}}, [\![\mathbf{e}_2]\!]^{\mathbf{a}}_{\mathbf{s}} \rangle$$

  where τ represents an internal invisible step of computation.

- **become**:
  The program $\mathbf{become}(\mathbf{C}, \mathbf{e}).\mathbf{P}'$ changes the state of the actual actor from active to idle:

  $${}^{\mathbf{a}}[\mathbf{become}(\mathbf{C}, \mathbf{e}).\mathbf{P}']_{\mathbf{s}} \xrightarrow{\tau} ({}^{\mathbf{d}}[\mathbf{P}'\{\mathbf{a}/\mathbf{self}\}]_{\mathbf{s}})\backslash\mathbf{d} \mid {}^{\mathbf{a}}\mathbf{C}_{[\![\mathbf{e}]\!]^{\mathbf{a}}_{\mathbf{s}}} \qquad \text{with } \mathbf{d} \text{ fresh}$$

  The primitive **become** is the only one that permits to change the state according to the expression $\mathbf{e}$; we sometimes omit $\mathbf{e}$ if the state is left unchanged (i.e. $\mathbf{e} = \mathbf{state}$). The continuation $\mathbf{P}'$ is executed by the new actor ${}^{\mathbf{d}}[\mathbf{P}'\{\mathbf{a}/\mathbf{self}\}]_{\mathbf{s}}$. This actor will never receive other messages (i.e. it is unreachable) as its name $\mathbf{d}$ cannot be known to any other actor. Indeed, the expression **self**, which is the only one that returns the value $\mathbf{d}$, is changed in order to refer to the name $\mathbf{a}$ of the initial actor.

- **create**:
  The program $\mathbf{create}(\mathbf{b}, \mathbf{C}, \mathbf{e}).\mathbf{P}'$ creates a new idle actor having state $\mathbf{s}$ and behaviour $\mathbf{C}$:

$$^{\mathbf{a}}[\mathbf{create}(\mathbf{b},\mathbf{C},\mathbf{e}).\mathbf{P}']_{\mathbf{s}} \xrightarrow{\tau} (^{\mathbf{a}}[\mathbf{P}'\{\mathbf{d}/\mathbf{b}\}]_{\mathbf{s}} \mid {}^{\mathbf{d}}\mathbf{C}_{[\![\mathbf{e}]\!]^{\mathbf{a}}_{\mathbf{s}}})\backslash\mathbf{d} \qquad \text{with } \mathbf{d} \text{ fresh}$$

The new actor receives a fresh name **d**. This new name is initially known only to the creating actor, in fact a restriction on the new name **d** is introduced.

- $\mathbf{e}_1{:}\mathbf{P}_1 + \ldots + \mathbf{e_n}{:}\mathbf{P_n}$:
  In the agent $\mathbf{e}_1{:}\mathbf{P}_1 + \ldots + \mathbf{e_n}{:}\mathbf{P_n}$, the expressions $\mathbf{e_i}$ are supposed to be boolean expressions with value **true** or **false**. The branch $\mathbf{P_i}$ can be chosen only if the value of the corresponding expression $\mathbf{e_i}$ is **true**:

$$^{\mathbf{a}}[\mathbf{e}_1{:}\mathbf{P}_1 + \ldots + \mathbf{e_n}{:}\mathbf{P_n}]_{\mathbf{s}} \xrightarrow{\tau} {}^{\mathbf{a}}[\mathbf{P_i}]_{\mathbf{s}} \qquad \text{if } [\![\mathbf{e_i}]\!]^{\mathbf{a}}_{\mathbf{s}} = \mathbf{true}$$

The function **n** returns the set of the actor names appearing in an expression, a program, or an actor term. Given the actor term **A**, the set **n**(**A**) is partitioned in **fn**(**A**) (the free names in **A**) and **bn**(**A**) (the bound names in **A**) where the bound names are defined as those names **a** appearing in **A** only under the scope of some restriction on **a**. We use **act**(**A**) to denote the set of the names of the actors in **A**. An actor term is well formed if and only if it does not contain two distinct actors with the same name. In the following we will consider only well formed agents, and we will use Γ to denote the set of well formed terms (**A**, **B**, **D**, **E**, **F**,... will range only over Γ).

We model the operational semantics of our language following the approach of Milner [23] which consists in separating the laws which govern the static relation among actors (for instance $\mathbf{A}|\mathbf{B}$ is equivalent to $\mathbf{B}|\mathbf{A}$) from the laws which rules their interaction. This is achieved by defining a static structural equivalence relation over syntactic terms and a dynamic relation by means of a labelled transition system [29].

Definition 1. - **Structural congruence,** *is the smallest congruence relation over actor terms ($\equiv$) satisfying:*

(i)	$^{\mathbf{a}}[\surd]_{\mathbf{s}} \equiv \mathbf{0}$	**(v)**	$\mathbf{0}\backslash\mathbf{a} \equiv \mathbf{0}$	
(ii)	$\mathbf{A}\|\mathbf{0} \equiv \mathbf{A}$	**(vi)**	$(\mathbf{A}\backslash\mathbf{a})\backslash\mathbf{b} \equiv (\mathbf{A}\backslash\mathbf{b})\backslash\mathbf{a}$	
(iii)	$\mathbf{A}\|\mathbf{B} \equiv \mathbf{B}\|\mathbf{A}$	**(vii)**	$(\mathbf{A}\|\mathbf{B})\backslash\mathbf{a} \equiv \mathbf{A}\|(\mathbf{B}\backslash\mathbf{a})$	*where* $\mathbf{a} \notin \mathbf{fn}(\mathbf{A})$
(iv)	$(\mathbf{A}\|\mathbf{B})\|\mathbf{D} \equiv \mathbf{A}\|(\mathbf{B}\|\mathbf{D})$	**(viii)**	$\mathbf{A}\backslash\mathbf{a} \equiv \mathbf{A}\{\mathbf{b}/\mathbf{a}\}\backslash\mathbf{b}$	*where* **b** *is fresh*

Definition 2. - **Computations.** *A transition system modelling computations in the actor algebra is represented by the triple $(\Gamma, T, \{\stackrel{\alpha}{\rightarrow} \mid \alpha \in T\})$. $T = \{\tau\} \cup \{av, \overline{av}L \mid a \in \mathcal{A},\ v \in \mathcal{V},\ L \subseteq_{fin} \mathcal{A}\}$ is a set of labels, where τ is the invisible action standing for internal autonomous steps of computation; av and $\overline{av}L$ respectively represent the receiving and the emission of the message with receiver a and contents v. The set L in the label $\overline{av}L$ represents the set of actor names in the expression v which were initially under the scope of some restriction. $\stackrel{\alpha}{\rightarrow}$ is the minimal transition relation satisfying the axioms and rules presented in Table 1.*

Table 1. Operational semantics.

Rule		Condition
Send	${}^{a}[send(e_1, e_2).P]_s \xrightarrow{\tau} {}^{a}[P]_s \mid \langle [\![e_1]\!]^a_s, [\![e_2]\!]^a_s \rangle$	
Deliver	$\langle a, v \rangle \xrightarrow{\overline{av}\emptyset} 0$	
Become	${}^{a}[become(C, e).P']_s \xrightarrow{\tau} ({}^{d}[P'\{a/self\}]_s)\backslash d \mid {}^{a}C_{[\![e]\!]^a_s}$	d fresh
Create	${}^{a}[create(b, C, e).P']_s \xrightarrow{\tau} ({}^{a}[P'\{d/b\}]_s \mid {}^{d}C_{[\![e]\!]^a_s})\backslash d$	d fresh
Receive	${}^{a}C_s \xrightarrow{av} {}^{a}[P\{v/message\}]_s$	if $C \stackrel{def}{=} P$
Guard	${}^{a}[e_1{:}P_1 + \ldots + e_n{:}P_n]_s \xrightarrow{\tau} {}^{a}[P_i]_s$	if $[\![e_i]\!]^a_s = true$
Res	$\dfrac{A \xrightarrow{\alpha} A'}{A\backslash a \xrightarrow{\alpha} A'\backslash a}$	$a \notin n(\alpha)$
Open	$\dfrac{A \xrightarrow{\overline{av}L} A'}{A\backslash b \xrightarrow{\overline{av}L \cup \{b\}} A'}$	$a \neq b \ \wedge \ b \in n(v)$
Par	$\dfrac{A \xrightarrow{\alpha} A'}{A \vert B \xrightarrow{\alpha} A' \vert B}$	if $\alpha = \overline{av}L$ then $a \notin act(B) \wedge$ $L \cap fn(B) = \emptyset$
Sync	$\dfrac{A \xrightarrow{av} A' \qquad B \xrightarrow{\overline{av}L} B'}{A \vert B \xrightarrow{\tau} (A' \vert B') \setminus L}$	
Cong	$\dfrac{B \equiv A \quad A \xrightarrow{\alpha} A' \quad A' \equiv B'}{B \xrightarrow{\alpha} B'}$	

The rules *Send*, *Become*, *Create* and *Guard* have been already discussed. Rule *Deliver* states that the term $\langle a, v \rangle$ (representing a message v sent to the actor a) is able to deliver its contents to the receiver by performing the action $\overline{av}\emptyset$. The corresponding receiving action labelled with av can be performed by the actor a when it is idle (rule *Receive*). Note here the use of the expression *message* which is replaced by the content of the incoming message v in the program P. The other rules are simply adaptation to our calculus of the standard laws for the π–calculus. The most interesting difference is due to the fact that in our calculus, more than one restriction can be extended by one single delivering operation. In fact, in our case the contents of a message is an expression instead of a unique name. This is the reason why we have added the set L to the label $\overline{av}L$. Another difference is in the rule *Par*: the actor term $A|B$ can deliver a message inferred by A (*i.e.,* execute an emission action $\overline{av}L$), only if B does not contain the target actor (see side condition $a \notin act(B)$).

The fact that an output label can be performed by an agent only if it does not contain the target actor introduces in our calculus an interesting strong form of

unique receptiveness property. Indeed, the operational semantics not only forces that a message can be consumed uniquely by its receptor, but also ensures that external actors are neither able to observe the presence of a pending message sent to another actor. We will show in the remainder of the paper nice features of our calculus related to this property.

3.1 Equivalence of Actor Terms

As already stated, one of the advantages of having introduced a semantics for actors based on a labelled transition system is that standard observational semantics for process algebras can be used. In this section we investigate two of them based on the notion of bisimulation: the *weak bisimulation* [22] (only bisimulation in the following) and the *asynchronous weak bisimulation* [19,5] (only asynchronous bisimulation in the following) which is the corresponding equivalence for languages based on asynchronous communication.

In order to define equivalences which do not take into account the τ steps, we recall the notion of *weak* transition which allows to contract successive τ-steps:

$$P \stackrel{\tau}{\Longrightarrow} P' \text{ iff } P(\stackrel{\tau}{\longrightarrow})^{\star} P'$$
$$P \stackrel{\alpha}{\Longrightarrow} P' \text{ iff exists } P'' \text{ and } P''' \text{ s.t. } P \stackrel{\tau}{\Longrightarrow} P'' \stackrel{\alpha}{\longrightarrow} P''' \stackrel{\tau}{\Longrightarrow} P' \quad (\text{for } \alpha \neq \tau)$$

Observe that given $P \stackrel{\tau}{\Longrightarrow} P'$ P' can be simply P if no τ transition is performed.

Definition 3. - Bisimulation. *A symmetric relation $\mathcal{R}$ on actor terms ($\mathcal{R} \subseteq \Gamma \times \Gamma$) is a* bisimulation *if $(A, B) \in \mathcal{R}$ implies:*

- *if $A \stackrel{\alpha}{\longrightarrow} A'$ then there exists B' such that $B \stackrel{\alpha}{\Longrightarrow} B'$ and $(A', B') \in \mathcal{R}$.*

Two actors A and B are bisimilar*, written $A \approx B$, if there exists a bisimulation $\mathcal{R}$ such that $(A, B) \in \mathcal{R}$.*

Observe that we are dealing with the standard *weak* bisimulation in which a τ step can be simulated by simply making no action. This kind of bisimulation is not a congruence in the presence of a general choice composition operator; in our case, the unique composition operators for actor terms are the parallel $A|A'$ and the restriction $A \setminus a$ operators, and it is not difficult to see that the bisimulation equivalence we have defined is a congruence w.r.t. these operators (*i.e.,* if $A \approx B$ then for every actor term D and actor name a, $A|D \approx B|D$ and $A \setminus a \approx B \setminus a$).

Example 1. Consider the actor term $A = {}^{a}Double$ which receives messages represented as pairs (b, v) where the first argument is an actor name and the second argument is an integer, and sends to the actor b the integer $2 * v$. This behaviour is defined formally below:

$$Double \stackrel{def}{=} send(1st(message), 2 * 2nd(message)).become(Double)$$

Suppose now that we want to build an interface that receives messages and

forward them to an actor which duplicates them. This job is performed by the actor term:

$$B = \ {}^{a}Forward \mid {}^{b}Double$$

where the behaviour *Forward* is:

$$Forward \stackrel{def}{=} send(b, message).become(Forward)$$

The actor terms A and B are not bisimilar because the term B has two addresses that can be reached from the outside (the action bv cannot be observed in the term A).

In order to make the actor ${}^{b}Double$ unreachable from the outside, we can think to introduce the following restriction on the actor name b:

$$B' = \ ({}^{a}Forward \mid {}^{b}Double)\backslash b$$

As the action bv is no more observable, we have that $B' \approx A$. In order to prove this equivalence, it is enough to see that the forward operation is composed of only unobservable τ labelled steps.

For languages based on asynchronous communication a new notion of *asynchronous bisimulation* has been introduced in [19] and formally analyzed in [5]. The basic difference between the asynchronous bisimulation and the standard (synchronous) one, is that in the asynchronous case, the action of removing a message and immediately reintroducing it, is considered as unobservable. This difference reflects the idea that an observer cannot synchronise with the observed system but can only send messages and look at what may come out of it.

Definition 4. - Asynchronous bisimulation. *A symmetric relation $\mathcal{R}$ on actor terms ($\mathcal{R} \subseteq \Gamma \times \Gamma$) is an* asynchronous bisimulation *if $(A, B) \in \mathcal{R}$ implies:*

- *if $A \stackrel{\alpha}{\longrightarrow} A'$ where $\alpha \neq av$ then there exists B' such that $B \stackrel{\alpha}{\Longrightarrow} B'$ and $(A', B') \in \mathcal{R}$.*
- *if $A \stackrel{av}{\longrightarrow} A'$ then there exists B' such that $B \stackrel{av}{\Longrightarrow} B'$ and $(A', B') \in \mathcal{R}$ or $B \stackrel{\tau}{\Longrightarrow} B'$ and $(A', B' | \langle a, v \rangle) \in \mathcal{R}$.*

Two actors A and B are asynchronous bisimilar*, written $A \approx_a B$, if there exists an asynchronous bisimulation $\mathcal{R}$ such that $(A, B) \in \mathcal{R}$.*

As for the standard bisimulation, it is not difficult to prove that also the asynchronous bisimulation is a congruence.

It is clear from the definition that the asynchronous bisimulation is coarser with respect to the standard bisimulation, *i.e.,* if $A \approx B$ then also $A \approx_a B$. For example, the equivalences described in the example 1 hold also if we take into account $\approx_a$ instead of $\approx$.

The following example shows the need to move to the asynchronous bisimulation in order to formally analyze interesting aspects of the actor model.

Example 2. We consider two actors implementing two different communication medias: a queue and an ether, *i.e.,* an unordered set (mailbox) of messages [22]. The behaviours of the two actors are defined as follows:

$QUEUE \stackrel{def}{=}$
$(1st(message) = get \ \wedge \ empty(state))$: $send(self, message).$
$become(QUEUE)+$
$(1st(message) = get \ \wedge \ \neg empty(state))$: $become(QUEUE, rest(state)).$
$send(2nd(message), 1st(state))+$
$(1st(message) = put)$: $become(QUEUE, insert_last(2nd(message), state))$

$ETHER \stackrel{def}{=}$
$(1st(message) = get \ \wedge \ empty(state))$: $send(self, message).$
$become(ETHER)+$
$(1st(message) = get \ \wedge \ \neg empty(state))$: $become(ETHER, rest(state)).$
$send(2nd(message), 1st(state))+$
$(1st(message) = put)$: $become(ETHER, insert_rand(2nd(message), state))$

The two actor programs are assumed to receive messages with the following structure: $(put, item)$ or $(get, sender)$. The two programs differ only for the functions *insert_last* and *insert_rand*, the first inserts a message at the end of a sequence and the second in a random position.

The property of arrival-order non-determinism in message delivery holding in the actor model, suggests that there is no way to distinguish between the behaviour of the *ETHER* and the *QUEUE*. Indeed, an external observer has no way to force a particular order of reception of the data to introduce in the buffer, so we can always consider that data are received by the *QUEUE* in the same order as they are stored in the *ETHER*.

Formally, we have that this equivalence is not captured by the standard bisimulation as it requires that each input operation should be matched exactly by an equivalent input operation, that is, the *ETHER* and the *QUEUE* are forced to consume data in the same order. In order to capture the above equivalence we need to move to the asynchronous bisimulation; indeed, it is not difficult to prove that ${}^{b}ETHER \approx_a {}^{b}QUEUE$ for any actor name b. We do not report the complete proof for this equivalence, because standard, but we only point out some interesting related aspects.

When we exploit the asynchronous bisimulation we can assume that an input operation consuming a message with contents (put, d) performed by ${}^{b}ETHER$ could be matched by ${}^{b}QUEUE$ by performing no action, provided that the agent ${}^{b}QUEUE|\langle b, (put, d)\rangle$ is equivalent to the term reached by the actor ${}^{b}ETHER$. In order to prove this, we observe that ${}^{b}ETHER_{\{d\}} \approx_a {}^{b}QUEUE|\langle b, (put, d)\rangle$. It is interesting to observe that this equivalence holds due to the previously described *unique repectiveness* property, indicating that no external actor can observe the presence of the pending message $\langle b, (put, d)\rangle$.

Example 3. The actor model does not provide an explicit primitive for receiving only data having a certain structure or a certain contents. In fact an actor

can read a message only when it is idle and each available message can be read independently from its sender or its contents. On the other hand, in real applications it is often necessary to specify the kind of data that are needed to be received in a particular point of the computation.

In this example we use the asynchronous bisimulation to prove the correctness of the implementation of a new command that explicitly permits to receive only messages containing a particular identifier. As we want to have more information than the identifier inside the contents of the message, we consider that the messages are records with a field *id*, containing the identifier, and possibly other fields that we left unspecified.

In general we denote a record r by $(f_1{:}e_1, \ldots, f_n{:}e_n)$ and the selection of the value in the field f_i is written as $r.f_i$.

We consider a new primitive $\mathit{receive}(e, x)$ which forces to receive only a message with the value e in the field *id*. Formally, we extend the syntax of the language by allowing also programs of the following kind:

$$P ::= \ \mathit{receive}(e, x).P$$

having the following operational semantics:

$$^{a}[\mathit{receive}(e, x).P]_s \xrightarrow{av} {}^{a}[P\{v/x\}]_s \qquad \text{if } v.\mathit{id} = [\![e]\!]^{a}_{s}$$

We present an implementation of the new primitive in the initial algebra which preserves the asynchronous bisimulation semantics; in other words, we prove that for every actor containing such a new primitive, there exists an equivalent term which does not contain any *receive* commands.

Our idea for implementing the program $\mathit{receive}(e, x).P$ by a term of the algebra denoted by $[\![\mathit{receive}(e, x).P]\!]$ is to define a behaviour which executes the program P only if the message has the desired *id*; otherwise it resends the received message and becomes idle waiting for another one:

$$[\![\mathit{receive}(e, x).P]\!] \stackrel{def}{=} \mathit{become}(\mathit{REC}, \mathit{state})$$

where:

$$\begin{aligned} \mathit{REC} \ \stackrel{def}{=} \ & (\mathit{message.id} = e)\colon P\{\mathit{message}/x\} \ + \\ & (\mathit{message.id} \neq e)\colon \mathit{send}(\mathit{self}, \mathit{message}). \\ & \qquad\qquad\qquad\qquad \mathit{become}(\mathit{REC}, \mathit{state}) \end{aligned}$$

The correctness of our mapping is proven by the fact that the equivalence $^{a}[\mathit{receive}(e, x).P]_s \approx_a {}^{a}[[\![\mathit{receive}(e, x).P]\!]]_s$ holds for every a and s. On the other hand, the standard (synchronous) bisimulation is not preserved. This is because the implementation uses the technique of immediately reintroducing the received messages (when the sender is different from e) that, as stated above, is observed by the standard bisimulation and not by the asynchronous one.

One feature of this implementation is that the encoding of a *receive* command could introduce a busy waiting; for example, if only messages without the desired *id* are received by the actor, the messages are repeatedly received and resent. Even if the encoding could introduce this divergent behaviour, asynchronous bisimulation is preserved because it is not divergence sensitive.

4 A Specification of the CORBA Messaging Service

The actor model and the algebra presented in the previous sections provide an abstract representation of the CORE OMG object model in an untyped scenario. The further extension to our algebra is to add the distinction between clients and servers objects and an adequate abstract syntax for the CORBA models of requests invocation.

4.1 CORBA Clients

CORBA clients are represented as actors performing sequences of requests. To this aim we extend the syntax for the behaviour of our actor algebra by introducing also:

$$\begin{aligned} P ::= \; & \mathit{INVOKE}(a, e, x).P \mid \mathit{ONEWAY}(a, e).P \mid \mathit{SEND}(a, e).P \mid \\ & \mathit{GET_RESPONSE}(a, e, x).P \mid \mathit{SENDC}(a, e, P').P \mid \\ & \mathit{SENDP}(a, e, b).P \mid \mathit{IS_READY}(b, e, x).P \mid \\ & \mathit{POLL_GET_RESPONSE}(b, e, x).P \end{aligned}$$

Several kinds of parameters are considered: a indicates the name of the actor representing the server the client is asking for the service, e is a record that describes the request, x is the value variable that will contain the results returned by the server, P' is the program that the callback object will perform after having received the result of the request in the case of *SENDC*, and b is the name of the poller object in the case of *SENDP*.

According to this extended syntax an actor can perform the CORBA primitives specified above and the primitives of the actor algebra. We assume that a CORBA client can only perform CORBA primitives.

4.2 CORBA Servers

CORBA servers are represented as actors that reply to the client requests according to the specification of the provided services. As already stated, requests are described as records; we suppose that these records contain at least three fields: *id* that contains an identifier that uniquely characterizes the request, *to* that indicates the name of the object to which the server has to send the results of the request, and *client* which is filled with the name of the client asking for the service. It is interesting to note that the fileds *to* and *client* are not in general the same. For example, in the case of the asynchronous invocations *SENDC* and *SENDP* the client will ask to the server to send the results to the callback or to the poller object respectively and not to itself.

We assume the existence of a function $f(a, e)$ that, given a server a and a request e, returns the results that the server will compute. We suppose that $f(a, e)$ is a record that contains the field *id* filled with the identifier of the current request.

Usually servers are idle waiting for requests: whenever a request is delivered to a server, it deterministically computes the answer, sends the result to the correct destination and possibly modifies its state.

Idle servers are denoted by ${}^{a}S_{s}$ where a is the name of the server, S its behaviour, and s its state. The behaviour of a server can be defined formally as follows:

$$\langle a, e\rangle \mid {}^{a}S_{s} \stackrel{\tau}{\Longrightarrow} \langle b, v\rangle \mid A \quad \text{iff } b = e.to,\ v = f(a, e) \text{ and } A = {}^{a}S_{s'}$$

where s' is the new state of the server which is a function of the previous state s and of the request e.

In this way we provide a specification of CORBA servers which is independent from the model of request invocation used by clients. We also abstract from the server interface, and we assume that a server is able to execute a set of operations and that the name of the method and its arguments are transmitted in the request.

4.3 CORBA Models of Request Invocation

We present a characterization of the possible request invocations, giving the indication of how a client behaves in each possible case.

- **INVOKE**. The program $\mathit{INVOKE}(a, e, x).P$ implements the synchronous model of request invocation of CORBA, basically it is a remote procedure call where a is the name of the server object, e is the request containing the operation and the arguments and x is the result that can be used in the rest of the program P.
  We implement this primitive by a term $[\![\mathit{INVOKE}(a, e, x).P]\!]$ which is defined by exploiting the *send* and the *receive* operations of the actor algebra. When the answer (which is a record with the corresponding field *id*) is received, the program P can be executed.

  $$[\![\mathit{INVOKE}(a, e, x).P]\!] \stackrel{def}{=} \mathit{send}(a, e).\mathit{receive}(e.id, x).P$$

  Here we suppose that $e.to = \mathit{self}$ (i.e. the name of the client actor) in order to be sure that the results are sent back to the correct client.

- **ONEWAY.** According to this model of request invocation the answer of the server will never be consumed by the client. We model this by creating a new empty actor to which the server will send the results of the request.

  $$[\![\mathit{ONEWAY}(a, e).P]\!] \stackrel{def}{=} \mathit{create}(b, \surd).\mathit{send}(a, e).P$$

  Here we suppose that $e.to = b$ in order to ensure that the server will send to the new empty actor its results. In this way the produced results will be consumed by the new object that performs a sort of garbage collection.

- **SEND.** This is the deferred synchronous model of request invocation. The request $\mathit{SEND}(a, e)$ is transmitted to the server object through the ORB and then the client retrieves the answer exploiting $\mathit{GET_RESPONSE}(a, e, x)$ which binds the variable x to the result of the request.

  $$[\![\mathit{SEND}(a, e).P]\!] \stackrel{def}{=} \mathit{send}(a, e).P$$

Here we suppose that $e.to = \mathit{self}$. The answer will be consumed at the moment the client performs a *GET_RESPONSE*; in order to ask for the result of the correct request e, we use its unique identifier $e.id$ as handler:

$$[\![\mathit{GET_RESPONSE}(a, e, x).P]\!] \stackrel{def}{=} \mathit{receive}(e.id, x).P$$

- **SENDC.** This is an asynchronous model of invocation: $\mathit{SENDC}(a, e, P')$ creates a callback object that will manage the answer executing the program P' after having received the result of the request, and sends the request and the address of the callback object to the server object through the ORB. Finally it executes the rest of the program.

$$[\![\mathit{SENDC}(a, e, P').P]\!] \stackrel{def}{=} \begin{array}{l} \mathit{create}(d, \mathit{CALLBACK}_{eP'}). \\ \mathit{send}(a, e).P \end{array}$$

Here we suppose that $e.to = d$ and that the behaviour $\mathit{CALLBACK}_{eP'}$ is defined as follows:

$$\mathit{CALLBACK}_{eP'} \stackrel{def}{=} \begin{array}{l} (\mathit{message}.id = e.id)\colon P'\{\mathit{message}/\mathit{answer}\} + \\ (\mathit{message}.id \neq e.id)\colon \mathit{send}(\mathit{self}, \mathit{message}). \\ \qquad \mathit{become}(\mathit{CALLBACK}_{eP'}, \mathit{state}) \end{array}$$

where *answer* is a special expression used inside the program P in order to denote the content of the message that the callback object will receive from the server. Note that in this case the *to* field of the request is set to the address of the callback object d.

- **SENDP.** The polling model of request invocation creates a poller object that will be used by the client to retrieve the answers.

$$[\![\mathit{SENDP}(a, e, b).P]\!] \stackrel{def}{=} \mathit{create}(b, \mathit{POLLER}_e, (\mathit{empty}, \mathit{empty})).\mathit{send}(a, e).P$$

Here we suppose $e.to = b$ as in this case the server will have to send the response to the poller actor. The state of the poller actor is a pair, initialized to $(\mathit{empty}, \mathit{empty})$, that contains two kinds of information. The first is the name of the client actor that is introduced in the state of the poller whenever the client asks for the results that are not yet received by the poller object. The second element is used to store the results produced by the servers.
The client has the possibility to interact with the poller object by means of two primitives that are compiled as follows:

$$\begin{array}{lcl} [\![\mathit{IS_READY}(b, e, x).P]\!] & \stackrel{def}{=} & \mathit{send}(b, \mathit{is_ready}). \\ & & \mathit{receive}(e.id, x).P \\ [\![\mathit{POLL_GET_RESPONSE}(b, e, x).P]\!] & \stackrel{def}{=} & \mathit{send}(b, \mathit{poll_get_response}). \\ & & \mathit{receive}(e.id, x).P \end{array}$$

Finally, we can describe the behaviour of the poller as follows:

$$
\begin{aligned}
&POLLER_e \stackrel{def}{=} \\
&\qquad (message.id = e.id) and (1st(state) = empty): \\
&\qquad\quad become(POLLER_e, (empty, message)) + \\
&\qquad (message.id = e.id) and (1st(state) \neq empty): \\
&\qquad\quad send(1st(state), message) + \\
&\qquad (message = is_ready) and (2nd(state) = empty): \\
&\qquad\quad send(e.client, false).become(POLLER_e, state) + \\
&\qquad (message = is_ready) and (2nd(state) \neq empty): \\
&\qquad\quad send(e.client, true).become(POLLER_e, state) + \\
&\qquad (message = poll_get_response) and (2nd(state) = empty): \\
&\qquad\quad become(POLLER_e, (e.client, empty)) + \\
&\qquad (message = poll_get_response) and (2nd(state) \neq empty): \\
&\qquad\quad send(e.client, 2nd(state))
\end{aligned}
$$

This is a static specification of the poller object. A poller object always sends answers to the original client, *e.g.,* the client which sent the *SENDP* request. This feature can be used to prove properties of this model of request invocation, for instance the equivalence presented in the next section. A more dynamic specification of the poller object is also possible and it will be the subject of future work.

4.4 Some Properties

After having mapped the different kinds of request invocation to our algebra of actors, we can use the equivalences we have introduced in the previous section in order to prove formally some interesting, even if simple, properties.

First of all we consider the standard synchronous *INVOKE* and the deferred synchronous *SEND*. The difference between these two requests is that *INVOKE* blocks a client until a result is received, while *SEND* does not block a client and the result will be received at the moment the corresponding *GET_RESPONSE* operation is executed. The blocking behaviour of the *INVOKE* can be simply simulated also in the case of the deferred synchronous invocation by performing the *GET_RESPONSE* primitive after the *SEND* primitive. This is formalized by the following equivalence:

$$
\begin{gathered}
{}^{b}[INVOKE(a, e, x).P]_s \\
\approx_a \\
{}^{b}[SEND(a, e).GET_RESPONSE(a, e, x).P]_s
\end{gathered}
$$

Also a stronger equivalence result holds: the two terms obtained by substituting each CORBA primitive according to the corresponding definition are syntactically equals.

More interesting is to show that the synchronous *INVOKE* mechanism can be simulated also by using the asynchronous primitives *SENDC* and *SENDP*.

In the case of *SENDC*, the idea is that the client blocks after having asked for the service, until the callback object forwards the results. To this aim we force

the callback object to perform the operation $send(e.client, x)$ where $e.client$ is the name of the client and x is the variable that will be bound to the results produced by the server. This is formalized by the following equivalence:

$$
\begin{gathered}
{}^{b}[INVOKE(a, e, x).P]_{s} \mid {}^{a}S_{s'} \\
\approx_{a} \\
{}^{b}[SENDC(a, e, send(e.client, answer)).receive(e.id, x).P]_{s} \mid {}^{a}S_{s'}
\end{gathered}
$$

In this case we have to take into account configurations composed of both the client and the server. In this way, the communication between them is not observable; indeed, if an external actor can observe the contents of the exchanged messages, it is easy to discriminate between the *INVOKE* and the *SENDC* mechanisms as the contents e of the two kinds of request are different (e.g. the contents of the field $e.to$).

The above equivalence can be proven by recalling the equivalence in the Example 1, and observing that the *CALLBACK* object, which is dynamically created, behaves like the *Forward* actor defined in that example. Moreover, the *CALLBACK* object is not visible to the outside because the name of new created actors is local to the actor itself and the parent actor. A new feature with respect to the Example 1 is that in this case the restriction is on the name of the object performing the forward operation and not on the client name. Note also that the above equivalence does not hold exploiting the CORBA primitives only. We have to resort to the low level message passing operations of the actor algebra when we specify the behaviour of the callback object.

A similar idea can be used in the case of the *SENDP* primitive where the command *POLL_GET_RESPONSE* can be used in order to retrieve the results via the poller object. This is formalized by the following equivalence:

$$
\begin{gathered}
{}^{b}[INVOKE(a, e, x).P]_{s} \mid {}^{a}S_{s'} \\
\approx_{a} \\
{}^{b}[SENDP(a, e, b).POLL_GET_REQUEST(b, e, x).P]_{s} \mid {}^{a}S_{s'}
\end{gathered}
$$

In this case, the forward operation is performed by the *POLLER* object, which will receive a new restricted name at the moment the *SENDP* operation is executed.

5 A CORBA Programming Example

We illustrate our approach showing how a specification of a client behaviour can be extracted from a simple CORBA program. We assume to have defined a server which is able to compute the result of mathematical formulas. We also assume a display object which is able to show the answer to the user. We assume the following OMG IDL interfaces:

```
interface Compute {
  string compute (in string formula);
};
```

```
interface Display {
  void display (in string message);
};
```

Here we suppose that formulas and results are represented as strings. We define an user interface client which takes a formula from the user, asks the server to compute it, waits for the answer and finally sends the answer to the display object.

We illustrate our example exploiting the CORBA Scripting Language (CORBAScript) [8], an interpreted scripting language dedicated to CORBA environments. CORBAScript provides an high level interface to implement CORBA clients and tools, hiding some of the specific mechanisms of CORBA. For example in CORBAScript the access to an IDL interface is simply done by providing its IDL interface identifier as illustrated below.

```
server = Compute("IOR:....");
display = Display("IOR:.....");
answer = server.compute("Formula");
display.display(answer);
```

Here we suppose that *Formula* is the string representing the formula to compute. We use an IOR (Interoperability Object Reference) to access the CORBA objects. The IOR is a unique reference to a CORBA object and thus can be modelled as an actor name. The variables *server* and *display* are references to CORBA objects, and the operations *display.display* and *server.compute* are synchronous requests invocations (*i.e., INVOKE*).

This situation is modelled by the following client program in the algebra of actors where we assume that *server* and *display* are the names of actors implementing the two servers.

$$
\begin{aligned}
UICLIENT_1 \stackrel{def}{=}\ & message = compute: && INVOKE(server, Formula, x).\\
& && INVOKE(display, x, y).\\
& && become(UICLIENT_1, state) +\\
& message = other: && P
\end{aligned}
$$

A drawback of this kind of interaction between the client and the mathematical server, is that the client is blocked on the call *INVOKE(server,Formula,x)* waiting for the results. In this way, the user interface program is not able to accept the *other* input which activates the generic program P.

This problem can be solved by using the new asynchronous request invocation mechanisms. We represent this alternative approach in the algebra of actors only, because CORBAScript does not yet support the asynchronous messaging facilities.

$$
\begin{aligned}
& UICLIENT_2 \stackrel{def}{=}\\
& message = compute: && SENDC(server, Formula, INVOKE(display, answer, y)).\\
& && become(UICLIENT_2, state) +\\
& message = other: && P
\end{aligned}
$$

Here the client performs an asynchronous request and thus it does not block and it is able to deal with the *other* inputs activating the generic program P. The display request is performed by the callback object created at the time the *SENDC* primitive is executed.

Now, it is interesting to compare formally the two above client descriptions. The result we have obtained is that the two different approaches are equivalent:

$$
\begin{array}{c}
{}^{b}[\mathit{UICLIENT}_1]_s \mid {}^{\mathit{server}}S_{s'} \mid {}^{\mathit{display}}D_{s''} \\
\approx_a \\
{}^{b}[\mathit{UICLIENT}_2]_s \mid {}^{\mathit{server}}S_{s'} \mid {}^{\mathit{display}}D_{s''}
\end{array}
$$

where the actors ${}^{\mathit{server}}S_{s'}$ and ${}^{\mathit{display}}D_{s''}$, representing the mathematical and the displaying servers respectively, are defined according to the indication reported in Section 4.2.

This equivalence can be proven following the same lines described in Section 4.4. The most interesting aspect to observe is that the second client is able to start the execution of P even if the request has not been considered by the mathematical server yet. The execution of P is started whenever an *other* input is received from the client. In order to simulate the same behaviour, also the first client has to perform the same input action. This can happen only if the two server invocations have been completely executed. Even if this behaviour is different, it does not permit to distinguish between the two above agents; indeed, the two reached actors are equivalent:

$$
\begin{array}{c}
{}^{b}[P]_s \mid {}^{\mathit{server}}S_{s'} \mid {}^{\mathit{display}}D_{s''} \\
\approx_a \\
({}^{b}[P]_s \mid {}^{c}\mathit{CALLBACK}_{s'''} \mid \langle \mathit{server}, \mathit{request} \rangle) \setminus c \mid {}^{\mathit{server}}S_{s'} \mid {}^{\mathit{display}}D_{s''}
\end{array}
$$

This equivalence holds because the *CALLBACK* object, the message containing the *request* for the server, as also the messages that will be exchanged among the servers and the *CALLBACK* object, are not observable to the outside.

It is interesting to observe that in this case we have not followed the previous general way to implement the *INVOKE* primitive using the callback mechanism. Indeed, here the callback object is not a forwarder, but it is responsible for the execution of the displaying request. The approach used in this case is more efficient, because the client does not block waiting for the answer of the server forwarded by the callback object, but it directly delegates the execution of the continuation to the callback.

6 Related Work

Najm and Stefani [26] present a notion of computational model for open distributed processing which can also be exploited for modelling the new features of CORBA for instance the quality of service [14]. Since their model is based on

rewriting logic and they do not deal with the issue of modelling asynchronous requests, their work can be considered complementary to our work. An interesting point could be to analyse if their proposal for modelling quality of service can be successfully used in an asynchronous context.

Bastide et al. [6] propose to extend the interface definition of CORBA distributed objects by a specification of their behaviour expressed exploiting high level petri nets. Although our goal was different, *e.g.,* to provide a formal representation of CORBA clients which interact exploiting the new CORBA asynchronous messaging service, the algebra of actors could also be used to provide a formal account of CORBA services.

In the past few years, several advances have been achieved on the semantics of actors, dealing with aspects of communication and concurrency [4,3,30,31,21], but these papers do not investigate the relationships of the actor model with traditional process algebras, even though recently Robin Milner [24] suggested that it may be worthwhile to work in this direction. We believe that our approach is complementary to previous approaches to the semantics of actors, providing a new framework to discuss concurrency related aspects in this context. The reader interested in a more detailed comparison with other calculi can refer to [11,10,9].

7 Conclusion

The main result presented in this paper is the development of a process algebraic specification of the new CORBA models for request invocation.

The specification is based on an algebra of actors that enjoys a clean formal definition and a rich algebraic theory, inspired from the π-calculus, while preserving a basic object-oriented features such as object identity, asynchronous message passing, an implicit receive mechanism and support for dynamic object creation. This approach allows us to reuse standard results of the theory of concurrency for reasoning on CORBA models of request invocation.

The algebra of actors is well suited to specify most of the aspects of CORBA which concern interaction of objects, their creation and their deletion. Thus CORBA services which define these features [15], such as the Life Cycle Service, the Concurrency Service, the Transaction Service or the Naming Service are good candidates for being modelled with our algebra and they will be target of our future work. Moreover other CORBA services which describe internal aspects of the involved objects can also be modelled defining their interfaces and their functionalities in an abstract way. For example we have followed this approach in the formal definition of a CORBA server in Section 4.2.

CORBA Exceptions can be also modelled as messages of the process algebra provided that they are included in the server specification. Namely, the specification of the CORBA server that we have presented in Section 4.2 must be extended considering that one of the possible outcomes of a request invocation can be an exception.

On the other hand, aspects such as types [12], quality of service and objects by value [32] have not an immediate mapping in our algebra. Thus a formal specification of all the CORBA features is not feasible with the current version of our algebra, and a significant extension is required to achieve this ambitious goal. A less ambitious goal will be to provide a specification of the CORBAScript [8] language which is more abstract and hides most of the low-level mechanisms of CORBA. Neverthenless, we still claim that a more formal description of interaction aspects of the CORBA services will be useful both to CORBA implementors and programmers.

Finally, a number of additional research items still need to be carried out in this research. For instance: a study of how typing and inheritance issues, such as in [1] can be integrated in our framework to have a more real model of CORBA; the formulation of algebraic laws that characterize the equivalences of CORBA programs, for example an axiomatization for the asynchronous bisimulation and its application to CORBA; and the definition of a framework for formal reasoning about CORBA programs, *e.g.,* following the style of the Hennessy and Milner logic [22].

Acknowledgements

We would like to thank the anonymous referees for their detailed reports, and we acknowledge also interesting discussions with Carolyn Talcott as also her comments on a previous version of this paper.

References

1. M. Abadi and L. Cardelli. An Imperative Object Calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1995.
2. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.
3. G. Agha, I. Mason, S. F. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(1):1–69, January 1997.
4. G. Agha, I.A. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computation. In *Proc. of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 564–579. Springer Verlag, 1992.
5. R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.
6. R. Bastide, O. Sy, and P. Palanque. Formal Specification and Prototyping of CORBA Systems. In *Proc. of ECOOP99*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1999. This volume.
7. G. Boudol. Asynchrony and the $\pi-calculus$. Technical Report INRIA-92-1702, INRIA Sophia-Antipolis., 1992.
8. Laboratoire d'Informatique Fondamentale de Lille and Object Oriented Concepts Inc. CORBA Scripting Language. Technical Report orbos/98-12-09, Object Management Group, 1998.
9. M. Gaspari. Concurrency and knowledge-level communication in agent languages. *Artificial Intelligence*, 105(1-2):1–45, 1998.

10. M. Gaspari and G. Zavattaro. An actor algebra for specifying distributed systems: the hurried philosophers case study. In G. Agha and F. Decindio, editors, *Concurrent Object-Oriented Programming and Petri Nets*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1998. To appear.
11. Mauro Gaspari and Gianluigi Zavattaro. An algebra of actors. In *Proc. 3nd IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 3–18. Kluwer Academic Publishers, Feb 1999.
12. Object Management Group. OMG Architecture Guide: The OMG Object Model. Technical report, Object Management Group, 1992.
13. Object Management Group. Messaging Service RFP. Technical Report orbos/96-03-16, Object Management Group, Framingham, MA, 1996.
14. Object Management Group. CORBA Messaging Joint Revised Submission. Technical Report orbos/98-05-05, Object Management Group, Framingham, MA, 1998.
15. Object Management Group. CORBA Services. Technical Report formal/98-12-09, Object Management Group, 1998.
16. Object Management Group. *The Common Object Request Brocker: Architecture and Specification*. Object Management Group, 1998.
17. C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.
18. CAR. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
19. K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *The Fifth European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 141–162. Springer-Verlag, Berlin, 1991.
20. S.N. Khoshafian and G.P. Copeland. Object Identity. In *Proc. OOPSLA '86*, pages 406–416, September 1986.
21. I.A. Mason and C. Talcott. A Semantically Sound Actor Translation. In *Proc. of ICALP'97*, Lecture Notes in Computer Science. Springer Verlag, 1997.
22. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
23. R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
24. R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):79–89, January 1993.
25. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I and II. *Information and Computation*, 100(1):1–40 – 41–77, 1992.
26. E. Najm and JB. Stefani. Computational Models for Open Distributed Systems. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 157–176, Canterbury, UK, 1997. Chapmann & Hall.
27. C. Palamidessi. Comparing the expressive power of the Synchronous and the Asynchronous pi-calculus. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–265, 1997.
28. B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In *invited lecture at Theory and Practice of Parallel Programming (TPPP)*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215, Sendai, Japan, nov 1994. Springer-Verlag, Berlin.
29. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981.
30. C. Talcott. Interaction Semantics for Components of Distributed Systems. In *Proc. FMOODS'96*, pages 154–169. Chapmann & Hall, 1996.

31. C. L. Talcott. An actor rewriting theory. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, number 4 in Electronic Notes in Theoretical Computer Science, pages 360–383. North Holland, 1996.
32. S. Vinoski. New Features for CORBA 3.0. *Communications of the ACM*, 41(10), October 1998.
33. D. Walker. Objects in the π*–calculus*. *Information and Computation*, 116(2):253–271, 1995.

Object-Oriented Programming: Regaining the Excitement

Andrew P. Black

Oregon Graduate Institute of Science & Technology
Portland, Oregon, USA
black@cse.ogi.edu

Abstract. This paper is based on a speech delivered at the ECOOP'98 Conference Banquet. It is not a literal transcription of my talk, since no recording was made, but has been reconstructed *ex post facto* based upon my speaker's notes and my memory. I have also taken the opportunity to add some headings and references.

Distinguished Chairmen, Members of the Conference Committee, Representatives of the sponsoring organizations, distinguished Professors, conference participants, and friends: Good evening.

It's customary to start this kind of talk with a joke, at least in part to give the audience a chance to become accustomed to my strange accent before I start to say anything interesting, but I'm going to skip that tonight because I think that we have already heard quite enough jokes for one evening.

To set the record straight from the very first, I should make it clear that I do not work for IBM, nor have I worked for IBM in the past. Although I did once spend a very enjoyable year at IBM's Yorktown Heights Research Laboratory, IBM was very clear that I didn't work for them, even though I did turn up every day and they did pay me: I believe that the distinction had something to do with social security tax or health insurance.

I should also point out, particularly for the benefit of those of you around the corner who cannot see me, that yes, I am wearing a tie, but no, I don't use a mainframe. I used to use a mainframe, but one day the mainframe broke and all of the little beads came off. [At this point the reader will have to imagine a large broken abacus frame with bent wires and missing beads.]

1 Inventing the Future of Object Technology

The right thing to do in a talk of this nature is to predict the future of object technology, but prediction is hard, and predicting the future is especially hard! Alan Kay once said: "The best way to predict the future is to invent it". And he was remarkably accurate. Let me read you a quote from over 20 years ago.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 519–528, 1999.
© Springer-Verlag Berlin Heidelberg 1999

> In the 1990's there will be millions of personal computers. They will be the size of notebooks of today, have high resolution flat screen reflective displays, weigh less than 10 pounds, have 10 to 20 times the computing and storage capacity of an Alto. Let's call them Dynabooks.
>
> The purchase price will be about that of a color television set of the era...
>
> Though the Dynabook will have considerable local storage and do most computing locally, it will spend a large percentage of its time hooked to various large, global information utilities which will permit communication with others of ideas, data, working models as well as the daily chit-chat that organizations need in order to function.

This is from Alan's paper "The Early History of Smalltalk"[5], in which Alan quotes an internal Xerox PARC memo from around 1976. I recommend that paper most highly: if you find any wisdom in my remarks this evening, please attribute it to Alan Kay and not to me.

My goal tonight is to challenge you to put the excitement back into object-orientation, and to recapture some of the dynamism of those early Smalltalk days.

Is this possible? Or has object-orientation become like structured programming: the right idea, but no longer the focus of innovation, exactly because everyone is already doing it. For example, we don't have a European Conference on Structured Programming every year.

I don't think that object-orientation is yet at that point. There are still many hard problems to solve: scale and encapsulation are two that I will mention briefly tonight.

2 Programming Language Contributions Relevant to Object-Orientation

The programming language research community has historically been a great source of innovation. I believe that we should challenge ourselves to recapture that atmosphere of innovation; to help us on our way, I will take a brief look at some of the ideas that have been "brought to market" by influential programming languages over the last forty years. I'm not claiming that the listed languages invented the concepts in every case, but rather that these languages were among the first to popularize them.

2.1 Lisp (late 1950s)

Lisp was a startling language for its time: I believe that Lisp is one of Alan Kay's "almost new things". It contributed:

- heap allocation and garbage collection,
- interpreted execution, and
- conditional expressions.

2.2 Algol 60

Another "almost new thing", in contrast to the "better old things" of COBOL and FORTRAN. Its contributions include:

- a semi-formal, concise definition,
- grammars as a descriptive technique,
- block structure,
- recursion—but not the word "recursion";
  - recursion introduced the distinction between program text (or "static instance") and its execution (or "dynamic instance"). This distinction is the kernel of the idea that became Simula Objects [7]. Algol 60 also introduced
- declarations,
- call-by-name,
- "own" variables,
- the extension of the concept of expression to non-numeric values,
  - Boolean expressions, and
  - "designational expressions", and
- the idea of "Security", which we would probably today call semantic integrity.

Security is the principle that a program must either be rejected as incorrect by compile-time or run-time checks, or *its behavior must be understandable by reasoning based entirely on the language semantics*, independent of the implementation [2,7].

However, no language is perfect. Two "non-contributions" of Algol 60 were:

- that the non-numeric expressions that Algol 60 chose to include were labels, not procedures; and that
- input/output was relegated to a library.

2.3 Algol W, Pascal and Simula 67

Contributions:

- quasi-parallel execution (previously introduced in Knuth's SOL);
- user-definable types;
- static inheritance (Simula's prefix classes);
- use of prefix classes to provide "language dialects"
- Sum types, also known as variant types, or discriminated unions;
- the combination of data and operations in a single language construct; and
- the idea that binding of a value to a name requires only type *compatibility*, not type identity.

Non-contributions:

- Explicit reference variables, and
- Pascal's sacrifice of the security of Algol 60 by the variant record construct.

This last innovation is perhaps one of the reasons that Tony Hoare said of Algol 60: "here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors"[2].

2.4 Algol 68

Contributions:

- unification of statement and expression, that is, making the distinction between them one of type, rather than one of context-free syntax;
- the void type;
- coercion;
- environment enquiries;
- conceptual minimality, for example,
  - the use of references to eliminate the notion of variable, and
  - the use of procedures to eliminate call-by-name; and
- obtaining an extensible syntax—then an important goal—by means of the definition of new operators, and by that means alone.

Non-contributions:

- 2-level grammars as a descriptive technique, and
- once again, breaching security, this time by requiring that the programmer not export a stack-allocated variable outside of its scope, which is not something for which implementations can easily check.

2.5 CLU, Alphard, Modula (1974-78)

Contributions:

- encapsulation,
- named scopes (*i.e.*, the module concept),
- parametric polymorphism, and
- F-bounded polymorphism (CLU's where clauses)
  - but it's not clear that the CLU designers realized at the time what they had invented.

2.6 Emerald (1984-6)

Contributions:

- safe, static subtyping,
- F-bounded polymorphism (in the interpretation of conformity in where clauses)[4]
  - but we also did not yet know that it would be given this name,
- mobile objects,
- location-independent invocation,
- object constructors (which I believe are relevant to Ole Lehrmann Madsen's quest to integrate prototype and class-based programming styles),
- separation of type from implementation, and
- the "open world" assumption

This last is the idea that new objects and new classes (superclasses as well as subclasses) can be added to a system at any time, so that we don't have to shut down and recompile the internet.

2.7 Later Developments

What has happened since then?

- multi-paradigm languages (LIFE, Leda, *etc.*)
- higher-order functional languages with object-oriented concepts (Objective CAML, Haskel with its type classes, *etc.*)

These are interesting, but not earth-shaking.

And then, of course, we have Java. What are Java's advances over Emerald? The major one is of course replacing word pair brackets (like **do** ... **end**) with curly braces {}, but we should not overlook typed byte-code and byte-code verification.

The latter means that one can obtain a class that someone else has compiled and be sure that it is type-safe. This is a real advance, although proof-carrying code [6] may well be a more elegant and more widely applicable way of achieving the same effect.

3 A Research Agenda

To summarize this historical review, I believe that Programming Language Research has made *major* contributions to the state of computing today, but that recently the focus of research has moved away from pure OO towards hybrid languages and applications. These are interesting topics, but I do not believe that we have completed the development of object-orientation itself.

3.1 Objects Demonstrate Recursive Structure

My prescription is that we should re-examine the core of object-orientation. Going back to Smalltalk, we find that there are two key ideas:

- that objects localize *data structures* and the *code* that operates on them in the same place; and
- that objects, using the words of Alan Kay again: are *a recursion on the idea of computer itself.*

Kay writes: "The basic principle of recursive design is to make the parts have the same power as the whole". Rather than dividing the computer into "lesser stuffs", like data structures and procedures, we should divide it into lots of little computers that communicate together. This enables us to postpone representation decisions almost indefinitely (and those are the decisions that are almost always the cause of our troubles).

This sounds like a model for distributed computing to me.

The World Wide Web is the largest and most successful distributed object-oriented system ever built—but what a poor system it is! For example, where is location independent naming? And how does one introduce a new class? The

answer to the latter question is that we persuade the W3 consortium to call an international meeting to get agreement on a new protocol!

I propose that we take a long hard look at Kay's three basic laws of object-oriented programming.

1. Everything is an object.
2. Objects communicate by sending and receiving messages (in term of objects).
3. Objects have their own memory (in terms of objects).

3.2 Information Hiding

Notice that there is nothing in the concept of little computers communicating with each other and protecting their own data that speaks of *how* to make information hiding decisions.

Recall that the first key idea behind objects is "localization", not "encapsulation". Parnas' idea of *information hiding* is a strong and powerful idea in program structuring: implementations of abstractions should hide as much as possible about themselves (but no more)!

Hence, there are two parts to each information hiding decision: we must specify not only *what* to hide, but also from *whom.* The weakness of most encapsulation mechanisms is that they typically give the programmer fairly good control on the *what*, but very little on the *whom.*

Java gives us three sets for whom: methods in this object, methods in objects that subclass from this object, and the rest of the world. But this is inadequate. I will give just two examples.

– Think about allowing accessor and updater methods access to a slot, but prohibiting that access to other methods in the same object.
– Think about a persistent system: how can programmers evolve the contents of an object in the store if they can't access its fields?

Don't be misled into thinking that this last class of access somehow takes place "outside" of the system; our goal should be to describe the whole world as objects, and to use the reflective properties of those objects to examine and change their structure and their protocol. To put it another way: I want objects to take over the world, so I'm not prepared to start by excluding our own programming environments from the set of things that I can describe with objects. To the contrary: we should be using the reflective properties of objects to *build* our environments.

Rather than the "all or nothing" encapsulation paradigm, what we need is a concept of encapsulation that is much closer to what operating system workers have thought of as "protection mechanisms". For example, one programmer might take on several different roles in relation to the *same* object: she may be a maintenance programmer, then a user, and maybe later an administrator, and under those roles, the objects "in the programmer's hand" should have different access rights to various parts of the target object.

3.3 Objects as an Integrative Paradigm

The third research direction that I would like to propose is that of objects as an integrative paradigm. The OO model is strong enough to capture functional and procedural styles of programming, but doing this naively is likely to result in "write only programs".

Let me explain. Many "idioms" can be written and not read. For example, I have used the functional style to build a Smalltalk Interval class in which the step and the limit slots hold blocks that are the step function and the limit predicate. Smalltalk programmers will be taken by surprise by such code.

The reverse is also true: I can build objects in ML, but ML programmers will not be able to read the resulting code.

Consequently, I do believe that multi-paradigm languages have a place, but I also believe that the non-OO features should be mapped onto objects. For example, functions should be objects with the convention that they have exactly one operation called $\circ$ (meaning "apply").

Nevertheless, it is important that the new features should be given an appropriate and readable syntax. The approach that I am advocating should enable us to retain semantic simplicity and avoid unpleasant interactions between features.

During Tuesday's Panel session, one of the questioners stated that his "old professor" (I do hate that phrase, now that I am one) said that:

> Programming language people think that they can solve a problem by designing a language to express it.

I believe that this professor was right, and that such "language people" are also right! The reason is that *language shapes thought:* first we shape the tool, and then the tool shapes us.

So it is vital that the tool be well-made. Category theory in mathematics is a good example: by exposing similarities between different branches of mathematics, some proofs that were obscure and difficult become self-evident. Lazy functional languages bring some of that clarity to programming certain kinds of problems.

Does clarity come from having to force your problem into the mold of the language? Only if it works! Sometimes, the struggle to force your problem into a notation that does not seem appropriate can yield great insight; sometimes it is a waste of time.

3.4 Scale

The final challenge is scaling, but we must ask: in what dimension? The answer is:

- in space, that is, wide area distribution;
- in time, that is, Persistent Object Systems;
- and, of course, in their combination.

Scaling in Space. Systems like Emerald that successfully handle distribution in the small do so by seeking to hide the distinction between local and remote objects. But the failure modes of local and remote objects are inherently different, and performance is radically reduced by distribution. This means that object placement is absolutely critical to application performance—and the tools that are currently available to automate or even understand placement are minimal.

Another problem is how to replicate an object and still preserve its “object nature”.

The systems community has spent 20 years working on replication, and we now know how to implement reliable and resilient systems from replicated communicating objects. But once we have done so, how can we treat the resulting subsystem as a single object, in order to hide its internal structure from its clients?

A student (Mark Immel) and I presented one solution to this problem at ECOOP’93 [1], but I can’t believe that it is the only solution, or even that it is necessarily a very good one. I encourage you to work on this problem.

Scaling in Time. I will defer to the experts who will be giving tomorrow morning’s invited lecture on Persistence in Java—Malcolm Atkinson and Mick Jordan—by mentioning only briefly some of the issues that must be resolved:

- compilation distributed in time;
- explicit support for versioning;
- safe and unsafe evolution of types;
- new interfaces for old objects; and
- new objects for old interfaces.

4 Summary

The Programming Language Research community has contributed massively to our progress in object-orientation to date. But recently, the pace of innovation seems to have slowed: we are making better old things, but I don’t see the almost new things.

My message is the following.

- Go back to the roots of object-orientation:
  - build little computational engines that communicate with one another to mimic the real world that we seek to model.
- Use objects as an integrative paradigm:
  - build a semantic model that will serve for concurrent, functional, object-oriented and, if possible, constraint-based and logic programming; but
  - remember that the purpose of a program is to be read; and
  - that idioms reduce readability whereas well-designed language constructs enhance it.
- Deal with the hard problems: scale, persistence, and their cross-product.

– Take encapsulation seriously, perhaps learning from the OS community.

But I have to caution you to be careful of what you learn from the Operating Systems Community.

At today's conference, I am reminded of the 1983 ACM Symposium on Operating Systems Principles, which took place at the Mt. Washington Hotel at Bretton Woods in New Hampshire. This is the same hotel that was the site of the famous Bretton Woods monetary conference, which was held there toward the close of the Second World War. Although the plumbing hadn't been touched since, it was a wonderfully atmospheric old hotel. I should add that it has since been completely renovated.

I was young Assistant Professor, presenting my first paper at a conference, and was somewhat overawed by all of the "grand old men" of operating systems; the conference has been called the Symposium for OS Princi*pals*, and that is only partially a joke.

I remember two things about the conference. The first is that it marked one of the early replicated failures of a replicated resilient system.

The Grapevine nameservice had recently been built at Xerox PARC, and was the subject of a paper [9,8] to be presented at the conference that celebrated how, by supporting multiple servers that exchanged updates between them, the service had proved resilient to failure of one or even several of the individual servers.

Unfortunately, all of the servers ran the same code on the same hardware, so when a particular packet was sent to the first server, triggered a bug, and caused the server to crash, the client responded by re-sending the same packet to the second server, and crashing that, and so on up the replication chain. As I recall, this packet was sent just at the time that all of the principals who might have diagnosed the problem had left Palo Alto to travel to Bretton Woods.

The second thing that I recall about this conference is that it had shared "resources" in the form of bathrooms between adjacent pairs of rooms. It also had multiple processors, in the form of the occupants of the two rooms. Thus, to avoid collision, both processors had to correctly execute the bathroom door locking protocol.

At breakfast on the first morning of the conference, many of these notable OS wizards were missing. It turned out that they had deadlocked contending for the bathrooms and the hotel staff had to be called to execute a manual reset.

Thank you for listening so patiently; I hope that you enjoy the rest of your evening and the remainder of the conference.

References

1. Andrew P. Black and Mark P. Immel. Encapsulating plurality. In *Proceedings of European Conference on Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 57–79. Springer-Verlag, July 1993.
2. Charles Antony Richard Hoare. Hints on programming language design. Invited address at ACM SIGACT/SIGPLAN Symposium on Principles of Programming languages, (Reprinted in Horowitz [3], pages 35–40), October 1973.
3. Ellis Horowitz, editor. *Programming Languages: A Grand Tour*. Computer Science Press, third edition, 1987.
4. Norman Hutchinson. *Emerald: An Object-Oriented Language for Distributed Programming*. PhD thesis, University of Washington, Department of Computer Science, January 1987.
5. Alan C. Kay. The early history of Smalltalk. In *Preprints of the Second ACM SIGPLAN History of Programming Languages Conference (HOPL–II)*, pages 69–98, Cambridge, Massachusetts, March 1993. ACM SIGPLAN.
6. G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1998.
7. Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. In Richard L. Wexelblat, editor, *History of Programming Languages*, chapter IX, pages 439–493. Academic Press, 1981.
8. Michael D. Schroeder, Andrew D. Birell, and Roger M. Needham. Experience with Grapevine: The growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, February 1984.
9. Michael D. Schroeder, Andrew D.Birrell, and Roger M. Needham. Experience with Grapevine: The growth of a distributed system (Summary). In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 141–142. ACM SIGOPS, October 1983.

Author Index

Atul Adya 230
Ole Agesen 258

Rémi Bastide 474
Elisa Bertino 416
Andrew P. Black 519
Viviana Bono 43
John Boyland 205

Miguel Castro 230
Craig Chambers 279
Charles Consel 367
Krzysztof Czarnecki 18

David Detlefs 258
Sylvia Dieckman 92

Ulrich W. Eisenecker 18
Erik Ernst 67

Mauro Gaspari 495
Aaron Greenhouse 205
Giovanna Guerrini 416

Brant Hashii 449
Jifeng He 1
C.A.R Hoare 1
Urs Hölzle 92, 135
Wade Holst 304

Atsushi Igarashi 161

Günter Kniesel 351
Kresten Krab Thorup 186
Thomas Kühne 329

Julia L. Lawall 367
Raimondas Lencevicius 135
Yuri Leontiev 304
Barbara Liskov 230

Isabella Merlo 416
Marco Mesiti 416
Todd Millstein 279
Gilles Muller 367

Philippe Palanque 474
Raju Pandey 449
Candy Pang 304
Amit Patel 43
Wim De Pauw 116
Benjamin C. Pierce 161

Ulrik Pagh Schultz 367
Manuel Serrano 391
Gary Sevitski 116
Vitaly Shmatikov 43
Liuba Shrira 230
Ambuj K. Singh 135
Ousmane Sy 474
Duane Szafron 304

Mads Torgersen 186

Jim Waldo 441

Gianluigi Zavattaro 495