

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

University of Dortmund, Germany

Madhu Sudan

Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos

New York University, NY, USA

Doug Tygar

University of California, Berkeley, CA, USA

Moshe Y. Vardi

Rice University, Houston, TX, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Peter Sestoft (Ed.)

Programming Languages and Systems

15th European Symposium on Programming, ESOP 2006
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2006
Vienna, Austria, March 27-28, 2006
Proceedings



Springer

Volume Editor

Peter Sestoft
IT University of Copenhagen
Rued Langaardsvej 7, 2300 Copenhagen S, Denmark
E-mail: sestoft@itu.dk

Library of Congress Control Number: 2006922219

CR Subject Classification (1998): D.3, D.1, D.2, F.3, F.4, E.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN	0302-9743
ISBN-10	3-540-33095-X Springer Berlin Heidelberg New York
ISBN-13	978-3-540-33095-0 Springer 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. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2006
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11693024 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2006 was the ninth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (CC, ESOP, FASE, FOSSACS, TACAS), 18 satellite workshops (AC-CAT, AVIS, CMCS, COCV, DCC, EAAI, FESCA, FRCSS, GT-VMT, LDTA, MBT, QAPL, SC, SLAP, SPIN, TERMGRAPH, WITS and WRLA), two tutorials, and seven invited lectures (not including those that were specific to the satellite events). We received over 550 submissions to the five conferences this year, giving an overall acceptance rate of 23%, with acceptance rates below 30% for each conference. Congratulations to all the authors who made it to the final programme! I hope that most of the other authors still found a way of participating in this exciting event and I hope you will continue submitting.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate Program Committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for “unifying” talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2006 was organized by the Vienna University of Technology, in cooperation with:

- European Association for Theoretical Computer Science (EATCS);
- European Association for Programming Languages and Systems (EAPLS);
- European Association of Software Science and Technology (EASST);
- Institute for Computer Languages, Vienna;
- Austrian Computing Society;
- The *Bürgermeister der Bundeshauptstadt Wien*;
- Vienna Convention Bureau;
- Intel.

The organizing team comprised:

Chair:	Jens Knoop
Local Arrangements:	Anton Ertl
Publicity:	Joost-Pieter Katoen
Satellite Events:	Andreas Krall
Industrial Liaison:	Eva Kühn
Liaison with City of Vienna:	Ulrich Neumerkel
Tutorials Chair, Website:	Franz Puntigam
Website:	Fabian Schmied
Local Organization, Workshops Proceedings:	Markus Schordan

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Perdita Stevens (Edinburgh, Chair), Luca Aceto (Aalborg and Reykjavík), Rastislav Bodík (Berkeley), Maura Cerioli (Genova), Matt Dwyer (Nebraska), Hartmut Ehrig (Berlin), José Fiadeiro (Leicester), Marie-Claude Gaudel (Paris), Roberto Gorrieri (Bologna), Reiko Heckel (Leicester), Michael Huth (London), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Shriram Krishnamurthi (Brown), Kim Larsen (Aalborg), Tiziana Margaria (Göttingen), Ugo Montanari (Pisa), Rocco de Nicola (Florence), Hanne Riis Nielson (Copenhagen), Jens Palsberg (UCLA), Mooly Sagiv (Tel-Aviv), João Saraiva (Minho), Don Sannella (Edinburgh), Vladimiro Sassone (Southampton), Helmut Seidl (Munich), Peter Sestoft (Copenhagen), Andreas Zeller (Saarbrücken).

I would like to express my sincere gratitude to all of these people and organizations, the Program Committee chairs and PC members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, the many reviewers, and Springer for agreeing to publish the ETAPS proceedings. Finally, I would like to thank the Organizing Chair of ETAPS 2006, Jens Knoop, for arranging for us to have ETAPS in the beautiful city of Vienna.

Edinburgh
January 2006

Perdita Stevens
ETAPS Steering Committee Chair

Preface

This volume contains 21 papers presented at ESOP 2006, the annual European Symposium on Programming, in Vienna, Austria, 27–28 March 2006. The first ESOP was organized in 1986 by Bernard Robinet and Reinhard Wilhelm in Saarbrücken, so this marks the 20th anniversary of ESOP, but is the 15th symposium, since the symposia were initially held biannually. On occasion of the anniversary we are particularly happy that Reinhard Wilhelm agreed to join this year’s program committee.

The goal of ESOP has always been to bridge the gap between theory and practice, and the conferences continue to be devoted to addressing fundamental issues in the specification, analysis, and implementation of programming languages and systems.

The volume begins with a summary of Sophia Drossopoulou’s ESOP invited talk, continues with the contributed ESOP papers, and ends with the abstract of Benjamin Pierce’s ETAPS joint invited talk. The 21 ESOP papers were selected by the program committee from 87 full paper submissions, each reviewed by three or more reviewers, with four being the typical number. The reviews were done by the program committee and 143 additional referees, listed here. The accepted papers were selected during a two-week electronic discussion within the program committee.

Thanks go to the authors, the members of the program committee and the external referees for their excellent work, to the ESOP steering committee chair Hanne Riis Nielson, the ETAPS steering committee chair Perdita Stevens and the ETAPS 2006 local organization chair Jens Knoop for providing infrastructure and gentle reminders, and finally to the Online Conference System maintainer Martin Karrusseit for fixing server problems and adding desirable functionality.

Copenhagen, January 2006

Peter Sestoft

Organization

Program Chair

Peter Sestoft
Royal Veterinary and Agricultural University (KVL)
and IT University Copenhagen, Denmark

Program Committee

Anindya Banerjee	Kansas State University, USA
Anton Ertl	Technische Universitt Wien, Austria
David Warren	Stony Brook University, USA
Didier Rmy	INRIA Rocquencourt, France
Erik Meijer	Microsoft Corporation, USA
Eugenio Moggi	University of Genova, Italy
German Vidal	Technical University of Valencia, Spain
Giuseppe Castagna	cole Normale Suprieure, France
Joe Wells	Heriot-Watt University, UK
Kostis Sagonas	Uppsala University, Sweden
Michele Bugliesi	University of Venice, Italy
Mooly Sagiv	Tel-Aviv University, Israel
Nick Benton	Microsoft Research, UK
Peter O'Hearn	Queen Mary, University of London, UK
Peter Sestoft (chair)	KVL and IT University Copenhagen, Denmark
Peter Stuckey	Melbourne University, Australia
Peter Thiemann	Freiburg University, Germany
Pieter Hartel	Twente University, Netherlands
Reinhard Wilhelm	Saarland University, Germany
Stephanie Weirich	University of Pennsylvania, USA
Susan Eisenbach	Imperial College London, UK
Todd Veldhuizen	Indiana University, USA
Ulrik Pagh Schultz	University of Southern Denmark, Denmark

Additional Referees

E. Albert	C. Anderson	S. Barker
T. Amtoft	J. Avery	J. Bauer
D. Ancona	B. Aydemir	J. Berdine

G. Bierman	W. Heaven	B. Pierce
L. Birkedal	T. Hildebrandt	A. Pitts
V. Bono	T. Hirschowitz	A. Podelski
A. Bossi	J. Jaffar	F. Pottier
B. Brassel	N.D. Jones	G. Puebla
R. Brinkman	A. Kennedy	F. Puntigam
A. Brogi	D. Kesner	F. Ranzato
P. Buchlovsky	J. Knoop	J. Rehof
N. Busi	K. Kristoffersen	J. Reineke
R. Caballero	G. Lagorio	S. Rossi
N. Cameron	J. Lawall	C. Russo
V. Capretta	J. Lee	R. Rydhof Hansen
L. Cardelli	S. Lengrand	C. Sadler
J. Cederquist	M. Lenisa	A. Saptawijaya
R. Chatley	X. Leroy	A. Schmitt
S. Chong	P. Levy	C. Schrmann
A. Compagnoni	P. Li	C. Segura
B. Cook	H.H. Lvengreen	P. Sewell
R. Corin	M. Maffei	J. Silva
A. Cortesi	S. Maffeis	C. Skalka
S. Crafa	M. Maher	M. Smith
K. Crary	P. Maier	P. Sobocinski
D. Cunningham	H. Makhholm	Z. Somogyi
M. Czenko	Y. Mandelbaum	H. Sndergaard
F. Damiani	R. Manevich	A. Stoughton
R. Davies	L. Maranget	E. Sumii
J. den Hartog	C. McBride	D. Syme
D. Distefano	I. Mijajlovic	S. Tse
J. Doumen	A. Myers	D. Varacca
D. Dreyer	A. Mller	A. Villanueva
S. Drossopoulou	R. Mller Jensen	D. Vytiniotis
G. Dufay	S. Nanz	B. Wachter
N. Dulay	M. Neubauer	P. Wadler
M. Elsmann	U. Neumerkel	D. Walker
E. Ernst	S. Nishimura	G. Washburn
S. Escobar	B. Nordstrm	A. Wasowski
S. Fagorzi	L. Ong	H. Xi
T. Field	R. Pagh	H. Yang
A. Filinski	N.S. Papaspyrou	G. Yorsh
J. Foster	J. Parrow	F. Zappa Nardelli
C. Fournet	A. Petrounias	U. Zarfary
A. Francalanza	M. Pettersson	S. Zdancewic
R. Garcia	S. Peyton Jones	E. Zucca
P. Giannini	A. Phillips	A. Zych
D. Gorla	I. Phillips	

Table of Contents

Types for Hierarchic Shapes (Summary) <i>Sophia Drossopoulou, Dave Clarke, James Noble</i>	1
Linear Regions Are All You Need <i>Matthew Fluet, Greg Morrisett, Amal Ahmed</i>	7
Type-Based Amortised Heap-Space Analysis <i>Martin Hofmann, Steffen Jost</i>	22
Haskell Is Not Not ML <i>Ben Rudiak-Gould, Alan Mycroft, Simon Peyton Jones</i>	38
Coinductive Big-Step Operational Semantics <i>Xavier Leroy</i>	54
Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types <i>Amal Ahmed</i>	69
Approaches to Polymorphism in Classical Sequent Calculus <i>Alexander J. Summers, Steffen van Bakel</i>	84
Pure Pattern Calculus <i>Barry Jay, Delia Kesner</i>	100
A Verification Methodology for Model Fields <i>K. Rustan M. Leino, Peter Müller</i>	115
ILC: A Foundation for Automated Reasoning About Pointer Programs <i>Limin Jia, David Walker</i>	131
Bisimulations for Untyped Imperative Objects <i>Vasileios Koutavas, Mitchell Wand</i>	146
A Typed Assembly Language for Confidentiality <i>Dachuan Yu, Nayeem Islam</i>	162
Flow Locks: Towards a Core Calculus for Dynamic Flow Policies <i>Niklas Broberg, David Sands</i>	180
A Basic Contract Language for Web Services <i>Samuele Carpineti, Cosimo Laneve</i>	197

Types for Dynamic Reconfiguration	
<i>João Costa Seco, Luís Caires</i>	214
Size-Change Termination Analysis in k -Bits	
<i>Michael Codish, Vitaly Lagoon, Peter Schachte, Peter J. Stuckey</i>	230
Path Optimization in Programs and Its Application to Debugging	
<i>Akash Lal, Junghee Lim, Marina Polishchuk, Ben Liblit</i>	246
Inference of User-Defined Type Qualifiers and Qualifier Rules	
<i>Brian Chin, Shane Markstrum, Todd Millstein, Jens Palsberg</i>	264
Assertion Checking over Combined Abstraction of Linear Arithmetic and Uninterpreted Functions	
<i>Sumit Gulwani, Ashish Tiwari</i>	279
Embedding Dynamic Dataflow in a Call-by-Value Language	
<i>Gregory H. Cooper, Shriram Krishnamurthi</i>	294
Polymorphic Type Inference for the JNI	
<i>Michael Furr, Jeffrey S. Foster</i>	309
Type Safety of Generics for the .NET Common Language Runtime	
<i>Nicu G. Fruja</i>	325
The Weird World of Bi-directional Programming	
<i>Benjamin C. Pierce</i>	342
Author Index	343

Types for Hierarchic Shapes^{*}

(Summary)

Sophia Drossopoulou¹, Dave Clarke², and James Noble³

¹ Imperial College London, UK

² CWI, Amsterdam, The Netherlands

³ Victoria University of Wellington, Wellington, NZ

Abstract. Heap entities tend to contain complex references to each other. To manage this complexity, types which express shapes and hierarchies have been suggested. We survey type systems which describe such hierarchic shapes, how these types are used for reasoning about programs, and applications in concurrent programming.

Most imperative programs create and manipulate heap entities (objects, or records) which contain references to each other forming intricate topologies. This creates complexity, and makes programs difficult to understand and manipulate. Programmers, on the other hand, tend to think in terms of shapes, categorizations and hierarchies.

Thus, in the last decade, types describing shapes and hierarchies have been proposed to express programming intuitions, to support verification, and for synchronization and optimizations. We will discuss types for hierarchic shapes in terms of object oriented programming, because, even though the ideas are applicable to any imperative language, most of the related research was conducted in the context of object oriented languages.

1 Types for Hierarchic Shapes

Information hiding [28] was suggested as early as the 1970s, as a means to make programs more robust and easy to understand. Mechanisms that achieve information hiding by restricting the visibility of names, *e.g.*, private/protected annotations, are useful but insufficient. They prevent the *name* of an entity from being *used* outside a class or package, but do not prevent a *reference* to an entity from being *leaked* out of a structure [26].

To prevent such leaking of references, type systems have been suggested which give guarantees about the topology of the object graph, *i.e.*, about which objects may access which other objects.

Ownership types [15] introduce the concept of an object *owning* its nested objects; an *ownership context* is the set of objects with a given common owner.

^{*} Slides available from slurp.doc.ic.ac.uk/pubs.html#esop06.

Objects have a unique owner, thus ownership contexts are organized hierarchically into a tree structure. Furthermore, the owner controls access to the owned objects, because an object may only be accessed by its direct owner, or by objects (possibly indirectly) owned by the former object’s owner. Therefore, owners are *dominators* [15], where o_1 *dominates* o_2 , if any path from the “outside” (or “root” of the object graph) to o_2 goes through o_1 .

Ownership types can thus be used to characterize the runtime structure of object graphs. Analysis of the heaps of programs has demonstrated that indeed, object graphs tend to have structure: In [29] analysis of heap dumps for a corpus of programs demonstrated that the average nesting (ownership) depth of objects is 5. In [30] heap dumps for 60 object graphs from 35 programs demonstrated that the number of incoming and outgoing references follow a *power law*, whereby the *log* of the number of objects with k references is proportional to *log* of k , thus challenging the common perception that oriented programs are built out of layers of homogeneous components.

The owners as dominators approach, also known as *deep ownership*, gives very strong encapsulation properties which are natural in containers and nested structures [13]. The approach has been used in program visualization [25].

On the other hand, deep ownership makes coding some popular structures, notably iterators, rather cumbersome. To alleviate this, *shallow ownership* has given up on the notion of owners as dominators. In [10] inner classes have privileged access to the objects enclosed by the corresponding outer class object; *i.e.*, objects of an inner class may refer to objects owned by their outer class objects. In [14, 2] objects on the stack are allowed to break deep ownership, and to refer to the inside of an ownership context. A more refined approach [1] decouples the encapsulation policy from the ownership mechanism, by allowing multiple ownership *domains* (contexts in our terminology) per object, and by allowing the programmer to specify permitted aliasing between pairs of contexts.

Ownership types usually cannot easily handle change of owner, except for *externally* unique objects, *i.e.*, for objects for which references from the outside are unique [17].

The type of an object describes the owner of the object itself as well as the owners of the fields of the object; because these may be distinct, types are parameterized by *ownership parameters* which will be instantiated by objects enclosing the current object. This requires all types to be annotated by a number of ownership parameters.

Universes [22] suggest a more lightweight approach, whereby references to owned objects, or references to objects with the same owner may be used for modifications, and references to any other objects are readonly. Thus, universe type systems do not require ownership parameters, and instead only distinguish between **rep** for owned, **peer** for same owner, and **readonly** annotations. Types are coarser: **readonly** are readonly references which may point into any context.

Confined types, on the other hand, introduce the concept of classes confined to their defining package, and guarantee that instances of a confined class are only accessible by instances of classes from the same package; thus, they are only

manipulated by code belonging to the same package as the class [8]. The annotations required for confined types are simple, and the object graph structure is simpler in the sense that the ownership contexts represent the packages, and thus are statically known.

1.1 Hierarchic Shapes for Program Verification

The decomposition of heaps into disjoint sets of objects allows these objects to be treated together for the purposes of verification. Central issues in the context of program verification are that an object’s properties may depend on other objects’ properties, that objects’ invariants need to be temporarily broken and later re-established, and the treatment of abstraction layers, *e.g.*, when a **Set** is implemented in terms of a **List**. The notion of ownership is primarily related to the dependence of objects’ properties rather than the topology of object graphs.

Universes were developed with the aim to support *modular* program verification; in [24] universe types were applied to JML for the description of frame properties, where **modifies** clauses of method specifications define which objects may be modified. Modularity is achieved by a form of “underspecification” of the semantics, allowing method calls to modify objects *outside* the ownership context of the receiver without being mentioned in the relevant modifies-clause.

In [6] a methodology for program specification and verification is proposed, whereby an object’s invariants may depend on (possibly indirectly) owned objects. The state space of programs is enriched to express whether an object’s validity holds (*i.e.*, whether its invariant holds); there is support for explicitly altering an object validity, and explicit ownership transfer. Subclassing means that an object’s invariant may hold at the level of different superclasses of the given object. This approach is refined and adapted to universes in [21], and is implemented in Boogie, and further extended in [7] to allow invariants to be expressed over shared state.

However, the necessity to explicitly manipulate an object’s validity increases the overhead of verification; therefore, [23] defines implicitly in which execution states an object’s invariants must hold, based on an ownership model which is enforced by the type system.

Representation independence, which means that a class can safely be replaced by another “equivalent” class provided it is encapsulated, *i.e.*, its internal representation is owned by instances of that class, is proven in [4]. In [5] the approach is extended to deal with shared state, recursive methods and callbacks, and the application to program equivalence.

In a more fundamental approach, [20] develops a logic for reasoning about mutable data structures whereby the spatial conjunction operator $*$ splits the heap into two disjoint parts, usually one representing the part necessary for some execution, and the other representing the rest. In [19] the conjunction $*$ is used to separate the internal resources of a module from those accessed by its client, to support verification in the context of information hiding. Work in [27] introduces *abstract predicates*, which are treated atomically outside a data structure, but whose definition may be used within the data structure, thus

supporting reasoning about modules, ADTs and classes. In these approaches the heap is split afresh in each verification step; there is no hierarchy in that the heap is just split into two parts. The approaches are very flexible, but do not yet handle issues around the dependency of objects' properties and breaking/re-establishing of objects' invariants.

Using a simpler methodology, rather than attempt full-fledged verification, [14] describes read-write effects of methods in terms of the ownership contexts, and uses these to determine when method calls are *independent*, *i.e.*, their execution does affect each other. In [31] the approach is extended to describe read-effects of predicates, and to infer when some execution does not affect the validity of some predicate.

1.2 Applications of Hierarchic Shapes

Hierarchic shapes have successfully been applied in concurrent programming, garbage collection, and in deployment time checks of architectural invariants.

Guava [3] introduces additional type rules to Java which control synchronization by distinguishing between objects which can be shared across threads, and those which cannot. The former are monitors, and the latter are either thread-local, or encapsulated within a monitor.

In [11] race-free programs are obtained though an extension of ownership types, where an object may be owned not only by another object (as in the classical system) but also by the object itself, or by a thread (to express objects local to threads). By acquiring the lock at the root of an ownership tree, a thread acquires exclusive access to all the members of that tree. In [9] the approach is extended to prevent deadlocks, by requiring a partial order among all locks, and statically checking that threads holding more than one lock acquire them in descending order.

In real-time Java, timely reclamation of memory is achieved through *scoped types* [32, 12]. Scopes correspond to ownership contexts, in that they contain objects, are hierarchically organized into a tree, and outer scopes may not hold references to objects within more deeply nested inner scopes. When a thread working in scope S_1 enters scope S_2 , then S_1 becomes the owner of S_2 . When a thread enters a scope it is dynamically checked that it originated in its owner scope, thus guaranteeing nesting of scopes into a tree hierarchy. Scopes are released upon thread exit.

In [16] the architectural integrity constraints of the Enterprise Java Beans architecture, which require beans to be confined within their wrappers, are enforced through a lightweight confinement model and a deployment checker.

1.3 Inference of Hierarchic Shapes

The various systems for hierarchic shapes impose an extra burden of annotation to programmers, as they require each appearance of a class in a type description to be annotated by ownership parameters or restrictions such as **rep**.

Kacheck/J [18] is a tool which infers which classes are confined within a package in the sense of [8]. Applied on a corpus of 46,000 classes, it could deduce that around 25% of package scoped classes are confined.

In [2] an algorithm to infer ownership types is developed and successfully applied to 408 classes of the Java standard library. However, inferred types often contain too many ownership parameters, so precision needs to be improved.

2 Conclusions

Hierarchic shapes have successfully been used for program visualization and verification, in concurrent programming, garbage collection, and for architectural integrity constraints. Hierarchic shapes come in different flavours, and differ in whether they support change of owner, whether ownership implies restrictions on aliasing (through deep or shallow ownership) or dependence of properties, whether the ownership contexts correspond to objects, classes or packages, whether the number of ownership contexts is statically or dynamically known, whether ownership is checked statically or dynamically, how many annotations are required, and whether inference is supported.

Further work is required to combine the different uses of the shapes, to develop more lightweight yet powerful systems, to develop better inference tools to alleviate the process of annotating programs, to combine shape types with new trends in program development (most notably with aspect oriented programming), and finally to combine the ease of use offered by types with the flexibility offered by full-fledged verification as in separation logic.

References

1. Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating aliasing Policy from Mechanism. In *ECOOP*, 2004.
2. Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, November 2002.
3. David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA*, 2000.
4. Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *JACM*, 2005.
5. Anindya Banerjee and David A. Naumann. State based ownership, renentrance and encapsulation. In *ECOOP*, 2005.
6. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology*, 2004.
7. Mike Barnett and David A. Naumann. Friends Need a Bit More: Maintaining Invariants Over Shared State. In *Mathematics of Program Construction*, 2004.
8. Boris Bokowski and Jan Vitek. Confined Types. In *OOPSLA*, 1999.
9. Chandrasekar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.
10. Chandrasekar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *POPL*, 2003.

11. Chandrasekar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2002.
12. Chandrasekar Boyapati, Alexandru Salcianu, William Beebe, and Martin Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *PLDI*, June 2003.
13. Gustaf Cele and Sebastian Stureborg. Ownership Types in Practice. Technical Report TR-02-02, Stockholm University, 2002.
14. David Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*, 2002.
15. David Clarke, John Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.
16. David Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: Deployment-time confinement checking. In *OOPSLA*, 2003.
17. David Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In *ECOOP*, 2003.
18. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating Objects with Confined Types. In *OOPSLA*, 2001.
19. Peter W. O’ Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL*, 2004.
20. Samin Ishtiaq and Peter W. O’ Hearn. Bi as an assertion language for mutable data structures. In *POPL*, 2000.
21. K. Rustan M. Leino and Peter Müller. Object Invariants in Dynamic Contexts. In *ECOOP*, 2004.
22. Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*, 1999.
23. Peter Müller, Arnd Poetzsch-Heffter, and Gary Leavens. Modular Invariants for Layered Object Structures. Technical Report 424, ETH Zürich, 2004. *further development under way*.
24. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular Specification of Frame Properties in JML. *Concurrency and Computation Practice and Experience*, 2003.
25. James Noble. Visualising Objects: Abstraction, Encapsulation, Aliasing and Ownership. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Software Visualisation, State of the Art Survey*, pages 58–72. LNCS 2269, 2002.
26. James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In *ECOOP*, 1998.
27. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2004.
28. David Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACS*, 1972.
29. Alex Potanin and James Noble. Checking ownership and confinement properties. In *Formal Techniques for Java-like Programs*, 2002.
30. Alex Potanin, James Noble, Marcus Freen, and Robert Biddle. Scale-free geometry in OO programs. *Commun. ACM*, 2005.
31. Matthew Smith and Sophia Drossopoulou. Cheaper Reasoning with Ownership Types. In *IWACO*. 2003.
32. Tian Zhao, James Noble, and Jan Vitek. Scoped Types for Real-time Java. In *RTSS*, 2004.

Linear Regions Are All You Need

Matthew Fluet¹, Greg Morrisett², and Amal Ahmed²

¹Cornell University,
Ithaca, NY
`fluet@cs.cornell.edu`

²Harvard University,
Cambridge, MA
`{greg, amal}@eecs.harvard.edu`

Abstract. The type-and-effects system of the Tofte-Talpin region calculus makes it possible to safely reclaim objects without a garbage collector. However, it requires that regions have last-in-first-out (LIFO) lifetimes following the block structure of the language. We introduce λ^{rgnUL} , a core calculus that is powerful enough to encode Tofte-Talpin-like languages, and that eliminates the LIFO restriction. The target language has an extremely simple, substructural type system. To prove the power of the language, we sketch how Tofte-Talpin-style regions, as well as the first-class dynamic regions and unique pointers of the Cyclone programming language can be encoded in λ^{rgnUL} .

1 Introduction

Most type-safe languages rely upon a garbage collector to reclaim storage safely. But there are domains, such as device drivers and embedded systems, where today's garbage collection algorithms result in unacceptable space or latency overheads. In these settings, programmers have been forced to use languages, like C, where memory management can be tailored to the application, but where the lack of type-safety has led to numerous bugs. To address these concerns, we have been developing Cyclone [1], a type-safe dialect of C that is intended to give programmers as much control over memory management as possible while retaining strong, static typing.

The initial design of Cyclone was based upon the region type system of Tofte and Talpin [2]. Data are allocated within lexically-scoped *regions* and all of the objects in a region are deallocated at the end of the region's scope. Unfortunately, the last-in-first-out (LIFO) lifetimes of lexically-scoped regions place severe restrictions on when data can be effectively reclaimed, and we found many programs that resulted in (unbounded) leaks when compared to a garbage collected implementation.

To address these concerns, we added a number of new features to Cyclone, including *dynamic regions* and *unique pointers* that provide more control over memory management. Dynamic regions are not restricted to LIFO lifetimes and can be treated as first-class objects. They are particularly well suited for iterative computations, CPS-based computations, and event-based servers where

lexical regions do not suffice. Unique pointers are essentially lightweight, dynamic regions that hold exactly one object. To ensure soundness, both dynamic regions and unique pointers depend upon a notion of *linear capabilities* which must be carefully threaded through a program. To alleviate this tedium, Cyclone provides convenient mechanisms to temporarily “open” a dynamic region or unique pointer and treat it as if it were in a freshly allocated, lexically-scoped region.

The efficacy of these new memory management features was detailed in previous papers [3, 4], where we analyzed a range of applications, including a streaming media server, a space-conscious web server, and a Scheme runtime system with a copying garbage collector. And while the soundness of Cyclone’s lexical regions and type-and-effects system has been established [5, 6], a model that justifies the soundness of the new features has eluded our grasp, due to sheer complexity.

Therefore, the goal of this work is to provide a simple model where we can easily *encode* the key features of Cyclone in a uniform target language for which type soundness may be easily established. The first step of our encoding was detailed in a previous paper [6], where we gave a translation from a type-and-effects, region-based language to a monadic variant of **System F** called \mathbf{F}^{RGN} . This calculus is summarized in Section 2. The meat of this paper picks up where this translation left off by further translating \mathbf{F}^{RGN} to a substructural polymorphic lambda calculus where the internals of the indexed monad are exposed (Section 3). The target language and translation are extremely simple, yielding a relatively straightforward proof of soundness for lexically scoped regions. Then, in Section 5, we sketch how the features in the target language allow us to encode Cyclone’s dynamic regions and unique pointers, as well as their interactions with lexically-scoped regions. Throughout, it is our intention that the target calculus serve as a compiler intermediate language and as vehicle for formal reasoning, not as a high-level programming language.

2 Source Calculus: \mathbf{F}^{RGN}

Launchbury and Peyton Jones introduced the **ST** monad to encapsulate stateful computations within the pure functional language Haskell [7]. Three key insights give rise to a safe and efficient implementation of stateful computations. First, a stateful computation is represented as a *store transformer*, a description of commands to be applied to an initial store to yield a final store. Second, the store can not be duplicated, because the state type is opaque and all primitive store transformers use the store in a single-threaded manner; hence, a stateful computation can update the store in place. Third, parametric polymorphism can be used to safely encapsulate and run a stateful computation.

All of these insights can be carried over to the region case, where we interpret stores as stacks of regions. We introduce the types and operations associated with the **rgn** monad:

$$\tau ::= \dots \mid \mathbf{rgn} \, s \, \tau \mid \mathbf{ref} \, s \, \tau \mid \mathbf{hnd} \, s \mid \mathbf{pf} \, (s_1 \leq s_2)$$

```

return :  $\forall \varsigma. \forall \alpha. \alpha \rightarrow \mathbf{rgn} \varsigma \alpha$ 
then :  $\forall \varsigma. \forall \alpha, \beta. \mathbf{rgn} \varsigma \alpha \rightarrow (\alpha \rightarrow \mathbf{rgn} \varsigma \beta) \rightarrow \mathbf{rgn} \varsigma \beta$ 
new :  $\forall \varsigma. \forall \alpha. \mathbf{hnd} \varsigma \rightarrow \alpha \rightarrow \mathbf{rgn} \varsigma (\mathbf{ref} \varsigma \alpha)$ 
read :  $\forall \varsigma. \forall \alpha. \mathbf{ref} \varsigma \alpha \rightarrow \mathbf{rgn} \varsigma \alpha$ 
write :  $\forall \varsigma. \forall \alpha. \mathbf{ref} \varsigma \alpha \rightarrow \alpha \rightarrow \mathbf{rgn} \varsigma 1$ 
runRgn :  $\forall \alpha. (\forall \varsigma. \mathbf{rgn} \varsigma \alpha) \rightarrow \alpha$ 
letRgn :  $\forall \varsigma_1. \forall \alpha. (\forall \varsigma_2. \mathbf{pf} (\varsigma_1 \leq \varsigma_2) \rightarrow \mathbf{hnd} \varsigma_2 \rightarrow \mathbf{rgn} \varsigma_2 \alpha) \rightarrow \mathbf{rgn} \varsigma_1 \alpha$ 
coerceRgn :  $\forall \varsigma_1, \varsigma_2. \forall \alpha. \mathbf{pf} (\varsigma_1 \leq \varsigma_2) \rightarrow \mathbf{rgn} \varsigma_1 \alpha \rightarrow \mathbf{rgn} \varsigma_2 \alpha$ 
reflSub :  $\forall \varsigma. \mathbf{pf} (\varsigma \leq \varsigma)$ 
transSub :  $\forall \varsigma_1, \varsigma_2, \varsigma_3. \mathbf{pf} (\varsigma_1 \leq \varsigma_2) \rightarrow \mathbf{pf} (\varsigma_2 \leq \varsigma_3) \rightarrow \mathbf{pf} (\varsigma_1 \leq \varsigma_3)$ 

```

The type $\mathbf{rgn} \, s \, \tau$ is the type of computations which transform a stack indexed by s and deliver a value of type τ . The type $\mathbf{ref} \, s \, \tau$ is the type of mutable references allocated in the region at the top of the stack indexed by s and containing a value of type τ . The type $\mathbf{hnd} \, s$ is the type of handles for the region at the top of the stack indexed by s ; we require a handle to allocate a reference in a region, but do not require a handle to read or write a reference.

The operations **return** and **then** are the *unit* and *bind* operations of the **rgn** monad, the former lifting a value to a computation and the latter sequencing computations. The next three operations are primitive stack transformers. **new** takes a region handle and an initial value and yields a stack transformer, which, when applied to a stack of regions, allocates and initializes a fresh reference in the appropriate region, and delivers the reference and the augmented stack. Similarly, **read** and **write** yield computations that respectively query and update the mappings of references to values in the current stack of regions. Note that all of these operations require the stack index ς of **rgn** and **ref** to be equal.

Finally, the operation **runRgn** encapsulates a stateful computation. To do so, it takes a stack transformer as its argument, applies it to an initial empty stack of regions, and returns the result while discarding the final stack (which should be empty). Note that to apply **runRgn**, we instantiate α with the type of the result to be returned, and then supply a stack transformer, *which is polymorphic in the stack index* ς . The effect of this universal quantification is that the stack transformer makes no assumptions about the initial stack (e.g., the existence of pre-allocated regions or references). Furthermore, the instantiation of the type variable α occurs outside the scope of the stack variable ς ; this prevents the stack transformer from delivering a value whose type mentions ς . Thus, references or computations depending on the final stack cannot escape beyond the encapsulation of **runRgn**.

However, the above does not suffice to encode region-based languages. The difficulty is that, in a region-based language, it is critical to allocate variables in and read variables from an outer (older) region while in the scope of an inner (younger) region. To accommodate this essential idiom, we include a powerful **letRgn** operation that is similar to **runRgn** in the sense that it encapsulates a stateful computation. Operationally, **letRgn** transforms a stack by (1) creating a new region on the top of the stack, (2) applying a stack transformer to the augmented stack to yield a transformed stack, (3) destroying the region on the top of the transformed stack and yielding the bottom of the transformed stack.

<i>Kinds</i>	$\kappa ::= \text{STACK} \mid \star$
<i>Type-level Variables</i>	$\varepsilon, \varsigma, \alpha ::= T\text{Vars}$
<i>Stack Indices</i>	$s ::= \varsigma$
<i>Types</i>	$\tau ::= \alpha \mid \tau_1 \rightarrow \tau_2 \mid \mathbf{1} \mid \tau_1 \times \tau_2 \mid \forall \varepsilon:\kappa. \tau$ $\quad \text{rgn } s \tau \mid \text{ref } s \tau \mid \text{hnd } s \mid \text{pf } (s_1 \leq s_2)$
<i>Type-level Terms</i>	$\epsilon ::= s \mid \tau$
<i>Type-level Contexts</i>	$\Delta ::= \bullet \mid \Delta, \varepsilon:\kappa$
<i>rgn Monad Operations</i>	$\text{ops} ::= \text{runRgn} \mid \text{coerceRgn} \mid \text{transSub} \mid$ $\quad \text{return} \mid \text{then} \mid \text{letRgn} \mid \text{new} \mid \text{read} \mid \text{write}$
<i>Expressions</i>	$e ::= \text{ops} \mid x \mid \lambda x:\tau. e \mid e_1 e_2 \mid \langle \rangle \mid \text{let } \langle \rangle = e_1 \text{ in } e_2 \mid$ $\quad \langle e_1, e_2 \rangle \mid \text{let } \langle x_1, x_2 \rangle = e_1 \text{ in } e_2 \mid \Lambda \varepsilon:\kappa. e \mid e[\epsilon]$
<i>Expression-level Contexts</i>	$\Gamma ::= \bullet \mid \Gamma, x:\tau$

Fig. 1. Syntax of \mathbf{F}^{RGN}

Ignoring for the moment the argument of type $\text{pf}(s_1 \leq s_2)$, we see that we may apply exactly the same reasoning as applied to **runRgn**: the computation makes no assumptions about the newly augmented stack s_2 , nor can the newly augmented stack s_2 be leaked through the return value.

What, then, is the role of the $\text{pf}(s_1 \leq s_2)$? The answer lies in the fact that the stack index s_2 does not denote an arbitrary stack; rather, it should denote a stack that is related to s_1 by the addition of a newly created region (i.e., $s_2 \equiv r::s_1$). In fact, we may consider s_1 to be a subtype of s_2 , since every region in the stack s_1 is also in the stack s_2 ; values of type $\text{pf}(s_1 \leq s_2)$ are witnesses of this relationship. The operation **coerceRgn** applies a subtyping witness to a stack transformer for the substack to yield a stack transformer for the superstack; intuitively, the operation is sound as a stack transformer may simply ignore extra regions. The operations **reflSub** and **transSub** are combinators witnessing the reflexivity and transitivity of the subtyping relation.

Figure 1 gives the complete syntax for \mathbf{F}^{RGN} , which is a natural extension of **System F**. We introduce a simple kind system to support abstraction over both types and stack indices. (In the text, we often omit kind annotations, using the convention that ς stands for a type-level variable of **STACK** kind, and α of \star .)

We adopt the standard type system for **System F**; the only typing judgement of interest is $\Delta; \Gamma \vdash e : \tau$ meaning that expression e has type τ , where Δ records the free type-level variables and their kinds and Γ records the free expression-level variables and their types. The types for the **rgn** monad operations are as given in the text above.

Our previous work [6] gave an operational semantics for \mathbf{F}^{RGN} and proved the type soundness of \mathbf{F}^{RGN} . However, the operational semantics of \mathbf{F}^{RGN} is somewhat cumbersome, due to the intertwining of contexts for pure evaluation and monadic evaluation. Hence, in the present setting, we will *define* the operational behavior of \mathbf{F}^{RGN} by its translation into the target language of Section 3.

3 Target Calculus: λ^{rgnUL}

In another line of work [8], we introduced λ^{URAL} , a core substructural polymorphic λ -calculus, and then extended it to λ^{refURAL} by adding a rich collection of mutable references. Providing four sorts of substructural qualifiers (unrestricted, relevant, affine, and linear) allowed us to encode and study the interactions of different forms of uniqueness that appear in other high-level programming languages. Notable features of λ^{refURAL} include: deallocation of references; strong (type-varying) updates; and storage of unique objects in shared references.

Here, we augment λ^{refURAL} by adding region primitives, and also simplify the language by removing features, such as the relevant and affine qualifiers, that do not play a part in the translation. We call the resulting language λ^{rgnUL} .

In contrast to the **letRgn** operation of \mathbf{F}^{RGN} , which encapsulates the creation and destruction of a region, the primitives of λ^{rgnUL} include **newrgn** and **freergn** for separately creating and destroying a region. All access to a region (for allocating, reading, and writing references) is mediated by a linear *capability* that is produced by **newrgn** and consumed by **freergn**.

As noted above, λ^{rgnUL} is a *substructural* polymorphic λ -calculus. A *substructural* type system provides the core mechanisms necessary to restrict the number and order of uses of data and operations. In our calculus, types and variables are qualified as unrestricted (U) or linear (L). Essentially, unrestricted variables are allowed to be used an arbitrary number of times, while linear variables are allowed to be used exactly once.

Figure 2 gives the syntax for λ^{rgnUL} , excluding intermediate terms that would appear in an operational semantics. Many of the types and expressions are based on a traditional polymorphic λ -calculus.

We structure our types τ as a qualifier q applied to a pre-type $\bar{\tau}$, yielding the two sorts of types noted above. The qualifier of a type dictates the number of uses of variables of the type, while the pre-type dictates the introduction and elimination forms. The pre-types $\mathbf{1}_{\otimes}$, $\tau_1 \otimes \cdots \otimes \tau_n$, and $\tau_1 \multimap \tau_2$ correspond to the unit, product, and function types of the polymorphic λ -calculus. Quantification over qualifiers, region names, pre-types, and types is provided by the pre-types $\forall \varepsilon : \kappa. \tau$ and $\exists \varepsilon : \kappa. \tau$. (In the text, we often omit kind annotations, using the convention that ξ stands for a type-level variable of **QUAL** kind, ϱ of **RGN**, $\bar{\alpha}$ of $\bar{\star}$, and α of \star .)

The pre-types **ref** r τ and **hnd** r are similar to the corresponding types in \mathbf{F}^{RGN} ; the former is the type of mutable references allocated in the region r and the latter is the type of handles for the region r . The pre-type **cap** r is the type of capabilities for accessing the region named r . We shall shortly see how linear capabilities effectively mediate access to a region.

Space precludes us from giving a detailed description of the type system for λ^{rgnUL} ; the major features are entirely standard for a substructural setting [9, 8]. First, in order to ensure the correct relationship between a data structure and its components, we extend the lattice ordering on constant qualifiers to arbitrary qualifiers ($\Delta \vdash q \preceq q'$), types ($\Delta \vdash \tau \preceq q'$), and contexts ($\Delta \vdash \Gamma \preceq q'$). Second, we introduce a judgement $\Delta \vdash \Gamma_1 \boxdot \Gamma_2 \leadsto \Gamma$ that splits the assumptions in Γ between the contexts Γ_1 and Γ_2 . Splitting the context is necessary to ensure that

<i>Kinds</i>	$\kappa ::= \text{QUAL} \mid \text{RGN} \mid \bar{\tau} \mid \star$
<i>Type-level Variables</i>	$\varepsilon, \xi, \varrho, \bar{\alpha}, \alpha ::= T\text{Vars}$
<i>Constant Qualifiers</i>	$q \in \text{Quals} = \{\text{U}, \text{L}\} \quad \text{U} \sqsubseteq \text{L}$
<i>Qualifiers</i>	$q ::= \xi \mid q$
<i>Constant Region Names</i>	$\mathfrak{r} \in R\text{Names}$
<i>Region Names</i>	$r ::= \varrho \mid \mathfrak{r}$
<i>PreTypes</i>	$\bar{\tau} ::= \bar{\alpha} \mid \tau_1 \multimap \tau_2 \mid \mathbf{1}_{\otimes} \mid$ $\tau_1 \otimes \cdots \otimes \tau_n \mid \forall \varepsilon:\kappa. \tau \mid \exists \varepsilon:\kappa. \tau \mid$ $\text{ref } r \tau \mid \text{hnd } r \mid \text{cap } r$
<i>Types</i>	$\tau ::= \alpha \mid {}^q\bar{\tau}$
<i>Type-level Terms</i>	$\epsilon ::= q \mid r \mid \bar{\tau} \mid \tau$
<i>Type-level Contexts</i>	$\Delta ::= \bullet \mid \Delta, \varepsilon:\kappa$
<i>Region Primitives</i>	$\text{prims} ::= \text{newrgn} \mid \text{freergn} \mid \text{new} \mid \text{read} \mid \text{write}$
<i>Expressions</i>	$e ::= \text{prims} \mid x \mid {}^q\lambda x:\tau. e \mid e_1 e_2 \mid {}^q\langle \rangle \mid \text{let } \langle \rangle = e_1 \text{ in } e_2 \mid$ ${}^q\langle e_1, \dots, e_n \rangle \mid \text{let } \langle x_1, \dots, x_n \rangle = e_1 \text{ in } e_2 \mid$ ${}^q\Lambda \varepsilon:\kappa. e \mid e[\epsilon] \mid {}^q\text{pack}(\varepsilon:\kappa, e) \mid \text{let pack}(\varepsilon:\kappa, x) = e_1 \text{ in } e_2$
<i>Expression-level Contexts</i>	$\Gamma ::= \bullet \mid \Gamma, x:\tau$

Fig. 2. Syntax of λ^{rgnUL}

variables are used appropriately by sub-expressions. Note that \Box must ensure that an L assumption appears in exactly one sub-context, while U assumptions may appear in both sub-contexts.

The main typing judgement has the form $\Delta; \Gamma \vdash e : \tau$; Figure 3 gives typing rules for each of the expression forms of λ^{rgnUL} .

Finally, we assign types for each of the region primitives of λ^{rgnUL} :

$$\begin{aligned}
\text{newrgn} &: {}^{\text{U}}(\mathbf{1}_{\otimes} \multimap {}^{\text{L}}\exists \varrho. {}^{\text{L}}(\text{cap } \varrho \otimes {}^{\text{U}}\text{hnd } \varrho)) \\
\text{freergn} &: {}^{\text{U}}\forall \varrho. {}^{\text{U}}({}^{\text{L}}(\text{cap } \varrho \otimes {}^{\text{U}}\text{hnd } \varrho) \multimap {}^{\text{L}}\mathbf{1}_{\otimes}) \\
\text{new} &: {}^{\text{U}}\forall \varrho. {}^{\text{U}}\forall \bar{\alpha}. {}^{\text{U}}({}^{\text{L}}(\text{cap } \varrho \otimes {}^{\text{U}}\text{hnd } \varrho \otimes {}^{\text{U}}\bar{\alpha}) \multimap {}^{\text{L}}(\text{cap } \varrho \otimes {}^{\text{U}}(\text{ref } \varrho {}^{\text{U}}\bar{\alpha}))) \\
\text{read} &: {}^{\text{U}}\forall \varrho. {}^{\text{U}}\forall \bar{\alpha}. {}^{\text{U}}({}^{\text{L}}(\text{cap } \varrho \otimes {}^{\text{U}}(\text{ref } \varrho {}^{\text{U}}\bar{\alpha})) \multimap {}^{\text{L}}(\text{cap } \varrho \otimes {}^{\text{U}}\bar{\alpha})) \\
\text{write} &: {}^{\text{U}}\forall \varrho. {}^{\text{U}}\forall \bar{\alpha}. {}^{\text{U}}({}^{\text{L}}(\text{cap } \varrho \otimes {}^{\text{U}}(\text{ref } \varrho {}^{\text{U}}\bar{\alpha}) \otimes {}^{\text{U}}\bar{\alpha}) \multimap {}^{\text{L}}(\text{cap } \varrho \otimes {}^{\text{U}}\mathbf{1}_{\otimes}))
\end{aligned}$$

We have purposefully “streamlined” the type of the reference primitives in order to simplify the exposition. For example, note that we may only allocate, read, and write references whose contents are unrestricted. However, there is no fundamental difficulty in adopting a richer set of reference primitives (à la λ^{refURAL} [8]), which would allow references to contain arbitrary values.

Space again precludes us from giving a detailed description of the operational semantics for λ^{rgnUL} ; however, it is entirely standard for a region-based language. The small-step operational semantics is defined by a relation between configurations of the form (ψ, e) , where ψ is a global heap mapping region names to regions and regions are mappings from pointers to values.

The primitives **newrgn** and **freergn** perform the complementary actions of creating and destroying a region in the global heap. Note that the type of **newrgn** specifies that it returns an existential package, hiding the name of the fresh re-

$$\boxed{\Delta; \Gamma \vdash e : \tau}$$

$$\begin{array}{c}
\frac{}{\Delta; \bullet, x : \tau \vdash x : \tau} \quad \frac{\Delta; \Gamma_1 \boxtimes \Gamma_2 \rightsquigarrow \Gamma \quad \Delta \vdash \Gamma_1 \preceq \mathbf{U} \quad \Delta; \Gamma_2 \vdash e : \tau}{\Delta; \Gamma \vdash e : \tau} \\
\\
\frac{\Delta \vdash q : \text{QUAL}}{\Delta; \bullet \vdash {}^q \langle \rangle : {}^q \mathbf{1}_{\otimes}} \quad \frac{\Delta \vdash \Gamma_1 \boxtimes \Gamma_2 \rightsquigarrow \Gamma \quad \Delta; \Gamma_1 \vdash e_1 : {}^q \mathbf{1}_{\otimes} \quad \Delta; \Gamma_2 \vdash e_2 : \tau}{\Delta; \Gamma \vdash \text{let } \langle \rangle = e_1 \text{ in } e_2 : \tau} \\
\\
\frac{\Delta \vdash \Gamma_1 \boxtimes \dots \boxtimes \Gamma_n \rightsquigarrow \Gamma \quad \Delta; \Gamma_1 \vdash e_1 : \tau_1 \quad \Delta \vdash \tau_1 \preceq q \quad \dots \quad \Delta; \Gamma_n \vdash e_n : \tau_n \quad \Delta \vdash \tau_n \preceq q}{\Delta; \Gamma \vdash {}^q \langle e_1, \dots, e_n \rangle : {}^q (\tau_1 \otimes \dots \otimes \tau_n)} \quad \frac{\Delta \vdash \Gamma_1 \boxtimes \Gamma_2 \rightsquigarrow \Gamma \quad \Delta; \Gamma_1 \vdash e_1 : {}^q (\tau_1 \otimes \dots \otimes \tau_n) \quad \Delta; \Gamma_2, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e_2 : \tau}{\Delta; \Gamma \vdash \text{let } \langle x_1, \dots, x_n \rangle = e_1 \text{ in } e_2 : \tau} \\
\\
\frac{\Delta \vdash \Gamma \preceq q \quad \Delta; \Gamma, x : \tau_x \vdash e : \tau}{\Delta; \Gamma \vdash {}^q \lambda x : \tau_x. e : {}^q (\tau_x \multimap \tau)} \quad \frac{\Delta \vdash \Gamma_1 \boxtimes \Gamma_2 \rightsquigarrow \Gamma \quad \Delta; \Gamma_1 \vdash e_1 : {}^q (\tau_x \multimap \tau) \quad \Delta; \Gamma_2 \vdash e_2 : \tau_x}{\Delta; \Gamma \vdash e_1 e_2 : \tau} \\
\\
\frac{\Delta \vdash \Gamma \preceq q \quad \Delta, \varepsilon : \kappa; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash {}^q \Lambda \varepsilon : \kappa. e : {}^q (\forall \varepsilon : \kappa. \tau)} \quad \frac{\Delta; \Gamma \vdash e_1 : {}^q (\forall \varepsilon : \kappa. \tau) \quad \Delta \vdash e_2 : \kappa}{\Delta; \Gamma \vdash e_1 [\varepsilon_2] : \tau [\varepsilon_2 / \varepsilon]} \\
\\
\frac{\Delta \vdash e_1 : \kappa \quad \Delta; \Gamma \vdash e_2 : \tau [\varepsilon_1 / \varepsilon] \quad \Delta \vdash \tau [\varepsilon_1 / \varepsilon] \preceq q}{\Delta; \Gamma; \Sigma \vdash {}^q \text{pack}(\varepsilon_1 : \kappa, e_2) : {}^q (\exists \varepsilon : \kappa. \tau)} \quad \frac{\Delta \vdash \Gamma_1 \boxtimes \Gamma_2 \rightsquigarrow \Gamma \quad \Delta \vdash \tau' : \star \quad \Delta; \Gamma_1 \vdash e_1 : {}^q (\exists \varepsilon : \kappa. \tau) \quad \Delta, \varepsilon : \kappa; \Gamma_2, x : \tau \vdash e_2 : \tau'}{\Delta; \Gamma \vdash \text{let pack}(\varepsilon : \kappa, x) = e_1 \text{ in } e_2 : \tau'}
\end{array}$$

Fig. 3. Static Semantics of λ^{rgnUL}

gion. The primitives **new**, **read**, and **write** behave precisely as their counterparts in any region-based language. Additionally, their types specify that they thread $\mathbf{L}_{\text{cap } q}$ values through the evaluation; the capability is simply presented at each access of a region and returned to allow future access. In the semantics, the capability is represented as a dummy token, which has no run-time significance.

As expected, the type system for λ^{rgnUL} is sound with respect to its operational semantics:

Theorem 1 (λ^{rgnUL} Safety). *If $\bullet; \bullet \vdash e_1 : \tau$ and $(\{\}, e_1) \mapsto^* (\psi_2, e_2)$, then either there exists v such that $e_2 \equiv v$ or there exists ψ_3 and e_3 such that $(\psi_2, e_2) \mapsto (\psi_3, e_3)$.*

We have formally verified this result (for a rich superset of λ^{rgnUL}) in the Twelf system [10] using its metatheorem checker [11]. The mechanized proof can be obtained at <http://www.cs.cornell.edu/People/fluet/research/substruct-regions>.

4 Translation: \mathbf{F}^{rgn} to λ^{rgnUL}

Having introduced both our source and target calculi, we are in a position to consider a (type-preserving) translation from \mathbf{F}^{rgn} to λ^{rgnUL} . Before giving the details, we discuss a few of the high-level issues.

First, we note that \mathbf{F}^{RGN} has no notion of linearity in the syntax or type system. Rather, all variables and types are implicitly considered unrestricted. Hence, we can expect that the translation of all \mathbf{F}^{RGN} expressions will yield λ^{rgnUL} expressions with a \mathbf{U} qualified type.

On the other hand, we claimed that a stateful region computation could be interpreted as a stack transformer. Recall that the type $\text{rgn } s \tau$ is the type of computations which transform a stack indexed by s and deliver a value of type τ . A key characteristic of \mathbf{F}^{RGN} is that all primitive stack transformers are meant to use the stack in a single-threaded manner; hence, a stateful computation can update the stack in place. This single-threaded behavior is precisely the sort of resource management that may be captured by a substructural type system. Hence, we can expect that the representation of a stack of regions in λ^{rgnUL} will be a value with \mathbf{L} qualified type. In particular, we will represent a stack of regions as a sequence of linear capabilities, formed out of nested linear tuples.

Third, we must be mindful of a slight mismatch between the **hnd** and **ref** types in \mathbf{F}^{RGN} and the corresponding types in λ^{rgnUL} . Recall that, in \mathbf{F}^{RGN} , **hnd** s and **ref** $s \tau$ are handles for and references allocated in the region at the top of the stack indexed by s . Whereas, in λ^{rgnUL} , **hnd** r and **ref** $r \tau$ explicitly name the region of the handle or reference. This subtle distinction (whether the region is implicit or explicit) will need to be handled by the translation.

Bearing these issues in mind, we turn our attention to the translation of \mathbf{F}^{RGN} type-level terms given in Figure 4. $\mathcal{S}_\star \llbracket s \rrbracket$ translates a \mathbf{F}^{RGN} term of **STACK** kind to a λ^{rgnUL} term of \star kind. As the **STACK** kind of \mathbf{F}^{RGN} is inhabited only by variables, the translation is trivial: in λ^{rgnUL} , ς is considered a variable of \star kind.

$$\begin{array}{ll}
\mathbf{F}^{\text{RGN}} \text{ STACK } to \lambda^{\text{rgnUL}} \star & \mathbf{F}^{\text{RGN}} \star to \lambda^{\text{rgnUL}} \overline{\star} \text{ (functional types)} \\
\mathcal{S}_\star \llbracket \varsigma \rrbracket = \varsigma & \mathcal{T}_\star \llbracket \alpha \rrbracket = \overline{\alpha} \\
\mathbf{F}^{\text{RGN}} \star to \lambda^{\text{rgnUL}} \star & \mathcal{T}_\star \llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \mathcal{T}_\star \llbracket \tau_1 \rrbracket \multimap \mathcal{T}_\star \llbracket \tau_2 \rrbracket \\
\mathcal{T}_\star \llbracket \tau \rrbracket = {}^U \mathcal{T}_\star \llbracket \tau \rrbracket & \mathcal{T}_\star \llbracket \mathbf{1} \rrbracket = \mathbf{1}_\otimes \\
& \mathcal{T}_\star \llbracket \tau_1 \times \tau_2 \rrbracket = \mathcal{T}_\star \llbracket \tau_1 \rrbracket \otimes \mathcal{T}_\star \llbracket \tau_2 \rrbracket \\
& \mathcal{T}_\star \llbracket \forall \alpha: \star. \tau \rrbracket = \forall \overline{\alpha}: \overline{\star}. \mathcal{T}_\star \llbracket \tau \rrbracket \\
& \mathcal{T}_\star \llbracket \forall \varsigma: \text{STACK}. \tau \rrbracket = \forall \varsigma: \star. \mathcal{T}_\star \llbracket \tau \rrbracket \\
\\
\mathbf{F}^{\text{RGN}} \star to \lambda^{\text{rgnUL}} \overline{\star} \text{ (rgn monad types)} & \\
\mathcal{T}_\star \llbracket \text{rgn } s \tau \rrbracket = \mathcal{S}_\star \llbracket s \rrbracket \multimap {}^L (\mathcal{S}_\star \llbracket s \rrbracket \otimes \mathcal{T}_\star \llbracket \tau \rrbracket) & \\
\mathcal{T}_\star \llbracket \text{pf } (s_1 \leq s_2) \rrbracket = \exists \beta: \star. \mathbf{Iso}(\mathcal{S}_\star \llbracket s_2 \rrbracket, {}^L (\mathcal{S}_\star \llbracket s_1 \rrbracket \otimes \beta)) & \\
\mathcal{T}_\star \llbracket \text{hnd } s \rrbracket = \exists \varrho: \text{RGN}. {}^U ({}^U \exists \beta: \star. \mathbf{Iso}(\mathcal{S}_\star \llbracket s \rrbracket, {}^L (\beta \otimes {}^L (\text{cap } \varrho))) \otimes {}^U (\text{hnd } \varrho)) & \\
\mathcal{T}_\star \llbracket \text{ref } s \tau \rrbracket = \exists \varrho: \text{RGN}. {}^U ({}^U \exists \beta: \star. \mathbf{Iso}(\mathcal{S}_\star \llbracket s \rrbracket, {}^L (\beta \otimes {}^L (\text{cap } \varrho))) \otimes {}^U (\text{ref } \varrho \mathcal{T}_\star \llbracket \tau \rrbracket)) & \\
\\
\lambda^{\text{rgnUL}} \text{ Type-level Macros} & \\
\mathbf{Iso}(\tau_1, \tau_2) = {}^U ({}^U (\tau_1 \multimap \tau_2) \otimes {}^U (\tau_2 \multimap \tau_1)) &
\end{array}$$

Fig. 4. Type-level Translation

$\mathcal{T}_\star[\tau]$ and $\mathcal{T}_\star[\tau]$ translate a \mathbf{F}^{RGN} term of \star kind to λ^{rgnUL} terms of \star and $\bar{\star}$ kinds, respectively. As we observed above, when we translate a \mathbf{F}^{RGN} type to a λ^{rgnUL} type, we ensure that the result is a \mathbf{U} qualified type. The $\mathcal{T}_\star[\tau]$ translation is straightforward on the functional types. (However, note that a \mathbf{F}^{RGN} variable α of \star kind is translated to a λ^{rgnUL} variable $\bar{\alpha}$ of $\bar{\star}$ kind; this ensures that *every* type corresponding to a \mathbf{F}^{RGN} type is manifestly qualified with \mathbf{U} .)

More interesting are the translations of the types associated with the **rgn** monad. In the translation of the **rgn** $s\tau$ type, we see the familiar store (stack) passing interpretation of computations. Since the representation of a stack of regions is linear, the resulting store/value pair is qualified with \mathbf{L} . Next, consider the translation of the **pf** $(s_1 \leq s_2)$ type. Recall that it is the type of witnesses to the fact that the stack indexed by s_1 is a subtype of the stack indexed by s_2 . Hence, we translate to a type that expresses the isomorphism between $\mathcal{S}_\star[s_2]$ and $\mathbf{L}(\mathcal{S}_\star[s_1] \otimes \beta)$, for some “slack” β . Note that while the types $\mathcal{S}_\star[s_2]$, $\mathcal{S}_\star[s_1]$, and β may be linear, the pair of functions witnessing the isomorphism is unrestricted. This corresponds to the fact that the *proof* that s_1 is a subtype of s_2 is *persistent*, while the *existence* of the stacks s_1 and s_2 are *ephemeral*.

The translation of the **hnd** s and **ref** $s\tau$ types are similar. An existentially bound region name ϱ fixes the region for the λ^{rgnUL} handle or reference, while an isomorphism witnesses the fact that ϱ may be found within the stack $\mathcal{S}_\star[s]$.

With the translation of \mathbf{F}^{RGN} type-level terms in place, the translation of \mathbf{F}^{RGN} expressions follows almost directly. We elide the translation of the introduction and elimination forms for the functional types in \mathbf{F}^{RGN} (it is simply the homomorphic mapping of the given expression translations) and focus on the translation of the **rgn** monad operations. For readability, we give translations for fully applied region primitives only, assuming that partially applied primitives have been eta-expanded. The translation of **return** and **then** follow directly from our store (stack) passing interpretation of **rgn** $s\tau$ types:

$$\begin{aligned} \mathcal{E}[\mathbf{return}[s][\tau_\alpha]e] &= \text{let } res:\mathcal{T}_\star[\tau_\alpha] = \mathcal{E}[e] \text{ in} \\ &\quad \mathbf{U}\lambda stk:\mathcal{S}_\star[s].\mathbf{L}\langle stk, res \rangle \\ \mathcal{E}[\mathbf{then}[s][\tau_\alpha][\tau_\beta]e_1e_2] &= \text{let } f:\mathcal{T}_\star[\mathbf{rgn}s\tau_\alpha] = \mathcal{E}[e_1] \text{ in} \\ &\quad \text{let } g:\mathcal{T}_\star[\tau_\alpha \rightarrow \mathbf{rgn}s\tau_\beta] = \mathcal{E}[e_2] \text{ in} \\ &\quad \mathbf{U}\lambda stk:\mathcal{S}_\star[s].\text{let } \langle stk, res \rangle = f\ stk \text{ in} \\ &\quad \quad g\ res\ stk \end{aligned}$$

The translation of **letRgn** is the most complicated, but breaks down into conceptually simple components. We bracket the execution of the inner computation with a **newrgn/freergn** pair, creating and destroying a new region. We construct the representation of the new stack stk_2 for the inner computation by pairing the old stack stk_1 with the new region capability cap . Finally, we construct isomorphisms witnessing the relationships between the new region capability and the new stack and between the old stack and the new stack. We carefully chose the isomorphism types so that the identity function suffices as a witness. Putting all of these pieces together, we have the following:

$$\begin{aligned}
\mathcal{E} \llbracket \text{letRgn } [s_1] [\tau_\alpha] e \rrbracket = & \\
& \text{let } f: T_\star \llbracket \forall \varsigma_2: \text{STACK}. \text{pf } (s_1 \leq \varsigma_2) \rightarrow \text{hnd } \varsigma_2 \rightarrow \text{rgn } \varsigma_2 \tau_\alpha \rrbracket = \mathcal{E} \llbracket e \rrbracket \text{ in} \\
& {}^U \lambda \text{stk}_1: \mathcal{S}_\star \llbracket s_1 \rrbracket. \text{let pack}(\varrho: \text{RGN}, \langle \text{cap}, \text{hnd} \rangle) = \text{newrgn } {}^L \langle \rangle \text{ in} \\
& \quad \text{let } id = {}^U \lambda \text{stk}: {}^L (\mathcal{S}_\star \llbracket s_1 \rrbracket \otimes {}^L \text{cap } \varrho). \text{stk} \text{ in} \\
& \quad \text{let } \text{ppf} = {}^U \text{pack}({}^L (\text{cap } \varrho): \star, {}^U \langle id, id \rangle) \text{ in} \\
& \quad \text{let } \text{phnd} = {}^U \text{pack}(\varrho: \text{RGN}, {}^U \langle {}^U \text{pack}(\mathcal{S}_\star \llbracket s_1 \rrbracket): \star, {}^U \langle id, id \rangle), \text{hnd} \rangle) \text{ in} \\
& \quad \text{let } \text{stk}_2 = {}^L \langle \text{stk}_1, \text{cap} \rangle \text{ in} \\
& \quad \text{let } \langle \text{stk}_2, \text{res} \rangle = f [{}^L (\mathcal{S}_\star \llbracket s_1 \rrbracket \otimes {}^L (\text{cap } \varrho))] \text{ppf } \text{phnd } \text{stk}_2 \text{ in} \\
& \quad \text{let } \langle \text{stk}_1, \text{cap} \rangle = \text{stk}_2 \text{ in} \\
& \quad \text{let } \langle \rangle = \text{freergn} [\varrho] {}^L \langle \text{cap}, \text{hnd} \rangle \text{ in} \\
& \quad {}^L \langle \text{stk}_1, \text{res} \rangle
\end{aligned}$$

We can see the isomorphisms in action in the translation of `coerceRgn`:

$$\begin{aligned}
\mathcal{E} \llbracket \text{coerceRgn } [s_1] [s_2] [\tau_\alpha] e_1 e_2 \rrbracket = & \\
& \text{let } \text{ppf}: T_\star \llbracket \text{pf } (s_1 \leq s_2) \rrbracket = \mathcal{E} \llbracket e_1 \rrbracket \text{ in} \\
& \text{let } f: T_\star \llbracket \text{rgn } s_1 \tau_\alpha \rrbracket = \mathcal{E} \llbracket e_2 \rrbracket \text{ in} \\
& {}^U \lambda \text{stk}_2: \mathcal{S}_\star \llbracket s_2 \rrbracket. \text{let pack}(\beta: \star, \langle \text{spl}, \text{cmb} \rangle) = \text{ppf} \text{ in} \\
& \quad \text{let } \langle \text{stk}_1, \text{stk}_\beta \rangle = \text{spl } \text{stk}_2 \text{ in} \\
& \quad \text{let } \langle \text{stk}_1, \text{res} \rangle = f \text{stk}_1 \text{ in} \\
& \quad \text{let } \text{stk}_2 = \text{cmb } {}^L \langle \text{stk}_1, \text{stk}_\beta \rangle \text{ in} \\
& \quad {}^L \langle \text{stk}_2, \text{res} \rangle
\end{aligned}$$

Note how the the stack “slack” stk_β is split out and then combined in, bracketing the execution of the $\text{rgn } s_1 \tau_\alpha$ computation.

As a final example, we can see an “empty” stack (represented by a ${}^L \mathbf{1}_\otimes$ value) being provided as the initial stack in the translation of `runRgn`:

$$\begin{aligned}
\mathcal{E} \llbracket \text{runRgn } [\tau_\alpha] e \rrbracket = & \\
& \text{let } f: T_\star \llbracket \forall \varsigma: \text{STACK}. \text{rgn } \varsigma \tau_\alpha \rrbracket = \mathcal{E} \llbracket e \rrbracket \text{ in} \\
& \text{let } \langle \langle \rangle, \text{res} \rangle = f [{}^L \mathbf{1}_\otimes] {}^L \langle \rangle \text{ in } \text{res}
\end{aligned}$$

The translations of the remaining `rgn` monad primitives are given in Figure 5. We strongly believe, but have not mechanically verified, that the translation is type preserving.

5 Extensions

The primary advantage of working at the target level is that we can expose the capabilities for regions as first-class objects instead of indirectly manipulating a stack of regions. In turn, this allows us to avoid the last-in-first-out lifetimes dictated by a lexically-scoped `letRgn`. For example, we can now explain the semantics for Cyclone’s *dynamic regions* and *unique pointers* using the concepts in the target language.

Dynamic Regions. In Cyclone, a dynamic region r is represented by a *key* (`key r`) which is treated linearly by the type system. At the target level, a key can be represented by a pair of the region’s capability and its handle:

$$\text{key } r = {}^L ({}^L \text{cap } r \otimes {}^U \text{hnd } r)$$

Then creating a new key is accomplished by calling `newrgn`, and destroying the key is accomplished by calling `freergn`.

$$\begin{aligned}
\mathcal{E} \llbracket \text{new } [s] [\tau_\alpha] e_1 e_2 \rrbracket &= \\
&\text{let } phnd : T_\star \llbracket \text{hnd } s \rrbracket = \mathcal{E} \llbracket e_1 \rrbracket \text{ in} \\
&\text{let } x : T_\star \llbracket \tau_\alpha \rrbracket = \mathcal{E} \llbracket e_2 \rrbracket \text{ in} \\
&\quad \text{let } \lambda stk : S_\star \llbracket s \rrbracket . \text{let pack}(\varrho : \text{RGN}, \langle \text{pack}(\beta : \star, \langle prj, inj \rangle), hnd \rangle) = phnd \text{ in} \\
&\quad \quad \text{let } \langle stk_\beta, cap \rangle = prj \ stk \text{ in} \\
&\quad \quad \text{let } \langle cap, ref \rangle = \text{new } [\varrho] [\mathcal{T}_\star \llbracket \tau_\alpha \rrbracket] \text{ }^L \langle cap, hnd, x \rangle \text{ in} \\
&\quad \quad \text{let } pref = \text{pack}(\varrho : \text{RGN}, \text{ }^U \langle \text{pack}(\beta : \star, \text{ }^U \langle prj, inj \rangle), ref \rangle) \text{ in} \\
&\quad \quad \text{let } stk = inj \text{ }^L \langle stk_\beta, cap \rangle \text{ in} \\
&\quad \quad \text{ }^L \langle stk, pref \rangle \\
\\
\mathcal{E} \llbracket \text{read } [s] [\tau_\alpha] e \rrbracket &= \\
&\text{let } pref : T_\star \llbracket \text{ref } s \tau_\alpha \rrbracket = \mathcal{E} \llbracket e \rrbracket \text{ in} \\
&\quad \text{let } \lambda stk : S_\star \llbracket s \rrbracket . \text{let pack}(\varrho : \text{RGN}, \langle \text{pack}(\beta : \star, \langle prj, inj \rangle), ref \rangle) = pref \text{ in} \\
&\quad \quad \text{let } \langle stk_\beta, cap \rangle = prj \ stk \text{ in} \\
&\quad \quad \text{let } \langle cap, res \rangle = \text{read } [\varrho] [\mathcal{T}_\star \llbracket \tau_\alpha \rrbracket] \text{ }^L \langle cap, ref \rangle \text{ in} \\
&\quad \quad \text{let } stk = inj \text{ }^L \langle stk_\beta, cap \rangle \text{ in} \\
&\quad \quad \text{ }^L \langle stk, res \rangle \\
\\
\mathcal{E} \llbracket \text{write } [s] [\tau_\alpha] e_1 e_2 \rrbracket &= \\
&\text{let } pref : T_\star \llbracket \text{ref } s \tau_\alpha \rrbracket = \mathcal{E} \llbracket e_1 \rrbracket \text{ in} \\
&\text{let } x : T_\star \llbracket \tau_\alpha \rrbracket = \mathcal{E} \llbracket e_2 \rrbracket \text{ in} \\
&\quad \text{let } \lambda stk : S_\star \llbracket s \rrbracket . \text{let pack}(\varrho : \text{RGN}, \langle \text{pack}(\beta : \star, \langle prj, inj \rangle), ref \rangle) = pref \text{ in} \\
&\quad \quad \text{let } \langle stk_\beta, cap \rangle = prj \ stk \text{ in} \\
&\quad \quad \text{let } \langle cap, res \rangle = \text{write } [\varrho] [\mathcal{T}_\star \llbracket \tau_\alpha \rrbracket] \text{ }^L \langle cap, ref, x \rangle \text{ in} \\
&\quad \quad \text{let } stk = inj \text{ }^L \langle stk_\beta, cap \rangle \text{ in} \\
&\quad \quad \text{ }^L \langle stk, res \rangle \\
\\
\mathcal{E} \llbracket \text{reflSub } [s] \rrbracket &= \\
&\text{let } spl = \text{ }^U \lambda stk : S_\star \llbracket s \rrbracket . \text{let } su = \text{ }^L \langle stk, \text{ }^L \langle \rangle \rangle \text{ in } su \text{ in} \\
&\text{let } cmb = \text{ }^U \lambda su : \text{ }^L (S_\star \llbracket s \rrbracket \otimes \text{ }^L \mathbf{1}_\otimes) . \text{let } \langle stk, \langle \rangle \rangle = su \text{ in } stk \text{ in} \\
&\quad \text{ }^U \text{pack}(\text{ }^L \mathbf{1}_\otimes : \star, \text{ }^U \langle spl, cmb \rangle) \\
\\
\mathcal{E} \llbracket \text{transSub } [s_1] [s_2] [s_3] e_{1 \rightsquigarrow 2} e_{2 \rightsquigarrow 3} \rrbracket &= \\
&\text{let } ppf_{1 \rightsquigarrow 2} : T_\star \llbracket \text{pf } (s_1 \leq s_2) \rrbracket = \mathcal{E} \llbracket e_{1 \rightsquigarrow 2} \rrbracket \text{ in} \\
&\text{let } ppf_{2 \rightsquigarrow 3} : T_\star \llbracket \text{pf } (s_2 \leq s_3) \rrbracket = \mathcal{E} \llbracket e_{2 \rightsquigarrow 3} \rrbracket \text{ in} \\
&\text{let pack}(\alpha : \star, \langle spl_{2 \rightsquigarrow 1 \otimes \alpha}, cmp_{1 \otimes \alpha \rightsquigarrow 2} \rangle) = ppf_{1 \rightsquigarrow 2} \text{ in} \\
&\text{let pack}(\beta : \star, \langle spl_{3 \rightsquigarrow 2 \otimes \beta}, cmp_{2 \otimes \beta \rightsquigarrow 3} \rangle) = ppf_{2 \rightsquigarrow 3} \text{ in} \\
&\text{let } spl = \text{ }^U \lambda stk_3 : S_\star \llbracket s_3 \rrbracket . \text{let } \langle stk_2, stk_\beta \rangle = spl_{3 \rightsquigarrow 2 \otimes \beta} \ stk_3 \text{ in} \\
&\quad \quad \text{let } \langle stk_1, stk_\alpha \rangle = spl_{2 \rightsquigarrow 1 \otimes \alpha} \ stk_2 \text{ in} \\
&\quad \quad \text{let } sss = \text{ }^L \langle stk_1, \text{ }^L \langle stk_\beta, stk_\alpha \rangle \rangle \text{ in} \\
&\quad \quad \quad sss \quad \text{in} \\
&\text{let } cmb = \text{ }^U \lambda sss : \text{ }^L (S_\star \llbracket s_1 \rrbracket \otimes \text{ }^L (\beta \otimes \alpha)) . \text{let } \langle stk_1, \langle stk_\beta, stk_\alpha \rangle \rangle = sss \text{ in} \\
&\quad \quad \text{let } stk_2 = cmb_{1 \otimes \alpha \rightsquigarrow 2} \text{ }^L \langle stk_1, stk_\alpha \rangle \text{ in} \\
&\quad \quad \text{let } stk_3 = cmb_{2 \otimes \beta \rightsquigarrow 3} \text{ }^L \langle stk_2, stk_\beta \rangle \text{ in} \\
&\quad \quad \quad stk_3 \quad \text{in} \\
&\quad \text{ }^U \text{pack}(\text{ }^L (\beta \otimes \alpha) : \star, \text{ }^U \langle spl, cmb \rangle)
\end{aligned}$$

Fig. 5. Translation of rgn Monad Operations

To access a value allocated in a dynamic region, or to allocate a value in a dynamic region, Cyclone requires that the region be *opened* by presenting its key. The `openDRgn` is similar to a `letRgn` in that it conceptually pushes the dynamic region onto the stack of regions, executes the body, and then pops the region off the stack. During execution of the `openDRgn`'s body, the key becomes inaccessible, ensuring that the region cannot be deallocated. At the end of the `openDRgn` scope, the key is given back. The programmer is then able to destroy the region or later re-open it.

The `openDRgn` primitive can be implemented as a higher-order function with a signature like this (eliding the L and U qualifiers, and using source-level `rgn` to abbreviate the store passing translation):

$$\text{openDRgn} : \forall \varrho, \varsigma, \alpha. \text{key } \varrho \multimap (\text{hnd } \varrho \multimap \text{rgn } (\varsigma \otimes \text{cap } \varrho) \alpha) \multimap \text{rgn } \varsigma (\alpha \otimes \text{key } \varrho)$$

The function takes the key for ϱ and a computation, which, when given the handle for ϱ , expects to run on a stack of the form $\varsigma \otimes \text{cap } \varrho$ for some ς . Once applied, `openDRgn` returns a computation, which, when run on a stack ς , opens up the key to get the capability and handle, pushes the capability for ϱ on the stack, passes the handle to the computation and runs it in the extended stack to produce an α value. Then, it pops the capability, and returns a (linear) pair of the result and the re-packaged key. (We leave the definition of `openDRgn` as an exercise for the reader.)

Furthermore, since keys are first-class objects, they can be placed in data structures. For example, in our space-conscious web server [3], we use a list of dynamic regions, each of which holds data corresponding to a particular connection. When we receive data from a connection, we find the corresponding key, open it up, and then place the data in the region. When a connection is terminated, we pull the corresponding key out of the queue, perform `freeDRgn` on it, and thus deallocate all of the data associated with the connection. The price paid for this flexibility is that the list of keys must be treated linearly to avoid creating multiple aliases to the keys.

Unique Pointers. Cyclone's unique pointers are anonymous dynamic regions, without the handle. Like the keys of dynamic regions, they can be destroyed at any time and the type system treats them linearly. At the target level, a unique pointer to a τ object can be represented as a term with type:

$${}^L\exists \varrho. {}^L(\text{cap } \varrho \otimes {}^U({}^L\text{cap } \varrho \multimap {}^U\mathbf{1}_\otimes) \otimes {}^U(\text{ref } \varrho \tau))$$

Note that the actual reference is unrestricted, whereas the capability is linear; the handle is not available for further allocations, but is caught up in the function closure, which may be applied to free the region. This encoding allows us to “open” a unique pointer, just as we do dynamic regions, and for a limited scope, freely access (and duplicate) the underlying reference. Of course, during the scope of the open, we temporarily lose the capability to deallocate the object, but regain the capability upon exit from the scope.

In practice, the ability to open dynamic regions and unique pointers has proven crucial for integrating these facilities into the language. They make it

relatively easy to access a data structure and mitigate some of the pain of threading linear resources through the program. Furthermore, they make it possible to write re-usable libraries that manipulate data allocated in lexically-scoped regions, dynamic regions, or as unique objects.

Phase-Splitting. In our target language, we represented capabilities and proof witnesses as explicit terms. But we have also crafted the language and translation so that these values are never actually needed at run-time. For instance, our witnesses only manipulate (products of) capabilities, which are themselves operationally irrelevant. These objects are only used to make the desired safety properties easy to check statically. So in principle, we should be able to erase the capabilities and witnesses before running a program.

To realize this goal, we should introduce a phase distinction via another modality, where we treat capabilities and proof witnesses as static objects, and all other terms as dynamic. The modality would demand that, as usual, static computations cannot depend upon dynamic values. Furthermore, we must be sure that witness functions (i.e., proof objects) are in fact total, effect-free functions so that they and their applications to capabilities may be safely erased. This sort of phase-splitting is effectively used in other settings that mix programming languages and logics, such as Xi *et al.*'s Applied Type System [12] and Sheard's Omega [13]. Perhaps the most promising approach is suggested by Mandelbaum, Walker, and Harper's work [14], where they developed a two-level language for reasoning about effectful programs.

The primary reason we did not introduce phase splitting here is that it complicates the translation and the target language, and thus obscures what is actually a relatively simple and straightforward encoding. A secondary reason is that, as demonstrated by the cited work above, there are many domains that would benefit from a general solution to the problem of type relevant, but operationally irrelevant, values.

6 Related Work and Open Issues

There has been much prior work aimed at relaxing the stack discipline imposed on region lifetimes by the Tofte-Talpin (TT) approach. The ML Kit [15] uses a storage-mode analysis to determine when it is safe to deallocate data in a region (known as region resetting) prior to the deallocation of the region itself. The safety of the storage-mode analysis has not been established formally.

Aiken *et al.* [16] eliminate the requirement that region allocation and deallocation should coincide with the beginning and end of the scope of region variables introduced by the `letregion` construct. They use a *late allocation/early deallocation* approach that delays the allocation of a region until just before its first access, and deallocates the region just after its last access. We believe that the results of their analysis can be encoded explicitly in our target language.

Unlike the previous two approaches which build on TT, Henglein *et al.* [17] present a region system that (like ours) replaces the `letregion` primitive with explicit commands to create and free a region. To ensure safety, they use a Hoare-logic-based region type system and consequently have no support for higher-order

functions. While they provide an inference algorithm to annotate programs with region manipulation commands, we intend for our system to serve as a target language for programs annotated using TT region inference, or those written in languages like Cyclone. The Calculus of Capabilities [18] is also intended as a target for TT-annotated programs, but unlike λ^{rgnUL} , it is defined in terms of a continuation-passing style language and does not support first-class regions.

The region system presented by Walker and Watkins [19] is perhaps the most closely related work. Like our target, they require a linear capability to be presented upon each access to a region. However, they provide a primitive, similar to **letregion**, that allows a capability to be temporarily treated as unrestricted for convenience’s sake. We have shown that no such primitive is needed. Rather, we use a combination of monadic encapsulation (to thread capabilities) coupled with *unrestricted witnesses* to achieve the same flexibility. In particular, our open construct for dynamic regions (and unique pointers) achieves the same effect as the Walker-Watkin’s primitive.

A related body of work has used regions as a low-level primitive on which to build type-safe garbage collectors [20, 21, 22]. Each of these approaches requires non-lexical regions, since, in a copying collector, the from- and to-spaces have non-nested lifetimes. Hawblitzel *et al.* [22] introduce a very low-level language in which they begin with a single linear array of words, construct lists and arrays out of the basic linear memory primitives, introduce *type sequences* for building regions of nonlinear data. Such a foundational approach is admirable, but there is a large semantic gap between a high-level language and such a target. Hence, λ^{rgnUL} serves as a useful intermediate point, and we may envision further translation from λ^{rgnUL} to such a low-level language.

The Vault language [23, 24] includes many of the features described in our target, including linear capabilities for accessing resources and a mechanism, called adoption, for temporarily transferring ownership of a capability to another capability, for a limited scope. But Vault also includes support for strong (i.e., type-changing) updates on linear resources, as well as features for temporarily treating an unrestricted resource as if it were linear. On the other hand, to the best of our knowledge, there exists no formal model that justifies the soundness of all of these mechanisms. We believe that it may be possible to combine λ^{rgnUL} with our previous work on strong updates [25, 26] to achieve this.

References

1. Cyclone, version 0.9. (2005) <http://www.eecs.harvard.edu/~greg/cyclone/>.
2. Tofte, M., Talpin, J.P.: Region-based memory management. *Information and Computation* **132**(2) (1997) 109–176
3. Hicks, M., Morrisett, G., Grossman, D., Jim, T.: Experience with safe manual memory-management in Cyclone. In: *Proc. International Symposium on Memory Management*. (2004) 73–84
4. Fluet, M., Wang, D.: Implementation and performance evaluation of a safe runtime system in Cyclone. In: *Proc. SPACE Workshop*. (2004)

5. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in Cyclone. In: *Proc. Programming Language Design and Implementation*. (2002) 282–293
6. Fluett, M., Morrisett, G.: Monadic regions. In: *Proc. International Conference on Functional Programming*. (2004) 103–114
7. Launchbury, J., Peyton Jones, S.: State in Haskell. *Lisp and Symbolic Computation* **8**(4) (1995) 293–341
8. Ahmed, A., Fluett, M., Morrisett, G.: A step-indexed model of substructural state. In: *Proc. International Conference on Functional Programming*. (2005) 78–91
9. Walker, D.: Substructural type systems. In Pierce, B., ed.: *Advanced Topics in Types and Programming Languages*. MIT Press, Cambridge, MA (2005) 3–43
10. Pfenning, F., Schürmann, C.: Twelf – a meta-logic framework for deductive systems. In: *Proc. Conference on Automated Deduction*. (1999) 202–206
11. Schürmann, C., Pfenning, F.: A coverage checking algorithm for LF. In: *Proc. Theorem Proving in Higher Order Logics*. (2003) 120–135 LNCS 2758.
12. Chen, C., Xi, H.: Combining programming with theorem proving. In: *Proc. International Conference on Functional Programming*. (2005) 66–77
13. Sheard, T., Pasalic, E.: Meta-programming with built-in type equality (extended abstract). In: *International Workshop on Logical Frameworks and Meta-Languages*. (2004)
14. Mandelbaum, Y., Walker, D., Harper, R.: An effective theory of type refinements. In: *Proc. International Conference on Functional Programming*. (2003) 213–225
15. Tofte, M., Birkedal, L., Elmsan, M., Hallenberg, N., Olesen, T.H., Sestoft, P.: Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen (2002)
16. Aiken, A., Fähndrich, M., Levien, R.: Better static memory management: Improving region-based analysis of higher-order languages. In: *Proc. Programming Language Design and Implementation*. (1995) 174–185
17. Henglein, F., Makhholm, H., Niss, H.: A direct approach to control-flow sensitive region-based memory management. In: *Proc. Principles and Practice of Declarative Programming*. (2001) 175–186
18. Walker, D., Crary, K., Morrisett, G.: Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems* **24**(4) (2000) 701–771
19. Walker, D., Watkins, K.: On regions and linear types. In: *Proc. International Conference on Functional Programming*. (2001) 181–192
20. Wang, D., Appel, A.: Type-preserving garbage collectors. In: *Proc. Principles of Programming Languages*. (2001) 166–178
21. Monnier, S., Saha, B., Shao, Z.: Principled scavenging. In: *Proc. Programming Language Design and Implementation*. (2001) 81–91
22. Hawblitzel, C., Wei, E., Huang, H., Krupski, E., Wittie, L.: Low-level linear memory management. In: *Proc. SPACE Workshop*. (2004)
23. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: *Proc. Programming Language Design and Implementation*. (2001) 59–69
24. Fähndrich, M., DeLine, R.: Adoption and focus: Practical linear types for imperative programming. In: *Proc. Programming Language Design and Implementation*. (2002) 13–24
25. Morrisett, G., Ahmed, A., Fluett, M.: \mathbf{L}^3 : A linear language with locations. In: *Proc. Typed Lambda Calculi and Applications*. (2005) 293–307
26. Ahmed, A., Fluett, M., Morrisett, G.: \mathbf{L}^3 : A linear language with locations. Technical Report TR-24-04, Harvard University (2004)

Type-Based Amortised Heap-Space Analysis

Martin Hofmann¹ and Steffen Jost²

¹ LMU München, Institut für Informatik

² University of St Andrews, School of Computer Science

Abstract. We present a type system for a compile-time analysis of heap-space requirements of Java style object-oriented programs with explicit deallocation.

Our system is based on an amortised complexity analysis: the data is arbitrarily assigned a potential related to its size and layout; allocations must be “payed for” from this potential. The potential of each input then furnishes an upper bound on the heap space usage for the computation on this input.

We successfully treat inheritance, downcast, update and aliasing. Example applications for the analysis include destination-passing style and doubly-linked lists.

Type inference is explicitly not included; the contribution lies in the system itself and the nontrivial soundness theorem. This *extended abstract* elides most technical lemmas and proofs, even nontrivial ones, due to space limitations. A full version is available at the authors’ web pages.

1 Introduction

Consider a Java-like class-based object-oriented language without garbage collection, but with explicit deallocation in the style of C’s `free()`. Such programs may be evaluated by maintaining a set of free memory units, the freelist. Upon object creation a number of heap units required to store the object is taken from the 3 provided it contains enough units; each deallocated heap unit is returned to the freelist. An attempt to create a new object with an insufficient freelist causes unsuccessful abortion of the program. This also happens upon attempts to access a deallocated object via a stale pointer.

It is now natural to ask what initial size the freelist must have so that a given program may be executed without causing unsuccessful abortion due to penury of memory. If we know such a bound on the initial freelist size we can then execute our program within a fixed amount of memory which can be useful in situations where memory is a scarce resource like embedded controllers or SIM cards. It may also be useful to calculate such bounds for individual parts of a program so that several applications can be run simultaneously even if their maximum memory needs exceed the available capacity [6].

Typically, the required initial freelist size will depend on the data, for example the size of some initial data structure, e.g., a phone book or an HTML document.

We therefore seek to determine an upper bound on the required freelist size as a function of the input size. We propose to approach this input dependency by a type-based version of amortised analysis in the sense of Tarjan [17].

Amortised Analysis. In amortised analysis data structure(s) are assigned an arbitrary nonnegative number, the *potential*. The amortised cost of an operation is its total cost (time or space) plus the difference in potential before and after the operation. The sum of the amortised costs plus the potential of the initial data structure then bounds (from above) the actual cost of a sequence of operations. If the potential is cleverly chosen then the amortised cost of individual operations is zero or a constant even when their actual cost is difficult to determine. The simplest example is an implementation of a queue using two stacks A and B . Enqueuing is performed on A , dequeuing is performed on B unless B is empty in which case the whole contents of A are moved to B prior to dequeuing. Thus, dequeuing sometimes takes time proportional to the size of A . If we decree that the size of A is the potential of the data then enqueuing has an amortised cost of 2 (one for the actual cost, one for the increase in potential). Dequeuing on the other hand has an amortised cost of 1 since the cost of moving A over to (the empty stack) B cancels out against the decrease in potential. Thus, the actual cost of a sequence of operations is bounded by the initial size of A plus twice the number of enqueues plus the number of dequeues. In this case, one can also see this directly by observing that each element is moved exactly three times: once into A , once from A to B , once out of B .

Type-Based Potential. In the above queue example, both stacks have the same type, but each element of A contributes 1 to the overall potential, whereas each element of B contributes a potential of 0. We recorded this information within the type by adding a number to each type constructor in our previous work [9]. However, object-oriented languages require a more complex approach due to aliasing and inheritance: Where in a purely functional setting a *refined type* might consist of a simple type together with a number a refined (class) type will consist of a number together with refined types for the attributes and methods. In order to break the recursive nature of this requirement we resort to explicit names for refined types as is common practice in Java (though not in OCaml): we introduce a set of names, the *views*. A *view* on an object shall determine its contribution to the potential. This is formally described in Section 3, but we shall convey a good intuition here. A refined type then consists of a class C and a view r and is written C^r . We sometimes conveniently use a refined type where only a class or a view is needed.

The fact that views are in some sense orthogonal to class types caters for typecasting. If, e.g., x has refined type C^r , then $(D)x$ will have refined type D^r .¹

¹ Peter Thiemann, Freiburg, independently and simultaneously used a similar approach in as yet unpublished work on a generic type-based analysis for Java. His main application is conformance of XML-document generators to standards and his treatment of aliasing is different from ours.

The meaning of views is given by three maps \Diamond defining potentials, A defining views of attributes, and M defining refined method types. More precisely, $\Diamond : \text{Class} \times \text{View} \rightarrow \mathbb{Q}^+$ assigns each class its potential according to the employed view. Next, $A : \text{Class} \times \text{View} \times \text{Field} \rightarrow \text{View} \times \text{View}$ determines the refined types of the fields. A different view may apply according to whether a field is read from (get-view) or written to (set-view), hence the codomain $\text{View} \times \text{View}$. Subtyping of refined types is behavioural and covariant in the get-view and contravariant in the set-view.

Finally, $M : \text{Class} \times \text{View} \times \text{Method} \rightarrow \mathcal{P}(\text{Views of Arguments} \rightarrow \text{Effect} \times \text{View of Result})$ assigns refined types and effects to methods. The effect is a pair of numbers representing the potential consumed before and released after method invocation. We allow polymorphism in the sense that, as implied by the powerset symbol \mathcal{P} one method may have more than one (or no) refined typing.

One and the same runtime object can have several refined types at once, since it can be regarded through different views at the same time. In fact, each *access path* leading from the current scope via field access to an object will determine its individual view on the object by the repeated application of A . The overall potential of a runtime configuration is the (possibly infinite) sum over all access paths in scope that lead to an actual object. Thus, if an object has several access paths leading to it (aliasing) it may make several contributions to the total potential. Our typesystem has an explicit contraction rule: If a variable is used more often, the associated potential is split by assigning different views to each use. The potential also depends on the dynamic class types of each object. However, our runtime model is the standard one which does not include any view/potential related information.

Our main contribution is the proof that the total potential plus the heap used never increases during execution. In other words, any object creation must be paid for from the potential in scope and the potential of the initial configuration furnishes an upper bound on the total heap consumption.

In this way, we can model data-dependent memory usage without manipulating functions and recurrences, as is the case in approaches based on sized types and also without any alteration to the runtime model.

We will now describe our approach in more detail using three suggestive examples: a copying function for lists, imperative append in destination passing style, and doubly-linked lists. These examples show many of the salient features of our methods: heap usage proportional to input size (the copying example), correct accounting for aliasing (destination passing style), circular data (doubly-linked lists). Especially the last example seems to go beyond the scope of current methods based on sized types or similar.

Example: Copying singly-linked lists in an object-oriented style:

```
abstract class List { abstract List copy(); }
class Nil extends List { List copy() { return this; }}
class Cons extends List { int elem; List next;
    List copy() { Cons res = new Cons(); res.elem = this.elem;
                res.next = this.next.copy(); return res; }}
```

It is clear that the memory consumption of a call $\mathbf{x}.\text{copy}()$ will equal the length of the list \mathbf{x} . To calculate this formally we construct a view \mathbf{a} which assigns to **List** itself the potential 0, to **Nil** the potential 0 and to **Cons** the potential 1. Another view is needed to describe the result of $\text{copy}()$ for otherwise we could repeatedly copy lists without paying for it. Thus, we introduce another view \mathbf{b} that assigns potential 0 to all classes. The complete specification of the two views is shown here, together with other views used later:

$\Diamond(\cdot)$	\mathbf{a}	\mathbf{b}	\mathbf{c}	\mathbf{d}	\mathbf{n}
List	0	0	0	0	0
Nil	0	0	0	0	0
Cons	1	0	0	1	0

	$\text{Cons}^{\mathbf{a}}$	$\text{Cons}^{\mathbf{b}}$	$\text{Cons}^{\mathbf{c}}$	$\text{Cons}^{\mathbf{d}}$	$\text{Cons}^{\mathbf{n}}$
$\text{A}^{\text{get}}(\cdot, \text{next})$	\mathbf{a}	\mathbf{b}	\mathbf{a}	\mathbf{n}	\mathbf{n}
$\text{A}^{\text{set}}(\cdot, \text{next})$	\mathbf{a}	\mathbf{b}	\mathbf{a}	\mathbf{a}	\mathbf{a}

(1.1)

$$\mathbf{M}(\{\text{List}^{\mathbf{a}}, \text{Cons}^{\mathbf{a}}, \text{Nil}^{\mathbf{a}}\}, \text{copy}) = () \xrightarrow{0/0} \mathbf{b}$$

The call $\mathbf{x}.\text{copy}()$ is well-typed and of type $\text{List}^{\mathbf{b}}$ if \mathbf{x} has refined type $\text{List}^{\mathbf{a}}$, $\text{Nil}^{\mathbf{a}}$ or $\text{Cons}^{\mathbf{a}}$. It is ill-typed if \mathbf{x} has refined type, e.g., $\text{List}^{\mathbf{b}}$. Its effect $0/0$ will not decrement the freelist beyond the amount implicit in the potential of \mathbf{x} (which equals in this case the length of the list pointed to by \mathbf{x}) and will not return anything to the freelist beyond the amount implicit in the potential of the result (which equals zero due to its refined type $\text{List}^{\mathbf{b}}$). Thus, the typing amounts to saying that the memory consumption of this call is equal to the length of \mathbf{x} .

Let us explain why the potential of \mathbf{x} indeed equals its length. Suppose for the sake of the example that \mathbf{x} points to a list of length 2. The potential is worked out as the sum over all access paths emanating from \mathbf{x} and not leading to null or being undefined. In this case, these access paths are $\mathbf{p}_1 = \mathbf{x}$, $\mathbf{p}_2 = \mathbf{x}.\text{next}$, $\mathbf{p}_3 = \mathbf{x}.\text{next}.\text{next}$. Each of these has a dynamic type: **Cons** for \mathbf{p}_1 and \mathbf{p}_2 ; **Nil** for \mathbf{p}_3 . Each of them also has a view worked out by chaining the view of \mathbf{x} along the get-views. Here it is view \mathbf{a} in each case. For each access paths we now look up the potential annotation of its dynamic type under its view. It equals 1 in case of \mathbf{p}_1 and \mathbf{p}_2 given $\Diamond(\text{List}^{\mathbf{a}}) = 1$ and 0 for \mathbf{p}_3 by $\Diamond(\text{Nil}^{\mathbf{a}}) = 0$, yielding a sum of 2. Notice that if the very same list had been accessed via a variable \mathbf{y} of type $\text{List}^{\mathbf{b}}$ the potential ascribed would have been 0.

The Typing Judgement. The type system allows us to derive assertions of the form $\Gamma \vdash_{m'}^m e : C^r$ where e is an expression or program phrase, C is a Java class, r is a view (so C^r is a refined type). Γ maps variables occurring in e to refined types; we often write Γ_x instead of $\Gamma(x)$. Finally m, m' are nonnegative numbers. The meaning of such a judgement is as follows. If e terminates successfully in some environment η and heap σ with unbounded memory resources available then it will also terminate successfully with a bounded freelist of size at least m plus the potential ascribed to η, σ with respect to the typings in Γ . Furthermore, the freelist size upon termination will be at least m' plus the potential of the result with respect to the view r .

For the typing of `copy()` to be accepted we must derive the judgements

$$\text{this:Nil}^c \vdash_0^0 e_{\text{Nil}} : \text{List}^b \quad \text{this:Cons}^c \vdash_0^1 e_{\text{Cons}} : \text{List}^b \quad (1.2)$$

where e_{Nil} and e_{Cons} are the bodies of `copy` in classes `Nil` and `Cons`, respectively. View c is basically the view a with the toplevel potential 1 stripped off. In exchange we get access to this toplevel potential in the form of the superscript 1 of the “turnstile”. This weaker typing of `this` allows us to read attributes from `this` as if its typing were List^a but it precludes the use of any toplevel potential which is right for otherwise there would be an unjustified duplication of potential.

Formally, this “stripping-off” is achieved through the coinductive definition of a relation $\forall(r|\mathcal{D})$ between views r and multisets of views \mathcal{D} which asserts that a variable of view r may be used multiple times provided the different occurrences are given the views in \mathcal{D} and it is this relation that appears as a side condition to the typing rule for methods. Entry of a method body is the only point where potential becomes available for use, but it can be used anywhere inside the method’s body and even passed on to further method calls.

In particular for the views listed in (1.1), we have $\forall(a|\{d, c\})$, $\forall(b|\{b, b, \dots\})$, and $\forall(a|\{a, n, n, \dots\})$, but neither $\forall(a|\{a, a\})$ (because $1 + 1 \neq 1$) nor $\forall(c|\{c, c\})$ (because the get-view of `next` in c is a), nor $\forall(a|\{a, b\})$ (because the set-view of `next` in b is not a , but the set-view has to be preserved upon sharing).

Typing “Copy”. Let us now see how we can derive the required typings in (1.2). The typing of e_{Cons} works as follows. The creation of a new object of class `Cons`^b incurs a cost of 1 (zero potential plus one physical heap unit – note that the physical cost of object creation can be chosen arbitrarily for each class to suit the applicable memory model). Thus, it remains to justify

```

this:Consc, res:Consb  $\vdash_0^0$ 
  res.elem=this.elem; res.next=this.next.copy(); return res;

```

The threefold use of `res` in this term relies on the sharing relation $\forall(b|\{b, b, b\})$. Let us now consider the assignments in order. The first assignment being of scalar type is trivial. The set-view of `res.next` is b but so is the view of `this.next.copy()` thus justifying the second assignment. The view of `res` equals the announced return view b so we are done.

The body e_{Nil} of `copy()` in `Nil` is simply `return this;`. The type of `this` is Nil^c which is unfortunately not a subtype of the required type List^b , which we ignore here for the sake of simplicity. A proper way to avoid this problem would be to have non-unique `Nil` objects for each list, which makes sense if the `Nil`-node has a function. Otherwise one could abandon the `Nil` class and use a null pointer instead, which would somehow defeat our example. A third solution would be to include mechanisms for static objects in our theory.

Example: Destination Passing Style. We augment `List` by two methods:

```
abstract void appAux(List y, Cons dest);
List append(List y) { Cons dest = new Cons(); this.appAux(y,dest);
                     List result = dest.next; free(dest); return result; }
```

The call `this.appAux(y,dest)` to be implemented in the subclasses `Nil` and `Cons` should imperatively append `y` to `this` and place the result into the `dest.next`. The point is that `appAux` admits a tail-recursive implementation which corresponds to a while-loop.

```
/* In Nil */      void appAux(List y, Cons dest){ dest.next <- y; }
/* In Cons */    void appAux(List y, Cons dest){ dest.next <- this;
                                                         this.next.appAux(y, this); }
```

We propose the following refined typings for these newly introduced methods:

$$\begin{aligned} M(\text{List}^a, \text{append}) &= \text{List}^a \xrightarrow{1/1} \text{List}^a \\ M(\text{List}^a, \text{appAux}) &= (\text{List}^a, \text{Cons}^n) \xrightarrow{0/0} \text{void} \end{aligned}$$

We focus here on the most interesting judgement

`this:Lista, dest:Listn $\xrightarrow{0/0}$ dest.next <- this; this.next.appAux(y, this):void`

Here we have decided not to glean any potential from `this` in the method body so that `this` is available as of type `Lista`. We split `this:Lista` using $\forall(a|\{a,n\})$ and `dest:Listn`. The set-view of `dest.next` is `a` coinciding with the view of `this` thus the assignment is justified. This example shows that the potential is correctly chained through the `appAux` method despite of heavy aliasing.

Example: Doubly-Linked Lists. Our final example illustrates doubly-linked lists which brings more aliasing and even circular data.

```
abstract class DList { }
class DNil extends DList{ }
class DCons extends DList{ Object elem; DList next; DList previous;
                           int getNext() { return this.next;}}
```

We would like to be able to implement methods `toList()` and `toDList()` which non-destructively transform singly-linked lists into doubly-linked ones and vice versa. To make this possible we need views on doubly-linked lists defined in such a way that the potential of a doubly-linked list is proportional to its length. This can be achieved as follows with two views `q` and `r`.

$\Diamond(\cdot)$	$ q r$		$DCons^q$	$DCons^r$	(1.3)
$DList$	$ 0 0$	$A^{get}(\cdot, next)$	q	r	
$DNil$	$ 0 0$	$A^{set}(\cdot, next)$	q	q	
		$A^{get}(\cdot, previous)$	r	r	
$DCons$	$ 1 0$	$A^{set}(\cdot, previous)$	r	r	

It is irrelevant what these views are at the other classes `Nil`, `Cons`, `List`.

The potential of a \mathbf{DList}^a equals its length, whereas the potential of a \mathbf{DList}^r is zero. The potential is defined as an infinite sum ranging over all access paths, i.e. $p \in \{\mathbf{next}, \mathbf{previous}\}^*$. However, due to the fact that field $\mathbf{previous}$ has view r and that in r even the \mathbf{next} attribute has view r , only access paths of the form \mathbf{next}^i for $i < \text{the length of the list}$ make a nonzero contribution.

It is now possible to include and justify in \mathbf{DList}^a a method that computes a singly-linked copy $M(\mathbf{DList}^a, \mathbf{toList}) = () \xrightarrow{1/0} \mathbf{List}^b$. The effect shows the cost of the additional object of type \mathbf{Nil}^b that is required. Similarly, a method $\mathbf{toList}()$ can be defined.

We remark that a circular singly-linked list can be constructed, with any fixed potential unrelated to its length, e.g. of type \mathbf{List}^b with an overall potential 0.

Related Work. A commonly found approach to bound memory usage is the use of sized types as initially proposed by Hughes and Pareto [10]. However, as pointed out by Vasconcelos [21], these systems have difficulties, e.g. with algorithms that divide and merge their input, such as the list splitting found in the popular quick-sort algorithm: the chosen pivot could be already minimal/maximal, hence each list originating from the splitting has its size bounded by $n - 1$. Merging these lists then results in an overall size of $2n - 1$ instead of n and thus to an exponential size for the resulting list of the quick-sort algorithm. Our amortised analysis does not suffer from this flaw, as the potential can be properly split and merged without this kind of loss of information.

A system employing sized types for an object-oriented language is presented by Rinard et al. [3]. Their system also depends upon a deallocation primitive like ours and in addition incorporates an alias control via usage aspects. We think it is fair to say that [3] bundles together known techniques into a single system to form an actual implementation that can deal with sizeable examples. Due to the lack of worked out examples in the paper it is difficult to compare exactly the strengths and weaknesses of loc. cit. and our approach. In any case, we feel that the topic is important and new enough to justify several competing approaches for some time until it will eventually be found out which one is better.

Another widespread approach is the use of a region based memory management as initially proposed by Tofte and Talpin [19] and realized in the ML Kit Compiler [18], which primarily aims at efficient memory usage rather than obtaining provable bounds. However, Berger et al. suggest in [1] that region based approaches suffer from increased memory consumption due to retarded deallocation if the programmer is unwilling to adjust his or her programming style to suit the region approach and they propose a more generalised version of regions.

Yet another way to obtain quantity bounds on memory usage is abstract interpretation and symbolic evaluation [20, 7, 8], which aim at identifying code portions which do not affect the overall memory usage of a program. An exhaustive search of all paths of computation is then performed on the remaining abstracted code parts. However, this exhaustive search might still lead to performance problems as reported in [20], which then leads to further abstraction jeopardising provable bounds in favour of estimates.

Finally, approaches based on formal specification and theorem proving are beginning to emerge [14]. From our own experience the current state of theorem provers does not suffice to automatically prove space assertions of the kind of examples we are interested in and able to treat. However, it may be that future progress in theorem proving will eventually make analyses like ours and indeed most other program analyses redundant.

2 Featherweight Java with Update

Our formal model of Java, FJEU, is an extension of Featherweight Java (FJ) [11] with attribute update, conditional and explicit deallocation. It is thus similar to Flatt et al. Classic Java [5].

We refer to our full paper for a formal definition of its syntax and semantics and content ourselves with an informal description here.

An FJEU program \mathcal{C} is a partial finite map from class names to class definitions, which we also refer to as *class table*. Each class table \mathcal{C} implies a subtyping relation $<:$ among the class names in the standard way by inheritance. Throughout the following sections we will consider a fixed (but arbitrary) class table \mathcal{C} for the ease of notation.

Each class consists of a super-class, a set of attributes (or fields) with their types, and a set of methods with their types and bodies. A method body is an expression in let normal form (nested expressions flattened out using a sequence of let-definitions). Classes have only one implicit constructor that initialises all attributes to a nil-value.

An *access path* is a list of attribute names, written $a.b.\dots.c = \mathbf{p}$. It is convenient to write $A(C, \mathbf{p})$ for the class type reached by following the access path \mathbf{p} , i.e. $A(C, \mathbf{p}.b) = A(A(C, \mathbf{p}), a)$. The typing judgement of FJEU takes the form $\Gamma \vdash e : C$ where Γ is a finite partial mapping from identifiers to class names. It is defined as a standard extension of the FJ typing rules and is omitted for lack of space.

For reasons of convenience, field update differs slightly and interdefinably from Java: the term $x.a \leftarrow y$ evaluates to the value of x after the update rather than y as in Java.

One new feature of FJEU is the presence of an explicit deallocation construct $\text{free}(x)$ that deallocates the object pointed to by x .

The typing judgement of FJEU takes the form $\Gamma \vdash e : C$ where Γ is a finite partial mapping from identifiers to class names. It is defined as a standard extension of the FJ typing rules. To illustrate FJEU we give here the corresponding version of the list copy example from Sect. 1:

```
List copy(){ let re1 = new Cons in
  let re2 = let elem = this.elem in re1.elem <- elem in
  let re3 = let next = this.next in let nres = next.copy() in
    re2.next <- nres in return re3; }
```

The dynamic semantics of FJEU is based on a global store (“heap”) mapping locations to object records as usual. We use the judgement $\eta, \sigma \vdash e \rightsquigarrow v, \tau$ shall mean that the expression e evaluates successfully to the value v , beginning with stack η , heap σ and ending with heap τ .

We also use the judgement $\eta, \sigma \frac{m}{m'} e \rightsquigarrow v, \tau$ to mean that the evaluation succeeds with an initial freelist of size at least m and leaves a freelist of size at least m' upon completion.

Both judgements are given as an inductive definition which increase and decrease resource counters m, m' as expected.

Unsuccessful evaluations such as null pointer access are not modelled explicitly in the semantics. For example, when e is `free(null)` then $\eta, \sigma \frac{m}{m'} e \rightsquigarrow v, \tau$ never holds. We assume that object creation `new` always returns a fresh location never seen before (and increments m by the size of the allocated object). Deallocation, on the other hand, overwrites an object record with a special value (`invalid`). In addition, the counter m' will be increased by the size of the deallocated object. We allow pointers to such disposed objects (“stale pointers”), however, any attempt to access a deallocated object via such a pointer leads to unsuccessful termination just as a null pointer access.

This abstract and essentially storeless [15, 2, 4, 13] semantics abstracts away from two important aspects of freelist based memory management: a) accidental “reanimation” of stale pointers through recycling of previously issued locations and b) fragmentation. Our strategy is to deal with those separately using known or orthogonal approaches.

To counter the problem with recycled locations, we can employ indirect pointers (symbolic handles) used by earlier implementations of the Sun JVM for the compacting garbage collector. Alternatively, we can statically reject programs that might access stale pointers using the alias types by Walker and Morrisett [22], or the bunched implication logic as practised by Ishtiaq and O’Hearn [12]. For those programs, our abstract semantics coincides with a concrete implementation using a freelist.

In order to deal with fragmentation in the freelist model one has several known possibilities that interact smoothly with the resource counting in the abstract semantics: allocating all objects with the same size or as linked lists of such blocks; maintaining several independent freelists for objects of various sizes (a slight change to the typing rules is then required to prevent trading objects of different sizes against each other), and, finally, compacting garbage collection. In the last case, we would simply run a compacting garbage collection as soon as the freelist does no longer contain a contiguous portion of the required size. Of course, the total memory requirement would be twice the one predicted by our analysis as usual with compacting garbage collection.

We find that the abstract operational semantics used here provides an adequate modular interface between the resource analysis and concrete implementation issues that have been and are being treated elsewhere.

3 Definition of RAJA

We now extend FJEU to an annotated version, RAJA, (Resource Aware JAva) as announced in the Introduction. A *RAJA program* is an annotation of an FJEU class table \mathcal{C} or more precisely a sextuple $\mathcal{R} = (\mathcal{C}, \mathcal{V}, \Diamond(\cdot), \mathbf{A}^{\text{get}}(\cdot, \cdot), \mathbf{A}^{\text{set}}(\cdot, \cdot), \mathbf{M}(\cdot, \cdot))$ specified as follows:

1. \mathcal{V} is a possibly infinite set of views.

For each class $C \in \text{dom}(\mathcal{C})$ and for each view $r \in \mathcal{V}$ the pair C^r is called a RAJA class (or *refined type*). If C^r is a RAJA type then we denote by $|C^r| = C$ the underlying FJEU type C and by $\langle\langle C^r \rangle\rangle = r$ its view. However, we allow ourselves to omit these projections if it is clear from the context whether the view or the FJEU class is required.

2. $\Diamond(\cdot)$ assigns to each RAJA class C^r a number $\Diamond(C^r) \in \mathbb{D}$.

This number will be used to define the potential of a heap configuration under a given static RAJA typing. For convenience, we extend the notation $\Diamond(\cdot)$ to possibly undefined meta-expressions by putting $\Diamond(\langle expr \rangle) = 0$ if $\langle expr \rangle$ is undefined.

3. $\mathbf{A}^{\text{get}}(\cdot, \cdot)$ and $\mathbf{A}^{\text{set}}(\cdot, \cdot)$ assign to each RAJA class C^r and attribute $a \in \mathbf{A}(C)$ two views $q = \mathbf{A}^{\text{get}}(C^r, a)$ and $s = \mathbf{A}^{\text{set}}(C^r, a)$.

The intention is that if $D = \mathbf{A}(C, a)$ is the FJEU type of attribute a in C then the RAJA type D^q will be the type of an access to a , whereas the (intendedly stronger) type D^s must be used when updating a . The stronger typing is needed since an update will possibly affect several aliases.

4. $\mathbf{M}(\cdot, \cdot)$ assigns to each RAJA class C^r and method $m \in \mathbf{M}(C)$ having method type $E_1, \dots, E_j \rightarrow E_0$ of arity j a j -ary polymorphic RAJA method type $\mathbf{M}(C^r, m)$.

A *j -ary polymorphic RAJA method type* is a (possibly empty or infinite) set of j -ary monomorphic RAJA method types. A *j -ary monomorphic RAJA method type* consists of $j + 1$ views and two numbers $p, q \in \mathbb{D}$, written $r_1, \dots, r_j \xrightarrow{p/q} r_0$.

The idea is that if m (of FJEU-type $E_1, \dots, E_j \rightarrow E_0$) has (among others) the monomorphic RAJA method type $r_1, \dots, r_j \xrightarrow{p/q} r_0$ then it may be called with arguments $v_1:E_1^{r_1}, \dots, v_j:E_j^{r_j}$, whose associated potential will be consumed, as well as an additional potential of p . Upon successful completion the return value will be of type $E_0^{r_0}$ hence carry an according potential. In addition to this a potential of another q units will be gained.

We note at this point that if a variable is to be used more than once, e.g., as an argument to a method, then the different occurrences must be given different types which are chosen such that the individual potentials assigned to each occurrence add up to the total potential available.

We sometimes write $E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{p/q} E_0^{r_0}$ to denote an FJEU method type combined with a corresponding monomorphic RAJA method type.

We will now define when such a RAJA-annotation of an FJEU class table is indeed valid; in particular this will require that each method body is typable with each of the monomorphic RAJA method types given in the annotation.

RAJA Subtyping Relation. We intend to define a preorder $r \sqsubseteq s$ on views as a largest fixpoint. If $\sqsubseteq_{var} \subseteq \mathcal{V} \times \mathcal{V}$ and $C <: D$ in \mathcal{C} and $r, s \in \mathcal{V}$ we define

$$\begin{aligned} Compat(\sqsubseteq_{var}, C, D, r, s) &\iff \\ \diamond(C^r) &\geq \diamond(D^s) \end{aligned} \quad (3.1)$$

$$\forall a \in A(D) . A^{\text{get}}(C^r, a) \sqsubseteq_{var} A^{\text{get}}(D^s, a) \quad (3.2)$$

$$\forall a \in A(D) . A^{\text{set}}(D^s, a) \sqsubseteq_{var} A^{\text{set}}(C^r, a) \quad (3.3)$$

$$\forall m \in M(D) . \forall \beta \in M(D^s, m) . \exists \alpha \in M(C^r, m) . \alpha \sqsubseteq_{var} \beta \quad (3.4)$$

where we extend \sqsubseteq_{var} to monomorphic RAJA method types as follows: if $\alpha = r_1, \dots, r_j \xrightarrow{p/q} r_0$ and $\beta = s_1, \dots, s_j \xrightarrow{t/u} s_0$ then $\alpha \sqsubseteq_{var} \beta$ is defined as $p \leq t$ and $q \geq u$ and $r_0 \sqsubseteq_{var} s_0$ and $s_i \sqsubseteq_{var} r_i$ for $i = 1, \dots, j$.

The subtyping relation $r \sqsubseteq s$ between views is now defined as the largest relation \sqsubseteq such that

$$r \sqsubseteq s \implies Compat(\sqsubseteq, C, C, r, s) \text{ for all } C$$

It is easy to see that \sqsubseteq is a preorder because if \sqsubseteq_{var} is a preorder, so is $\forall C. Compat(\sqsubseteq, C, C, \cdot, \cdot)$. We extend subtyping to RAJA-classes by

$$C^r <: D^s \iff C <: D \text{ and } r \sqsubseteq s \quad (3.5)$$

It is possible to define a more fine-grained subtyping relation directly on RAJA-classes, which would in particular give the subtyping $\text{Nil}\langle c \rangle <: \text{Nil}\langle b \rangle$ required in the copying example. We choose not to do this here because it unduly clutters notation and clarity. A practical implementation should, however, include this feature. Note that since both \sqsubseteq and $<:$ on FJEU are reflexive and transitive so is $<:$ on RAJA.

Definition 1 (Sharing Relation). We define the sharing relation between a single view r and a multiset of views \mathcal{D} written $\forall(r|\mathcal{D})$ as the largest relation \forall , such that if $\forall(r|\mathcal{D})$ then for all $C \in \mathcal{C}$:

$$\diamond(C^r) \geq \sum_{s \in \mathcal{D}} \diamond(C^s) \quad (3.6)$$

$$\forall s \in \mathcal{D} . r \sqsubseteq s \quad (3.7)$$

$$\forall a \in \text{dom}(A(C)) . \forall (A^{\text{get}}(C^r, a) \mid A^{\text{get}}(C^{\mathcal{D}}, a)) \quad (3.8)$$

where $A^{\text{get}}(C^{\mathcal{D}}, a) = \{A^{\text{get}}(C^s, a) \mid s \in \mathcal{D}\}$. When $\mathcal{D} = \{s_1, \dots, s_i\}$ is a finite multiset, we also write $\forall(r|s_1, \dots, s_i)$ for $\forall(r|\mathcal{D})$. We remark that, it would be possible to define $\forall(\cdot|\cdot)$ on the level of RAJA-classes rather than views.

Lemma 1.

$$\forall(r|\emptyset) \quad (3.9)$$

$$\forall(r|\{r\}) \quad (3.10)$$

$$\forall(r|\mathcal{D}) \iff \forall \text{ finite } \mathcal{E} \subset \mathcal{D} . \forall(r|\mathcal{E}) \quad (3.11)$$

$$\forall(r|\mathcal{D} \cup \{s\}) \wedge \forall(s|\mathcal{E}) \implies \forall(r|\mathcal{D} \cup \mathcal{E}) \quad (3.12)$$

$$r' \sqsubseteq r \wedge s' \sqsubseteq s \wedge \forall(r|\mathcal{D} \cup \{s'\}) \implies \forall(r'|\mathcal{D} \cup \{s\}) \quad (3.13)$$

Typing RAJA. We now give the formal definition of the RAJA-typing judgement. RAJA-typing is defined in Curry style, i.e., the terms being typed contain no RAJA-type annotations whatsoever. The intuitive meaning of the typing judgement $\Gamma \vdash_n^r e : C^r$ has already been given in the introduction.

$$\begin{array}{c}
\frac{\emptyset \vdash_0^{\diamond(C^r) + \text{SIZE}(C)} \text{new } C : C^r}{\text{new } C : C^r} (\diamond \text{NEW}) \quad \frac{}{x : C^r \vdash_{\diamond(C^r) + \text{SIZE}(C)}^0 \text{free}(x) : E^r} (\diamond \text{FREE}) \\
\frac{s = \text{A}^{\text{get}}(C^r, a) \quad D = C.a}{x : C^r \vdash_0^0 x.a : D^s} (\diamond \text{ACCESS}) \quad \frac{\text{A}^{\text{set}}(C^r, a) = s \quad C.a = D}{x : C^r, y : D^s \vdash_0^0 x.a < -y : C^r} (\diamond \text{UPDATE}) \\
\frac{C <: E}{x : E^r \vdash_0^0 (C)x : C^r} (\diamond \text{CAST}) \quad \frac{\forall (s \mid q_1, q_2) \quad \Gamma, y : D^{q_1}, z : D^{q_2} \vdash_n^r e : C^r}{\Gamma, x : D^s \vdash_n^r e[x/y, x/z] : C^r} (\diamond \text{SHARE}) \\
\frac{}{\emptyset \vdash_0^0 \text{null} : C^r} (\diamond \text{NULL}) \quad \frac{\Gamma_1 \vdash_{n'}^r e_1 : D^s \quad \Gamma_2, x : D^s \vdash_{n'}^r e_2 : C^r}{\Gamma_1, \Gamma_2 \vdash_{n''}^r \text{let } x = e_1 \text{ in } e_2 : C^r} (\diamond \text{LET}) \\
\frac{(E_1^{q_1}, \dots, E_j^{q_j} \xrightarrow{n/n'} E_0^{q_0}) \in \mathbf{M}(C^r, m)}{x : C^r, y_1 : E_1^{q_1}, \dots, y_j : E_j^{q_j} \vdash_{n'}^r x.m(y_1, \dots, y_j) : E_0^{q_0}} (\diamond \text{INVOCATION}) \\
\frac{x \in \Gamma \quad \Gamma \vdash_{n'}^r e_1 : C^r \quad \Gamma \vdash_{n'}^r e_2 : C^r}{\Gamma \vdash_{n'}^r \text{if } x \text{ instanceof } E \text{ then } e_1 \text{ else } e_2 : C^r} (\diamond \text{CONDITIONAL}) \\
\frac{n \geq u \quad n + u' \geq n' + u \quad \Theta \vdash_{u'}^u e : D^s \quad \forall x \in \Theta. \Gamma_x <: \Theta_x \quad D^s <: C^r}{\Gamma \vdash_{n'}^r e : C^r} (\diamond \text{WASTE})
\end{array}$$

Class Table. A RAJA-program $\mathcal{R} = (\mathcal{C}, \mathcal{V}, \diamond(\cdot), \text{A}^{\text{get}}(\cdot, \cdot), \text{A}^{\text{set}}(\cdot, \cdot), \mathbf{M}(\cdot, \cdot))$ is *well-typed* if for all $C \in \mathcal{C}$ and $r \in \mathcal{V}$ the following conditions are satisfied:

$$S(C) = D \implies \text{Compat}(\sqsubseteq, C, D, r, r) \quad (3.14)$$

$$\forall a \in \mathbf{A}(C) . \text{A}^{\text{set}}(C^r, a) \sqsubseteq \text{A}^{\text{get}}(C^r, a) \quad (3.15)$$

$$\forall m \in \mathbf{M}(C) . \forall \alpha \in \mathbf{M}(C^r, m) . \exists q, s \in \mathcal{V} . \forall (r \mid q, s) \wedge$$

$$\text{this} : C^q, x_1 : E_1^{r_1}, \dots, x_j : E_j^{r_j} \vdash_{n'}^{\frac{n + \diamond(C^s)}{n'}} \mathbf{M}_{\text{body}}(C, m) : E_0^{r_0} \quad (3.16)$$

$$\text{where } C.m = E_1, \dots, E_j \rightarrow E_0 \text{ and } \alpha = r_1, \dots, r_j \xrightarrow{n/n'} r_0$$

4 Main Result

Our main result involves the following concepts which we explain informally here; the full version contains formal definitions and motivations. We write $\llbracket (v:r).p \rrbracket_\sigma^{\text{stat}}$ for the view on the object reached from v (of view r) via access path p when accessed in this way. For example, if v points to a doubly-linked list of length 2 in σ then $\llbracket (v:\mathbf{q}).\text{next}.\text{next} \rrbracket_\sigma^{\text{stat}} = \mathbf{q}$.

We write $\Phi_\sigma(v : r)$ and $\Phi_\sigma(\eta : \Gamma)$ for the potentials of the data structures reachable from v , resp. η when viewed through r , resp., Γ . For example, $\Phi_\sigma(v : r) = \sum_{\mathbf{p}} \diamond(D^s)$, where D is the dynamic type of the record reached from v in σ via \mathbf{p} , whereas $s = \llbracket (v:r) \cdot \mathbf{p} \rrbracket_\sigma^{\text{stat}}$.

Finally, if Γ is a RAJA typing context with underlying FJEU context $|\Gamma|$, we write $\sigma \models \eta : \Gamma$ to mean that $\sigma \models \eta : |\Gamma|$ and, moreover, for each location ℓ reachable from η there exists a view r (its *proto-view*) such that $\forall(r \mid \mathcal{V}_{\sigma, \eta, \Gamma}(\ell))$ where $\mathcal{V}_{\sigma, \eta, \Gamma}(\ell)$ is the multiset consisting of all assumable views on location ℓ by σ, η, Γ , formally, $\mathcal{V}_{\sigma, \eta, \Gamma}(\ell) = \{ \llbracket (\eta_x : \langle \Gamma_x \rangle) \cdot \mathbf{p} \rrbracket_\sigma^{\text{stat}} \mid x \in \Gamma, \llbracket \eta_x \cdot \mathbf{p} \rrbracket_\sigma = \ell \}$.

This definition is a crucial invariant needed in the proof of the main result; it does not appear in the corollary intended for end users.

Theorem 1. *Fix a well-typed RAJA program \mathcal{R} . If*

$$\Gamma \vdash_{n'}^n e : C^r \quad (4.1)$$

$$\eta, \sigma \vdash^\circ e \rightsquigarrow v, \tau \quad (4.2)$$

$$\sigma \models \eta : (\Gamma, \Delta) \quad (4.3)$$

then

$$\eta, \sigma \vdash_{n' + \Phi_\tau(v : r) + \Phi_\tau(\eta : \Delta)}^{n + \Phi_\sigma(\eta : \Gamma) + \Phi_\sigma(\eta : \Delta)} e \rightsquigarrow v, \tau \quad (4.1)$$

$$\tau \models \eta[x_{\text{res}} \mapsto v] : (\Delta, x_{\text{res}} : C^r) \quad (4.2)$$

where x_{res} is assumed to be an unused auxiliary variable, i.e. $x_{\text{res}} \notin \Gamma, \Delta$. Note that (4.3) implies $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$ by definition of notation.

Proof. (Sketch) The proof is by induction on the operational semantics and a subordinate induction on typing derivations. Several of the cases present interesting difficulties. To give a flavour of the proof we sketch the case of field update here where we essentially have to show that a field update leaves the total potential unchanged and that newly created aliases admit a “proto-view”. We describe the crucial observation to give a flavour of the proof. Suppose that at runtime the update is $\ell.a := v$, ℓ being a location, a a field, v a value. For locations ℓ_1, ℓ_2 let $P(\ell_1, \ell_2)$ stand for access paths from ℓ_1 to ℓ_2 and $Q(\ell_1, \ell_2)$ stand for the subset of $P(\ell_1, \ell_2)$ consisting of those paths that do not go through $\ell.a$ and are thus unaffected by the update. After the update we have

$$P(\ell_1, \ell_2) = Q(\ell_1, \ell_2) + Q(\ell_1, \ell)(aQ(v, \ell))^* aQ(v, \ell_2)$$

since any access path either does not meet the updated location at all or goes through it a finite number of times. Since the right hand side of this identity comprises paths that are not affected by the update, information about it can be obtained from the assumptions that describe the situation before the update. A number of technical lemmas about the sharing relation and potentials are of course needed to flesh this out.

The following corollary is a direct consequence of the main result and it is in this form that we intend to use it. The apparently clumsy form of the main result is needed in order to enable an inductive proof.

Corollary 1. *Suppose that \mathcal{C} is an FJEU program containing (in Java notation) a class `List` of singly-linked lists with boolean entries, a class `C` containing a method `void C.main(List args)`, and arbitrary other classes and methods.*

Suppose furthermore, that there exists a RAJA-annotation of this program containing a view a where $\Diamond(\text{List}^a) = k \in \mathbb{N}$ and $\text{A}^{\text{get}}(\text{List}^a, \text{next}) = a$ then evaluating `C.main(args)` in a heap where `args` points to a linked list of length l requires at most kl memory cells.

5 Conclusion

We have presented a generic method for using potentials in the sense of amortised complexity to count memory allocations and deallocations. Our method allows for input dependent analysis without explicitly manipulating size expressions. This sets it apart against more direct methods based on sized types. We have stated and proved a nontrivial soundness property which shows that our typing rules for sharing correctly account for aliased and even circular data.

Inference. We have not studied the problem of view inference and not even algorithmic type checking since these two tasks are independent of soundness which was our main concern here. But of course inference and automatic type checking are of paramount importance for the viability of our method. We therefore briefly comment on how we plan to attack these issues.

First, we remark that if the structure of the views, i.e., the views without their potential annotations are known, the latter quantities can be efficiently found by LP-solving as was done in the precursor of this work [9]. Indeed, the system presented there can be faithfully embedded into RAJA and for this fragment automatic inference is unproblematic.

Likewise, the intermediate views that do not appear in class tables but only within method bodies typically take the form of fragments of already declared views in the sense that some fields are set to a zero view like `n` in Sect. 1. We are confident that these views can be generated automatically and on the fly during algorithmic type checking for example by reformulating the sharing rule in an algorithmic fashion.

A simple kind of view polymorphism should also be within reach if one applies the generic type and effect discipline [16] to our system.

Going beyond these low-hanging fruit will probably require the isolation of several fragments or high-level systems built on top of RAJA, supporting for example particular styles or patterns of programming.

Lastly, we mention that the possible access paths emanating from each class define an infinite regular tree, e.g., the tree consisting of the paths in $\text{next}^*(\text{elem} \cup \{\epsilon\})$ in the case of `Cons`. The set of views on a class in a (finite!) RAJA program defines a regular tiling of that tree and can perhaps be found using automata- or language-theoretic methods.

Extensions. Our focusing on heap space usage was a rather arbitrary choice. We believe that by slight modifications we can use the amortised method for other quantitative resources such as stack size, multiple freelists, number of open files, etc., in a similar fashion.

Limitations. Rule \Diamond_{UPDATE} contains a source of possible over-approximation because it does not reconstitute the potential contained in the overwritten data. This can lead to sound but not typable examples the simplest of which is as follows: if a is an object with a field f whose get- and set-types differ then $a.f < -a.f$ is not typable yet obviously sound since its effect is zero.

Another limitation of the system stems from the fact that object types do not change after a method invocation. This is mediated by the linear formulation of the type system: after a call $x.m()$ the reference x with its type is “used up”; a further invocation of a method on the object referenced by x can only happen through a prior invocation of \Diamond_{SHARE} and hence in general with a different type. Nevertheless, exploring type change after method invocation could be worthwhile.

We also note that our method estimates resource usage as a function of the input. Thus, programs whose resource usage depends on other parameters cannot be analysed. A concrete example is the numerical solution of a boundary value problem by solving successively larger and larger linear systems of equations.

Other limitations stem from the type inference problem. While it is in many cases possible to find a typing it might be difficult to come up with an inference scheme that encompasses those. On a positive side we note that the earlier system by the authors [9] can be faithfully mapped into the present system.

Acknowledgements. We thank Olha Shkaravska for pointing out the relationship of our previous work [9] with amortised complexity. We had been teaching the latter for years but failed to see the connection. We also thank Peter O’Hearn for a long and lively phone conversation on the topic of operational semantics of “free”. The authors acknowledge financial support by the GKLI Munich and the EU FET-IST projects IST-510255 (EmBounded), IST-15905 (Möbius), IST-2001-33149 (MRG).

References

1. E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2002.
2. M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation (PEPM)*, pages 55–65. ACM, 2003.
3. W.-N. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *The 12th International Static Analysis Symposium (SAS)*. LNCS, Sep 2005.
4. A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k -limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

5. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 171–183, 1998.
6. A. Galland and M. Baudet. Controlling and Optimizing the Usage of One Resource. In A. Ohori, editor, *1st Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science*, pages 195–211, Beijing, China, Nov. 27–29, 2003. Springer-Verlag.
7. G. Gómez and Y. A. Liu. Automatic time-bound analysis for a higher-order language. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 75–86. ACM Press, 2002.
8. B. Grobauer. *Topics in Semantics-based Program Manipulation*. PhD thesis, BRICS Aarhus, 2001.
9. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003.
10. J. Hughes and L. Pareto. Recursion and dynamic data structures in bounded space: towards embedded ML programming. In *Proc. International Conference on Functional Programming (ICFP). Paris, September 1999.*, pages 70–81, 1999.
11. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, volume 34(10), pages 132–146, N.Y., 1999.
12. S. S. Ishtiaq and P. W. O’Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 14–26. ACM, 2001.
13. H. Jonkers. Abstract storage structures. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981.
14. J. Krone, W. F. Ogden, and M. Sitaraman. Modular verification of performance constraints. Technical report, Dep. of Comp. Sci., Clemson Univeristy, May 2003.
15. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 296–309, New York, NY, USA, 2005. ACM Press.
16. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *J. Funct. Program.*, 2(3):245–271, 1992.
17. R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.
18. M. Tofte, L. Birkedal, M. Elsmann, N. Hallenberg, T. Olesen, and P. Sestoft. Programming with regions in the ml kit, April 2002. IT University of Copenhagen <http://www.itu.dk/research/mlkit/>.
19. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
20. L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate live memory analysis for garbage-collected languages. In *Proceedings of The Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 2001.
21. P. Vasconcelos. *Space Cost Modeling for Concurrent Resource Sensitive Systems*. PhD thesis, School of Comp. Sci., University of St Andrews, Scotland, to appear.
22. D. Walker and G. Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177+, 2001.

Haskell Is Not Not ML

Ben Rudiak-Gould¹, Alan Mycroft¹, and Simon Peyton Jones²

¹ University of Cambridge Computer Laboratory

² Microsoft Research Cambridge

{br276, am}@cl.cam.ac.uk, simonpj@microsoft.com

Abstract. We present a typed calculus IL (“intermediate language”) which supports the embedding of ML-like (strict, eager) and Haskell-like (non-strict, lazy) languages, without favoring either. IL’s type system includes negation (continuations), but not implication (function arrow). Within IL we find that lifted sums and products can be represented as the double negation of their unlifted counterparts. We exhibit a compilation function from IL to AM—an abstract von Neumann machine—which maps values of ordinary and doubly negated types to heap structures resembling those found in practical implementations of languages in the ML and Haskell families. Finally, we show that a small variation in the design of AM allows us to treat any ML value as a Haskell value at runtime without cost, and project a Haskell value onto an ML type with only the cost of a Haskell `deepSeq`. This suggests that IL and AM may be useful as a compilation and execution model for a new language which combines the best features of strict and non-strict functional programming.

1 Introduction

Every functional language in use today is either strict and eagerly evaluated or non-strict and lazily evaluated. Though most languages make some provision for both evaluation strategies, it is always clear which side their bread is buttered on: one evaluation strategy is automatic and transparent, while the other requires micromanagement by the programmer.

The dichotomy is surprising when one considers how similar lazy functional programs and eager functional programs look in practice. Most of the differences between SML and Haskell are independent of evaluation order (syntax, extensible records, module systems, type classes, monadic effect system, rank-2 types. . .). Were it not for those differences, it would in many cases be difficult to tell which language a given code fragment was actually written in. Why, then, is there no *hybrid language* which can understand code like

```
foldl f z l = case l of []      -> z
                  (x:xs) -> foldl f (f z x) xs
```

in some way that abstracts over possible evaluation orders?

Designing such a language turns out to be quite difficult. Reducing the significant notational and cognitive burdens of mixed strict/non-strict programming

is an open research problem, which we do not attempt to solve in this paper. Instead we address a prerequisite for the success of any hybrid language, which is the possibility of implementing it easily and efficiently enough to be competitive with dedicated strict and lazy languages. That is, we discuss the optimization and back end phases of a hybrid language compiler, leaving the front end for future work.

In Section 2 we introduce a compiler intermediate language (“IL”) which can support conventional strict and non-strict source languages through different front-end translations. IL is the centerpiece and the main contribution of this paper. Section 3 presents toy strict and non-strict languages (“SL” and “LL”) and their translations to IL. SL and LL are entirely conventional in design, and are included only as examples of translations to IL. Section 4 introduces an abstract machine (“AM”) with a von Neumann architecture, which can act as an execution environment for IL. Our goal is that compilation of SL or LL via IL to AM should be competitive (in code size and execution speed) with compilation of SL or LL to purpose-designed, incompatible abstract machines.

With this architecture we can compile ML-like and Haskell-like source code to a common abstract machine with a single heap. The strict and non-strict languages cannot share data structures, since they have incompatible type systems, but they can exchange data via appropriate marshalling code (written in IL). In Section 5 we aim to get rid of the marshalling and enable direct sharing. By carefully choosing the representation of the heap data, we can arrange that the natural injection from ML values into Haskell values is a no-op at runtime, and the natural projection from Haskell values to pointed ML values carries only the cost of a Haskell `deepSeq` operation (in particular, it need not perform copying).

Space constraints have forced us to omit or gloss over many interesting features of IL in this conference paper. Interested readers may find additional material at the project website [7].

1.1 A Note on Types

We handle recursive types with a form $\nu\alpha.T$, which denotes the type T with free occurrences of α replaced by T itself. The ν form can be seen as extending the tree of types to a graph with cycles; $\nu\alpha$ simply gives a name to the graph node which it annotates. For example, $\nu\alpha.\neg\alpha$ describes a single negation node pointing to itself, while $\nu\alpha.\alpha$ is meaningless, since α refers to no node. In this paper we will treat the ν form as a metasyntactic description of a type graph, rather than a feature of the concrete syntax of types. This allows us to omit rules for syntactic manipulation of ν forms, which would complicate the presentation and yield no new insight.

Rather than attempt to distinguish between inductive types (à la ML) and coinductive types (à la Haskell), we make all types coinductive, with the caveat that types may contain values which have no representation as terms in the language. This is already true of recursive datatypes in Haskell (for example, uncountably many values inhabit the Haskell list type `[Bool]`, but there are only countably many expressions of type `[Bool]`) and it is true of functions in both ML and Haskell (if S is infinite then $S \rightarrow \text{Bool}$ is uncountable).

Deciding whether to include parametric polymorphism was difficult. There are several cases in which we would like to speak of polymorphic types, and one case in which we must do so; on the other hand the introduction of polymorphism into our languages has no novel features, and would in most cases simply clutter the discussion and the figures. Instead we adopt a compromise approach. Within IL, SL and LL, type variables (except those quantified by ν) simply represent unknown types; there is no instantiation rule and no polymorphic quantification. We permit ourselves, however, to speak of instantiation within the text of the paper. E.g. we may say that a term E has the “type” $\forall\alpha. (\alpha \& \alpha)$, by which we mean that for any type T , $E[T/\alpha]$ is well-typed and has the type $(T \& T)$.

2 IL

IL is a continuation-passing calculus with a straightforward syntax and semantics. It makes no explicit mention of strictness or non-strictness, but it contains both notions in a way which will be explained in Section 2.3.

IL types are shown in Fig. 1, and IL expressions in Fig. 2. We will use the words “expression” and “term” interchangeably in this paper. The type $\mathbf{0}$ has a special role in IL; the distinction between $\mathbf{0}$ and non- $\mathbf{0}$ types, and associated terms and values, will recur throughout this paper.¹ Therefore we adopt the convention that T ranges only over non- $\mathbf{0}$ types (mnemonic *True*), while U ranges over all types. For expressions, E ranges only over those of non- $\mathbf{0}$ type; F ranges over those of type $\mathbf{0}$ (mnemonic: *False*); and G ranges over all expressions (*General*). Note in particular that Figs. 1 and 2 imply that we do not permit types such as $\neg\mathbf{0}$ and $(\mathbf{0} \vee \mathbf{1})$: the only type containing $\mathbf{0}$ is $\mathbf{0}$ itself.

In the spirit of Martin-Löf type theory and the Curry-Howard isomorphism, we give both a logical and an operational interpretation to IL types and expressions. Logically, types are formulas, and expressions are proofs of those formulas; operationally, types are sets of values and expressions evaluate to a value from the set.

Syntax	Logical meaning	Operational meaning
$\mathbf{0}$	Contradiction.	The empty type.
$\mathbf{1}$	Tautology.	The unit type.
$T_1 \& T_2$	Conjunction (there are proofs of T_1 and T_2).	Unlifted product.
$T_1 \vee T_2$	Disjunction (there is a proof of T_1 or of T_2).	Unlifted sum.
$\neg T$	From T it is possible to argue a contradiction.	Continuation (see text).
α, β, γ	Free type variables (of non- $\mathbf{0}$ type).	

Fig. 1. Syntax and gloss of IL types

¹ Our type $\mathbf{0}$ is conventionally called \perp , but in this paper we use the symbol \perp for another purpose.

Syntax	Logical meaning	Operational meaning
x, y, k	Free variables.	
$\lambda(x:T).F$	A <i>reductio ad absurdum</i> proof of $\neg T$.	Builds a closure.
$E_1 \bowtie E_2$	Proves 0 (a contradiction) from E_1 , of type $\neg T$, and E_2 , of type T .	Enters a closure.
$()$	Proves 1 .	Unit constructor.
(E_1, E_2)	Proves a conjunction by proving its conjuncts.	Pair constructor.
inl E	Proves a disjunction by its left case.	Union constructor.
inr E	Proves a disjunction by its right case.	Union constructor.
fst E	Proves T_1 , where $E : (T_1 \& T_2)$.	Pair deconstructor.
snd E	Proves T_2 , where $E : (T_1 \& T_2)$.	Pair deconstructor.
case \dots	case $E \{(x)G_1; (y)G_2\}$ is a case analysis of a disjunction.	Union deconstructor.

Fig. 2. Syntax and gloss of IL expressions

A striking feature of IL, when compared with most programming calculi, is that its type system includes logical negation, but no implication (function arrow). Operationally, $\neg T$ is a *continuation* which takes a value of type T and never returns. Logically, the only way to prove $\neg T$ is by *reductio ad absurdum*: to prove $\neg T$, we show that any proof of T can be used to construct a proof of **0**. We introduce continuations with the symbol λ and eliminate them with infix \bowtie , reserving λ and juxtaposition for function types (used later in SL and LL).

In its use of \neg rather than a function arrow, IL resembles Wadler’s dual calculus [9]. IL, however, has only terms (no coterms), and is intuitionistic (not classical).

To simplify the discussion, we will often omit type signatures when they are uninteresting, and we will allow tuple-matching anywhere a variable is bound, so for example $\lambda(x, y). \dots x \dots y \dots$ is short for $\lambda w. \dots \mathbf{fst} \ w \dots \mathbf{snd} \ w \dots$.

2.1 Typing and Operational Semantics of IL

Fig. 3 lists the typing rules for IL expressions, and Fig. 4 gives a small-step operational semantics for IL, with the reduction relation \rightsquigarrow . These satisfy the subject reduction property that if $\Gamma \vdash G : U$ and $G \rightsquigarrow G'$, then $\Gamma \vdash G' : U$. The proof is straightforward.

We say that a term is a *value* if it has no free variables and is not subject to any reduction rules. The values in IL are

$$V ::= () \mid (V, V) \mid \mathbf{inl} \ V \mid \mathbf{inr} \ V \mid \lambda(x:T).F$$

Note that there are no values of type **0**.

2.2 Nontermination in IL

There is no **fix** or **letrec** form in IL, but we can construct nonterminating expressions even without it. For example, let E be $(\lambda(x : \nu\alpha. \neg\alpha). x \bowtie x)$; then

$\frac{}{\Gamma, (x:T) \vdash x : T}$	$\frac{\Gamma, (x:T) \vdash F : \mathbf{0}}{\Gamma \vdash \lambda(x:T).F : \neg T}$	$\frac{\Gamma \vdash E_1 : \neg T \quad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 \bowtie E_2 : \mathbf{0}}$	
$\frac{\Gamma \vdash E_1 : T_1 \quad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E_1, E_2) : (T_1 \& T_2)}$	$\frac{\Gamma \vdash E : (T_1 \& T_2)}{\Gamma \vdash \mathbf{fst} E : T_1}$	$\frac{\Gamma \vdash E : (T_1 \& T_2)}{\Gamma \vdash \mathbf{snd} E : T_2}$	
$\frac{\Gamma \vdash E : T_1}{\Gamma \vdash \mathbf{inl} E : (T_1 \vee T_2)}$	$\frac{\Gamma \vdash E : T_2}{\Gamma \vdash \mathbf{inr} E : (T_1 \vee T_2)}$	$\frac{}{\Gamma \vdash () : \mathbf{1}}$	
$\frac{\Gamma \vdash E : (T_1 \vee T_2) \quad \Gamma, (x:T_1) \vdash G_1 : U \quad \Gamma, (y:T_2) \vdash G_2 : U}{\Gamma \vdash \mathbf{case} E \{(x)G_1; (y)G_2\} : U}$			

Fig. 3. Well-typed IL expressions

$\mathbf{fst} (E_1, E_2) \rightsquigarrow E_1$ $\mathbf{snd} (E_1, E_2) \rightsquigarrow E_2$ $\mathbf{case} (\mathbf{inl} E) \{(x)G_1; (y)G_2\} \rightsquigarrow G_1[E/x]$ $\mathbf{case} (\mathbf{inr} E) \{(x)G_1; (y)G_2\} \rightsquigarrow G_2[E/y]$ $(\lambda x. F) \bowtie E \rightsquigarrow F[E/x]$ $\frac{G_1 \rightsquigarrow G_2}{C[G_1] \rightsquigarrow C[G_2]}$	Evaluation context $C[] ::= []$ $\mid C[] \bowtie E \mid E \bowtie C[]$ $\mid (C[], E) \mid (E, C[])$ $\mid \mathbf{inl} C[] \mid \mathbf{inr} C[]$ $\mid \mathbf{fst} C[] \mid \mathbf{snd} C[]$ $\mid \mathbf{case} C[] \{(x)G_1; (y)G_2\}$
---	---

Fig. 4. Operational semantics of IL

$E \bowtie E$ is well-typed, and proves $\mathbf{0}$. We will refer to this term by the name **diverge**. It is analogous to $(\lambda x. x x)(\lambda x. x x)$ in the untyped lambda calculus. It is well-typed in IL because we permit recursive types via the ν construct. In its logical interpretation, this proof is known variously as Curry’s paradox or Löb’s paradox; exorcising it from a formal system is not easy. In IL, we do not try to exorcise it but rather welcome its presence, since any logically consistent language would not be Turing-complete.

But in IL, unlike ML and Haskell, nontermination cannot be used to inhabit arbitrary types. The type systems of ML and Haskell, interpreted logically, are inconsistent in the classical sense of triviality: the expression $(\mathbf{letrec} x() = x() \mathbf{in} x())$ can appear anywhere in a program, and can be given any type whatsoever.² IL is not trivial; rather, it is what is known as *paraconsistent*. More precisely, IL has the following properties:

- *Confluence*: for all expressions G_1, G_2, G_3 , if $G_1 \rightsquigarrow^* G_2$ and $G_1 \rightsquigarrow^* G_3$, then there is a G_4 such that $G_2 \rightsquigarrow^* G_4$ and $G_3 \rightsquigarrow^* G_4$.

² A similar expression can be constructed without **letrec** by using an auxiliary recursive type, as in IL.

- *Strong normalization*: for any expression E (of non-**0** type) there is an integer n such that any sequence of reductions from E has length at most n .

Together these properties imply that any IL expression of non-**0** type reduces in finitely many steps to a value which does not depend on the order of reduction. In contrast, we see that *no* IL expression of type **0** reduces to a value, since there are no values of type **0**. If evaluation terminates it can only be on a non-value, such as $x \bowtie E$.

2.3 Lifted and Pointed Types in IL

For any type T , there is a natural map from the expressions (including values) of type T into the values of type $\neg\neg T$: we construct a continuation of type $\neg\neg T$ which, given a continuation of type $\neg T$, passes the value of type T to it. Symbolically, we take E to $(\lambda k. k \bowtie E)$. We will call this mapping **lift**.

Similarly, for any type T , there is a natural map from the values of type $\neg\neg\neg T$ onto the values of type $\neg T$: we construct a continuation of type $\neg T$ which takes a value of type T , converts it to a value of type $\neg\neg T$ by **lift**, and passes that to the continuation of type $\neg\neg\neg T$. Symbolically, we take E to $(\lambda x. E \bowtie (\lambda k. k \bowtie x))$. We will call this mapping **colift**. It is easy to see that **colift** \circ **lift** is the identity on types $\neg T$, so **lift** is an injection and **colift** is a surjection.

There is, however, no natural map from $\neg\neg T$ to T when T does not begin with \neg (i.e. when T is not of the form $\neg T'$). In particular, $(\lambda(x:\neg T). \mathbf{diverge})$ of type $\neg\neg T$ has no counterpart in T , if T has no leading \neg .

If we say that types T_1 and T_2 are *semantically equivalent* if there exist (co)lifting maps from T_1 to T_2 and from T_2 to T_1 , then the types $\neg^k T$ for T without a leading \neg fall into the three semantic equivalence classes shown below, where we write $\neg^k T$ for k successive negations of T .

1. $\neg^0 T$
2. $\neg^1 T, \neg^3 T, \neg^5 T, \neg^7 T, \dots$
3. $\neg^2 T, \neg^4 T, \neg^6 T, \neg^8 T, \dots$

There is a natural identification of classes 1 and 3 with the types of SL and LL *values*, respectively, and of class 2 with *evaluation contexts* in both SL and LL—more precisely, the types in class 2 are the types of the continuations which receive the computed values. We will motivate this identification informally with some examples; later, explicit translations from SL/LL to IL will make it precise.

We have already observed that **lift** is an injection: what values in $\neg\neg T$ does it miss? A value of type $\neg\neg T$ is called with a continuation of type $\neg T$. It may, perhaps after some computation, pass a result V of type T to the continuation; because of IL's purity, any such value is indistinguishable from **lift** V . But there are also values of type $\neg\neg T$ which do not call the continuation at all. In principle there may be many values in this category—one could imagine aborting execution with a variety of error messages, or transferring control to an exception-handling continuation—but following tradition we will lump all of these together as a single value $\perp_{\neg\neg T}$. Clearly \perp exists not just in types $\neg\neg T$ but

in any type $\neg T$: consider $\lambda(x:T).\mathbf{diverge}$. Types of the form $\neg T$ are *pointed*, and types of the form $\neg\neg T$ are *lifted*. This applies even if T itself begins with \neg , so each successive double negation adds a level of lifting: e.g. the type $\neg\neg\neg\neg\mathbf{1}$ contains distinguishable values $\mathbf{lift\ lift}()$, $\mathbf{lift}\ \perp_{\neg\neg\mathbf{1}}$, and $\perp_{\neg\neg\neg\neg\mathbf{1}}$.

We have likewise observed that **colift** is a surjection, and by a similar argument we can show that it merges the two outermost levels of lifting: in the case of $\neg\neg\neg\neg\mathbf{1}$ it maps $\mathbf{lift\ lift}()$ to $\mathbf{lift}()$ and maps both $\mathbf{lift}\ \perp_{\neg\neg\mathbf{1}}$ and $\perp_{\neg\neg\neg\neg\mathbf{1}}$ to the single value $\perp_{\neg\neg\mathbf{1}}$.

The maps **lift** and **colift** resemble the **unit** and **join** operations of a lifting monad, except that **colift** is slightly more general than **join**. In a lifting monad, **unit** would map from T to $\neg\neg T$ and **join** from $\neg\neg\neg\neg T$ to $\neg\neg T$.

Our discussion above overlooks the possibility that a continuation of type $\neg T$ might be called *more* than once. Multiple calls passing the same value are indistinguishable from a single call because of purity, but there are interesting terms that pass two or more distinguishable values to their continuation. An example is $\lambda k.k \bowtie \mathbf{inr}(\lambda x.k \bowtie \mathbf{inl}\ x)$ of type $\forall\alpha.\neg\neg(\alpha \vee \neg\alpha)$, which is an IL interpretation of the story of the devil's offer from [9]. Inasmuch as we intend to use double negation to model Haskell lifting, we would like to forbid such values. We do not discuss this problem further in this paper.

3 SL and LL

SL and LL are simple strict and non-strict programming languages which have the same syntax, but different translations to IL. They are pure functional languages; side effects are assumed to be handled via monads or some comparable approach. SL has no provision for lazy evaluation, and LL has no provision for eager evaluation.

Fig. 5 shows the syntax of SL/LL expressions (ranged over by e) and types (ranged over by t). The typing rules and operational semantics are standard, and we will not give them here. We use **case** rather than **fst** and **snd** to deconstruct pairs because it simplifies the translation slightly. The term **error** stands for a generic divergent expression like $1/0$ or Haskell's **undefined**.

x, y, z		Free variables
α, β		Free type vars.
$\lambda(x:t).e$	$t \rightarrow t$	Functions
ee		
$()$	$\mathbf{1}$	Unit
(e, e)	$t \otimes t$	Pairs
$\mathbf{case}\ e\ \{(x, y)e\}$		
$\mathbf{inl}\ e, \mathbf{inr}\ e$	$t \oplus t$	Unions
$\mathbf{case}\ e\ \{(x)e; (y)e\}$		
error		Proves any type

Fig. 5. SL/LL expressions and types

SL/LL type	SL to IL	LL to IL
$\mathcal{D}_v \llbracket t \rrbracket$	$\mathcal{D}_b \llbracket t \rrbracket$	$\neg\neg\mathcal{D}_b \llbracket t \rrbracket$
$\mathcal{D}_k \llbracket t \rrbracket$	$\neg\mathcal{D}_b \llbracket t \rrbracket$	
$\mathcal{D}_b \llbracket \mathbf{1} \rrbracket$	$\mathbf{1}$	
$\mathcal{D}_b \llbracket t_1 \otimes t_2 \rrbracket$	$\mathcal{D}_v \llbracket t_1 \rrbracket \ \& \ \mathcal{D}_v \llbracket t_2 \rrbracket$	
$\mathcal{D}_b \llbracket t_1 \oplus t_2 \rrbracket$	$\mathcal{D}_v \llbracket t_1 \rrbracket \ \vee \ \mathcal{D}_v \llbracket t_2 \rrbracket$	
$\mathcal{D}_b \llbracket t_1 \rightarrow t_2 \rrbracket$	$\neg(\mathcal{D}_v \llbracket t_1 \rrbracket \ \& \ \mathcal{D}_k \llbracket t_2 \rrbracket)$	
$\mathcal{D}_b \llbracket \alpha \rrbracket$	α	

Fig. 6. Translation of SL/LL to IL types

3.1 Translation to IL

Translation of SL/LL types to IL types is shown in Fig. 6. In order to model the distinction between values and expression contexts mentioned in Section 2.3 we use three different type translations, written $\mathcal{D}_b \llbracket t \rrbracket$, $\mathcal{D}_v \llbracket t \rrbracket$, and $\mathcal{D}_k \llbracket t \rrbracket$. $\mathcal{D}_v \llbracket t \rrbracket$ is the type of values on the heap (and of bindings to variables). $\mathcal{D}_k \llbracket t \rrbracket$ is the type of a *k*ontinuation which receives the result of evaluating an expression of type t . $\mathcal{D}_b \llbracket t \rrbracket$ is a “bare” type which has not yet been converted to a value or continuation type by suitable negation.

In the interest of simplicity, SL and LL support only anonymous sums and products; there is no provision for declaring new datatypes. It is worth noting that LL’s type system is consequently not quite expressive enough to represent many Haskell datatypes, because Haskell does not lift at every opportunity. For example, Haskell’s `Bool` type is isomorphic (as a “bare” type) to IL’s $(1 \vee 1)$, while the closest LL equivalent, $1 \oplus 1$, maps to the IL type $(\neg\neg 1 \vee \neg\neg 1)$. Accommodating Haskell types requires a more complex translation, which, however, introduces no new difficulties.

The representation of functions is interesting. Logical implication $P \Rightarrow Q$ is classically equivalent to $\neg P \vee Q$, but this will not work as a function type in our intuitionistic calculus. An IL value of type $\neg P \vee Q$ is either a value from $\neg P$ or a value from Q ; the former includes all divergent functions and the latter all constant functions, but there are no values which can accept an argument and return an answer that depends on that argument. The type $\neg(P \& \neg Q)$, again classically equivalent to implication, does not share this problem. Its operational interpretation is that a function is a continuation which takes two values, one of type P (the argument) and one of type $\neg Q$ (somewhere to send the result). This is exactly how function calls work in practical abstract machines: the two arguments to this continuation are the two values—argument and return address—pushed onto the stack before jumping to the function’s entry point.

The translations from SL and LL terms to IL terms are shown in Fig. 7. These are the familiar continuation-passing translations of Plotkin [6]. Because

(In the LL to IL translation $\llbracket e \rrbracket_v$ abbreviates $\lambda k. \llbracket e \rrbracket \triangleright k$)

SL/LL term	Translation (SL to IL)	Translation (LL to IL)
$\llbracket x \rrbracket \triangleright E$	$E \bowtie x$	$x \bowtie E$
$\llbracket e e' \rrbracket \triangleright E$	$\llbracket e \rrbracket \triangleright \lambda x. \llbracket e' \rrbracket \triangleright \lambda x'. x \bowtie (x', E)$	$\llbracket e \rrbracket \triangleright \lambda x. x \bowtie (\llbracket e' \rrbracket_v, E)$
$\llbracket (e, e') \rrbracket \triangleright E$	$\llbracket e \rrbracket \triangleright \lambda x. \llbracket e' \rrbracket \triangleright \lambda x'. E \bowtie (x, x')$	$E \bowtie (\llbracket e \rrbracket_v, \llbracket e' \rrbracket_v)$
$\llbracket \text{inl } e \rrbracket \triangleright E$	$\llbracket e \rrbracket \triangleright \lambda x. E \bowtie \text{inl } x$	$E \bowtie \text{inl } \llbracket e \rrbracket_v$
$\llbracket \text{inr } e \rrbracket \triangleright E$	$\llbracket e \rrbracket \triangleright \lambda x. E \bowtie \text{inr } x$	$E \bowtie \text{inr } \llbracket e \rrbracket_v$
$\llbracket \lambda x. e \rrbracket \triangleright E$	$E \bowtie \lambda(x, k). \llbracket e \rrbracket \triangleright k$ $E \bowtie ()$ diverge $\llbracket e_1 \rrbracket \triangleright \lambda z. \text{case } z \{ (x) \llbracket e_2 \rrbracket \triangleright E; (y) \llbracket e_3 \rrbracket \triangleright E \}$ $\llbracket e_1 \rrbracket \triangleright \lambda(x, y). \llbracket e_2 \rrbracket \triangleright E$	
$\llbracket () \rrbracket \triangleright E$		
$\llbracket \text{error} \rrbracket \triangleright E$		
$\llbracket \text{case } e_1 \{ (x)e_2; (y)e_3 \} \rrbracket \triangleright E$		
$\llbracket \text{case } e_1 \{ (x, y)e_2 \} \rrbracket \triangleright E$		

Fig. 7. Translation of SL/LL to IL terms (type signatures omitted)

IL has explicit continuations while the continuations in SL/LL are implicit, the translation must be done in the context of an IL continuation. $\llbracket e \rrbracket \triangleright E$ denotes the IL expression which passes e 's value on to the IL continuation E . This should be treated as a syntactic unit; \triangleright has no meaning on its own. For LL we write $\llbracket e \rrbracket_v$ as a shorthand for $\lambda k. \llbracket e \rrbracket \triangleright k$; this notation will prove useful in Section 4.2.

Type signatures have been omitted for space and readability reasons. Restoring the type signatures and adding standard typing rules for SL/LL terms, it can be shown that the translation of a well-typed SL/LL term is a well-typed IL term. In fact, we can show that if $e : t$ and $(\llbracket e \rrbracket \triangleright E) : \mathbf{0}$, then $E : \mathcal{D}_k \llbracket t \rrbracket$, and (in LL) that if $e : t$, then $\llbracket e \rrbracket_v : \mathcal{D}_v \llbracket t \rrbracket$. Note the translations of $\llbracket x \rrbracket \triangleright E$, which capture the essential difference between “ML-like” and “Haskell-like” embeddings in IL.

Translation Examples. For a first example we consider the SL/LL expression **inl error**. In SL this expression will clearly always diverge when evaluated, and our SL-to-IL translation turns out to yield **diverge** directly:

$$\llbracket \mathbf{inl\ error} \rrbracket \triangleright k = \llbracket \mathbf{error} \rrbracket \triangleright \lambda x. k \bowtie \mathbf{inr\ } x = \mathbf{diverge}$$

The LL-to-IL translation instead boxes the divergence:

$$\llbracket \mathbf{inr\ error} \rrbracket \triangleright k = k \bowtie \mathbf{inr\ } (\lambda k'. \llbracket \mathbf{error} \rrbracket \triangleright k') = k \bowtie \mathbf{inr\ } (\lambda k'. \mathbf{diverge})$$

The translations of the nested function application $p(qr)$ are interesting. From SL we have the following translation, where $\stackrel{*}{=}$ denotes a sequence of several (trivial) translation steps, and $\stackrel{*}{\rightsquigarrow}$ denotes a sequence of “clean-up” beta reductions after the translation proper.

$$\begin{aligned} \llbracket p(qr) \rrbracket \triangleright k &= \llbracket p \rrbracket \triangleright \lambda f. \llbracket qr \rrbracket \triangleright \lambda x. f \bowtie (x, k) \\ &= \llbracket p \rrbracket \triangleright \lambda f. \llbracket q \rrbracket \triangleright \lambda f'. \llbracket r \rrbracket \triangleright \lambda x'. f' \bowtie (x', \lambda x. f \bowtie (x, k)) \\ &\stackrel{*}{=} (\lambda f. (\lambda f'. (\lambda x'. f' \bowtie (x', \lambda x. f \bowtie (x, k)))) \bowtie r) \bowtie q \bowtie p \\ &\stackrel{*}{\rightsquigarrow} q \bowtie (r, (\lambda x. p \bowtie (x, k))) \end{aligned}$$

For LL we have

$$\begin{aligned} \llbracket p(qr) \rrbracket \triangleright k &= \llbracket p \rrbracket \triangleright \lambda f. f \bowtie ((\lambda k'. \llbracket qr \rrbracket \triangleright k'), k) \\ &= \llbracket p \rrbracket \triangleright \lambda f. f \bowtie ((\lambda k'. \llbracket q \rrbracket \triangleright \lambda f'. f' \bowtie ((\lambda k''. \llbracket r \rrbracket \triangleright k''), k')), k) \\ &\stackrel{*}{=} p \bowtie \lambda f. f \bowtie ((\lambda k'. q \bowtie (\lambda f'. f' \bowtie ((\lambda k''. r \bowtie k''), k'))), k) \end{aligned}$$

Our operational semantics cannot simplify this term. But it can be shown that η reduction is safe in IL (in the sense of contextual equivalence), so we may reduce it to $p \bowtie \lambda f. f \bowtie ((\lambda k'. q \bowtie (\lambda f'. f' \bowtie (r, k'))), k)$. Using the **lift** operation from section 2.3, and renaming k' to x , we get $p \bowtie \mathbf{lift}\ ((\lambda x. q \bowtie \mathbf{lift}\ (r, x)), k)$, which, modulo lifting, is surprisingly similar to its SL counterpart.

4 AM

AM is an abstract machine designed to run IL code. The primitives of AM are chosen to resemble machine-code or byte-code instructions. AM is untyped for reasons of simplicity.

The purpose of AM is to provide a framework for discussing the low-level optimizations that make subtyping possible, which are described in Section 5. We are not concerned with a formal treatment of compilation as such, nor are we interested in most of the optimizations found in existing abstract machines. Therefore we will define AM only informally, and will gloss over most performance issues.

AM is a register machine. Throughout this section register names will be written in **typewriter face**. There are two special registers **env** and **arg**, which are used when entering a continuation, as well as a collection of compile-time constants, which are never assigned to, but are otherwise indistinguishable from registers. All other registers are local temporaries. There is never a need to save registers across function calls, because every call is a tail call. AM has no stack.

Registers hold *machine words*, which can be pointers to heap objects, pointers to addressable code, or tag values (which will be discussed later).

There are just five instructions in AM:

$M ::= x \leftarrow y$	Sets register x equal to register y .
$x \leftarrow y[i]$	Indexed load: Register x gets the value at offset i within the heap object pointed to by register y .
$x \leftarrow \mathbf{new} \ y1, \dots, yn$	Allocates n consecutive words from the heap, places the address of the allocated memory in register x , and stores the operands $y1, \dots, yn$ at locations $x[0], \dots, x[n-1]$.
if $x1 = x2$ then M^* else M^*	Compares two registers and executes the first instruction sequence if they are equal, the second instruction sequence if they are not equal. M^* denotes a sequence of zero or more instructions.
jump x	Transfers control to the instruction sequence whose address is in register x .

To make code more concise, we allow indexed-load expressions $r[i]$ wherever a register operand is expected. For example, the instruction **jump** **env**[0] is equivalent to the two-instruction sequence **tmp** \leftarrow **env**[0] ; **jump** **tmp**.

While IL is indifferent as regards evaluation order, we choose to use eager evaluation when we compile it to AM.

4.1 Compilation

We have already noted that IL expressions divide naturally into those of type **0** and those not of type **0**. In AM this has the following concrete meaning:

- IL expressions of type **0** compile to *addressable instruction sequences*. These have an entry point which we jump to when calling a continuation, and they terminate by jumping to another addressable instruction sequence.
- IL expressions of other types compile to *non-addressable instruction sequences*: these appear within addressable instruction sequences and construct values on the heap.

Compilation form	Expansion
$\mathcal{F} \llbracket E_1 \bowtie E_2 \rrbracket \Gamma$	$\mathcal{E} \llbracket E_1 \rrbracket \Gamma \text{ tmp1} ; \mathcal{E} \llbracket E_2 \rrbracket \Gamma \text{ tmp2}$ $\text{env} \leftarrow \text{tmp1} ; \text{arg} \leftarrow \text{tmp2}$ $\text{jump env}[0]$
$\mathcal{F} \llbracket \text{case } E \{ (x)F_1 ; (y)F_2 \} \rrbracket \Gamma$	$\mathcal{E} \llbracket E \rrbracket \Gamma \text{ tmp}$ $\text{if tmp}[0] = \text{tagLeft}$ $\quad \text{then } \mathcal{F} \llbracket F_1 \rrbracket (\Gamma[\text{tmp}[1]/x])$ $\quad \text{else } \mathcal{F} \llbracket F_2 \rrbracket (\Gamma[\text{tmp}[1]/y])$
$\mathcal{E} \llbracket x \rrbracket \Gamma \mathbf{r}$	$\mathbf{r} \leftarrow \Gamma(x)$
$\mathcal{E} \llbracket () \rrbracket \Gamma \mathbf{r}$	$\mathbf{r} \leftarrow \text{new tagUnit}$
$\mathcal{E} \llbracket (E_1, E_2) \rrbracket \Gamma \mathbf{r}$	$\mathcal{E} \llbracket E_1 \rrbracket \Gamma \text{ tmp1} ; \mathcal{E} \llbracket E_2 \rrbracket \Gamma \text{ tmp2}$ $\mathbf{r} \leftarrow \text{new tagPair, tmp1, tmp2}$
$\mathcal{E} \llbracket \text{inl } E \rrbracket \Gamma \mathbf{r}$	$\mathcal{E} \llbracket E \rrbracket \Gamma \text{ tmp} ; \mathbf{r} \leftarrow \text{new tagLeft, tmp}$
$\mathcal{E} \llbracket \text{inr } E \rrbracket \Gamma \mathbf{r}$	$\mathcal{E} \llbracket E \rrbracket \Gamma \text{ tmp} ; \mathbf{r} \leftarrow \text{new tagRight, tmp}$
$\mathcal{E} \llbracket \text{fst } E \rrbracket \Gamma \mathbf{r}$	$\mathcal{E} \llbracket E \rrbracket \Gamma \text{ tmp} ; \mathbf{r} \leftarrow \text{tmp}[1]$
$\mathcal{E} \llbracket \text{snd } E \rrbracket \Gamma \mathbf{r}$	$\mathcal{E} \llbracket E \rrbracket \Gamma \text{ tmp} ; \mathbf{r} \leftarrow \text{tmp}[2]$
$\mathcal{E} \llbracket \lambda(x:T). F \rrbracket \Gamma \mathbf{r}$	$\mathbf{r} \leftarrow \text{new code, } \Gamma(v_1), \dots, \Gamma(v_n)$ <i>(see text)</i>
$\mathcal{E} \llbracket \text{case } E_1 \{ (x)E_2 ; (y)E_3 \} \rrbracket \Gamma \mathbf{r}$	$\mathcal{E} \llbracket E_1 \rrbracket \Gamma \text{ tmp}$ $\text{if tmp}[0] = \text{tagLeft}$ $\quad \text{then } \mathcal{E} \llbracket E_2 \rrbracket (\Gamma[\text{tmp}[1]/x]) \mathbf{r}$ $\quad \text{else } \mathcal{E} \llbracket E_3 \rrbracket (\Gamma[\text{tmp}[1]/y]) \mathbf{r}$

Fig. 8. AM compilation rules

Fig. 8 lists the rules for compiling an IL expression to an instruction sequence. There is a separate set of rules for non-0 types (\mathcal{E}) and type 0 (\mathcal{F}). The \mathcal{E} rules take an extra parameter \mathbf{r} , the register which receives the result value.

Within the context of each expansion, register names beginning **tmp** are instantiated with fresh temporary register names. Each such register is assigned to exactly once; thus the code is in SSA form as regards temporary registers. This does not apply to the special registers **env** and **arg**, which are overwritten just before each **jump** instruction. Note that there is no need to save the old values of **env** and **arg** or of any temporary register, since continuations never return.

When expanding $\mathcal{E} \llbracket \lambda x. F \rrbracket \Gamma \mathbf{r}$, the compiler also expands

$$\mathcal{F} \llbracket F \rrbracket (x \mapsto \text{arg}, y_1 \mapsto \text{env}[1], \dots, y_n \mapsto \text{env}[n]),$$

where $\{y_1, \dots, y_n\} = \text{fv}(\lambda x. F)$, and places the expanded code somewhere in the code segment. The local (fresh) constant register **code** is set to the entry point of this code.

4.2 Updating

There is one optimization that every non-strict language implementation must perform, because compiled code may run exponentially slower without it. This is *thunk memoization* or *thunk updating*, and it is the difference between lazy

evaluation and normal-order reduction. It is described, for example, in [4]. The details of this process are ugly, and a full treatment would complicate IL and AM significantly; but we cannot ignore it entirely, since it interacts non-trivially with the subtyping system of the next section.

For our purposes, a *thunk* is a heap object constructed by executing $\mathcal{E} \llbracket E \rrbracket Fr$, where E was produced by the $\llbracket e \rrbracket_v$ rule during translation from LL to IL. Such an expression has the form $\lambda(k : \neg T). F$, where F either diverges or computes a value V and passes it to k . If F successfully computes a value, then before passing that value to k we must *update* the thunk by physically overwriting its heap representation with a new object equivalent to $(\lambda(k : \neg T). k \bowtie V)$. The new object is indistinguishable from the old as far as the programmer is concerned: we know, by virtue of having just evaluated F , that its effect is just $k \bowtie V$. (And since IL is referentially transparent it cannot have any side effect.)

We might model updating by extending AM with a new instruction

$$M ::= \dots$$

$ \mathbf{x}[i] \leftarrow \mathbf{y}$	Indexed store: Overwrites the word at index i in the heap object pointed to by \mathbf{x} with the word in \mathbf{y} .
---	---

in terms of which the updating step may be written `thunk[0] ← ret_payload`; `thunk[1] ← val`, where `thunk` points to the heap object to be updated, `val` points to the heap representation of V , and `ret_payload` points to the code $\mathcal{F} \llbracket k \bowtie v \rrbracket$ ($k \mapsto \mathbf{arg}, v \mapsto \mathbf{env}[1]$). When `ret_payload` is called, `env[1]` will contain the computed value that we stored in `thunk[1]`. This form of updating is similar to updating with an *indirection node* in GHC [4].

5 Subtyping and Auto-Lifting

It turns out that with a small variation in the design of AM, the natural embedding from unlifted to lifted types becomes a subtyping relationship, allowing us to treat any ML value as a Haskell value at runtime without cost, and project a Haskell value onto an ML type with only the cost of a Haskell `deepSeq`.

Suppose that we have a value V of type T , and wish to construct a value V' of type $\neg\neg T$, most likely for the purpose of marshalling data from eager to lazy code. If $T = (T_1 \& T_2)$, then V can only be (V_1, V_2) for some values V_1 and V_2 . Then $V' = \lambda k. k \bowtie (V_1, V_2)$. But we cannot compile a fresh addressable instruction sequence $k \bowtie (V_1, V_2)$ for each V_1 and V_2 , since V_1 and V_2 are not known at compile time. Instead we compile a single instruction sequence $\mathcal{F} \llbracket k \bowtie (v_1, v_2) \rrbracket$ ($k \mapsto \mathbf{arg}, v_1 \mapsto \mathbf{env}[1], v_2 \mapsto \mathbf{env}[2]$) and place pointers to V_1 and V_2 in the appropriate environment slots at run time.

Similarly, if $T = (T_1 \vee T_2)$, then V is `inl` V_1 or `inr` V_2 , so we compile $\mathcal{F} \llbracket k \bowtie \mathbf{inl} \ v_1 \rrbracket$ ($k \mapsto \mathbf{arg}, v_1 \mapsto \mathbf{env}[1]$) and $\mathcal{F} \llbracket k \bowtie \mathbf{inr} \ v_2 \rrbracket$ ($k \mapsto \mathbf{arg}, v_2 \mapsto \mathbf{env}[1]$), and place V_1 or V_2 in the environment slot.

In short, the lifted pair $(\lambda k. k \bowtie (V_1, V_2))$ will be represented on the heap by an object of three words, the first being a pointer to the code for $k \bowtie (v_1, v_2)$ and the second and third being pointers to V_1 and V_2 . The lifted left injection will

be represented by an object of two words, the first being a pointer to the code for $k \bowtie \text{inl } v_1$ and the second being a pointer to V_1 ; and similarly for the right injection. These heap objects are the same size as the heap objects we would construct for the unlifted values V ; except for the first word, the layout is the same; and since we compiled just three instruction sequences, the first word of the lifted values can contain just three different pointers, which are in one-to-one correspondence with the three tags `tagPair`, `tagLeft`, `tagRight`. So if we simply *define* our sum and product tags to be pointers to the appropriate instruction sequence, then a heap representation of any value of an IL sum or product type is also a value of that type’s double negation. We will call this *auto-lifting*.

Auto-lifting also works for `tagUnit`, but a slightly different approach is needed for function types, and more generally for any type beginning with \neg . Further discussion of this may be found at the web site [7].

An obvious but nonetheless interesting observation about auto-lifting is that it is often *polynomially* faster than explicit lifting. Explicitly converting from a strict list to a non-strict list in IL is $\Theta(n)$ in the size of the list, while auto-lifting is free of cost independently of n .

5.1 Coercing LL Values to SL Values

The function `deepSeq` is not built in to Haskell, but can be defined using type classes. Its operational effect is to traverse a data structure, *forcing* each node as it goes. By forcing we mean, in IL terms, calling each continuation $\neg\neg T$ and ignoring the result T (except for purposes of further traversal). Applying `deepSeq` to a data structure has the referentially transparent “side effect” of causing all nodes in the data structure to be updated with values of the form $\lambda k. k \bowtie V$ (see Section 4.2). If there is no such value—if \perp is hiding anywhere in the data structure—then `deepSeq` diverges.

We have already arranged that a fully-evaluated LL value *is* an SL value in AM. It would seem that if we define our forcing operation in such a way that it overwrites thunks with valid SL values, then provided `deepSeq` does not diverge, we could subsequently treat its argument as having the corresponding SL type.

Unfortunately, this does not quite work. The trouble is that we cannot always overwrite a thunk with a valid SL value. Consider, for example, the thunk $\lambda k. k \bowtie (x, x)$. This has one free variable (x) , and so its heap representation in AM occupies two words (the other being the code pointer). Its SL counterpart, on the other hand, requires *three* words (one for the tag and two for the fields of the pair). We can solve this in some cases by setting aside extra space when we allocate the thunk, but this is not always possible in a practical implementation with larger tuples and polymorphism. To handle the remaining cases, we are forced to (re-)introduce indirection nodes. But indirection nodes are not SL values!

Fortunately, the solution is not difficult. We must think of `deepSeq` not as a procedure but as a function that returns an SL value as its result. If in-place updating is possible, `deepSeq` returns its argument (which is then a valid SL value); if in-place updating is not possible, `deepSeq` updates with an indirection and returns the *target* of that indirection, which is again a valid SL value.

A related complication arises when we use `deepSeq` on a data structure with a mixture of strict and non-strict fields, such as might be defined in a hybrid language. In such cases we must update not only thunks but also the fields of SL values. Because of space constraints we do not discuss the details.

6 Conclusions

In this paper we defined an intermediate language IL, containing continuations but not functions, which can encode naturally both strict and non-strict languages; and we exhibited an abstract machine, AM, which can execute IL (and, via translation, strict and non-strict source languages) with an efficiency comparable to existing ML and Haskell implementations modulo known optimization techniques, while also supporting efficient interconversion between ML data structures and Haskell data structures.

IL seems to capture fundamental aspects of the relationship between strict and non-strict languages which we had previously understood only in an ad hoc manner. The fact that a structure resembling a lifting monad appears within IL, without any attempt to place it there (Section 2.3) is one example of this. In fact IL’s three negation classes do the lifting monad one better, since they predict that lifting leads to a semantic distinction (in the sense of Section 2.3) only in a value context, not in an expression (continuation) context. It follows that, in a hybrid strict/lazy language, it makes sense to annotate the strictness of function *arguments*, but not of function *results*—a fact that we recognized long before discovering IL, but for which we had never had a satisfactory theoretical model. In this and other ways IL seems to be predictive where previous systems were phenomenological, and this is its primary appeal.

6.1 Related Work

The benefits of continuation-passing style for compilation, and the existence of call-by-name and call-by-value CPS translations, have been known for decades [6]. The notion of continuations as negations was introduced by Griffin [3]. Recently several authors have introduced computational calculi to demonstrate the call-by-value/call-by-name duality within a classical framework, including Curien and Herbelin’s lambda-bar calculus [2], Wadler’s dual calculus [9], and van Bakel, Lengrand, and Lescanne’s \mathcal{X} [8]. Wadler explicitly defines the function arrow in terms of negation, conjunction and disjunction. On the practical side, a previous paper by Peyton Jones, Launchbury, Shields, and Tolmach [5] studies the same practical problem as the present work, proposing a monad-based intermediate language also called IL.

The present work represents, we believe, a happy medium between the theoretical and practical sides of the problem we set out to solve. IL is straightforwardly and efficiently implementable on stock hardware, while retaining some of the interesting features of its theoretical cousins; it is also substantially simpler than the previous proposal by Peyton Jones et al, while preserving its

fundamental design goal. The notion of auto-lifting described in this paper may also be new to the literature, though it was known to the authors before this research began.

6.2 Future Work

The work described in this paper is part of an ongoing research project. Again we invite interested readers to visit the project web site [7], which will contain additional material omitted from this paper as well as updates on further progress.

As of this writing, and undoubtedly as of publication time, a large amount of work remains to be done. We have not implemented a compiler based on IL and AM, and many issues must be investigated and resolved before we can do so. Some optimizations can be accommodated quite well within IL as it stands—for example, the “vectored return” optimization of the STG-machine [4] is valid as a consequence of de Morgan’s law. Others require further work. The minimalist design of AM can accommodate extensions for stack-based evaluation, register arguments and the like, if these can be represented neatly in IL. Since the use of continuation-passing calculi as intermediate languages is well understood [1], it seems likely that this can be done using known techniques.

Adding polymorphism to IL is not difficult, and the translation from the source language to IL remains straightforward as long as the source language is strict or non-strict. Unfortunately, attempts to introduce unrestricted polymorphism into a hybrid language lead to subtle difficulties. IL does not cause these difficulties, but only exposes them; we hope that further study of IL will expose a sensible solution as well.

Acknowledgements

This work was supported by a studentship from Microsoft Research Cambridge. We are grateful to Philip Wadler and the anonymous reviewers for helpful comments.

References

1. Andrew W. Appel, *Compiling With Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7.
2. Pierre-Louis Curien and Hugo Herbelin, *The duality of computation*. Proc. ICFP 2000. <http://pauillac.inria.fr/~herbelin/habilitation/icfp-CurHer00-duality+errata.ps>
3. Timothy G Griffin, *A formulae-as-types notion of control*. Proc. POPL 1990.
4. Simon Peyton Jones, *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*. Journal of Functional Programming, 2(2):127–202, 1992.

5. Simon Peyton Jones, John Launchbury, Mark Shields, and Andrew Tolmach, *Bridging the gulf: a common intermediate language for ML and Haskell*. Proc. POPL 1998. http://www.cartesianclosed.com/pub/intermediate_language/neutral.ps
6. Gordon D. Plotkin, *Call-by-name, call-by-value and the λ -calculus*. Theoretical Computer Science 1:125–159, 1975. http://homepages.inf.ed.ac.uk/gdp/publications/cbn_cbv_lambda.pdf
7. Ben Rudiak-Gould, *The NotNotML project website*. <http://www.cl.cam.ac.uk/~br276/notnotML/>
8. Steffen van Bakel, Stéphane Lengrand, and Pierre Lescanne, *The language \mathcal{X} : circuits, computations and Classical Logic*. Proc. ICTCS 2005. <http://www.doc.ic.ac.uk/~svb/Research/Abstracts/vBLL.html>
9. Philip Wadler, *Call-by-value is dual to call-by-name*. Proc. ICFP 2003, pp. 189–201. <http://homepages.inf.ed.ac.uk/wadler/papers/dual/dual.pdf>

Coinductive Big-Step Operational Semantics

Xavier Leroy

INRIA Rocquencourt,
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France
Xavier.Leroy@inria.fr

Abstract. This paper illustrates the use of coinductive definitions and proofs in big-step operational semantics, enabling the latter to describe diverging evaluations in addition to terminating evaluations. We show applications to proofs of type soundness and to proofs of semantic preservation for compilers.

1 Introduction

There exist two popular styles of structured operational semantics: big-step semantics, relating programs to final configurations, and small-step semantics, where a one-step reduction relation is repeatedly applied to form reduction sequences. Small-step semantics is more expressive since it can describe the evaluation of both terminating and non-terminating programs, as finite or infinite reduction sequences, respectively. In contrast, big-step semantics describes only the evaluation of terminating programs, and fails to distinguish between non-terminating programs and programs that “go wrong”. For this reason, small-step semantics is generally preferred, in particular for proving the soundness of type systems.

However, big-step semantics is more convenient than small-step semantics for some applications. One that is dear to our heart is proving the correctness (preservation of program behaviours) of program transformations, especially compilation of a high-level language down to a lower-level language. Our experience and that of others [14, 12, 19] is that fairly complex, optimizing compilation passes can be proved correct relatively easily using big-step semantics, by induction on the structure of big-step evaluation derivations. In contrast, compiler correctness proofs using small-step semantics are significantly harder even for simple, non-optimizing compilation schemes [10, 8].

In this paper, we illustrate how coinductive definitions and proofs enable big-step semantics to describe both finite and infinite evaluations. The target of our study is a simple call-by-value functional language. We study two approaches: the first, initially proposed by Cousot and Cousot [4], complements the normal inductive big-step evaluation rules for finite evaluations with coinductive big-step rules describing diverging evaluations; the second simply interprets coinductively the normal big-step evaluation rules, thus enabling them to describe both terminating and non-terminating evaluations. These semantics are

defined in section 2. The main technical results of the paper are: connections between coinductive big-step semantics and finite or infinite reduction sequences in small-step semantics (section 3); a novel approach to stating and proving the soundness of type systems (section 4); and proofs of semantic preservation for compilation down to an abstract machine (section 5).

An originality of this paper is that all results were not only proved using a proof assistant (the Coq system), but even developed in interaction with this tool, and only then transcribed to standard mathematical notations in this paper. The Coq proof assistant [3] provides built-in support for coinductive definitions and proofs by a limited form of coinduction called guarded structural coinduction. (See [6, 2, 3] for descriptions of this approach to coinduction.) Such proofs are easier than the standard, on-paper proofs by coinduction; in particular, there is no need to exhibit F -consistent relations [5]. This enables us to play fast and loose with coinduction in the proof sketches given in this paper; the skeptical reader is referred to the corresponding Coq development [13] for details. Another benefit of using Coq is that our formalization and proofs use rather modest mathematics: just syntactic definitions, no domain theory, and constructive logic plus the axiom of excluded middle from classical logic. (The proofs that use excluded middle are marked “*classical*”.)

2 The Language and Its Big-Step Semantics

The language we consider in this paper is the λ -calculus extended with constants: the simplest functional language that exhibits run-time errors (terms that “go wrong”). Its syntax is as follows:

Variables: x, y, z, \dots

Constants: $c ::= 0 \mid 1 \mid \dots$

Terms: $a, b, v ::= x \mid c \mid \lambda x. a \mid a \ b$

We write $a[x \leftarrow b]$ for the capture-avoiding substitution¹ of b for all free occurrences of x in a . We say that a term v is a value, and write $v \in \text{Values}$, if a is either a constant c or an abstraction $\lambda x. b$.

The standard call-by-value semantics in big-step style for this language is defined by the following inference rules, interpreted inductively.

$$\begin{array}{c}
 c \Rightarrow c \quad (\Rightarrow\text{-const}) \qquad \qquad \qquad \lambda x. a \Rightarrow \lambda x. a \quad (\Rightarrow\text{-fun}) \\
 \\
 \frac{a_1 \Rightarrow \lambda x. b \quad a_2 \Rightarrow v_2 \quad b[x \leftarrow v_2] \Rightarrow v}{a_1 \ a_2 \Rightarrow v} \quad (\Rightarrow\text{-app})
 \end{array}$$

More precisely, the relation $a \Rightarrow v$ (read: “ a evaluates to v ”) is the smallest fixpoint of the rules above. Equivalently, $a \Rightarrow v$ holds if and only if it is the conclusion of a finite derivation tree built from the rules above.

¹ The Coq development does not treat terms modulo α -conversion, therefore the substitution $a[x \leftarrow b]$ is capture-avoiding only if b is closed. However, this suffices to define evaluation and reduction of closed source terms.

Lemma 1. *If $a \Rightarrow v$, then $v \in \text{Values}$.*

Proof sketch. Induction on the derivation of $a \Rightarrow v$.

The rules above capture only terminating evaluations. Writing $\delta = \lambda x. x$ and $\omega = \delta \delta$, we have for instance:

Lemma 2. *$\omega \Rightarrow v$ is false for all terms v .*

Proof sketch. We show that $a \Rightarrow v$ implies $a \neq \omega$ by induction on the derivation of $a \Rightarrow v$.

Following Cousot and Cousot [4] and more recent work by Grall [7], we define divergence (infinite evaluations) by the following inference rules, interpreted coinductively:²

$$\begin{array}{c}
 \frac{a_1 \overset{\infty}{\Rightarrow}}{a_1 \ a_2 \overset{\infty}{\Rightarrow}} \ (\overset{\infty}{\Rightarrow}\text{-app-l}) \qquad \frac{a_1 \Rightarrow v \quad a_2 \overset{\infty}{\Rightarrow}}{a_1 \ a_2 \overset{\infty}{\Rightarrow}} \ (\overset{\infty}{\Rightarrow}\text{-app-r}) \\
 \\
 \frac{a_1 \Rightarrow \lambda x.b \quad a_2 \Rightarrow v \quad b[x \leftarrow v] \overset{\infty}{\Rightarrow}}{a_1 \ a_2 \overset{\infty}{\Rightarrow}} \ (\overset{\infty}{\Rightarrow}\text{-app-f})
 \end{array}$$

More precisely, the relation $a \overset{\infty}{\Rightarrow}$ (read: “ a diverges”) is the greatest fixpoint of the rules above, or, equivalently, the conclusions of infinite derivation trees built from these rules. Note that we have imposed (arbitrarily) a left-to-right evaluation order for applications.

Lemma 3. *$\omega \overset{\infty}{\Rightarrow}$ holds.*

Proof sketch. By coinduction. Assume $\omega \overset{\infty}{\Rightarrow}$ as coinduction hypothesis. We can derive $\omega \overset{\infty}{\Rightarrow}$ with rule ($\overset{\infty}{\Rightarrow}$ -app-f), using the coinduction hypothesis as third premise.

Lemma 4. *$a \Rightarrow v$ and $a \overset{\infty}{\Rightarrow}$ are mutually exclusive.*

Proof sketch. By induction on the derivation of $a \Rightarrow v$ and inversion on $a \overset{\infty}{\Rightarrow}$.

An alternate attempt to describe both terminating and non-terminating evaluations at the same time is to interpret coinductively the standard evaluation rules for terminating evaluations.

$$\begin{array}{c}
 c \overset{\infty}{\Rightarrow} c \ (\overset{\infty}{\Rightarrow}\text{-const}) \qquad \lambda x.a \overset{\infty}{\Rightarrow} \lambda x.a \ (\overset{\infty}{\Rightarrow}\text{-fun}) \\
 \\
 \frac{a_1 \overset{\infty}{\Rightarrow} \lambda x.b \quad a_2 \overset{\infty}{\Rightarrow} v_2 \quad b[x \leftarrow v_2] \overset{\infty}{\Rightarrow} v}{a_1 \ a_2 \overset{\infty}{\Rightarrow} v} \ (\overset{\infty}{\Rightarrow}\text{-app})
 \end{array}$$

² Throughout this paper, double horizontal lines in inference rules denote inference rules that are to be interpreted coinductively; single horizontal lines denote the inductive interpretation.

The relation $a \overset{\text{co}}{\Rightarrow} b$ (read: “ a coevaluates to b ”) is therefore the greatest fixpoint of the standard evaluation rules. It holds if and only if $a \overset{\text{co}}{\Rightarrow} b$ is the conclusion of a finite *or infinite* derivation tree built from these rules.

Naively, we could expect that $\overset{\text{co}}{\Rightarrow}$ is the union of \Rightarrow and $\overset{\infty}{\Rightarrow}$. This intuition is supported by the following properties:

Lemma 5. *If $a \Rightarrow v$, then $a \overset{\text{co}}{\Rightarrow} v$.*

Proof sketch. By induction on the derivation of $a \Rightarrow v$.

Lemma 6. $\omega \overset{\text{co}}{\Rightarrow} v$ for all terms v .

Proof sketch. By coinduction, using rule ($\overset{\text{co}}{\Rightarrow}$ -app) with the coinduction hypothesis as third premise.

Lemma 7. *If $a \overset{\text{co}}{\Rightarrow} v$, then either $a \Rightarrow v$ or $a \overset{\infty}{\Rightarrow}$.*

Proof sketch (classical). We show that $a \overset{\text{co}}{\Rightarrow} v$ and $\neg(a \Rightarrow v)$ implies $a \overset{\infty}{\Rightarrow}$. The result then follows by excluded middle on $a \Rightarrow v$. The auxiliary property is proved by coinduction and case analysis on a . The cases for variables, constants and abstractions trivially contradict one of the hypotheses. If $a = a_1 a_2$, inversion on the hypothesis $a \overset{\text{co}}{\Rightarrow} v$ shows that $a_1 \overset{\text{co}}{\Rightarrow} \lambda x.b$ and $a_2 \overset{\text{co}}{\Rightarrow} v_2$ and $b[x \leftarrow v_2] \overset{\text{co}}{\Rightarrow} v$. Using excluded middle, it must be that at least one of these three terms does not evaluate, otherwise, $a \Rightarrow v$ would hold. The result follows by applying the rule for $\overset{\infty}{\Rightarrow}$ that matches which term does not evaluate, and using the coinduction hypothesis.

However, the reverse implication does not hold: there exists terms that diverge but do not coevaluate. Consider for instance $a = \omega (0\ 0)$. It is true that $a \overset{\infty}{\Rightarrow}$, but there is no term v such that $a \overset{\text{co}}{\Rightarrow} v$, because the coevaluation of the argument $0\ 0$ goes wrong (there is no v such that $0\ 0 \overset{\text{co}}{\Rightarrow} v$).

Another unusual feature of coevaluation is that it is not deterministic. For instance, $\omega \overset{\text{co}}{\Rightarrow} v$ for any term v . However, $\overset{\text{co}}{\Rightarrow}$ is deterministic for terminating terms, in the following sense:

Lemma 8. *If $a \Rightarrow v$ and $a \overset{\text{co}}{\Rightarrow} v'$, then $v' = v$.*

Proof sketch. By induction on the derivation of $a \Rightarrow v$ and inversion on $a \overset{\text{co}}{\Rightarrow} v'$.

Moreover, there exists diverging terms that coevaluate to only one value. An example is $(\lambda x.0)\ \omega$, which coevaluates to 0 but not to any other term.

3 Relation with Small-Step Semantics

The one-step reduction relation \rightarrow is defined by the call-by-value β -reduction axiom plus two context rules for reducing under applications, assuming left-to-right evaluation order.

$$\begin{array}{c}
\frac{v \in \text{Values}}{(\lambda x.a) v \rightarrow a[x \leftarrow v]} \quad (\rightarrow\beta) \\
\\
\frac{a_1 \rightarrow a_2}{a_1 b \rightarrow a_2 b} \quad (\rightarrow\text{-app-l}) \qquad \frac{a \in \text{Values} \quad b_1 \rightarrow b_2}{a b_1 \rightarrow a b_2} \quad (\rightarrow\text{-app-r})
\end{array}$$

There are three kinds of reduction sequences of interest. The first, written $a \xrightarrow{*} b$ (“ a reduces to b in zero, one or several steps”), is the normal reflexive transitive closure of \rightarrow ; it captures finite reductions. The second, $a \xrightarrow{\infty} b$ (“ a reduces infinitely”) captures infinite reductions. The third, $a \xrightarrow{\text{co}*} b$ (“ a reduces to b in zero, one, several or infinitely many steps”) is the coinductive interpretation of the rules for reflexive transitive closure; it captures both finite and infinite reductions. These relations are defined by the following rules, interpreted inductively for $\xrightarrow{*}$ and coinductively for $\xrightarrow{\infty}$ and $\xrightarrow{\text{co}*}$.

$$\begin{array}{ccc}
\frac{a \xrightarrow{*} a}{a \rightarrow b \quad b \xrightarrow{*} c} \quad a \xrightarrow{*} c & \frac{a \rightarrow b \quad b \xrightarrow{\infty}}{a \xrightarrow{\infty}} & \frac{a \rightarrow b \quad b \xrightarrow{\text{co}*} c}{a \xrightarrow{\text{co}*} c}
\end{array}$$

In contrast with the evaluation predicates of section 2, it is true that $\xrightarrow{\text{co}*}$ is the union of $\xrightarrow{*}$ and $\xrightarrow{\infty}$.

Lemma 9. $a \xrightarrow{\text{co}*} b$ if and only if $a \xrightarrow{*} b$ or $a \xrightarrow{\infty}$.

Proof sketch (classical). For the “if” part, we show that $a \xrightarrow{*} b \implies a \xrightarrow{\text{co}*} b$ by induction on $a \xrightarrow{*} b$, and that $a \xrightarrow{\infty} \implies a \xrightarrow{\text{co}*} b$ by coinduction. For the “only if” part, we show that $a \xrightarrow{\text{co}*} b \wedge \neg(a \xrightarrow{*} b) \implies a \xrightarrow{\infty}$ by coinduction. The result follows by excluded middle over $a \xrightarrow{*} b$.

We now turn to relating the reduction relations (small-step) and the evaluation relations (big-step). (Some of these results were proved earlier on paper by Grall [7], using a variant of the F -consistent relation approach.) It is well known that normal evaluation is equivalent to finite reduction to a value:

Lemma 10. $a \Rightarrow v$ if and only if $a \xrightarrow{*} v$ and $v \in \text{Values}$.

Proof sketch. The “only if” part is an easy induction on $a \Rightarrow v$. For the “if” part, we first show the following two lemmas: (1) $v \Rightarrow v$ if $v \in \text{Values}$, and (2) $a \Rightarrow v$ if $a \rightarrow b$ and $b \Rightarrow v$. The result follows by induction on the derivation of $a \xrightarrow{*} v$.

Similarly, divergence ($\xrightarrow{\infty}$) is equivalent to infinite reduction ($\xrightarrow{\infty}$). The proof uses the following lemma:

Lemma 11. For all terms a , either $a \xrightarrow{\infty}$, or there exists b such that $a \xrightarrow{*} b$ and $b \not\rightarrow$, that is, $\forall c, \neg(b \rightarrow c)$.

Proof sketch (classical). We first show that $\forall b, a \xrightarrow{*} b \implies \exists c, b \rightarrow c$ implies $a \xrightarrow{\infty}$ by coinduction. We then argue by excluded middle on $a \xrightarrow{\infty}$.

Lemma 12. $a \xrightarrow{\infty}$ if and only if $a \xrightarrow{\infty}$.

Proof sketch (classical). For the “only if” part, we first show that $a \xrightarrow{\infty}$ implies $\exists b, a \rightarrow b \wedge b \xrightarrow{\infty}$ by structural induction on a , then conclude by coinduction. For the “if” part, we proceed by coinduction and case analysis over a . The only non-trivial case is $a = a_1 a_2$. Using lemma 11, we distinguish three cases: (1) a_1 reduces infinitely; (2) a_1 reduces to a value but a_2 reduces infinitely; (3) a_1 and reduce to values $\lambda x.b$ and v respectively, and $b[x \leftarrow v]$ reduces infinitely. The result $a \xrightarrow{\infty}$ then follows from the coinduction hypothesis in all three cases.

For coevaluations $\xrightarrow{\infty}$ and coreductions $\xrightarrow{\text{co}*}$, the equivalence holds in one direction only.

Lemma 13. $a \xrightarrow{\infty} v$ implies $a \xrightarrow{\text{co}*} v$.

Proof sketch. Using classical logic, this follows from lemmas 7, 10, 12 and 9. However, the result can be proved directly in constructive logic. We first show that $a \xrightarrow{\infty} v \implies a \in \text{Values} \vee \exists b, a \rightarrow b \wedge b \xrightarrow{\infty} v$ by induction on a . The result follows by coinduction.

An example where the reverse implication does not hold is $a = (\lambda x. 0) \omega$ and $v = 1$. Since $a \xrightarrow{\infty}$, we have $a \xrightarrow{\text{co}*} v$. However, $a \xrightarrow{\infty} v$ does not hold since the only term to which a coevaluates is 0.

4 Type Soundness Proofs

We now turn to using our coinductive evaluation and reduction relations for proving the soundness of type systems. To be more specific, we will use the simply-typed λ -calculus with recursive types as our type system. We obtain recursive types by interpreting the type algebra $\tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2$ coinductively, as in [5]. The typing rules are recalled below. Γ ranges over type environments, that is, finite maps from variables to types.

$$\begin{array}{c}
 \dfrac{E(x) = \tau}{E \vdash x : \tau} \qquad \qquad \qquad E \vdash c : \text{int} \\
 \\
 \dfrac{E + \{x : \tau'\} \vdash a : \tau}{E \vdash \lambda x. a : \tau' \rightarrow \tau} \qquad \qquad \qquad \dfrac{E \vdash a_1 : \tau' \rightarrow \tau \quad E \vdash a_2 : \tau'}{E \vdash a_1 a_2 : \tau}
 \end{array}$$

Enabling recursive types makes the type system non-normalizing and allows interesting programs to be written. In particular, the call-by-value fixpoint operator $Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (\lambda y. (x x) y))$ is well-typed, with types $((\tau \rightarrow \tau') \rightarrow \tau \rightarrow \tau') \rightarrow \tau \rightarrow \tau'$ for all types τ, τ' .

4.1 Type Soundness Proofs Using Small-Step Semantics

Felleisen and Wright [20] introduced a proof technique for showing type soundness that relies on small-step semantics and is standard nowadays. The proof relies on the twin properties of *type preservation* (also called *subject reduction*) and *progress*:

Lemma 14 (Preservation). *If $a \rightarrow b$ and $\emptyset \vdash a : \tau$, then $\emptyset \vdash b : \tau$*

Lemma 15 (Progress). *If $\emptyset \vdash a : \tau$, then either $a \in \text{Values}$ or $\exists b, a \rightarrow b$.*

The formal statement of type soundness in Felleisen and Wright’s approach is the following:

Lemma 16 (Type soundness, 1). *If $\emptyset \vdash a : \tau$ and $a \xrightarrow{*} b$, then either $a \in \text{Values}$ or there exists b such that $a \rightarrow b$.*

Proof sketch. We first show that $\emptyset \vdash b : \tau$ by induction over $a \xrightarrow{*} b$, using the preservation lemma. We then conclude with the progress lemma.

Authors that follow this approach then conclude that well-typed closed terms either reduce to a value or reduce infinitely. However, this conclusion is generally not expressed nor proved formally. In our approach, it is easy to do so:

Lemma 17 (Type soundness, 2). *If $\emptyset \vdash a : \tau$, then either $a \xrightarrow{\infty}$, or there exists v such that $a \xrightarrow{*} v$ and $v \in \text{Values}$.*

Proof sketch (classical). By lemma 11, either $a \xrightarrow{\infty}$ or $\exists v, a \xrightarrow{*} v \wedge v \not\rightarrow$. The result is obvious in the first case. In the second case, we note that $\emptyset \vdash v : \tau$ as a consequence of the preservation lemma, then use the progress lemma to conclude that $v \in \text{Values}$.

An alternate, equivalent formulation of this theorem uses the coreduction relation $\xrightarrow{\text{co}*}$.

Lemma 18 (Type soundness, 3). *If $\emptyset \vdash a : \tau$, then there exists v such that $a \xrightarrow{\text{co}*} v$ and $v \in \text{Values}$.*

Proof sketch (classical). Follows from lemmas 17 and 9.

An arguably nicer characterisation of “programs that do not go wrong” is given by the relation $a \xrightarrow{\text{safe}}$ (read: “ a reduces safely”), defined coinductively by the following rules:

$$\frac{v \in \text{Values}}{v \xrightarrow{\text{safe}}} \qquad \frac{a \rightarrow b \quad b \xrightarrow{\text{safe}}}{a \xrightarrow{\text{safe}}}$$

These rules are interpreted coinductively so that $a \xrightarrow{\text{safe}}$ holds if a reduces infinitely. We can then state and show type soundness without recourse to classical logic:

Lemma 19 (Type soundness, 4). *If $\emptyset \vdash a : \tau$, then $a \xrightarrow{\text{safe}}$.*

Proof sketch. By coinduction. Applying the progress lemma, either $a \in \text{Values}$ and we are done, or $a \rightarrow b$ for some b . In the latter case, $\emptyset \vdash b : \tau$ by the preservation property, and the result follows from the coinduction hypothesis.

4.2 Type Soundness Proofs Using Big-Step Semantics

The standard big-step semantics (the \Rightarrow relation) is awkward for proving type soundness because it does not distinguish between terms that diverge and terms that go wrong: in both cases, there is no value v such that $a \Rightarrow v$. Consequently, the obvious type soundness statement “if $\emptyset \vdash a : \tau$, there exists v such that $a \Rightarrow v$ ” is false for all type systems that do not guarantee normalization. The best result we can prove, then, is the following big-step equivalent to the preservation lemma:

Lemma 20. *If $a \Rightarrow v$ and $\emptyset \vdash a : \tau$, then $\emptyset \vdash v : \tau$.*

The standard approach is to provide inductive inference rules to define a predicate $a \Rightarrow \text{err}$ characterizing terms that go wrong, and prove the weaker type soundness statement “if $\emptyset \vdash a : \tau$, then it is not the case that $a \Rightarrow \text{err}$ ”. This approach is unsatisfactory for two reasons: (1) extra rules must be provided to define $a \Rightarrow \text{err}$, which increases the size of the semantics; (2) there is a risk that the rules for $a \Rightarrow \text{err}$ are incomplete and miss some cases of “going wrong”, in which case the type soundness statement does not guarantee that well-typed terms either evaluate to a value or diverge.

Let us revisit these trade-offs in the light of our characterizations of divergence and coevaluation. We can now formally state what it means for a term to evaluate or to diverge. This leads to the following alternate statement of type soundness:

Lemma 21 (Type soundness, 5). *If $\emptyset \vdash a : \tau$, then either $a \xRightarrow{\infty}$ or there exists v such that $a \Rightarrow v$.*

This result follows from the lemma below (a big-step analogue to the progress lemma) and from excluded middle applied to $\exists v. a \Rightarrow v$.

Lemma 22. *If $\emptyset \vdash a : \tau$ and $\forall v, \neg(a \Rightarrow v)$, then $a \xRightarrow{\infty}$.*

Proof sketch (classical). The proof is by coinduction and case analysis over a . The cases $a = x$, $a = c$ and $a = \lambda x.b$ lead to contradictions: variables are not typeable in the empty environment; constants and abstractions evaluate to themselves. The interesting case is therefore $a = a_1 a_2$. By excluded middle, either a_1 evaluates to some value v_1 , or not. In the latter case, $a \xRightarrow{\infty}$ follows from rule ($\xRightarrow{\infty}$ -app-l) and from $a_1 \xRightarrow{\infty}$, which we obtain by coinduction hypothesis. In the former case, v_1 has a function type $\tau' \rightarrow \tau$ by lemma 20, and therefore $v_1 = \lambda x.b$ for some x and b . Moreover, $\{x : \tau'\} \vdash b : \tau$. Using excluded middle again, either a_2 evaluates to some value v_2 , or not. In the latter case, $a \xRightarrow{\infty}$

follows from rule ($\overset{\infty}{\Rightarrow}$ -app-r) and the coinduction hypothesis. In the former case, $\emptyset \vdash v_2 : \tau'$. Since typing is stable by substitution, $\emptyset \vdash b[x \leftarrow v_2] : \tau$. Using excluded middle for the third time, it must be that $\forall v. \neg(b[x \leftarrow v_2] \Rightarrow v)$, otherwise a would evaluate to some value. The result $a \overset{\infty}{\Rightarrow}$ then follows from rule ($\overset{\infty}{\Rightarrow}$ -app-f) and the coinduction hypothesis.

The proof above is an original alternative to the standard approach of showing $\neg(a \Rightarrow \text{err})$ for all well-typed terms a . From a methodological standpoint, our proof addresses one of the shortcomings of the standard approach, namely the risk of not putting enough error rules. If we forget some divergence rules, the proof of lemma 22 will, in all likelihood, not go through. Moreover, it is improbable to put too many rules for divergence and still have the property that $a \Rightarrow v$ and $a \overset{\infty}{\Rightarrow}$ are mutually exclusive. Therefore, this novel approach to proving type soundness using big-step semantics appears rather robust with respect to mistakes in the specification of the semantics.

The other methodological shortcoming remains, however: just like the “not goes wrong” approach, our approach requires more evaluation rules than just those for normal evaluations, namely the rules for divergence. This can easily double the size of the specification of a dynamic semantics, which is a serious concern for realistic languages where the normal evaluation rules number in dozens.

The coevaluation relation $\overset{\infty}{\Rightarrow}$ is attractive for these pragmatic reasons, as it has the same number of rules as normal evaluation. Of course, we have seen that $a \overset{\infty}{\Rightarrow} v$ is not equivalent to $a \Rightarrow v \vee a \overset{\infty}{\Rightarrow}$, but the example we gave was for a term a that is not typeable and where an early diverging evaluation “hides” a later evaluation that goes wrong. Since type systems ensure that all subterms of a term do not go wrong, we could hope that the following conjecture holds:

Conjecture 1 (Type soundness, 6). If $\emptyset \vdash a : \tau$, there exists v such that $a \overset{\infty}{\Rightarrow} v$.

We were able to prove this conjecture for some uninteresting but nonetheless non-normalizing type systems, such as simply-typed λ -calculus without recursive types, but with a predefined constant of type $\mathbf{int} \rightarrow \mathbf{int}$ that diverges when applied. However, the conjecture is false for simply-typed λ -calculus with recursive types, and probably for all type systems with a general fixpoint operator. Andrzej Filinski provided the following counterexample. Consider

$$Y \ F \ 0 \quad \text{where} \quad F = \lambda f. \lambda x. (\lambda g. \lambda y. g \ y) (f \ x).$$

The term $Y \ F \ 0$ is well-typed with type $\mathbf{int} \rightarrow \mathbf{int}$, yet it fails to coevaluate: the only possible value v such that $Y \ F \ 0 \overset{\infty}{\Rightarrow} v$ is an infinite term, $\lambda y. (\lambda y. (\lambda y. \dots y) \ y) \ y$.

5 Compiler Correctness Proofs

We now return to the original motivation of this work: proving semantic preservation for compilers both for terminating and diverging programs, using big-step semantics. We demonstrate this approach on the compilation of call-by-value λ -calculus down to a simple abstract machine.

5.1 Big-Step Semantics with Environments and Closures

Our abstract machine uses closures and environments indexed by de Bruijn indices. It is therefore convenient to reformulate the big-step evaluation predicates in these terms. Variables, written x_n , are now identified by their de Bruijn indices n . Values (which are no longer a subset of terms) and environments are defined as:

Values: $v ::= c$ integer values
 $\quad \quad \quad | (\lambda a)[e]$ function closures
 Environments: $e ::= \varepsilon \mid v.e$ sequences of values

Figure 1 shows the inference rules defining the three evaluation relations:

$e \vdash a \Rightarrow v$ finite evaluations (inductive)
 $e \vdash a \stackrel{\infty}{\Rightarrow}$ infinite evaluations (coinductive)
 $e \vdash a \stackrel{\text{co}}{\Rightarrow} v$ coevaluations (coinductive)

We will not formally study these relations, but note that they enjoy the same properties as the environment-less relations studied in section 2.

5.2 The Abstract Machine and its Compilation Scheme

The abstract machine we use as target of compilation follows the call-by-value strategy and the “eval-apply” model. It is close in spirit to the SECD, CAM,

$\frac{e = v_1 \dots v_n \dots}{e \vdash x_n \Rightarrow v_n}$	$e \vdash c \Rightarrow c$	$e \vdash \lambda a \Rightarrow (\lambda a)[e]$
$\frac{e \vdash a_1 \Rightarrow (\lambda b)[e'] \quad e \vdash a_2 \Rightarrow v_2 \quad v_2.e' \vdash b \Rightarrow v}{e \vdash a_1 a_2 \Rightarrow v}$		
$\frac{e \vdash a_1 \stackrel{\infty}{\Rightarrow}}{e \vdash a_1 a_2 \stackrel{\infty}{\Rightarrow}}$	$\frac{e \vdash a_1 \Rightarrow v \quad e \vdash a_2 \stackrel{\infty}{\Rightarrow}}{e \vdash a_1 a_2 \stackrel{\infty}{\Rightarrow}}$	
$\frac{e \vdash a_1 \Rightarrow (\lambda b)[e'] \quad e \vdash a_2 \Rightarrow v \quad v.e' \vdash b \stackrel{\infty}{\Rightarrow}}{e \vdash a_1 a_2 \stackrel{\infty}{\Rightarrow}}$		
$\frac{e = v_1 \dots v_n \dots}{e \vdash x_n \stackrel{\text{co}}{\Rightarrow} v_n}$	$e \vdash c \stackrel{\text{co}}{\Rightarrow} c$	$e \vdash \lambda a \stackrel{\text{co}}{\Rightarrow} (\lambda a)[e]$
$\frac{e \vdash a_1 \stackrel{\text{co}}{\Rightarrow} (\lambda b)[e'] \quad e \vdash a_2 \stackrel{\text{co}}{\Rightarrow} v_2 \quad v_2.e' \vdash b \stackrel{\text{co}}{\Rightarrow} v}{e \vdash a_1 a_2 \stackrel{\text{co}}{\Rightarrow} v}$		

Fig. 1. Big-step evaluation rules with closures and environments

FAM and CEK machines. The machine state has three components: a code sequence, a stack and an environment. The syntax for these components is as follows.

Instructions:	$I ::= \mathbf{Var}(n)$ push the value of variable number n $\quad \mid \mathbf{Const}(c)$ push the constant c $\quad \mid \mathbf{Clos}(C)$ push a closure for code C $\quad \mid \mathbf{App}$ perform a function application $\quad \mid \mathbf{Ret}$ return to calling function
Code:	$C ::= \varepsilon \mid I, C$ instruction sequences
Machine values:	$V ::= n$ integer values $\quad \mid C[E]$ code closures
Machine environments:	$E ::= \varepsilon \mid V.E$
Stacks:	$S ::= \varepsilon$ empty stack $\quad \mid V.S$ pushing a value $\quad \mid (C, E).S$ pushing a return frame

The behaviour of the abstract machine is defined by the following rules, as a transition relation $C; S; E \rightarrow C'; S'; E'$ that relates the machine state before $(C; S; E)$ and after $(C'; S'; E')$ the execution of the first instruction of the code C .

$$\begin{array}{llll}
(\mathbf{Var}(n), C); S; & E \rightarrow C; V_n.S; & E & \text{if } E = V_1 \dots V_n \dots \\
(\mathbf{Const}(c), C); S; & E \rightarrow C; c.S; & E & \\
(\mathbf{Clos}(C'), C); S; & E \rightarrow C; C'[E].S; & E & \\
(\mathbf{App}, C); & V.C'[E'].S; & E \rightarrow C'; (C, E).S; & V.E' \\
(\mathbf{Ret}, C); & V.(C', E').S; & E \rightarrow C'; V.S; & E'
\end{array}$$

As in section 3, we consider the following closures of the one-step transition relation:

$$\begin{array}{ll}
C; S; E \xrightarrow{*} C'; S'; E' & \text{zero, one or several transitions (inductive)} \\
C; S; E \xrightarrow{+} C'; S'; E' & \text{one or several transitions (inductive)} \\
C; S; E \xrightarrow{\infty} & \text{infinitely many transitions (coinductive)} \\
C; S; E \xrightarrow{\text{co}*} C'; S'; E' & \text{zero, one, several or infinitely many transitions (coind.)}
\end{array}$$

The compilation scheme from terms to code is straightforward:

$$\begin{array}{ll}
\llbracket x_n \rrbracket = \mathbf{Var}(n) & \llbracket c \rrbracket = \mathbf{Const}(c) \\
\llbracket \lambda a \rrbracket = \mathbf{Clos}(\llbracket a \rrbracket, \mathbf{Ret}) & \llbracket a_1 \ a_2 \rrbracket = \llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket, \mathbf{App}
\end{array}$$

The intended effect for the code $\llbracket a \rrbracket$ is to evaluate the term a and push its value at the top of the machine stack, leaving the rest of the stack and the environment unchanged.

5.3 Proofs of Semantic Preservation

We expect the compilation to abstract machine code to preserve the semantics of the source term, in the following general sense. Consider a closed term a and

start the abstract machine in the initial state corresponding to a . If a diverges, the machine should perform infinitely many transitions. If a evaluates to the value v , the machine should reach a final state corresponding to v in a finite number of transitions. Here, the initial state corresponding to a is $\llbracket a \rrbracket; \varepsilon; \varepsilon$. The final state corresponding to the result value v is $\varepsilon; \llbracket v \rrbracket. \varepsilon; \varepsilon$, that is, the code has been entirely consumed and the machine value $\llbracket v \rrbracket$ corresponding to the source-level value v is left on top of the stack. The correspondence between source-level and machine values is defined by:

$$\llbracket c \rrbracket = c \quad \llbracket (\lambda a)[e] \rrbracket = (\llbracket a \rrbracket, \text{Ret})[\llbracket e \rrbracket] \quad \llbracket v_1 \dots v_n \rrbracket = \llbracket v_1 \rrbracket \dots \llbracket v_n \rrbracket$$

Semantic preservation is easy to show for terminating terms a using the big-step semantics. We just need to strengthen the statement of preservation so that it lends itself to induction over the derivation of $e \vdash a \Rightarrow v$. (See the Coq development [13] for the full proof.)

Lemma 23. *If $e \vdash a \Rightarrow v$, then $(\llbracket a \rrbracket, C); S; \llbracket e \rrbracket \xrightarrow{+} C; \llbracket v \rrbracket.S; \llbracket e \rrbracket$ for all codes C and stacks S .*

It is impossible, however, to prove semantic preservation for diverging terms using only the standard big-step semantics. This led several authors to prove semantic preservation for compilation to abstract machines using small-step semantics with explicit substitutions [10, 17]. Such proofs are difficult, however, because the obvious simulation property

$$\text{If } a[e] \rightarrow a'[e'] \text{ then } \llbracket a \rrbracket; S; \llbracket e \rrbracket \xrightarrow{+} \llbracket a' \rrbracket; S'; \llbracket e' \rrbracket \text{ (for some } S')$$

does not hold: the transitions of the abstract machine do not follow the reductions of the source term. Instead, the proofs in [10, 17] rely on a *decompilation* relation that maps intermediate machine states back to source-level terms. With the help of this decompilation relation, it is possible to prove simulation diagrams that imply the desired semantic preservation properties. However, decompilation relations are difficult to define, especially for optimizing compilation schemes (see [8] for an example).

The coinductive big-step semantics studied in this paper provide a simpler way to prove semantic preservation for non-terminating terms. Namely, the following two theorems hold, showing that compilation preserves divergence and coevaluation as characterized by the $\overset{\infty}{\Rightarrow}$ and $\overset{\infty}{\Leftarrow}$ predicates.

Lemma 24. *If $e \vdash a \overset{\infty}{\Rightarrow}$, then $(\llbracket a \rrbracket, C); S; \llbracket e \rrbracket \overset{\infty}{\Leftarrow}$.*

Lemma 25. *If $e \vdash a \overset{\infty}{\Leftarrow} v$, then $(\llbracket a \rrbracket, C); S; \llbracket e \rrbracket \overset{\infty}{\Rightarrow^*} C; \llbracket v \rrbracket.S; \llbracket e \rrbracket$.*

The full proofs can be found in [13]. Both lemmas cannot be proved directly by structural coinduction and case analysis over a . The problem is in the application case $a = a_1 a_2$, where the code component of the initial machine state is of the form $\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket, \text{App}, C$. It is not possible to invoke the coinduction hypothesis to reason over the execution of $\llbracket a_1 \rrbracket$, because this use of the coinduction hypothesis

is not guarded by an inference rule for the $\overset{\infty}{\Rightarrow}$ relation, or in other terms because no machine instruction is evaluated before invoking the hypothesis.

There are two ways to address this issue. The first is to modify the compilation scheme for applications, in order to insert a “no operation” instruction in front of the generated sequence: $\llbracket a_1 \ a_2 \rrbracket = \text{Nop}, \llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket$. The **Nop** operation has the obvious machine transition $(\text{Nop}, C); S; E \rightarrow C; S; E$. With this modification, the coinductive proof for lemma 24 performs a **Nop** transition before invoking the coinduction hypothesis to deal with the evaluation of $\llbracket a_1 \rrbracket$. This makes the coinductive proof properly guarded and acceptable to Coq.

Of course, it is inelegant to pepper the generated code with **Nop** instructions just to make one proof get through. We therefore used an alternate approach where the compilation scheme for applications is unchanged, but we exploit the fact that the number of such recursive calls that do not perform a machine transition is necessarily finite, because our term algebra is finite. The proof of lemma 24 exploits this fact by defining a variant of the $\overset{\infty}{\Rightarrow}$ relation that enables a finite number of “stuttering steps” (where no instructions are executed) between executions of instructions. The finite number in question is the length of the left application spine of the term being compiled. The problem and the solution are similar to those described by Bertot [2] in his coinductive presentation and proof of Eratosthenes’ sieve algorithm.

6 Related Work

There are few instances of coinductive definitions and proofs for big-step semantics in the literature. Cousot and Cousot [4] proposed the coinductive big-step characterization of divergence that we use in this paper and studied its applicability for abstract interpretation. Grall [7] applied this approach to call-by-value λ -calculus; unlike our \Rightarrow and $\overset{\infty}{\Rightarrow}$ predicates, his big-step semantics also generate finite or infinite traces of elementary computation steps, traces which he uses to define observational equivalences. Gunter and Rémy [9] and Stoughton [18] have the same initial goal as us, namely describe both terminating and diverging computations with big-step semantics, but use increasing sequences of finite, incomplete derivations to do so, instead of infinite derivations. We do not know yet how their approach relates to our $\overset{\infty}{\Rightarrow}$ and $\overset{\omega}{\Rightarrow}$ relations.

Milner and Tofte [16] and later Leroy and Rouaix [15] used coinduction in the context of a big-step semantics for functional and imperative languages, not to describe diverging evaluations, but to capture safety properties over possibly cyclic memory stores.

Of course, coinductive techniques are routinely used in the context of small-step semantics, especially for the labeled transition systems arising from process calculi. The flavours of coinduction used there, especially proofs by bisimulations, are quite different from the present work.

The infinitary λ -calculus [11, 1] studies diverging computations from a very different angle: not only the authors use reduction semantics, but their terms are also infinite, and they use topological tools (metrics, convergence, etc) instead of coinduction.

7 Conclusions

We investigated two coinductive approaches to giving big-step semantics for non-terminating computations. The first, based on [4] and using separate evaluation rules for terminating terms and diverging terms, appears very well-behaved: it corresponds exactly to finite and infinite reduction sequences, and lends itself well to type soundness proofs and to compiler correctness proofs. The second approach, consisting in a coinductive interpretation of the standard evaluation rules, is less satisfactory: while amenable to compiler correctness proofs as well, it captures only a subset of the diverging computations of interest — and it is not yet clear which subset exactly.

A natural continuation of this work, following Grall’s work [7], is to develop coinductive, big-step, trace semantics for imperative languages that capture not only the final outcome of the evaluation (divergence or result value), but also a possibly infinite trace of the observable effects (such as input/output) performed during evaluation. Such trace semantics would enable stronger statements of observational equivalence between source code and compiled code in the context of compiler certification. However, the existence of suitable traces for infinite evaluations cannot be proved constructively, nor with just the axiom of excluded middle. It is not clear yet what classical axioms (probably variants of the axiom of choice) need to be added to Coq.

Acknowledgments

Andrzej Filinski disproved the conjecture from section 4.2 very shortly after it was stated. We thank the anonymous reviewers and the participants of the 22nd meeting of IFIP Working Group 2.8 (Functional Programming) for their feedback.

References

1. A. Berarducci and M. Dezani-Ciancaglini. Infinite lambda-calculus and types. *Theor. Comp. Sci.*, 212(1-2):29–75, 1999.
2. Y. Bertot. Filters on coinductive streams, an application to Eratosthenes’ sieve. In *Typed Lambda Calculi and Applications (TLCA ’05)*, volume 3461 of *LNCS*, pages 102–115. Springer-Verlag, 2005.
3. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development – Coq’Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2004.
4. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *19th symp. Principles of Progr. Lang.*, pages 83–94. ACM Press, 1992.
5. V. Gapeyev, M. Levin, and B. Pierce. Recursive subtyping revealed. *J. Func. Progr.*, 12(6):511–548, 2003.
6. E. Giménez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs. International Workshop TYPES ’94*, volume 996 of *LNCS*, pages 39–59. Springer-Verlag, 1994.

7. H. Grall. *Deux critères de sécurité pour l'exécution de code mobile*. PhD thesis, École Nationale des Ponts et Chaussées, Dec. 2003.
8. B. Grégoire. *Compilation des termes de preuves: un (nouveau) mariage entre Coq et OCaml*. PhD thesis, University Paris 7, 2003.
9. C. A. Gunter and D. Rémy. A proof-theoretic assessment of runtime type errors. Research Report 11261-921230-43TM, AT&T Bell Laboratories, 1993.
10. T. Hardin, L. Maranget, and B. Pagano. Functional runtimes within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.
11. R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Infinitary lambda calculus. *Theor. Comp. Sci.*, 175(1):93–125, 1997.
12. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Mar. 2004. To appear in ACM TOPLAS.
13. X. Leroy. Coinductive big-step operational semantics – the Coq development. Available from pauillac.inria.fr/~xleroy, 2005.
14. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symp. Principles of Progr. Lang*, pages 42–54. ACM Press, 2006.
15. X. Leroy and F. Rouaix. Security properties of typed applets. In J. Vitek and C. Jensen, editors, *Secure Internet Programming – Security issues for Mobile and Distributed Objects*, volume 1603 of *LNCS*, pages 147–182. Springer-Verlag, 1999.
16. R. Milner and M. Tofte. Co-induction in relational semantics. *Theor. Comp. Sci.*, 87:209–220, 1991.
17. M. Rittri. Proving the correctness of a virtual machine by a bisimulation. Licentiate thesis, Göteborg University, 1988.
18. A. Stoughton. An operational semantics framework supporting the incremental construction of derivation trees. In *Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II)*, volume 12 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.
19. M. Strecker. Compiler verification for C0. Technical report, Université Paul Sabatier, Toulouse, April 2005.
20. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. and Comp.*, 115(1):38–94, 1994.

Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types

Amal Ahmed

Harvard University, Cambridge, MA
`amal@eecs.harvard.edu`

Abstract. We present a sound and complete proof technique, based on syntactic logical relations, for showing contextual equivalence of expressions in a λ -calculus with recursive types and impredicative universal and existential types. Our development builds on the step-indexed PER model of recursive types presented by Appel and McAllester. We have discovered that a direct proof of transitivity of that model does not go through, leaving the “PER” status of the model in question. We show how to extend the Appel-McAllester model to obtain a logical relation that we can prove is transitive, as well as sound and complete with respect to contextual equivalence. We then augment this model to support relational reasoning in the presence of quantified types.

Step-indexed relations are indexed not just by types, but also by the number of steps available for future evaluation. This stratification is essential for handling various circularities, from recursive functions, to recursive types, to impredicative polymorphism. The resulting construction is more elementary than existing logical relations which require complex machinery such as domain theory, admissibility, syntactic minimal invariance, and $\top\top$ -closure.

1 Introduction

Proving equivalence of programs is important for verifying the correctness of compiler optimizations and other program transformations, as well as for establishing that program behavior is independent of the representation of an abstract type. This representation independence principle guarantees that if one implementation of an abstraction is exchanged for another, client modules will not be able to detect a difference.

Program equivalence is generally defined in terms of *contextual equivalence*. We say that two programs are contextually equivalent if they have the same observable behavior when placed in any program context C . Unfortunately, proving contextual equivalence is difficult in general, since it involves quantification over *all* possible contexts. As a result, there’s been much work on finding tractable techniques for proving contextual equivalence. Many of these are based on the method of *logical relations*.

Logical relations specify relations on well-typed terms via structural induction on the syntax of types. Thus, for instance, logically related functions take

logically related arguments to related results, while logically related pairs consist of components that are related pairwise. Logical relations may be based on denotational models (e.g. [1, 2, 3]) or on the operational semantics of a language [4, 5, 6, 7]. The latter are also known as syntactic logical relations [8] and it is this flavor that is the focus of this paper.

To prove the soundness of a logical relation, one must prove the Fundamental Property (also called the Basic Lemma) which says that any well-typed term is related to itself. For simple type systems, it is fairly straightforward to prove the Fundamental Property in the absence of nontermination. The addition of recursive functions, however, complicates matters: establishing the Fundamental Property now requires proving additional “unwinding” lemmas [9, 6, 7, 10] which show that in any terminating computation a recursively defined function is approximated by its finite unrollings. More challenging still is the addition of recursive types and impredicative quantified types¹ since the logical relation can no longer be defined by induction on types. Thus, showing the existence of a relational interpretation of recursive types requires proving a nontrivial *minimal invariance* property [3, 10, 8, 11, 12].

Appel and McAllester [13] proposed a radically different solution to the problem of recursive types. They defined *intensional* types, based on the operational semantics of the language, that are indexed by the number of available (future) execution steps. This extra information is sufficient to solve recursive equations on types. Appel and McAllester also presented a PER (relational) model of recursive types, which we build on in this paper. The advantage of step-indexed logical relations is that they avoid complex machinery like domain theory, admissibility, syntactic minimal invariance, and $\top\top$ -closure (biorthogonality). The approach is promising since unary step-indexed models have scaled well to advanced features like impredicative quantified types and general references (i.e., mutable references that can store functions, recursive types, other references, and even impredicative quantified types) [14, 15].

Appel and McAllester proved the Fundamental Property for their PER model of equi-recursive types, and conjectured that their model was sound with respect to contextual equivalence. We show that their claim is correct — to be precise, we show soundness for a calculus with iso-recursive types, but the essence of the model is the same.

We discovered, however, that the expected proof of transitivity for the Appel-McAllester model does not go through. To definitively show that their model is not transitive we tried to find a counterexample, but could not. Thus, we note that the transitivity of the Appel-McAllester model remains an open problem.

In Section 2 we consider a λ -calculus with iso-recursive types and present a sound and complete logical relation for the language. We also show how a direct proof of transitivity of the Appel-McAllester model fails, and discuss some of the peculiarities of the step-indexed approach. In Section 3 we extend the logical relation to support quantified types. Proofs of all lemmas in the paper and

¹ A quantified type such as $\forall\alpha. \tau$ is impredicative if α may be instantiated with *any* type, including $\forall\alpha. \tau$ itself.

several examples to illustrate the use of our logical relation are given in the accompanying technical report [16].

2 Recursive Types

We consider a call-by-value λ -calculus with iso-recursive types (dubbed the λ^{rec} -calculus). Figure 1 presents the syntax and small-step operational semantics for the language, which supports booleans and pairs in addition to recursive types. We define the operational semantics for λ^{rec} as a relation between closed terms e . We use evaluation contexts to lift the primitive rewriting rules to a standard, left-to-right, innermost-to-outermost, call-by-value interpretation of the language. We say that a term e is irreducible ($\text{irred}(e)$) if e is a value ($\text{val}(e)$) or if e is a “stuck” expression to which no operational rule applies. We also use $e \Downarrow$ as an abbreviation for $\exists e'. e \mapsto^* e' \wedge \text{val}(e')$.

<i>Types</i>	$\tau ::= \text{bool} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \alpha \mid \mu\alpha. \tau$
<i>Expressions</i>	$e ::= x \mid \text{tt} \mid \text{ff} \mid \text{if } e_0, e_1, e_2 \mid \langle e_1, e_2 \rangle \mid \text{let } \langle x_1, x_2 \rangle = e_1 \text{ in } e_2 \mid \lambda x. e \mid e_1 e_2 \mid \text{fold } e \mid \text{unfold } e$
<i>Values</i>	$v ::= \text{tt} \mid \text{ff} \mid \langle v_1, v_2 \rangle \mid \lambda x. e \mid \text{fold } v$
<i>Eval Ctxts</i>	$E ::= [\cdot] \mid \text{if } E, e_1, e_2 \mid \text{let } \langle x_1, x_2 \rangle = E \text{ in } e \mid E e \mid v E \mid \text{fold } E \mid \text{unfold } E$
(iftrue)	$\text{if } \text{tt}, e_1, e_2 \mapsto e_1$
(iffalse)	$\text{if } \text{ff}, e_1, e_2 \mapsto e_2$
(letpair)	$\text{let } \langle x_1, x_2 \rangle = \langle v_1, v_2 \rangle \text{ in } e \mapsto e[v_1/x_1][v_2/x_2]$
(app)	$(\lambda x. e) v \mapsto e[v/x]$
(unfold)	$\text{unfold } (\text{fold } v) \mapsto v$
(ctxt)	$\frac{e \mapsto e'}{E[e] \mapsto E[e']}$

Fig. 1. λ^{rec} Syntax and Operational Semantics

Typing judgments in λ^{rec} have the form $\Gamma \vdash e : \tau$ where the context Γ is defined as follows:

$$\text{Value Context } \Gamma ::= \bullet \mid \Gamma, x : \tau$$

Thus, Γ is used to track the set of variables in scope, along with their (closed) types. There may be at most one occurrence of a variable x in Γ . The λ^{rec} static semantics is entirely conventional (see, e.g., [17]) so we only show selected rules in Figure 2. We use the abbreviated judgment $\vdash e : \tau$ when the value context is empty.

Theorem 1 (λ^{rec} Safety). *If $\bullet \vdash e : \tau$ and $e \mapsto^* e'$, then either e' is a value, or there exists an e'' such that $e' \mapsto e''$.*

$$\boxed{\Gamma \vdash e : \tau}$$

$$\begin{array}{c}
(\text{Var}) \frac{}{\Gamma \vdash x : \Gamma(x)} \quad (\text{Fn}) \frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \quad (\text{App}) \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \\
(\text{Fold}) \frac{\Gamma \vdash e : \tau[\mu\alpha. \tau/\alpha]}{\Gamma \vdash \text{fold } e : \mu\alpha. \tau} \quad (\text{Unfold}) \frac{\Gamma \vdash e : \mu\alpha. \tau}{\Gamma \vdash \text{unfold } e : \tau[\mu\alpha. \tau/\alpha]}
\end{array}$$

Fig. 2. λ^{rec} Static Semantics (Selected Rules)

2.1 λ^{rec} : Contextual Equivalence

A context C is an expression with a single hole $[\cdot]$ in it. Typing judgments for contexts have the form $\Gamma_1 \vdash C : (\Gamma \triangleright \tau) \rightsquigarrow \tau_1$, where $(\Gamma \triangleright \tau)$ indicates the type of the hole — that is, if $\Gamma \vdash e : \tau$, then $\Gamma_1 \vdash C[e] : \tau_1$.

Definition 2 (λ^{rec} Contextual Approximation \preceq^{ctx} & Equivalence \simeq^{ctx}). *If $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$, we write $\Gamma \vdash e \preceq^{\text{ctx}} e' : \tau$ to mean*

$$\forall C, \tau_1. \bullet \vdash C : (\Gamma \triangleright \tau) \rightsquigarrow \tau_1 \wedge C[e] \Downarrow \implies C[e'] \Downarrow.$$

Two terms are contextually equivalent if they contextually approximate one another:

$$\Gamma \vdash e \simeq^{\text{ctx}} e' : \tau \stackrel{\text{def}}{=} \Gamma \vdash e \preceq^{\text{ctx}} e' : \tau \wedge \Gamma \vdash e' \preceq^{\text{ctx}} e : \tau.$$

2.2 λ^{rec} : Logical Relation

Our step-indexed logical relation for λ^{rec} is based on the PER model for equi-recursive types presented by Appel and McAllester [13] (henceforth AM). The latter claimed, but did not prove, that their PER model was sound with respect to contextual equivalence. We have proved that this is indeed the case. However, “PER” may be somewhat of misnomer for the AM model since the status of transitivity is unclear, as we shall show.

In both models, the relational interpretation $\mathcal{RV}[\tau]$ of a type τ is a set of triples of the form (k, v, v') where k is a natural number (called the *approximation index* or *step index*), and v and v' are (closed) values. Intuitively, $(k, v, v') \in \mathcal{RV}[\tau]$ says that in any computation running for no more than k steps, v approximates v' at the type τ . Our model differs from the AM model in that whenever $(k, v, v') \in \mathcal{RV}[\tau]$, we additionally require that $\bullet \vdash v' : \tau$. This additional constraint enables us to prove the transitivity of our logical relation. Moreover, restricting the model to terms that are well-typed seems essential for completeness with respect to contextual equivalence, as others have also noted [12]. We defer an explanation of why we don’t also require $\bullet \vdash v : \tau$ till Section 2.3.

Figure 3 gives the definition of our logical relation; shaded parts of the definitions have no analog in the AM model. We use the meta-variable χ to denote sets of tuples of the form (k, v, v') , where v and v' are closed values ($v, v' \in CValues$).

$$\begin{aligned}
Rel_\tau &\stackrel{\text{def}}{=} \{\chi \in 2^{Nat \times CValues \times CValues} \mid \forall (j, v, v') \in \chi. \quad \bullet \vdash v' : \tau \wedge \\
&\quad \forall i \leq j. (i, v, v') \in \chi\} \\
\lfloor \chi \rfloor_k &\stackrel{\text{def}}{=} \{(j, v, v') \mid j < k \wedge (j, v, v') \in \chi\} \\
\mathcal{RV} \llbracket \alpha \rrbracket \rho &= \rho^{\text{sem}}(\alpha) \\
\mathcal{RV} \llbracket \text{bool} \rrbracket \rho &= \{(k, v, v') \mid \vdash v' : \text{bool} \wedge \\
&\quad (v = v' = \mathbf{tt} \vee v = v' = \mathbf{ff})\} \\
\mathcal{RV} \llbracket \tau_1 \times \tau_2 \rrbracket \rho &= \{(k, \langle v_1, v_2 \rangle \langle v'_1, v'_2 \rangle) \mid \vdash \langle v'_1, v'_2 \rangle : (\tau_1 \times \tau_2)^{[\rho]} \wedge \\
&\quad (k, v_1, v'_1) \in \mathcal{RV} \llbracket \tau_1 \rrbracket \rho \wedge (k, v_2, v'_2) \in \mathcal{RV} \llbracket \tau_2 \rrbracket \rho\} \\
\mathcal{RV} \llbracket \tau_1 \rightarrow \tau_2 \rrbracket \rho &= \{(k, \lambda x. e, \lambda x. e') \mid \vdash \lambda x. e' : (\tau_1 \rightarrow \tau_2)^{[\rho]} \wedge \\
&\quad \forall j < k, v, v'. \\
&\quad (j, v, v') \in \mathcal{RV} \llbracket \tau_1 \rrbracket \rho \implies \\
&\quad (j, e[v/x], e'[v'/x]) \in \mathcal{RC} \llbracket \tau_2 \rrbracket \rho\} \\
\mathcal{RV} \llbracket \mu\alpha. \tau \rrbracket \rho &= \{(k, \text{fold } v, \text{fold } v') \mid \vdash \text{fold } v' : (\mu\alpha. \tau)^{[\rho]} \wedge \\
&\quad \forall j < k. \\
&\quad \text{let } \chi = \lfloor \mathcal{RV} \llbracket \mu\alpha. \tau \rrbracket \rho \rfloor_{j+1} \text{ in} \\
&\quad (j, v, v') \in \mathcal{RV} \llbracket \tau \rrbracket \rho[\alpha \mapsto (\chi, (\mu\alpha. \tau)^{[\rho]})]\} \\
\mathcal{RC} \llbracket \tau \rrbracket \rho &= \{(k, e, e') \mid \forall j < k, e_f. \\
&\quad e \mapsto^j e_f \wedge \text{irred}(e_f) \implies \\
&\quad \exists e'_f. e' \mapsto^* e'_f \wedge (k - j, e_f, e'_f) \in \mathcal{RV} \llbracket \tau \rrbracket \rho\} \\
\mathcal{RG} \llbracket \bullet \rrbracket &= \{(k, \emptyset, \emptyset)\} \\
\mathcal{RG} \llbracket \Gamma, x:\tau \rrbracket &= \{(k, \gamma[x \mapsto v], \gamma'[x \mapsto v']) \mid (k, \gamma, \gamma') \in \mathcal{RG} \llbracket \Gamma \rrbracket \wedge (k, v, v') \in \mathcal{RV} \llbracket \tau \rrbracket \emptyset\} \\
\Gamma \vdash e \leq e' : \tau &\stackrel{\text{def}}{=} \Gamma \vdash e : \tau \wedge \Gamma \vdash e' : \tau \wedge \\
&\quad \forall k \geq 0. \forall \gamma, \gamma'. \\
&\quad (k, \gamma, \gamma') \in \mathcal{RG} \llbracket \Gamma \rrbracket \implies (k, \gamma(e), \gamma'(e')) \in \mathcal{RC} \llbracket \tau \rrbracket \emptyset \\
\Gamma \vdash e \sim e' : \tau &\stackrel{\text{def}}{=} \Gamma \vdash e \leq e' : \tau \wedge \Gamma \vdash e' \leq e : \tau
\end{aligned}$$

Fig. 3. λ^{rec} Relational Model (Shaded \notin Appel-McAllester)

For any set χ , we define the k -approximation of the set (written $\lfloor \chi \rfloor_k$) as the subset of its elements whose indices are less than k .

We define Rel_τ (where τ is a closed syntactic type) as the set of those sets $\chi \in 2^{Nat \times CValues \times CValues}$ that have the following two properties: if $(k, v, v') \in \chi$, then v' must be well-typed with type τ , and χ must be closed with respect to a decreasing step-index.

We use the meta-variable ρ to denote type substitutions. These are partial maps from type variables α to pairs (χ, τ) where χ is the semantic substitution for α and τ (a closed syntactic type) is the syntactic substitution for α . We note that our definitions ensure that if $\rho(\alpha) = (\chi, \tau)$ then $\chi \in Rel_\tau$. Since types in λ^{rec} may contain free type variables, the interpretation of a type τ is parametrized by a type substitution ρ such that $FTV(\tau) \subseteq \text{dom}(\rho)$. We use the following abbreviations:

- Let $\rho(\alpha) = (\chi, \tau)$. Then $\rho^{\text{sem}}(\alpha) = \chi$ and $\rho^{\text{syn}}(\alpha) = \tau$.
- Let $\rho = \{\alpha_1 \mapsto (\chi_1, \tau_1), \dots, \alpha_n \mapsto (\chi_n, \tau_n)\}$.
Then $\tau^{[\rho]}$ is an abbreviation for $\tau[\tau_1/\alpha_1, \tau_2/\alpha_2, \dots, \tau_n/\alpha_n]$.

Next, we consider the relational interpretation $\mathcal{RV} \llbracket \tau \rrbracket \rho$ of each type τ . In each case, note that if $(k, v, v') \in \mathcal{RV} \llbracket \tau \rrbracket \rho$ then $\vdash v' : (\tau)^{[\rho]}$.

Booleans. Two values are related at the type **bool** for any number of steps $k \geq 0$, if they are both **tt** or both **ff**.

Pairs. The pairs $\langle v_1, v_2 \rangle$ and $\langle v'_1, v'_2 \rangle$ are related at type $\tau_1 \times \tau_2$ for k steps if v_i and v'_i are related for k steps at the type τ_i (for $i \in \{1, 2\}$).

Functions. Since functions are suspended computations, their interpretation is given in terms of the interpretation of types as computations (see below). Two functions are related if they map related arguments to related results. Specifically, $\lambda x. e$ and $\lambda x. e'$ are related at the type $\tau_1 \rightarrow \tau_2$ for k steps if, at some point in the future, when there are $j < k$ steps left to execute, and there are arguments v_a and v'_a that are related at the type τ_1 for j steps, then $e[v_a/x]$ and $e'[v'_a/x]$ are related as computations of type τ_2 for j steps.

Recursive Types. One would expect the values **fold** v and **fold** v' to be related at the type $\mu\alpha. \tau$ for k steps if v and v' are related at the type $\tau[\mu\alpha. \tau/\alpha]$ for $j < k$ steps. We show that the latter is equivalent to what is required by the definition in Figure 3. Note that by the definition of $\llbracket \cdot \rrbracket_k$

$$(j, v, v') \in \mathcal{RV} \llbracket \tau[\mu\alpha. \tau/\alpha] \rrbracket \rho \Leftrightarrow (j, v, v') \in \llbracket \mathcal{RV} \llbracket \tau[\mu\alpha. \tau/\alpha] \rrbracket \rho \rrbracket_{j+1}.$$

We prove a type substitution lemma (see [16]) that allows us to conclude that if $\chi = \llbracket \mathcal{RV} \llbracket \mu\alpha. \tau \rrbracket \rho \rrbracket_{j+1}$ then:

$$\llbracket \mathcal{RV} \llbracket \tau[\mu\alpha. \tau/\alpha] \rrbracket \rho \rrbracket_{j+1} = \llbracket \mathcal{RV} \llbracket \tau \rrbracket \rho[\alpha \mapsto (\chi, (\mu\alpha. \tau)^{[\rho]})] \rrbracket_{j+1}.$$

Hence,

$$\begin{aligned} (j, v, v') &\in \mathcal{RV} \llbracket \tau[\mu\alpha. \tau/\alpha] \rrbracket \rho \\ &\Leftrightarrow (j, v, v') \in \llbracket \mathcal{RV} \llbracket \tau[\mu\alpha. \tau/\alpha] \rrbracket \rho \rrbracket_{j+1} && \text{by } \llbracket \cdot \rrbracket_k \\ &\Leftrightarrow (j, v, v') \in \llbracket \mathcal{RV} \llbracket \tau \rrbracket \rho[\alpha \mapsto (\chi, (\mu\alpha. \tau)^{[\rho]})] \rrbracket_{j+1} && \text{by type subst} \\ &\Leftrightarrow (j, v, v') \in \mathcal{RV} \llbracket \tau \rrbracket \rho[\alpha \mapsto (\chi, (\mu\alpha. \tau)^{[\rho]})] && \text{by } \llbracket \cdot \rrbracket_k \end{aligned}$$

which is exactly what is required by the definition of $\mathcal{RV} \llbracket \mu\alpha. \tau \rrbracket \rho$.

Computations. Two closed expressions e and e' are related as computations of type τ for k steps as follows. If e steps to an irreducible term e_f in $j < k$ steps, then e' must also step to some irreducible e'_f . Furthermore, both e_f and e'_f must be values that are related for the remaining $k - j$ steps.

What is surprising about this definition is that e must terminate in $j < k$ steps, while e' may terminate in *any* number of steps, say i . Hence, i may be greater than k . This has ramifications for transitivity in the AM model and we shall return to this point shortly.

Logical Relation. If $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$, then we write $\Gamma \vdash e \leq e' : \tau$ to mean that for all $k \geq 0$, if γ and γ' are mappings from variables x to closed values that are related for k steps at Γ , then $\gamma(e)$ and $\gamma'(e')$ are related for k steps as computations of type τ . We say e and e' are logically equivalent, written $\Gamma \vdash e \sim e' : \tau$, if they logically approximate one another.

We now have to prove that each type τ is a valid type — that is, that the relational interpretation of τ belongs to Rel_τ (i.e., $\mathcal{RV}[\![\tau]\!] \rho \in Rel_{\tau[\rho]}$). This involves showing well-typedness and closure under decreasing step-index.

Next, we prove a number of nontrivial lemmas (see the technical report [16]). Specifically, we prove that the logical relation defined in Figure 3 has the compatibility and substitutivity properties (see e.g., [9]). These allow us to show that the λ^{rec} typing rules preserve the logical relation, and hence prove the following lemma.

Lemma 3 (λ^{rec} Fundamental Property / Reflexivity).

If $\Gamma \vdash e : \tau$, then $\Gamma \vdash e \leq e : \tau$.

2.3 Transitivity and the Appel-McAllester Model

Let us ignore the shaded parts of Figure 3 and try to prove the following lemma with the resulting definitions.

Proposed Lemma (Transitivity: Appel-McAllester)

If $\Gamma \vdash e_1 \leq e_2 : \tau$ and $\Gamma \vdash e_2 \leq e_3 : \tau$, then $\Gamma \vdash e_1 \leq e_3 : \tau$.

Proof Attempt: Suppose $k \geq 0$ and $(k, \gamma, \gamma') \in \mathcal{RG}[\![I]\!]$.

Show $(k, \gamma(e_1), \gamma'(e_3)) \in \mathcal{RC}[\![\tau]\!] \emptyset$. Suppose $j < k$, $\gamma(e_1) \mapsto^j e_{f_1}$, and $\text{irred}(e_{f_1})$.

Show $\exists e_{f_3}. \gamma'(e_3) \mapsto^* e_{f_3} \wedge (k - j, e_{f_1}, e_{f_3}) \in \mathcal{RV}[\![\tau]\!] \emptyset$.

Instantiate $\Gamma \vdash e_1 \leq e_2 : \tau$ with $k \geq 0$ and $(k, \gamma, \gamma') \in \mathcal{RG}[\![I]\!]$.

Hence, $(k, \gamma(e_1), \gamma'(e_2)) \in \mathcal{RC}[\![\tau]\!] \emptyset$.

Instantiate this with $j < k$, $\gamma(e_1) \mapsto^j e_{f_1}$, and $\text{irred}(e_{f_1})$.

Hence, $\exists e_{f_2}, i$ such that $i \geq 0$, $\gamma'(e_2) \mapsto^i e_{f_2}$, and $(k - j, e_{f_1}, e_{f_2}) \in \mathcal{RV}[\![\tau]\!] \emptyset$.

Now we need to use the premise $\Gamma \vdash e_2 \leq e_3 : \tau$. But what should we instantiate this with? We consider two ways we could proceed.

- (i) Instantiate $\Gamma \vdash e_2 \leq e_3 : \tau$ with k, γ, γ' . Note that $k \geq 0$ and $(k, \gamma, \gamma') \in \mathcal{RG}[\![I]\!]$. Hence, $(k, \gamma(e_2), \gamma'(e_3)) \in \mathcal{RC}[\![\tau]\!] \emptyset$.

Problem: We could instantiate this with i and e_{f_2} , but at that point we are stuck since we cannot show $i < k$ (since i may be greater than k), and we cannot show $\gamma(e_2) \mapsto^i e_{f_2}$ (we only have $\gamma'(e_2) \mapsto^i e_{f_2}$).

- (ii) Instantiate $\Gamma \vdash e_2 \leq e_3 : \tau$ with $i + 1, \gamma', \gamma'$.

Problem: We cannot show $(i + 1, \gamma', \gamma') \in \mathcal{RG}[\![I]\!]$. All we know is that $(k, \gamma, \gamma') \in \Gamma$, where i may be greater than k .

We note that if we restrict our attention to closed terms e_1, e_2, e_3 , then the above lemma can be proved. In the case of open terms, however, the status of transitivity of the AM model is unclear as we have been unable to find a counterexample.

There are several things one could attempt in order to rectify the above problem with the AM model (unshaded parts of Figure 3). One problem we encountered was that i may be greater than k . To get around this, we could change the definition of $(k, e, e') \in \mathcal{RC}[\tau]$ to require that e' must terminate in less than k steps. Unfortunately, if we step back and examine the resulting meaning of $\Gamma \vdash e_1 \sim e_2 : \tau$, we see that the latter now requires that both e_1 and e_2 must terminate in *exactly the same number of steps*. Clearly such a logical relation would not be very useful (unless we are concerned with reasoning about timing leaks in an information-flow setting). Other formulations involving the use of not one, but two step-indices (where the second bounds the number of steps in which e' must terminate) also lead to models where both terms are required to terminate in exactly the same number of steps.

Since we want a logical relation that considers programs equivalent modulo the number of steps they take, we will not change the definition of $\mathcal{RC}[\tau]$. Instead we fix the problem with transitivity by moving to a typed setting where $(k, v, v') \in \mathcal{RV}[\tau] \emptyset$ implies $\vdash v' : \tau$. Assuming the definitions in Figure 3, including the shaded parts, let us again try to prove transitivity.

Lemma 4 (λ^{rec} : Transitivity). *(Our model: Figure 3, including shaded parts) If $\Gamma \vdash e_1 \leq e_2 : \tau$ and $\Gamma \vdash e_2 \leq e_3 : \tau$, then $\Gamma \vdash e_1 \leq e_3 : \tau$.*

Proof. We start at the point where we got stuck before. Now from $(k, \gamma, \gamma') \in \mathcal{RG}[\Gamma]$ we can conclude that $\vdash \gamma' : \Gamma$. By reflexivity (Fundamental Property, Lemma 3) it follows that $\vdash \gamma' \leq \gamma' : \Gamma$. Hence, we can show that for all $z \geq 0$, $(z, \gamma', \gamma') \in \mathcal{RG}[\Gamma]$ holds. Now we may instantiate $\Gamma \vdash e_2 \leq e_3 : \tau$ above with $i+1$ since we know that $(i+1, \gamma', \gamma') \in \mathcal{RG}[\Gamma]$. The rest of the proof is relatively straightforward and is given in the accompanying technical report [16].

Seemingly Asymmetric Well-Typedness Requirement. The definitions in Figure 3 may have left the reader with the impression that we only require terms on one side of our logical relation to be well-typed. This, however, is not the case. In particular, notice that in the definition of $\Gamma \vdash e \leq e' : \tau$, we require that *both* e and e' be well-typed. However, once we have picked a step-index k (i.e., once we have moved under the $\forall k$ quantifier), there is an asymmetry in the model in that when $(k, e, e') \in \mathcal{RC}[\tau]$, k pertains (as a bound) only to e and not to e' . As a result of this asymmetry, when working with a specific k (in the definition of $\mathcal{RV}[\tau]$) we do not need to know that v has type τ in the limit, while the converse is true of v' . Hence, at the value interpretation level $\mathcal{RV}[\tau]$, we chose only to require $\vdash v' : \tau$. One could add the requirement $\vdash v : \tau$ in the interest of symmetry, but it would simply lead to additional proof obligations being shuffled around. It would also complicate definitions when we get to quantified types as Rel_τ would have to be replaced by Rel_{τ_1, τ_2} (since in the presence of quantified types we wish to relate values of different types).

2.4 λ^{rec} : Soundness

To prove that our logical relation is sound with respect to contextual equivalence, we first define what it means for two contexts to be logically related.

Definition 5 (λ^{rec} Logical Relation: Contexts).

$$\Gamma_1 \vdash C \leq C' : (\Gamma \triangleright \tau) \rightsquigarrow \tau_1 \stackrel{\text{def}}{=} \forall e, e'. \Gamma \vdash e \leq e' : \tau \implies \Gamma_1 \vdash C[e] \leq C'[e'] : \tau_1$$

Next, we prove the compatibility lemmas for contexts, which allows us to prove the following.

Lemma 6 (λ^{rec} Reflexivity: Contexts).

If $\Gamma_1 \vdash C : (\Gamma \triangleright \tau) \rightsquigarrow \tau_1$, then $\Gamma_1 \vdash C \leq C : (\Gamma \triangleright \tau) \rightsquigarrow \tau_1$.

Theorem 7 (λ^{rec} Soundness: $\leq \subseteq \preceq^{ctx}$).

If $\Gamma \vdash e \leq e' : \tau$ then $\Gamma \vdash e \preceq^{ctx} e' : \tau$.

Proof. Suppose $\bullet \vdash C : (\Gamma \triangleright \tau) \rightsquigarrow \tau_1$ and $C[e] \Downarrow$. Hence, there exist v_f, k such that $C[e] \mapsto^k v_f$. We must show $C[e'] \Downarrow$.

Applying Lemma 6 to $\bullet \vdash C : (\Gamma \triangleright \tau) \rightsquigarrow \tau_1$, we have $\bullet \vdash C \leq C : (\Gamma \triangleright \tau) \rightsquigarrow \tau_1$.

Instantiate this with $\Gamma \vdash e \leq e' : \tau$. Hence, $\bullet \vdash C[e] \leq C[e'] : \tau_1$.

Instantiate this with $k + 1 \geq 0$ and $(k + 1, \emptyset, \emptyset) \in \mathcal{RG} \llbracket \bullet \rrbracket$.

Hence, $(k + 1, C[e], C[e']) \in \mathcal{RC} \llbracket \tau_1 \rrbracket \emptyset$.

Instantiate this with $k < k + 1$, $C[e] \mapsto^k v_f$, and *irred*(v_f).

Hence, exists v'_f such that $C[e'] \mapsto^* v'_f$. Hence, $C[e'] \Downarrow$.

2.5 λ^{rec} : Completeness

To show that our logical relation is complete with respect to contextual equivalence, we make use of the notion of *ciu-equivalence* introduced by Mason and Talcott [18]. Two closed terms of the same closed type are said to be *ciu-equivalent* if they have the same termination behavior in any evaluation context E (a use of the term). The relation is extended to open terms via closing substitutions (i.e., closed instantiations). We note that evaluation contexts E are a simply a subset of general contexts C and that only closed terms can be placed in an evaluation context.

Definition 8 (λ^{rec} Ciu Approximation \preceq^{ciu} & Equivalence \simeq^{ciu}). Let $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$.

$$\Gamma \vdash e \preceq^{ciu} e' : \tau \stackrel{\text{def}}{=} \forall \gamma, E, \tau_1. \bullet \vdash \gamma : \Gamma \wedge \bullet \vdash E : (\bullet \triangleright \tau) \rightsquigarrow \tau_1 \wedge E[\gamma(e)] \Downarrow \implies E[\gamma(e')] \Downarrow$$

$$\Gamma \vdash e \simeq^{ciu} e' : \tau \stackrel{\text{def}}{=} \Gamma \vdash e \preceq^{ciu} e' : \tau \wedge \Gamma \vdash e' \preceq^{ciu} e : \tau$$

Theorem 9 ($\lambda^{\text{rec}} : \preceq^{ctx} \subseteq \preceq^{ciu}$). If $\Gamma \vdash e \preceq^{ctx} e' : \tau$ then $\Gamma \vdash e \preceq^{ciu} e' : \tau$.

To prove that two *ciu-equivalent* terms are logically related, we will need the following lemma which shows that our logical relation respects *ciu* equivalence. Pitts [9] proves a similar property which he calls “equivalence-respecting”.

Lemma 10 (λ^{rec} Equivalence-Respecting: Closed Values). If $(k, v_1, v_2) \in \mathcal{RV} \llbracket \tau \rrbracket \emptyset$ and $\bullet \vdash v_2 \preceq^{ciu} v_3 : \tau$, then $(k, v_1, v_3) \in \mathcal{RV} \llbracket \tau \rrbracket \emptyset$.

Proof. By induction on k and nested induction on the structure of the (closed) type τ .

Theorem 11 ($\lambda^{\text{rec}} : \preceq^{\text{ciu}} \subseteq \leq$). *If $\Gamma \vdash e \preceq^{\text{ciu}} e' : \tau$ then $\Gamma \vdash e \leq e' : \tau$.*

Proof. Suppose $k \geq 0$ and $(k, \gamma, \gamma') \in \mathcal{RG}[\Gamma]$. Show $(k, \gamma(e), \gamma'(e')) \in \mathcal{RC}[\tau] \emptyset$.

Suppose $j < k$, $\gamma(e) \mapsto^j e_f$, and $\text{irred}(e_f)$.

Show $\exists e'_f. \gamma'(e') \mapsto^* e'_f \wedge (k - j, e_f, e'_f) \in \mathcal{RV}[\tau] \emptyset$.

From $\Gamma \vdash e \preceq^{\text{ciu}} e' : \tau$, we have $\Gamma \vdash e : \tau$. Applying Lemma 3 to $\Gamma \vdash e : \tau$, we have $\Gamma \vdash e \leq e : \tau$. Instantiate this with $k \geq 0$ and $(k, \gamma, \gamma') \in \mathcal{RG}[\Gamma]$. Hence, $(k, \gamma(e), \gamma'(e)) \in \mathcal{RC}[\tau] \emptyset$. Instantiate this with $j < k$, $\gamma(e) \mapsto^j e_f$, and $\text{irred}(e_f)$. Hence, $\exists e'_f$ such that $\gamma'(e) \mapsto^* e'_f$ and $(k - j, e_f, e'_f) \in \mathcal{RV}[\tau] \emptyset$. Hence, $e_f \equiv v_f$ and $e'_f \equiv v'_f$. Hence, $\gamma'(e) \Downarrow v'_f$.

Instantiate $\Gamma \vdash e \preceq^{\text{ciu}} e' : \tau$ with $\vdash \gamma' : \Gamma$ (follows from $(k, \gamma, \gamma') \in \mathcal{RG}[\Gamma]$), and $\bullet \vdash [\cdot] : (\bullet \triangleright \tau) \rightsquigarrow \tau$, and $\gamma'(e) \Downarrow$. Hence, $\exists v'_f$ such that $\gamma'(e') \mapsto^* v'_f$.

Remains to show: $(k - j, v_f, v'_f) \in \mathcal{RV}[\tau] \emptyset$.

This follows from Lemma 10 applied to $(k - j, v_f, v'_f) \in \mathcal{RV}[\tau] \emptyset$ and $v'_f \preceq^{\text{ciu}} v''_f : \tau$ (which follows from $\Gamma \vdash e \preceq^{\text{ciu}} e' : \tau$ and $\gamma'(e) \Downarrow v'_f$ and $\gamma'(e') \Downarrow v''_f$).

3 Type Abstraction

We now extend λ^{rec} with impredicative universal and existential types; we call the extended language the $\lambda^{\forall\exists}$ -calculus. The syntactic extensions to support quantified types are as follows:

<i>Types</i>	$\tau ::= \dots \mid \forall \alpha. \tau \mid \exists \alpha. \tau$
<i>Values</i>	$v ::= \dots \mid \Lambda. e \mid \text{pack } v$
<i>Expressions</i>	$e ::= \dots \mid e [] \mid \text{unpack } e_1 \text{ as } x \text{ in } e_2$

Note that terms are not decorated with types (which was also the case for λ^{rec}). Here we let the vestigial operators remain in the untyped syntax in order to preserve the operational semantics. For instance, the term $\Lambda. e$ is a suspended computation (normally written $\Lambda \alpha. e$); $e []$ runs the suspended computation. We extend the λ^{rec} operational semantics as follows:

<i>Evaluation Contexts</i>	$E ::= \dots \mid E [] \mid \text{unpack } E \text{ as } x \text{ in } e$
(inst)	$(\Lambda. e) [] \mapsto e$
(unpack)	$\text{unpack } (\text{pack } v) \text{ as } x \text{ in } e \mapsto e[v/x]$

$\lambda^{\forall\exists}$ typing judgments have the form $\Delta; \Gamma \vdash e : \tau$, where the context Γ is as before, and the context Δ is defined as follows:

$$\text{Type Context } \Delta ::= \bullet \mid \Delta, \alpha \ .$$

The type context Δ is used to track the set of type variables in scope. We modify the typing rules in Figure 2 by adding Δ to each typing judgment. Figure 4 gives the typing rules for the additional terms in $\lambda^{\forall\exists}$. We prove soundness of the $\lambda^{\forall\exists}$ typing rules, show that value and type substitution hold, and prove type safety.

$$\boxed{\Delta; \Gamma \vdash e : \tau}$$

$$\begin{array}{c}
\text{(All)} \quad \frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda. e : \forall \alpha. \tau} \qquad \text{(Inst)} \quad \frac{\Delta; \Gamma \vdash e : \forall \alpha. \tau \quad \Delta \vdash \tau_1}{\Delta; \Gamma \vdash e [] : \tau[\tau_1/\alpha]} \\
\text{(Pack)} \quad \frac{\Delta \vdash \tau_1 \quad \Delta; \Gamma \vdash e : \tau[\tau_1/\alpha]}{\Delta; \Gamma \vdash \text{pack } e : \exists \alpha. \tau} \qquad \text{(Unpack)} \quad \frac{\Delta; \Gamma \vdash e_1 : \exists \alpha. \tau_1 \quad \Delta \vdash \tau_2}{\Delta, \alpha; \Gamma, x : \tau_1 \vdash e_2 : \tau_2} \\
\Delta; \Gamma \vdash \text{unpack } e_1 \text{ as } x \text{ in } e_2 : \tau_2
\end{array}$$

Fig. 4. $\lambda^{\forall\exists}$ Static Semantics

Theorem 12 ($\lambda^{\forall\exists}$ Safety). *If $\bullet; \bullet \vdash e : \tau$ and $e \mapsto^* e'$, then either e' is a value, or there exists an e'' such that $e' \mapsto e''$.*

3.1 $\lambda^{\forall\exists}$: Contextual Equivalence

Typing judgments for contexts C now have the form $\Delta_1; \Gamma_1 \vdash C : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow \tau_1$ (where $(\Delta; \Gamma \triangleright \tau)$ represents the type of the hole) indicating that whenever $\Delta; \Gamma \vdash e : \tau$, then $\Delta_1; \Gamma_1 \vdash C[e] : \tau_1$.

Definition 13 ($\lambda^{\forall\exists}$ Contextual Approximation \preceq^{ctx}).

If $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma \vdash e' : \tau$, then we write $\Delta; \Gamma \vdash e \preceq^{ctx} e' : \tau$ to mean

$$\forall C, \tau_1. \bullet; \bullet \vdash C : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow \tau_1 \wedge C[e] \Downarrow \implies C[e'] \Downarrow .$$

3.2 $\lambda^{\forall\exists}$: Logical Relation

As in the case of λ^{rec} , the relational interpretation of a type $\mathcal{RV}[\![\tau]\!] \rho$ in $\lambda^{\forall\exists}$ is a set of triples of the form (k, v, v') . However, there is now one additional property (in addition to well-typedness of the second value of each tuple and closure under decreasing step-index) that every set χ in Rel_τ must satisfy. To motivate this property, we take the reader back to the proof of completeness of λ^{rec} , specifically to Lemma 10 which establishes that the relational value interpretation $\mathcal{RV}[\![\tau]\!]$ is equivalence-respecting. The proof of that lemma requires induction on k and nested induction on the structure of the closed type τ . In the case of $\lambda^{\forall\exists}$, when we get to the proof of the corresponding lemma, τ may have free type variables. Thus, one of the cases we must consider for the inner induction is $\tau = \alpha$. Assuming that $\rho(\alpha) = (\chi, \tau_\alpha)$, we will be required to show that if $(k, v_1, v_2) \in \mathcal{RV}[\![\alpha]\!] \rho \equiv \rho^{\text{sem}}(\alpha) \equiv \chi$ and $\vdash v_2 \preceq^{\text{ciu}} v_3 : \alpha^{[\rho]}$ (where $\alpha^{[\rho]} \equiv \tau_\alpha$), then $(k, v_1, v_3) \in \chi$. Note that $\chi \in \text{Rel}_{\tau_\alpha}$. Thus, we must add this requirement directly to the definition of Rel_τ .

A more informal justification is that in the presence of quantified types, we can instantiate a type variable with a relational interpretation of our own choosing. Thus, we have to show that the relation we pick satisfies certain properties, one of which is that it must be equivalence-respecting.

The modified definition of Rel_τ is given below. It makes use of a notion of ciu-equivalence restricted to closed values.

$$v \prec^{ciu} v' : \tau \stackrel{\text{def}}{=} \forall E, \tau_1. \bullet; \bullet \vdash E : (\bullet; \bullet \triangleright \tau) \rightsquigarrow \tau_1 \wedge E[v] \Downarrow \implies E[v'] \Downarrow$$

$$Rel_\tau \stackrel{\text{def}}{=} \{ \chi \in 2^{Nat \times CValues \times CValues} \mid \\ \forall (j, v, v') \in \chi. \vdash v' : \tau \wedge \\ \forall i \leq j. (i, v, v') \in \chi \wedge \\ (\forall v''. v' \prec^{ciu} v'' : \tau \implies (j, v, v'') \in \chi) \}$$

The relational interpretation of universal and existential types is given in Figure 5. Two values $\text{pack } v$ and $\text{pack } v'$ are related at the type $\exists \alpha. \tau$ for k steps if there exists a syntactic type τ_2 and a semantic interpretation $\chi \in Rel_{\tau_2}$ such that for all $j < k$, $(j, v, v') \in \mathcal{RV} \llbracket \tau \rrbracket \rho [\alpha \mapsto (\chi, \tau_2)]$. Here we only pick a type τ_2 for the second value v' while the type of v is left unrestricted. Intuitively, this suffices because when showing logical equivalence of two terms $(\Delta; \Gamma \vdash e \sim e' : \tau)$, we pick a type for v' while proving $\Delta; \Gamma \vdash e \leq e' : \tau$ and we pick a type for v while proving $\Delta; \Gamma \vdash e' \leq e : \tau$. The relational interpretation of universal types is the dual of existential types.

$$\mathcal{RV} \llbracket \forall \alpha. \tau \rrbracket \rho = \{ (k, \Lambda. e, \Lambda. e') \mid \vdash \Lambda. e' : (\forall \alpha. \tau)^{[\rho]} \wedge \\ \forall \tau_2, \chi. \chi \in Rel_{\tau_2} \implies \\ \forall j < k. (j, e, e') \in \mathcal{RC} \llbracket \tau \rrbracket \rho [\alpha \mapsto (\chi, \tau_2)] \}$$

$$\mathcal{RV} \llbracket \exists \alpha. \tau \rrbracket \rho = \{ (k, \text{pack } v, \text{pack } v') \mid \vdash \text{pack } v' : (\exists \alpha. \tau)^{[\rho]} \wedge \\ \exists \tau_2, \chi. \chi \in Rel_{\tau_2} \wedge \\ \forall j < k. (j, v, v') \in \mathcal{RV} \llbracket \tau \rrbracket \rho [\alpha \mapsto (\chi, \tau_2)] \}$$

$$\mathcal{RD} \llbracket \bullet \rrbracket = \{ \emptyset \}$$

$$\mathcal{RD} \llbracket \Delta, \alpha \rrbracket = \{ \rho [\alpha \mapsto (\chi, \tau_2)] \mid \rho \in \mathcal{RD} \llbracket \Delta \rrbracket \wedge \chi \in Rel_{\tau_2} \}$$

$$\mathcal{RG} \llbracket \bullet \rrbracket \rho = \{ (k, \emptyset, \emptyset) \}$$

$$\mathcal{RG} \llbracket \Gamma, x : \tau \rrbracket \rho = \{ (k, \gamma[x \mapsto v], \gamma'[x \mapsto v']) \mid (k, \gamma, \gamma') \in \mathcal{RG} \llbracket \Gamma \rrbracket \rho \wedge (k, v, v') \in \mathcal{RV} \llbracket \tau \rrbracket \rho \}$$

$$\Delta; \Gamma \vdash e \leq e' : \tau \stackrel{\text{def}}{=} \Delta; \Gamma \vdash e : \tau \wedge \Delta; \Gamma \vdash e' : \tau \wedge \\ \forall k \geq 0. \forall \rho, \gamma, \gamma'. \rho \in \mathcal{RD} \llbracket \Delta \rrbracket \wedge (k, \gamma, \gamma') \in \mathcal{RG} \llbracket \Gamma \rrbracket \rho \implies \\ (k, \gamma(e), \gamma'(e')) \in \mathcal{RC} \llbracket \tau \rrbracket \rho$$

Fig. 5. $\lambda^{\forall\exists}$ Relational Model

The relational interpretation of types as computations is defined exactly as before. The definition of the logical relation $\Delta; \Gamma \vdash e \leq e' : \tau$ appears in Figure 5.

We prove that each type τ is a valid type: $\mathcal{RV} \llbracket \tau \rrbracket \rho \in Rel_{\tau^{[\rho]}}$. Specifically, we have to show well-typedness, closure under decreasing step-index, and the following lemma.

Lemma 14 ($\lambda^{\forall\exists}$ Rel Equivalence-Respecting). *Let $\rho \in \mathcal{RD} \llbracket \Delta \rrbracket$ and $\Delta \vdash \tau$. If $(k, v_1, v_2) \in \mathcal{RV} \llbracket \tau \rrbracket \rho$ and $v_2 \prec^{ciu} v_3 : \tau^{[\rho]}$, then $(k, v_1, v_3) \in \mathcal{RV} \llbracket \tau \rrbracket \rho$.*

To show the Fundamental Property of the logical relation, we prove the new set of compatibility lemmas, as well as value and type substitutivity.

Lemma 15 ($\lambda^{\forall\exists}$ **Fundamental Property / Reflexivity**).

If $\Delta; \Gamma \vdash e : \tau$ then $\Delta; \Gamma \vdash e \leq e : \tau$.

3.3 $\lambda^{\forall\exists}$ **Soundness and Completeness**

We prove that the logical relation in Figure 5 is sound with respect to contextual equivalence. The overall proof structure is the same as for λ^{rec} .

Theorem 16 ($\lambda^{\forall\exists} : \leq \subseteq \preceq^{\text{ctx}}$). If $\Delta; \Gamma \vdash e \leq e' : \tau$ then $\Delta; \Gamma \vdash e \preceq^{\text{ctx}} e' : \tau$.

To establish completeness, we again rely on the notion of ciu-equivalence, which we define for $\lambda^{\forall\exists}$ as follows.

Definition 17 ($\lambda^{\forall\exists}$ **Ciu Approximation** \preceq^{ciu}).

Let $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma \vdash e' : \tau$. If δ is a mapping from type variables α to closed syntactic types τ , we write $\delta \models \Delta$ whenever $\text{dom}(\delta) = \Delta$.

$$\begin{aligned} \Delta; \Gamma \vdash e \preceq^{\text{ciu}} e' : \tau &\stackrel{\text{def}}{=} \forall \delta, \gamma, E, \tau_1. \delta \models \Delta \wedge \vdash \gamma : \delta(\Gamma) \wedge \\ &\bullet; \bullet \vdash E : (\bullet; \bullet \triangleright \delta(\tau)) \rightsquigarrow \tau_1 \wedge \\ &E[\gamma(e)] \Downarrow \implies E[\gamma(e')] \Downarrow \end{aligned}$$

Theorem 18 ($\lambda^{\forall\exists} : \preceq^{\text{ctx}} \subseteq \preceq^{\text{ciu}} \subseteq \leq$).

If $\Delta; \Gamma \vdash e \preceq^{\text{ctx}} e' : \tau$ then $\Delta; \Gamma \vdash e \preceq^{\text{ciu}} e' : \tau$.

If $\Delta; \Gamma \vdash e \preceq^{\text{ciu}} e' : \tau$ then $\Delta; \Gamma \vdash e \leq e' : \tau$.

3.4 **Example: Simple Existential Packages**

For lack of space, we present only one simple example (from Sumii and Pierce[19]) to illustrate the use of our logical relation to prove contextual equivalence. Additional examples involving existential packages, recursive types, and higher-order functions are given in the technical report [16].

Notation: Let χ be a set of tuples of the form (k, v, v') such that $\vdash v' : \tau$. We define the closure of χ under ciu approximation at type τ as follows:

$$\chi_\tau^* = \{(k, v_1, v_2) \mid (k, v_1, v_2) \in \chi \vee ((k, v_1, v) \in \chi \wedge v \preceq^{\text{ciu}} v_2 : \tau)\}$$

Example: Consider the following existential packages e and e' of type τ :

$$e = \text{pack} \langle 1, \lambda x. x \stackrel{\text{int}}{=} 0 \rangle \quad e' = \text{pack} \langle \text{tt}, \lambda x. \neg x \rangle \quad \tau = \exists \alpha. \alpha \times (\alpha \rightarrow \text{bool})$$

Show $\bullet; \bullet \vdash e \sim e' : \tau$. We only show $\bullet; \bullet \vdash e \leq e' : \tau$. $\bullet; \bullet \vdash e' \leq e : \tau$ is symmetric.

Suppose $k \geq 0$. Unwinding definitions, we must show $(k, e, e') \in \mathcal{RV} \llbracket \tau \rrbracket \equiv$

$$(k, \text{pack} \langle 1, \lambda x. x \stackrel{\text{int}}{=} 0 \rangle, \text{pack} \langle \text{tt}, \lambda x. \neg x \rangle) \in \mathcal{RV} \llbracket \exists \alpha. \alpha \times (\alpha \rightarrow \text{bool}) \rrbracket \emptyset.$$

Let $\chi_0 = \{(k', 1, \text{tt}) \mid k' \geq 0\}$. Take $\tau_2 = \text{bool}$ and $\chi = (\chi_0)_{\text{bool}}^*$.

Note that $\chi \in \text{Rel}_{\text{bool}}$ (from defn of $(\chi_0)_{\text{bool}}^*$). Suppose $j < k$.

Show $(j, \langle 1, \lambda x. x \stackrel{\text{int}}{=} 0 \rangle, \langle \text{tt}, \lambda x. \neg x \rangle) \in \mathcal{RV} \llbracket \alpha \times (\alpha \rightarrow \text{bool}) \rrbracket \emptyset[\alpha \mapsto (\chi, \text{bool})]$,

which follows from:

- $\vdash \langle \text{tt}, \lambda x. \neg x \rangle : (\alpha \times (\alpha \rightarrow \text{bool}))[\text{bool}/\alpha]$
- $(j, 1, \text{tt}) \in \mathcal{RV} \llbracket \alpha \rrbracket \emptyset[\alpha \mapsto (\chi, \text{bool})] \equiv (j, 1, \text{tt}) \in \chi$ (by defn of $\mathcal{RV} \llbracket \alpha \rrbracket \rho$)
which follows from $\chi \supseteq \chi_0 \supseteq \{(j, 1, \text{tt})\}$, which follows from defn of χ .

- $(j, (\lambda x. x \stackrel{\text{int}}{=} 0), (\lambda x. \neg x)) \in \mathcal{RV} \llbracket \alpha \rightarrow \text{bool} \rrbracket \emptyset[\alpha \mapsto (\chi, \text{bool})]$, which follows from:
 First, note that $\vdash \lambda x. \neg x : (\alpha \rightarrow \text{bool})[\text{bool}/\alpha] \equiv \vdash \lambda x. \neg x : \text{bool} \rightarrow \text{bool}$.
 Next, suppose $i < j$, and $(i, v_1, v'_1) \in \mathcal{RV} \llbracket \alpha \rrbracket \emptyset[\alpha \mapsto (\chi, \text{bool})]$.
 Note that $\mathcal{RV} \llbracket \alpha \rrbracket \emptyset[\alpha \mapsto (\chi, \text{bool})] \equiv \chi$ by defn of $\mathcal{RV} \llbracket \alpha \rrbracket \rho$. Hence,
 $(i, v_1, v'_1) \in \chi$.
 Then, from defn of χ , $v_1 = 1$ and, using more subtle reasoning, $v'_1 = \text{tt}$.
 Show: $(i, (x \stackrel{\text{int}}{=} 0)[v_1/x], (\neg x)[v'_1/x]) \in \mathcal{RC} \llbracket \text{bool} \rrbracket \emptyset[\alpha \mapsto (\chi, \text{bool})]$
 $\equiv (i, v_1 \stackrel{\text{int}}{=} 0, \neg v'_1) \in \mathcal{RC} \llbracket \text{bool} \rrbracket \emptyset[\alpha \mapsto (\chi, \text{bool})]$
 $\equiv (i, 1 \stackrel{\text{int}}{=} 0, \neg \text{tt}) \in \mathcal{RC} \llbracket \text{bool} \rrbracket \emptyset[\alpha \mapsto (\chi, \text{bool})]$.
 Note that $(1 \stackrel{\text{int}}{=} 0) \mapsto^1 \mathbf{ff}$ and $(\neg \text{tt}) \mapsto^* \mathbf{ff}$. Hence, remains to show:
 $(i - 1, \mathbf{ff}, \mathbf{ff}) \in \mathcal{RV} \llbracket \text{bool} \rrbracket \emptyset[\alpha \mapsto (\chi, \text{bool})]$, which is immediate.

4 Related Work and Conclusion

Logical relations were first developed for denotational semantics of typed λ -calculi (e.g., [1, 2]). Early examples of the use of logical relations based on operational semantics include Tait’s [4] proof of strong normalization of the simply typed λ -calculus, and Girard’s method of reducibility candidates [5] used to prove normalization for System F.

Pitts [7, 6, 9] developed syntactic logical relations for a λ -calculus with recursive functions and quantified types (but no recursive types). To support recursive functions without using denotational techniques, Pitts makes use of $\top\top$ -closure (or biorthogonality [12]). Relations that are $\top\top$ -closed can be immediately shown to be equivalence-respecting and admissible [9]. In comparison, we directly require that our relations be equivalence-respecting and closed under decreasing step-index — the latter, effectively, gives us admissibility.

Birkedal and Harper [10] and Crary and Harper [8] extended syntactic logical relations with recursive types (the latter also support polymorphic types) by adapting Pitts’ minimal invariance [3] technique for use in a purely syntactic setting. Mellès and Vouillon [12, 11] construct a realizability model of a language with recursive types and polymorphism based on intuitions from the ideal model of types [20]. They also present a relational model based on an orthogonality relation between quadruples of terms and contexts [12]. We note that to show completeness, they too must move to a typed setting. An issue that merits further investigation is the relationship between the different notions of approximation — i.e., syntactic projections [8], interval types [12], and step counts.

Contextual equivalence may also be proved using bisimulations. Sumii and Pierce [19] present a bisimulation for recursive and quantified types. Using their examples as a point of comparison (see [16]) we show that our logical relations are somewhat easier to use when proving contextual equivalence. Also, unlike logical relations, Sumii and Pierce note that their bisimulation cannot be used to derive free theorems [21] based only on types.

We have presented a step-indexed logical relation for recursive and impredicative quantified types. The construction is far more elementary than that of

existing logical relations for such types. In future work, we hope to scale this up to support dynamically allocated (ML-style) mutable references.

References

1. Plotkin, G.D.: Lambda-definability and logical relations. Memorandum SAI-RM-4, University of Edinburgh, Edinburgh, Scotland (1973)
2. Statman, R.: Logical relations and the typed λ -calculus. *Information and Control* **65**(2–3) (1985) 85–97
3. Pitts, A.M.: Relational properties of domains. *Information and Computation* **127**(2) (1996) 66–90
4. Tait, W.W.: Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic* **32**(2) (1967) 198–212
5. Girard, J.Y.: *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, Paris, France (1972)
6. Pitts, A.M.: Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science* **10** (2000) 321–359
7. Pitts, A.M.: Existential types: Logical relations and operational equivalence. *Lecture Notes in Computer Science* **1443** (1998) 309–326
8. Crary, K., Harper, R.: Syntactic logical relations over polymorphic and recursive types. Draft (2000)
9. Pitts, A.M.: Typed operational reasoning. In Pierce, B.C., ed.: *Advanced Topics in Types and Programming Languages*. MIT Press (2005)
10. Birkedal, L., Harper, R.: Relational interpretations of recursive types in an operational setting. In: *Theoretical Aspects of Computer Software (TACS)*. (1997)
11. Melliès, P.A., Vouillon, J.: Semantic types: A fresh look at the ideal model for types. In: *POPL*, Venice, Italy. (2004)
12. Melliès, P.A., Vouillon, J.: Recursive polymorphic types and parametricity in an operational framework. In: *LICS*, Chicago, Illinois. (2005)
13. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. *ACM TOPLAS* **23**(5) (2001) 657–683
14. Ahmed, A., Appel, A.W., Virga, R.: An indexed model of impredicative polymorphism and mutable references. Available at <http://www.cs.princeton.edu/~appel/papers/impred.pdf> (2003)
15. Ahmed, A.J.: *Semantics of Types for Mutable State*. PhD thesis, Princeton University (2004)
16. Ahmed, A.: Step-indexed syntactic logical relations for recursive and quantified types. Technical Report TR-01-06, Harvard University (2006)
17. Pierce, B.C.: *Types and Programming Languages*. MIT Press (2002)
18. Mason, I.A., Talcott, C.L.: Equivalence in functional languages with effects. *Journal of Functional Programming* **1**(3) (1991) 287–327
19. Sumii, E., Pierce, B.C.: A bisimulation for type abstraction and recursion. In: *POPL*, Long Beach, California. (2005) 63–74
20. MacQueen, D., Plotkin, G., Sethi, R.: An ideal model for recursive polymorphic types. *Information and Computation* **71**(1/2) (1986) 95–130
21. Wadler, P.: Theorems for free! In: *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, London (1989)

Approaches to Polymorphism in Classical Sequent Calculus

Alexander J. Summers and Steffen van Bakel

Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2AZ, U.K
{ajs300m,svb}@doc.ic.ac.uk

Abstract. \mathcal{X} is a relatively new calculus, invented to give a Curry-Howard correspondence with Classical Implicative Sequent Calculus. It is already known to provide a very expressive language; embeddings have been defined of the λ -calculus, Bloo and Rose's $\lambda\mathbf{x}$, Parigot's $\lambda\mu$ and Curien and Herbelin's $\bar{\lambda}\mu\tilde{\mu}$.

We investigate various notions of polymorphism in the context of the \mathcal{X} -calculus. In particular, we examine the first class polymorphism of System F, and the shallow polymorphism of ML. We define analogous systems based on the \mathcal{X} -calculus, and show that these are suitable for embedding the original calculi.

In the case of shallow polymorphism we obtain a more general calculus than ML, while retaining its useful properties. A type-assignment algorithm is defined for this system, which generalises Milner's \mathcal{W} .

1 Introduction

Polymorphism is a powerful aspect of most modern programming languages. It is a mechanism for allowing a program to be applied with various different types for its inputs (or outputs), and so allows flexibility and reuse of code. For example, in a polymorphic system, the identity function might be given the type $\forall X.(X \rightarrow X)$, where the \forall -bound type variable X ranges over all types. This correctly expresses that the identity may be typed with $A \rightarrow A$ for any and all formulas A . The rules for type-assignment typically allow this type to be *instantiated* several different times, so that it would be acceptable for the identity function to be applied to both an integer and a list in the same program.

\mathcal{X} is based on the work of [5] and [9], and has since been further studied in [10]. Like the $\lambda\mu$ -calculus of Parigot [7], it has been designed to have a Curry-Howard correspondence with Classical Logic. Unlike most existing calculi in this field (which, like $\lambda\mu$ are typically based on a Natural Deduction formulation of logic), \mathcal{X} corresponds to a Classical Sequent Calculus. The particular sequent calculus is defined by Urban [9].

In this paper we investigate various notions of polymorphism based on the logical \forall connective, in the context of the \mathcal{X} -calculus. We examine the first class polymorphism of System F, and the shallow polymorphism of ML. We define analogous systems based on the \mathcal{X} -calculus, and show that these systems are suitable for encoding System F and ML. In the case of shallow polymorphism we present a more general calculus than ML, and show that all the useful properties of ML still hold.

2 The \mathcal{X} -Calculus

In this section we will give a brief presentation of the \mathcal{X} -calculus; a more detailed description is given in [10]. We present here the syntax and reduction rules, and aim to give an intuition of how the calculus behaves.

Although \mathcal{X} provides a rather different computational behaviour to calculi based on the λ -calculus, it has been shown that it can faithfully encode many such calculi, including λ -calculus, λx , and the $\lambda\mu$ -calculus [10]. These calculi incorporate variable-symbols and (with the exception of λx) rely on an implicit concept of substitution to perform the basic computational steps. \mathcal{X} on the other hand features two separate categories of ‘connectors’, *plugs* and *sockets*, that act as input and output channels, and is defined without any notion of substitution.

Definition 1 (\mathcal{X} -Terms). The terms of the \mathcal{X} -calculus are defined by the following syntax, where x, y range over the infinite set of *sockets* and α, β over the infinite set of *plugs* (sockets and plugs together form the set of *connectors*).

$$\begin{array}{ll}
 P, Q ::= \langle x.\alpha \rangle & \text{capsule} \\
 | \widehat{y}P\widehat{\beta}.\alpha & \text{export} \\
 | P\widehat{\beta}[y]\widehat{x}Q & \text{mediator} \\
 | P\widehat{\alpha}\dagger\widehat{x}Q & \text{cut}
 \end{array}$$

The $\widehat{}$ symbolises that the connector underneath is bound in the circuit; notions of *free* and *bound* connectors are defined as usual. We will use $fp(P)$ to denote the free plugs of P , and similarly $fs(P)$ for free sockets.

The notion of reduction on \mathcal{X} -terms corresponds to the process of *cut elimination* on sequent calculus proofs. As such, the reduction rules define how cuts may be eliminated from an \mathcal{X} -term. If a cut binds a connector which occurs several times in the corresponding subterm, it may not be immediately eliminated, but rather must seek out each of these occurrences and make a copy of itself for each one. For example, if there are many occurrences of x in the term Q then the term $P\widehat{\alpha}\dagger\widehat{x}Q$ can reduce by ‘pushing’ the cut into the structure of Q , and making a cut between a copy of P and each x found. This process of ‘pushing’ is correctly referred to as propagation, in this example right-propagation (since we propagate the cut into the right-hand term). Once the cut reaches a level where a single α and x are immediately introduced in its two subterms, a *logical rule* specifies how the two subterms can communicate with one another through the cut. For example, a cut between an export and a mediator allows the body of the function from the export to be inserted between the two subterms of the mediator.

Definition 2 (Logical Rules). The logical rules are presented by:

$$\begin{array}{ll}
 (cap) : & \langle y.\alpha \rangle \widehat{\alpha}\dagger\widehat{x}\langle x.\beta \rangle \rightarrow \langle y.\beta \rangle \\
 (exp) : & (\widehat{y}P\widehat{\beta}.\alpha) \widehat{\alpha}\dagger\widehat{x}\langle x.\gamma \rangle \rightarrow \widehat{y}P\widehat{\beta}.\gamma \quad \alpha \notin fs(P) \\
 (med) : & \langle y.\alpha \rangle \widehat{\alpha}\dagger\widehat{x}(P\widehat{\beta}[x]\widehat{z}Q) \rightarrow P\widehat{\beta}[y]\widehat{z}Q \quad x \notin fs(P, Q) \\
 (exp-med) : & (\widehat{y}P\widehat{\beta}.\alpha) \widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}R) \rightarrow \left\{ \begin{array}{l} Q\widehat{\gamma}\dagger\widehat{y}(P\widehat{\beta}\dagger\widehat{z}R) \\ (Q\widehat{\gamma}\dagger\widehat{y}P)\widehat{\beta}\dagger\widehat{z}R \end{array} \right\} \quad \begin{array}{l} \alpha \notin fs(P), \\ x \notin fs(Q, R) \end{array}
 \end{array}$$

The first three logical rules above specify a renaming (reconnecting) procedure, whereas the last rule specifies the basic computational step: it links the exportation of a function, available on the plug α , to an adjacent mediator via the socket x (the resulting cuts may be bracketed either way, as shown).

A key element of the cut-elimination procedure of [9] is that cuts which are propagated to the left or right are marked as such.

Definition 3 (Active Cuts). The syntax is extended with two *flagged* or *active* cuts:

$$P ::= \dots \mid P_1 \hat{\alpha} \not\vdash \hat{x} P_2 \mid P_1 \hat{\alpha} \not\backslash \hat{x} P_2$$

We define two *cut-activation* rules.

$$\begin{aligned} (\text{act-L}) : P \hat{\alpha} \dagger \hat{x} Q &\rightarrow P \hat{\alpha} \not\vdash \hat{x} Q \text{ if } P \text{ does not introduce } \alpha \\ (\text{act-R}) : P \hat{\alpha} \dagger \hat{x} Q &\rightarrow P \hat{\alpha} \not\backslash \hat{x} Q \text{ if } Q \text{ does not introduce } x \end{aligned}$$

where: P introduces x : Either $P = Q \hat{\beta} [x] \hat{y} R$ and $x \notin \text{fs}(Q, R)$, or $P = \langle x.\alpha \rangle$.

P introduces α : Either $P = \hat{x} Q \hat{\beta} \cdot \alpha$ and $\alpha \notin \text{fp}(Q)$, or $P = \langle x.\alpha \rangle$.

An activated cut is processed by ‘pushing’ it systematically through the syntactic structure of the circuit in the direction indicated by the tilting of the dagger. Whenever an active cut meets a circuit exhibiting the connector it is trying to communicate with, a new (inactive) cut is ‘deposited’, representing an attempt to communicate at this level. The pushing of the active cut continues until the level of capsules is reached, where it is either deactivated or destroyed. Once again, the inactive cut can reduce via a logical rule, or pushing can continue in the other direction. This behaviour is expressed by the following propagation rules.

Definition 4 (Propagation Rules).

Left Propagation:

$$\begin{aligned} (\not\vdash) : & \langle y.\alpha \rangle \hat{\alpha} \not\vdash \hat{x} P \rightarrow \langle y.\alpha \rangle \hat{\alpha} \dagger \hat{x} P \\ (\not\text{cap}) : & \langle y.\beta \rangle \hat{\alpha} \not\vdash \hat{x} P \rightarrow \langle y.\beta \rangle, \quad \beta \neq \alpha \\ (\not\text{exp-outs}) : & (\hat{y} Q \hat{\beta} \cdot \alpha) \hat{\alpha} \not\vdash \hat{x} P \rightarrow (\hat{y}(Q \hat{\alpha} \not\vdash \hat{x} P) \hat{\beta} \cdot \gamma) \hat{\gamma} \dagger \hat{x} P, \quad \gamma \text{ fresh} \\ (\not\text{exp-ins}) : & (\hat{y} Q \hat{\beta} \cdot \gamma) \hat{\alpha} \not\vdash \hat{x} P \rightarrow \hat{y}(Q \hat{\alpha} \not\vdash \hat{x} P) \hat{\beta} \cdot \gamma, \quad \gamma \neq \alpha \\ (\not\text{med}) : & (Q \hat{\beta} [z] \hat{y} R) \hat{\alpha} \not\vdash \hat{x} P \rightarrow (Q \hat{\alpha} \not\vdash \hat{x} P) \hat{\beta} [z] \hat{y}(R \hat{\alpha} \not\vdash \hat{x} P) \\ (\not\text{cut}) : & (Q \hat{\beta} \dagger \hat{y} R) \hat{\alpha} \not\vdash \hat{x} P \rightarrow (Q \hat{\alpha} \not\vdash \hat{x} P) \hat{\beta} \dagger \hat{y}(R \hat{\alpha} \not\vdash \hat{x} P) \end{aligned}$$

Right Propagation:

$$\begin{aligned} (\not\backslash) : & P \hat{\alpha} \not\backslash \hat{x} \langle x.\beta \rangle \rightarrow P \hat{\alpha} \dagger \hat{x} \langle x.\beta \rangle \\ (\not\backslash \text{cap}) : & P \hat{\alpha} \not\backslash \hat{x} \langle y.\beta \rangle \rightarrow \langle y.\beta \rangle, \quad y \neq x \\ (\not\backslash \text{exp}) : & P \hat{\alpha} \not\backslash \hat{x} (\hat{y} Q \hat{\beta} \cdot \gamma) \rightarrow \hat{y}(P \hat{\alpha} \not\backslash \hat{x} Q) \hat{\beta} \cdot \gamma \\ (\not\backslash \text{med-outs}) : & P \hat{\alpha} \not\backslash \hat{x} (Q \hat{\beta} [z] \hat{y} R) \rightarrow P \hat{\alpha} \dagger \hat{z} ((P \hat{\alpha} \not\backslash \hat{x} Q) \hat{\beta} [z] \hat{y}(P \hat{\alpha} \not\backslash \hat{x} R)), \quad z \text{ fresh} \\ (\not\backslash \text{med-ins}) : & P \hat{\alpha} \not\backslash \hat{x} (Q \hat{\beta} [z] \hat{y} R) \rightarrow (P \hat{\alpha} \not\backslash \hat{x} Q) \hat{\beta} [z] \hat{y}(P \hat{\alpha} \not\backslash \hat{x} R), \quad z \neq x \\ (\not\backslash \text{cut}) : & P \hat{\alpha} \not\backslash \hat{x} (Q \hat{\beta} \dagger \hat{y} R) \rightarrow (P \hat{\alpha} \not\backslash \hat{x} Q) \hat{\beta} \dagger \hat{y}(P \hat{\alpha} \not\backslash \hat{x} R) \end{aligned}$$

The symmetry of the cut can be seen by these rules - it may (depending on the conditions on the activation rules) be propagated to the left or right, making copies of the right or left term respectively. Right-propagation is reminiscent of *substitution* of terms for term-variables; left-propagation $P\hat{\alpha} \not\vdash \hat{x}Q$ then is its dual: it expresses the connection of the continuation Q , accessible via x , to all the ‘calls’ α in P .

We write \rightarrow for the (reflexive, transitive, compatible) reduction relation generated by the logical, propagation and activation rules. The reduction relation \rightarrow is not confluent; this comes in fact from the critical pair that activates a cut $P\hat{\alpha} \not\vdash \hat{x}Q$ in two ways if P does not introduce α and Q does not introduce x .

Definition 5 ([10]). The interpretation of lambda terms into circuits of \mathcal{X} via the plug α , $\llbracket M \rrbracket_{\alpha}^{\lambda}$, is defined by:

$$\begin{aligned} \llbracket x \rrbracket_{\alpha}^{\lambda} &= \langle x.\alpha \rangle \\ \llbracket \lambda x.M \rrbracket_{\alpha}^{\lambda} &= \hat{x} \llbracket M \rrbracket_{\beta}^{\lambda} \hat{\beta}.\alpha, & \beta \text{ fresh} \\ \llbracket MN \rrbracket_{\alpha}^{\lambda} &= \llbracket M \rrbracket_{\gamma}^{\lambda} \hat{\gamma} \not\vdash \hat{x}(\llbracket N \rrbracket_{\beta}^{\lambda} \hat{\beta} [x] \hat{y}\langle y.\alpha \rangle), \quad x, y, \beta, \gamma \text{ fresh} \end{aligned}$$

In [10] it is shown that this interpretation respects (CBN/CBV) reduction and typeability.

Notice that every sub-circuit of $\llbracket M \rrbracket_{\alpha}^{\lambda}$ has exactly one free plug. This can be seen as an explicit notation for the output of the lambda term (outputs are not explicitly labelled in λ -calculus).

3 Type Assignment for \mathcal{X}

The notion of type assignment on \mathcal{X} that we present in this section is the basic implicative system for Classical Logic. The Curry-Howard property is easily achieved.

Definition 6 (Types and Contexts).

1. The set of types $\mathcal{T}_{\mathcal{C}}$, ranged over by A, B , is defined over a set of *atomic types* $\mathcal{V} = \{\varphi_1, \varphi_2, \varphi_3, \dots\}$ by the grammar:

$$A, B ::= \varphi \mid A \rightarrow B$$

These types are normally known as *Curry types*.

2. A *context of sockets* Γ is a mapping from sockets to types, denoted as a finite set of *statements* $x:A$, such that the *subjects* of the statements (the sockets) are distinct. We write $\Gamma, x:A$ for $\Gamma \cup \{x:A\}$. When writing a context as $\Gamma, x:A$, we indicate that either Γ is not defined on x or contains the same statement $x:A$. We write $\Gamma \setminus x$ for the context from which the statement concerning x , if any, has been removed. *Contexts of plugs* Δ , and the notations $\alpha:A, \Delta$ and $\Delta \setminus \alpha$ are defined in a similar way.
3. A pair $\langle \Gamma; \Delta \rangle$ is usually referred to simply as a *context*, and is a shorthand for the sequent $\Gamma \vdash \Delta$.

The notation $\Gamma \vdash \Delta$ will still usually be used when discussing sequents.

Definition 7 (Typing for \mathcal{X}).

1. *Type judgements* are expressed via a ternary relation $P : \cdot \Gamma \vdash \Delta$, where Γ is a context of *sockets* and Δ is a context of *plugs*, and P is an \mathcal{X} -term. We say that P is the *witness* of this judgement.
2. *Type assignment* is defined by the following sequent calculus:

$$\begin{aligned}
 (cap) : \frac{}{\langle y.\alpha \rangle : \cdot \Gamma, y:A \vdash \alpha:A, \Delta} \quad (med) : \frac{P : \cdot \Gamma \vdash \alpha:A, \Delta \quad Q : \cdot \Gamma, x:B \vdash \Delta}{P\hat{\alpha} [y] \hat{x}Q : \cdot \Gamma, y:A \rightarrow B \vdash \Delta} \\
 (exp) : \frac{P : \cdot \Gamma, x:A \vdash \alpha:B, \Delta}{\hat{x}P\hat{\alpha}.\beta : \cdot \Gamma \vdash \beta:A \rightarrow B, \Delta} \quad (cut) : \frac{P : \cdot \Gamma \vdash \alpha:A, \Delta \quad Q : \cdot \Gamma, x:A \vdash \Delta}{P\hat{\alpha} \dagger \hat{x}Q : \cdot \Gamma \vdash \Delta}
 \end{aligned}$$

We write $P : \cdot \Gamma \vdash \Delta$ if there exists a derivation that has this judgement in the bottom line.

Notice that, in $P : \cdot \Gamma \vdash \Delta$, Γ and Δ carry the types of the free connectors in P , as unordered sets. By the Curry-Howard correspondence, P represents a proof of the sequent $\Gamma \vdash \Delta$, so P is actually a witness to this sequent being derivable in the logic. Moreover, there is no notion of a single type for P itself, instead the derivable statement shows the consistency between the free connectors of P .

It is important to note that the typing rules include a notion of implicit contraction (just as the original sequent rules do); if a new statement is introduced on the bottom line of a rule, but it was already present in the context, then it is simply merged. We do not consider duplicate statements, as we consider contexts to be unordered sets.

We have the following result:

Theorem 8 (Witness Reduction [10]). *If $P : \cdot \Gamma \vdash \Delta$, and $P \rightarrow Q$, then $Q : \cdot \Gamma \vdash \Delta$.*

Also, the standard notion of Curry type assignment on lambda terms and the notion of type assignment on \mathcal{X} defined above are strongly linked:

Theorem 9 ([10]). *If $\Gamma \vdash_{\lambda} M : A$, then $\llbracket M \rrbracket_{\alpha}^{\lambda} : \cdot \Gamma \vdash \alpha:A$.*

In [11] a notion of *principal contexts* (principal typings, in the language of [12]) is defined by providing an algorithm pC that, given an \mathcal{X} -term P , returns a context $\langle \Gamma; \Delta \rangle$, with the following properties:

Theorem 10 (Soundness and Completeness of pC).

1. *Soundness:* If $pC(P) = \langle \Gamma; \Delta \rangle$, then $P : \cdot \Gamma \vdash \Delta$.
2. *Completeness:* If $P : \cdot \Gamma \vdash \Delta$, then there exist Γ_p and Δ_p , and a substitution S such that $pC(P) = \langle \Gamma_p; \Delta_p \rangle$, and $(S \Gamma_p) \subseteq \Gamma$ and $(S \Delta_p) \subseteq \Delta$.

4 System F in \mathcal{X}

In this section, we will examine the System F approach to polymorphism, and how it may be incorporated into the \mathcal{X} -calculus. We will present System F, and show it can be expressed in an \mathcal{X} setting, by giving an explicit encoding into a variant of the \mathcal{X} -calculus. We will show that typings and reductions are preserved by this encoding.

4.1 System F

System F (also known as the Polymorphic λ -calculus) was invented independently by Jean-Yves Girard [4] and John C. Reynolds [8]. We will give here a short overview of its main definitions, based largely on those of [3].

Definition 11 (System F Types). The types of System F (ranged over by A, B) are defined over an infinite set of *atomic types* (ranged over by φ), and one of type variable-symbols (ranged over by X, Y), in the following way:

$$A, B ::= \varphi \mid X \mid A \rightarrow B \mid \forall X. A$$

A type is *well-formed* if and only if it contains no free type variable-symbols (i.e. every such symbol X appears under a $\forall X$ binder). It is useful to consider types modulo some kind of alpha-conversion, for example we would like to identify the types $\forall X.(X \rightarrow X)$ and $\forall Y.(Y \rightarrow Y)$. From here on we will assume this.

Definition 12 (System F). The terms of System F (à la Church) are defined over an infinite set of typed term variable-symbols, $\{x^A, y^B, \dots\}$, where A, B can be any System F type. They are defined by the following syntax:

$$M, N ::= x^A \mid \lambda x^A. M^1 \mid MN \mid \Lambda \varphi. M^2 \mid MA$$

¹: if x^B appears free in M , then $B = A$.

²: φ does not appear in the type of a free term variable of M .

The syntax as described above is in fact rather too liberal; a notion of *well-formed terms* will be employed, which insists that terms must have a well-formed type. It is simple to derive the type of a particular System F term (unique, modulo alpha conversion) from the type information within the syntax. We will write $M :_F A$ to denote that A is the type of the term M .

Definition 13 (Type Derivation in System F). The procedure of type derivation is defined as follows:

$$\begin{array}{c} \frac{}{x^A :_F A} (Ax) \qquad \frac{M :_F B}{\lambda x^A. M :_F A \rightarrow B} (\rightarrow I) \qquad \frac{M :_F A \rightarrow B \quad N :_F A}{(MN) :_F B} (\rightarrow E) \\[10pt] \frac{M :_F A}{\Lambda \varphi. M :_F \forall X. A[X/\varphi]} (\forall I) \qquad \frac{M :_F \forall X. A}{(MB) :_F A[B/X]} (\forall E) \end{array}$$

For example, we would not consider the term $(x^A y^A)$ to be well-formed, since (according to the rules above) it does not have a type. In addition, we would not consider the term $\lambda x^X. x^X$ to be well-formed (its type is $X \rightarrow X$ where X is free). As an example of a well-formed term, the identity function would be represented in System F by the term $\Lambda \varphi. \lambda x^\varphi. x^\varphi$, which has the type $\forall X.(X \rightarrow X)$. From here onwards we will assume terms are well-formed unless otherwise stated.

Definition 14 (System F Reductions). There are two reduction rules:

$$\begin{array}{l} (\lambda x^A. M) N \rightarrow_F M[N/x^A] \\ (\Lambda \varphi. M) A \rightarrow_F M[A/\varphi] \end{array}$$

In general, we will write \rightarrow_F for the reflexive, transitive, compatible closure of the relation generated by these rules.

System F à la Church possesses a Curry-Howard correspondence with the ‘ \forall, \rightarrow ’-fragment of Intuitionistic Natural Deduction. Since each term carries only one type, the correspondence between terms and proofs is in fact one-to-one.

To illustrate the polymorphism in this system, we can find a typeable term analogous with the lambda term $(\lambda z.zz)(\lambda x.x)$. The term we would use is

$$(\lambda z^{\forall Z.(Z \rightarrow Z)}. \Lambda \varphi_1. ((z^{\forall Z.(Z \rightarrow Z)} (\varphi_1 \rightarrow \varphi_1)) (z^{\forall Z.(Z \rightarrow Z)} \varphi_1))) (\Lambda \varphi_2. \lambda x^{\varphi_2}. x^{\varphi_2})$$

4.2 Typed Polymorphic λ

One possible method of introducing polymorphism to λ is to go back to the sequent calculus rules, and encode the quantifier rules there into the syntax of a typed version of λ . This gives typed λ -terms which naturally carry polymorphic types. This approach is analogous to that of System F, where the original implicative calculus (typed λ -calculus) is extended with representations of the ‘ \forall ’ rules.

The \forall -rules from the sequent calculus are as follows:

$$\frac{\Gamma, A[B/X] \vdash \Delta}{\Gamma, \forall X. A \vdash \Delta} (\forall \mathcal{L}) \quad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash \forall X. A[X/\varphi], \Delta} (\forall \mathcal{R})^*$$

* if φ does not occur in Γ, Δ .

Notice that (as is typical of the sequent calculus rules) quantifiers are only introduced (and not eliminated), but may be introduced on the left of a sequent (which approximately corresponds to elimination in a Natural Deduction setting).

We introduce two new terms, representing the rules $(\forall \mathcal{L})$ and $(\forall \mathcal{R})$, and give a typed version of the existing syntax.

Definition 15 (Typed Polymorphic λ). The terms of Typed Polymorphic λ (hereafter denoted by λ^\forall) are defined by the following syntax:

$$P, Q ::= \langle x^A. \alpha^A \rangle \mid \widehat{y}^A P \widehat{\beta}^B. \alpha^{A \rightarrow B} \mid P \widehat{\beta}^A [y^{A \rightarrow B}] \widehat{x}^B Q \\ \mid P \widehat{\alpha}^A \dagger \widehat{x}^A Q \mid P \widehat{\alpha}^A \triangleleft_{\varphi} \beta^{\forall X. A[X/\varphi]} (*) \mid y^{\forall X. A} \triangleright_B \widehat{x}^{A[B/X]} Q$$

(*) φ does not appear in the type of a free connector of P , except (possibly) α^A .

The notation \triangleleft is chosen to indicate the *generalisation* of the output α , whereas the symbol \triangleright denotes the corresponding *instantiation*. Although instantiation is really a ‘Natural Deduction way’ of considering this mechanism (where there is an elimination rule to do the job), the concept still makes sense in reading the term from left to right, since this means reading the $(\forall \mathcal{L})$ rule from the bottom upwards.

Notice that no process of derivation is required in determining the type (context) of an λ^\forall term - the context may be immediately formed by taking a statement for each free connector in the term, with the type it has there. This is because outputs are labelled as well as inputs, so all of the pertinent information is present in the term. For example, the λ^\forall term $\widehat{y}^A \langle x^B. \beta^B \rangle \widehat{\beta}^B. \alpha^{A \rightarrow B}$ would be given the context $x:B \vdash \alpha:A \rightarrow B$. We write $P : \Gamma \vdash_{\forall} \Delta$ to indicate that $\Gamma \vdash \Delta$ is the context for the λ^\forall term P .

It is straightforward to convert the original \mathcal{X} reduction rules into their typed versions. The extra rules required to deal with the new syntax constructs are given in Appendix A. We will write \rightarrow_{\forall} for the reduction relation for \mathcal{X}^{\forall} .

We have the following result:

Theorem 16 (Witness Reduction for \mathcal{X}^{\forall}). *For all \mathcal{X}^{\forall} -terms P, Q , if $P : \cdot \Gamma_P \vdash_{\forall} \Delta_P$, and $P \rightarrow_{\forall} Q$ and $Q : \cdot \Gamma_Q \vdash_{\forall} \Delta_Q$, then $\Gamma_Q \subseteq \Gamma_P$ and $\Delta_Q \subseteq \Delta_P$.*

So our new formulation of the calculus is well-behaved with respect to the type-assignment proposed.

It is possible to encode System F à la Church into \mathcal{X}^{\forall} , just as \mathcal{X} can encode the original λ -calculus. The interpretation is based on the translation from Natural Deduction to Sequent Calculus proofs, as originally given in [2].

The interpretation function takes as input a System F term and a plug α (used to represent the output in the resulting \mathcal{X}^{\forall} term) and returns the corresponding \mathcal{X}^{\forall} term. It makes use of the derivation of the (unique) type of a System F term, of Definition 13.

Definition 17 (Encoding System F à la Church). The interpretation of System F into \mathcal{X}^{\forall} , via the plug α is defined recursively by:

$$\begin{array}{ll}
 \llbracket x^A \rrbracket_{\alpha}^{\forall} &= \langle x^A. \alpha^A \rangle & \llbracket MN \rrbracket_{\alpha}^{\forall} &= P\hat{\beta}^C \dagger \hat{x}^C (Q\hat{\gamma}^A [x^C] \hat{y}^B \langle y^B. \alpha^B \rangle) \\
 \llbracket \lambda x^A. M \rrbracket_{\alpha}^{\forall} &= \hat{x}^A P\hat{\beta}^B \cdot \alpha^C & \text{where} & M :_F A \rightarrow B \\
 & \text{where} & M :_F B & N :_F A \\
 & & P = \llbracket M \rrbracket_{\beta}^{\forall} & P = \llbracket M \rrbracket_{\beta}^{\forall} \\
 & & C = A \rightarrow B & Q = \llbracket N \rrbracket_{\gamma}^{\forall} \\
 & & & C = A \rightarrow B \\
 \llbracket \lambda \varphi. M \rrbracket_{\alpha}^{\forall} &= P\hat{\beta}^A \triangleleft_{\varphi} \alpha^B & \llbracket MB \rrbracket_{\alpha}^{\forall} &= P\hat{\beta}^C \dagger \hat{x}^C (x^C \triangleright_B \hat{y}^D \langle y^D. \alpha^D \rangle) \\
 & \text{where} & M :_F A & \text{where} & M :_F \forall X. A \\
 & & P = \llbracket M \rrbracket_{\beta}^{\forall} & & P = \llbracket M \rrbracket_{\beta}^{\forall} \\
 & & B = \forall X. A[X/\varphi] & & C = \forall X. A \\
 & & & & D = A[B/X]
 \end{array}$$

The following results show that we can simulate System F faithfully.

Theorem 18.

1. If $M \rightarrow_F N$ then $\llbracket M \rrbracket_{\alpha}^{\forall} \rightarrow_{\forall} \llbracket N \rrbracket_{\alpha}^{\forall}$.
2. If $M :_F A$ then there exists a Γ such that $\llbracket M \rrbracket_{\alpha}^{\forall} : \cdot \Gamma \vdash_{\forall} \alpha : A$.

4.3 Untyped Polymorphic \mathcal{X}

As an alternative to \mathcal{X}^{\forall} , it is possible to work in the style of System F à la Curry and deal with the original syntax of \mathcal{X} while allowing polymorphism to be represented only in the type system. This is essentially achieved by employing System F types, and by adding the following two type assignment rules to those standard for \mathcal{X} .

$$\frac{P : \cdot \Gamma, x : A[B/X] \vdash_{\mathbb{P}} \Delta}{P : \cdot \Gamma, x : \forall X. A \vdash_{\mathbb{P}} \Delta} \quad (\forall \mathcal{L}) \quad \frac{P : \cdot \Gamma \vdash_{\mathbb{P}} \alpha : A, \Delta}{P : \cdot \Gamma \vdash_{\mathbb{P}} \alpha : \forall X. A[X/\varphi], \Delta} \quad (\forall \mathcal{R})^*$$

*if φ does not occur in Γ, Δ .

We encode System F à la Curry by the usual encoding of the λ -calculus syntax into \mathcal{X} , as given in Definition 5. This encoding respects typeability and reductions.

5 Shallow Polymorphism

In this section, we will examine the style of polymorphism commonly associated with ML, that of *shallow polymorphism*. We will show that a shallow polymorphic type assignment can be naturally defined on \mathcal{X} -terms without the need to extend the syntax (in contrast to the case of the λ -calculus). We will show that ML can be encoded into \mathcal{X} , and that using this new type-assignment, typings and reductions are preserved. We will discuss the notions of principal types and typings [12] with respect to our shallow polymorphic version of \mathcal{X} , and present a type inference algorithm in the style of the algorithm \mathcal{W} of [6].

ML [6] is a calculus based upon the λ -calculus, which uses a different approach to System F for admitting polymorphism. To obtain decidability of type assignment, it permits only *shallow polymorphism*, which means that types are allowed to contain the \forall symbol only on the outside of their structure.

The syntax of the λ -calculus is extended the construct $\text{let } x = M \text{ in } N$ which (along with its typing rule) is designed to give a workaround for the situation when an application $(\lambda x.N)M$ would be untypeable, whereas the reduct $N[M/x]$ can be typed. The typing rule for let allows M to be given a shallow polymorphic type, and for this type to be used for x when trying to derive a type for N . This way, it may be that several instances of the polymorphic type are used for different occurrences of x within N .

Definition 19 (ML Expressions). The set \mathcal{L}_{ML} of ML expressions is defined by:

$$M, N ::= x \mid MN \mid \lambda x.M \mid \text{Fix } g.M \mid \text{let } x = M \text{ in } N$$

The construct $\text{Fix } g.M$ is included to allow recursion in the calculus. For simplicity in our discussions of polymorphism we choose to study the subset of ML expressions without Fix , and from hereon will consider ML expressions only within this subset.

Definition 20 (ML Reductions). The reduction rules in ML are as follows:

$$\begin{aligned} (\lambda x.N)M &\rightarrow_{\text{ML}} N[M/x] \\ (\text{let } x = M \text{ in } N) &\rightarrow_{\text{ML}} N[M/x] \end{aligned}$$

The typing rules for let provide the polymorphism in this system - it is allowed for each of the occurrences of x in N to be given a different instance of a polymorphic type found for M . This is in contrast to the usual way in which the term $(\lambda x.N)M$ would be treated, which would allow only *one* Curry type to be used for the variable x .

Definition 21 (Generic Types [6]). The set of *generic types* is built from the usual Curry types by allowing \forall quantifiers to be built on the outside. We will use A, B to range over the usual Curry types, and ψ to range over generic types, as defined below.

$$\begin{aligned}
A &::= \varphi \mid X \mid (A \rightarrow B) && \text{Curry types} \\
\psi &::= A \mid (\forall X.\psi) && \text{generic types}
\end{aligned}$$

As in the discussions in the previous section, we distinguish between atomic types φ and type variable symbols X (whereas Milner chooses not to), and again consider only types with no free type-variable symbols to be well-formed.

Definition 22 ([1]). *ML-type assignment* and *ML-derivations* are defined by the following deduction system.

$$\begin{array}{ll}
(ax) : \dfrac{}{\Gamma \vdash_{\text{ML}} x : \psi} \quad (x:\psi \in \Gamma) & (let) : \dfrac{\Gamma \vdash_{\text{ML}} M_1 : \psi \quad \Gamma, x:\psi \vdash_{\text{ML}} M_2 : B}{\Gamma \vdash_{\text{ML}} (\text{let } x = M_1 \text{ in } M_2) : B} \\
(\rightarrow I) : \dfrac{\Gamma, x:A \vdash_{\text{ML}} M : B}{\Gamma \vdash_{\text{ML}} \lambda x.M : A \rightarrow B} & (\rightarrow E) : \dfrac{\Gamma \vdash_{\text{ML}} M_1 : A \rightarrow B \quad \Gamma \vdash_{\text{ML}} M_2 : A}{\Gamma \vdash_{\text{ML}} M_1 M_2 : B} \\
(\forall I) : \dfrac{\Gamma \vdash_{\text{ML}} M : \psi}{\Gamma \vdash_{\text{ML}} M : \forall X.\psi[X/\varphi]} (*) & (\forall E) : \dfrac{\Gamma \vdash_{\text{ML}} M : \forall X.\psi}{\Gamma \vdash_{\text{ML}} M : \psi[B/X]}
\end{array}$$

*If φ is not free in Γ .

Notice that generic types ψ may not be used in the $(\rightarrow I)$ or $(\rightarrow E)$ rules - this reflects the fact that \forall -symbols may not appear inside an arrow type. However, when x is a variable not occurring under an abstraction, the rules allow more freedom - if x has a polymorphic type in the basis then the use of the (ax) and $(\forall E)$ rules allows a different instance of this type to be chosen each time x is used.

Although ML admits less polymorphism than System F does, it has the advantage of being very practical - not only is type assignment in ML decidable (in contrast to System F), but it has a principal type property. Milner presents an algorithm (called \mathcal{W}) that takes as input a pair of (*basis*, *term*) and returns a pair of (*substitution*, *type*), representing the most general typing for the term (if one exists) using a substitution instance of the basis.

6 ML in \mathcal{X}

The key to the use of polymorphism in ML is in the `let` construct, which is interpreted as a substitution both syntactically (according to its reduction rule) and semantically (see [6]). The polymorphism present in the (let) -rule essentially gives a way of typing the substitution about to take place, such that the multiple occurrences of the name to replace need not all be typed in the same way. The `let`-construct is a necessary extension to the syntax for a shallow polymorphic approach (short of allowing polymorphism to be used directly with abstractions and applications, which leads to System F), since there is nothing in the syntax of the λ -calculus to represent these substitutions.

In the \mathcal{X} -calculus, there is a construct already present which can be seen to represent substitution. The cut $P\hat{\alpha} \dagger \hat{x}Q$ can, depending on the structure of P and Q , be seen to represent the substitution of P for the x 's in Q , or symmetrically the substitution of Q for α 's in P .

A subtle problem occurs in defining a shallow polymorphic type assignment, which motivates a relaxation of Definition 6 to allow multiple statements in a context with the same subject. The main reason for this is in the manipulation of quantified types, when we wish to take several instances of a type in the same derivation. It should be noted that in a sequent calculus setting, instances are taken on the same side of the sequent as the quantified type appeared (see the $\forall\mathcal{L}$ rule of Definition 23 below). We wish many such instances to be available (to make full use of the polymorphism in the system), and this causes us a difficulty, since all must be types for the same connector. For example the (*med*)-rule

$$(med) : \frac{P \vdash \Gamma \vdash \alpha:A, \Delta \quad Q \vdash \Gamma, x:B \vdash \Delta}{P\hat{\alpha}[y] \hat{x}Q \vdash \Gamma, y:A \rightarrow B \vdash \Delta}$$

(which adds a type for y to the context Γ) would be expressed awkwardly: assume $y:C$ already occurs in Γ , then, given the polymorphic character of types, we can accept that $A \rightarrow B$ and C are different, as long as they are all instances of the same quantified type. In other words, we can assume that $y:\forall\vec{\varphi}.D \in \Gamma$, and ask that $A \rightarrow B$ can be obtained from D by instantiation. This would give a complicated side-condition to the rule.

Instead, we choose to relax Definition 6, in that we now allow multiple statements in a context with the same subject. However, in order to retain soundness, we insist that whenever the rules (*exp*), (*med*) and (*cut*) are employed, the connectors mentioned in the top line of the rule (which are bound in the construction of the respective terms) have a unique statement in the rule. This enforces that all the types for a connector disappear from the contexts when the connector is bound. We also insist that a derivation is not complete unless the subjects of the statements in the final sequent are unique (so the relaxation is only usable temporarily within a derivation). As a consequence of these restrictions, if several statements with the same subject (but different types) are used in a derivation, it will be necessary for the \forall rules to be applied until the types of these statements match, and they are contracted into a single statement. Until this takes place, it will be impossible to either bind the connective concerned, or complete the derivation.

Definition 23 (Shallow Polymorphic Type Assignment for \mathcal{X}). The shallow polymorphic type assignment for \mathcal{X} is defined by the following rules (where ψ represents a generic type of Definition 21):

$$\begin{aligned} (cap) : & \frac{}{\langle y.\alpha \rangle \vdash \Gamma, y:\psi \vdash_{\text{SP}} \alpha:\psi, \Delta} & (med) : & \frac{P \vdash \Gamma \vdash_{\text{SP}} \alpha:A, \Delta \quad Q \vdash \Gamma, x:B \vdash \Delta}{P\hat{\alpha}[y] \hat{x}Q \vdash \Gamma, y:A \rightarrow B \vdash_{\text{SP}} \Delta} \quad (2) \\ (exp) : & \frac{P \vdash \Gamma, x:A \vdash_{\text{SP}} \alpha:B, \Delta}{\hat{x}P\hat{\alpha}.\beta \vdash \Gamma \vdash_{\text{SP}} \beta:A \rightarrow B, \Delta} \quad (1) & (cut) : & \frac{P \vdash \Gamma \vdash_{\text{SP}} \alpha:\psi, \Delta \quad Q \vdash \Gamma, x:\psi \vdash_{\text{SP}} \Delta}{P\hat{\alpha} \dagger \hat{x}Q \vdash \Gamma \vdash_{\text{SP}} \Delta} \quad (3) \\ (\forall\mathcal{L}) : & \frac{P \vdash \Gamma, x:\psi[B/X] \vdash_{\text{SP}} \Delta}{P \vdash \Gamma, x:\forall X.\psi \vdash_{\text{SP}} \Delta} & (\forall\mathcal{R}) : & \frac{P \vdash \Gamma \vdash_{\text{SP}} \alpha:\psi, \Delta}{P \vdash \Gamma \vdash_{\text{SP}} \alpha:\forall X.\psi[X/\varphi], \Delta} \quad (4) \end{aligned}$$

¹: if $x \notin \Gamma$ and $\alpha \notin \Delta$. ^{2,3}: if $x \notin \Gamma$ and $\alpha \notin \Delta$. ⁴: if φ does not occur in Γ, Δ .

We include a notion of implicit contraction in the above rules (as for the type system presented in Section 3), so that if a derivation rule introduces a statement which was already present in the context, it is simply merged.

Notice that generic types are not used in the (*exp*) or (*med*) rules. This enforces the restriction that the \forall -symbol may not appear to the left of an ' \rightarrow ' in a type, and is similar to the way the ($\rightarrow I$) and ($\rightarrow E$) rules are treated in ML.

We have the following result:

Theorem 24 (Witness Reduction). *If $P : \Gamma \vdash_{\text{sp}} \Delta$, and $P \rightarrow Q$, then $Q : \Gamma \vdash_{\text{sp}} \Delta$.*

Using our previous observation concerning the fact that `let` and a cut both explicitly represent a substitution, we define an encoding of the language of ML into \mathcal{X} .

Definition 25 (Encoding ML in \mathcal{X}).

$$\begin{aligned} \llbracket x \rrbracket_{\alpha}^{\text{ML}} &= \langle x.\alpha \rangle \\ \llbracket \lambda x.M \rrbracket_{\alpha}^{\text{ML}} &= \widehat{x} \llbracket M \rrbracket_{\beta}^{\text{ML}} \widehat{\beta} \cdot \alpha \\ \llbracket MN \rrbracket_{\alpha}^{\text{ML}} &= \llbracket M \rrbracket_{\beta}^{\text{ML}} \widehat{\beta} \dagger \widehat{y} (\llbracket N \rrbracket_{\gamma}^{\text{ML}} \widehat{\gamma} [y] \widehat{z} \langle z.\alpha \rangle) \\ \llbracket \text{let } x = M \text{ in } N \rrbracket_{\alpha}^{\text{ML}} &= \llbracket M \rrbracket_{\beta}^{\text{ML}} \widehat{\beta} \dagger \widehat{x} \llbracket N \rrbracket_{\alpha}^{\text{ML}} \end{aligned}$$

where y, z, β, γ are fresh connectors.

We have the following results for our encoding:

Theorem 26. 1. $\llbracket M \rrbracket_{\beta}^{\text{ML}} \widehat{\beta} \dagger \widehat{x} \llbracket N \rrbracket_{\alpha}^{\text{ML}} \rightarrow \llbracket (N[M/x]) \rrbracket_{\alpha}^{\text{ML}}$.
 2. If $M \rightarrow_{\text{ML}} N$ then $\llbracket M \rrbracket_{\alpha}^{\text{ML}} \rightarrow \llbracket N \rrbracket_{\alpha}^{\text{ML}}$.
 3. If $\Gamma \vdash_{\text{ML}} M : \psi$ then $\llbracket M \rrbracket_{\beta}^{\text{ML}} : \Gamma \vdash_{\text{sp}} \beta : \psi$.

In fact, the converse of part 3 also holds if we restrict the right-context in our \mathcal{X} typing judgement to contain only a statement for β (any other information would be redundant since β is the only free plug in such a term). This implies that the possible typings for M in ML and $\llbracket M \rrbracket_{\beta}^{\text{ML}}$ in shallow-polymorphic \mathcal{X} are essentially the same. Since Wells proves in [12] that ML does not in general have principal typings (i.e. when the basis of assumptions is unspecified, there is no pair of basis and type which represents all other possible typings), this immediately implies that the same is the case of our shallow polymorphic version of \mathcal{X} .

On the other hand, it is well known that a notion of principal types for ML terms exists (as presented by Milner), with respect to a fixed basis Γ . We can define principal typings in our shallow polymorphic version of \mathcal{X} , with respect to a given context $\langle \Gamma; \Delta \rangle$ which gives a type to the free connectors in a term. Notice that such a context provides types for the outputs as well as the inputs.

We define an algorithm, based on the \mathcal{W} algorithm of [1], which takes as input an \mathcal{X} -term and a context $\langle \Gamma; \Delta \rangle$, and produces as output a substitution S , giving the most general solution to the problem of typing the term with a (substitution) instance of $\langle \Gamma; \Delta \rangle$. We require that no types in Δ contain the \forall symbol - the intention is that Γ provides any known licence to use polymorphism in the type search. In defining this algorithm (which we will name $\mathcal{W}^{\mathcal{X}}$), we require the following definition.

Definition 27 (\forall -closure). The \forall -closure of type ψ with respect to a context $\langle \Gamma; \Delta \rangle$, is defined by: $\forall\text{-closure } \psi \langle \Gamma; \Delta \rangle = \forall X_1 \dots \forall X_n. (\psi[X_i/\varphi_i])$ where $\varphi_1, \dots, \varphi_n$ are the atomic types occurring in ψ but not in $\langle \Gamma; \Delta \rangle$.

We are now in a position to define our type-inference algorithm.

Definition 28 ($\mathcal{W}^{\mathcal{X}}$). The procedure $\mathcal{W}^{\mathcal{X}} :: \langle \mathcal{X}, \langle \Gamma; \Delta \rangle \rangle \rightarrow \mathcal{S}$ is defined by:

$$\begin{array}{ll}
\mathcal{W}^{\mathcal{X}}(\langle x.\alpha \rangle, \langle \Gamma; \Delta \rangle) = S & \mathcal{W}^{\mathcal{X}}(P\hat{\alpha}[y] \hat{x}Q, \langle \Gamma; \Delta \rangle) = S_3 \circ S_2 \circ S_1 \\
\text{where } A = \text{instance } x \Gamma & \text{where } \varphi_1 = \text{fresh} \\
B = \text{instance } \alpha \Delta & \varphi_2 = \text{fresh} \\
S = \text{unify } A B & S_1 = \mathcal{W}^{\mathcal{X}}(P, \langle \Gamma; \Delta \cup \alpha : \varphi_1 \rangle) \\
& S_2 = \mathcal{W}^{\mathcal{X}}(Q, (S_1 \langle \Gamma \cup x : \varphi_2; \Delta \rangle)) \\
& A = (S_2 \circ S_1 \varphi_1) \\
& B = (S_2 \circ S_1 \varphi_2) \\
& C = \text{instance } y (S_2 \circ S_1 \Gamma) \\
& S_3 = \text{unify } C A \rightarrow B \\
\\
\mathcal{W}^{\mathcal{X}}(\hat{x}P\hat{\alpha}.\beta, \langle \Gamma; \Delta \rangle) = S_2 \circ S_1 & \\
\text{where } \varphi_1 = \text{fresh} & \\
\varphi_2 = \text{fresh} & \\
S_1 = \mathcal{W}^{\mathcal{X}}(P, \langle \Gamma \cup x : \varphi_1; & \\
\Delta \cup \alpha : \varphi_2 \rangle) & \mathcal{W}^{\mathcal{X}}(P\hat{\alpha} \dagger \hat{x}Q, \langle \Gamma; \Delta \rangle) = S_2 \circ S_1 \\
A = (S_1 \varphi_1) & \text{where } \varphi = \text{fresh} \\
B = (S_1 \varphi_2) & S_1 = \mathcal{W}^{\mathcal{X}}(P, \langle \Gamma; \Delta \cup \alpha : \varphi \rangle) \\
C = \text{instance } \beta (S_1 \Delta) & \psi = \forall\text{-closure } (S_1 \varphi) (S_1 \langle \Gamma; \Delta \rangle) \\
S_2 = \text{unify } C A \rightarrow B & S_2 = \mathcal{W}^{\mathcal{X}}(Q, \langle (S_1 \Gamma) \cup x : \psi; (S_1 \Delta) \rangle)
\end{array}$$

where *instance* is a mapping that takes the type associated to the given connective in the given context and replaces all \forall -bound type-variables by fresh atomic types. Note that since we prohibit Δ from containing the \forall symbol, our uses of *instance* on a plug α merely extract the type for α from the context.

In order to reason about this context being truly principal, we need a notion of ‘more general’ for quantified types.

Definition 29 (Generic Instance). A type scheme $\psi = \forall X_1 \dots \forall X_m. A$ has a *generic instance* $\psi' = \forall Y_1 \dots \forall Y_n. A'$ if there exists a type B such that:

1. There exist types B_1, \dots, B_m with $B = A[B_i/X_i]$.
2. There exist atomic types $\varphi_1, \dots, \varphi_n$ such that $A' = B[Y_i/\varphi_i]$, and the φ_i are not free in ψ .

We write $\psi' \preceq \psi$ in this case, read “ ψ' is a generic instance of ψ ”.

We extend this notion to contexts, by defining \preceq on contexts to be the least preorder such that:

$$\begin{aligned}
\psi' \preceq \psi &\Rightarrow \langle \Gamma; \Delta, \alpha : \psi' \rangle \preceq \langle \Gamma; \Delta, \alpha : \psi \rangle \\
\psi' \preceq \psi &\Rightarrow \langle \Gamma, x : \psi; \Delta \rangle \preceq \langle \Gamma, x : \psi'; \Delta \rangle
\end{aligned}$$

\forall -closure is extended to right-contexts by taking the closure of each statement.

Definition 30 (\forall -closure for Contexts). We define the closure of a context $\langle \Gamma; \Delta \rangle$ by

$$\text{closure } \langle \Gamma; \Delta \rangle = \langle \Gamma; \Delta' \rangle$$

where $\Delta' = \{ \alpha : \psi' \mid \alpha : \psi \in \Delta \text{ and } \psi' = \forall\text{-closure } \psi \langle \Gamma; \Delta \setminus \alpha \rangle \}$

We can now define our notion of principal contexts for shallow polymorphic \mathcal{X} , by running the algorithm $\mathcal{W}^{\mathcal{X}}$, applying the resulting substitution, and taking the closure of the result.

Definition 31 (Principal Contexts for Shallow Polymorphic \mathcal{X}). Given any \mathcal{X} -term P , and a context $\langle \Gamma; \Delta \rangle$ which provides types for exactly the free connectors in P , we define the *shallow polymorphic principal context* for P with respect to a $\langle \Gamma; \Delta \rangle$ by:

$$\begin{aligned} \text{sppc}(P, \langle \Gamma; \Delta \rangle) &= \text{closure}((S \langle \Gamma; \Delta \rangle)) \\ &\text{where } S = \mathcal{W}^{\mathcal{X}}(P, \langle \Gamma; \Delta \rangle) \end{aligned}$$

Notice that *sppc* may or may not succeed, depending on whether the call to $\mathcal{W}^{\mathcal{X}}$ does. The following result justifies this definition.

Theorem 32 (Soundness and Completeness of *sppc*). *Given an \mathcal{X} -term P and an initial context $\langle \Gamma_1; \Delta_1 \rangle$,*

1. *If $\text{sppc}(P, \langle \Gamma_1; \Delta_1 \rangle)$ succeeds and $\langle \Gamma; \Delta \rangle = \text{sppc}(P, \langle \Gamma_1; \Delta_1 \rangle)$ then $P : \cdot \vdash_{\text{sp}} \Delta$.*
2. *If $\langle \Gamma_2; \Delta_2 \rangle$ is an instance of $\langle \Gamma_1; \Delta_1 \rangle$ (i.e. can be obtained from the latter by substitution), and is such that $P : \cdot \vdash_{\text{sp}} \Delta_2$ then:*
 - (a) *$\text{sppc}(P, \langle \Gamma_1; \Delta_1 \rangle)$ succeeds.*
 - (b) *If $\langle \Gamma; \Delta \rangle = \text{sppc}(P, \langle \Gamma_1; \Delta_1 \rangle)$ then there is a substitution S such that $\langle \Gamma_2; \Delta_2 \rangle \preceq (S \langle \Gamma; \Delta \rangle)$.*

In summary, we have shown that in our shallow polymorphic formulation of \mathcal{X} we can faithfully simulate ML reductions, we have decidable type-assignment (at least as strong as that of ML) and principal typings with respect to a fixed basis of assumptions (in the style of \mathcal{W}). While we retain all these useful properties, our calculus is more general than ML because of its basis on Classical Sequent Calculus. We can give typeable programs which have no analogue in ML (for example, we can give a program that has Pierce's Law as a type), and can treat terms with multiple outputs. Furthermore, since cut elimination is well-known to be non-confluent, we can simulate non-determinism, a feature not present in ML. The precise computational content of these various extensions is the subject of ongoing research.

7 Future Work

We are interested in investigating further useful programming features in the context of \mathcal{X} , like recursion.

At present, we only model polymorphism in the same style that ML does, in generalising an output on the left of a cut and then taking instances for the various inputs on the right. Since \mathcal{X} is a very symmetric calculus, a natural idea to explore is the use of polymorphism in the opposite direction: to generalise the input of the right-hand term, and take instances for the (possibly several) outputs on the left. In logical terms, this can be seen as introducing the \exists connective to the system, and investigations into a system based on this observation are ongoing. While a 'dual' notion of polymorphism is fairly straightforward to define (where we allow \exists in the type system instead of \forall), it is more complicated to allow both kinds of polymorphism at once. This is a promising line of future work, and is expected to yield a very powerful decidable type system for \mathcal{X} .

Acknowledgements. We would like to thank Jayshan Raghunandan, Pierre Lescanne, Dragisa Zunic, Dorian Gaertner and the (anonymous) referees for their generous feedback and discussions on the subject of this paper.

References

1. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
2. Gerhard Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.
3. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
4. Jean-Yves Girard. Une extension de l’interprétation de Gödel à l’analyse et son application à l’élimination des coupures dans l’analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium (Univ. Oslo, Oslo, 1970)*, volume 63, pages 63–92. North-Holland, 1971.
5. Stéphane Lengrand. Call-by-value, call-by-name, and strong normalization for the classical sequent calculus. In Bernhard Gramlich and Salvador Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.
6. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
7. M. Parigot. An algorithmic interpretation of classical natural deduction. In *Proc. of Int. Conf. on Logic Programming and Automated Reasoning, LPAR’92*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, 1992.
8. J.C. Reynolds. Towards a Theory of Type Structures. In B. Robinet, editor, *Proceedings of ‘Colloque sur la Programmation’*, Paris, France, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.
9. Christian Urban. *Classical Logic and Computation*. PhD thesis, University of Cambridge, October 2000.
10. S. van Bakel, S. Lengrand, and P. Lescanne. The language \mathcal{X} : circuits, computations and classical logic. In *Proc. 10th Italian Conf. on Theoretical Computer Science (ICTCS’05)*, volume 3701 of *Lecture Notes in Computer Science*, pages 66–80. Springer-Verlag, 2005.
11. S. van Bakel, J. Raghunandan, and A. Summers. Term Graphs and Principal Types for \mathcal{X} . Unpublished, <http://www.doc.ic.ac.uk/~svb/Research>, 2005.
12. J. B. Wells. The essence of principal typings. In *Proc. 29th Int’l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.

A New Reduction Rules for \mathcal{X}^\forall

The new key logical rule which is introduced, is the following:

$$\begin{aligned}
 (poly) : \quad & (P \hat{\alpha}^A \triangleleft_{\beta}^{\forall X. A[X/\varphi]} \hat{\beta}^{\forall X. A[X/\varphi]} \dagger \hat{y}^{\forall X. A[X/\varphi]} (y^{\forall X. A[X/\varphi]} \triangleright_{\beta} \hat{x}^A[B/\varphi] Q) \\
 & \rightarrow (P[B/\varphi]) \hat{\alpha}^A[B/\varphi] \dagger \hat{x}^A[B/\varphi] Q \quad \text{if } \beta \notin fp(P), y \notin fs(Q)
 \end{aligned}$$

This rule is complex, more so than the existing logical rules, because of the need to account for the instantiation of the polymorphic type involved. It is necessary for a type substitution to be made throughout P , in order for the type of the output α to be able to

agree with the type of the input x . This is where the instantiation happens - once a copy of P has been propagated to meet a single term Q with which it can communicate, an appropriate instance of P is taken. Of course, many such copies of P may have been made by this stage, and it is the polymorphic type for α which allows each of these to be instantiated in different (and possibly non-unifiable) ways.

We omit the type information from the following rules for brevity. However, except in the case of the *poly* rule already described, the types are not necessary for understanding the operation of the rules.

Logical Rules

$$\begin{aligned} (\text{gen}) : & \quad (P\hat{\beta}\triangleleft\alpha)\hat{\alpha}\dagger\hat{x}\langle x.\gamma \rangle \rightarrow P\hat{\beta}\triangleleft\gamma \quad \alpha \notin fp(P) \\ (\text{inst}) : & \quad \langle y.\alpha \rangle\hat{\alpha}\dagger\hat{x}(x\triangleright\hat{z}P) \rightarrow y\triangleright\hat{z}P \quad x \notin fs(P) \\ (\text{poly}) : & \quad (P\hat{\beta}\triangleleft\alpha)\hat{\alpha}\dagger\hat{x}(x\triangleright\hat{y}Q) \rightarrow P\hat{\beta}\dagger\hat{y}Q \quad \alpha \notin fp(P), x \notin fs(Q) \end{aligned}$$

Activation Rules

$$\begin{aligned} (\text{act-L}) : & \quad P\hat{\alpha}\dagger\hat{x}Q \rightarrow P\hat{\alpha}\not\sim\hat{x}Q \text{ if } P \text{ does not introduce } \alpha \\ (\text{act-R}) : & \quad P\hat{\alpha}\dagger\hat{x}Q \rightarrow P\hat{\alpha}\not\sim\hat{x}Q \text{ if } Q \text{ does not introduce } x \end{aligned}$$

where:

P introduces x : $P = Q\hat{\beta}[x]\hat{y}R$ and $x \notin fs(Q, R)$, or $P = x\triangleright\hat{y}Q$ and $x \notin fs(Q)$, or $P = \langle x.\alpha \rangle$.

P introduces α : $P = \hat{x}Q\hat{\beta}.\alpha$ and $\alpha \notin fp(Q)$, or $P = Q\hat{\beta}\triangleleft\alpha$ and $\alpha \notin fp(Q)$, or $P = \langle x.\alpha \rangle$.

Left Propagation

$$\begin{aligned} (\not\sim\text{gen-outs}) : & \quad (Q\hat{\beta}\triangleleft\alpha)\hat{\alpha}\not\sim\hat{x}P \rightarrow ((Q\hat{\alpha}\not\sim\hat{x}P)\hat{\beta}\triangleleft\gamma)\hat{\gamma}\dagger\hat{x}P, \gamma \text{ fresh} \\ (\not\sim\text{gen-ins}) : & \quad (Q\hat{\beta}\triangleleft\gamma)\hat{\alpha}\not\sim\hat{x}P \rightarrow (Q\hat{\alpha}\not\sim\hat{x}P)\hat{\beta}\triangleleft\gamma, \quad \gamma \neq \alpha \\ (\not\sim\text{inst}) : & \quad (y\triangleright\hat{z}Q)\hat{\alpha}\not\sim\hat{x}P \rightarrow y\triangleright\hat{z}(Q\hat{\alpha}\not\sim\hat{x}P) \end{aligned}$$

Right Propagation

$$\begin{aligned} (\not\sim\text{gen}) : & \quad P\hat{\alpha}\not\sim\hat{x}(Q\hat{\beta}\triangleleft\gamma) \rightarrow (P\hat{\alpha}\not\sim\hat{x}Q)\hat{\beta}\triangleleft\gamma \\ (\not\sim\text{inst-outs}) : & \quad P\hat{\alpha}\not\sim\hat{x}(x\triangleright\hat{y}Q) \rightarrow P\hat{\alpha}\dagger\hat{z}(z\triangleright\hat{y}(P\hat{\alpha}\not\sim\hat{x}Q)), z \text{ fresh} \\ (\not\sim\text{inst-ins}) : & \quad P\hat{\alpha}\not\sim\hat{x}(z\triangleright\hat{y}R) \rightarrow z\triangleright\hat{y}(P\hat{\alpha}\not\sim\hat{x}R), \quad z \neq x \end{aligned}$$

Pure Pattern Calculus

Barry Jay¹ and Delia Kesner²

¹ University of Technology, Sydney
cbj@it.uts.edu.au

² PPS, CNRS and Université Paris 7
kesner@pps.jussieu.fr

Abstract. The pure pattern calculus generalises the pure lambda-calculus by basing computation on pattern-matching instead of beta-reduction. The simplicity and power of the calculus derive from allowing any term to be a pattern. As well as supporting a uniform approach to functions, it supports a uniform approach to data structures which underpins two new forms of polymorphism. *Path polymorphism* supports searches or queries along all paths through an arbitrary data structure. *Pattern polymorphism* supports the dynamic creation and evaluation of patterns, so that queries can be customised in reaction to new information about the structures to be encountered. In combination, these features provide a natural account of tasks such as programming with XML paths.

As the variables used in matching can now be eliminated by reduction it is necessary to separate them from the binding variables used to control scope. Then standard techniques suffice to ensure that reduction progresses and to establish confluence of reduction.

1 Introduction

The lambda-calculus is a theory of functions which is powerful enough to model arbitrary computations. In its pure form every term is a function, so that function arguments are themselves functions. Such higher-order functions give a clean account of recursion as the application of the fixpoint function. Also data structures such as pairs, lists and trees can be modelled as higher-order functions that take as arguments functions that are to act on the data stored within the structure. Central to the expressive power of the lambda-calculus is that a single rule, beta-reduction, is used to describe the evaluation of an arbitrary function. This uniformity allows a single function to be applied in a variety of different situations, i.e. supports function polymorphism. Unfortunately, the description of data structures is not so uniform. Although the lambda-calculus supports functions that act uniformly on all pairs, or all lists, it cannot support operations that exploit characteristics common to all data structures. These include operations for searching, updating and aggregating that are at the heart of data processing but do not make sense with lambda-abstractions.

In a way, this is surprising because such operations can be specified quite simply. Every data structure is either an *atom* or a *compound*. For example, every list is either empty or is compounded from a head and a tail, every tree

is either a leaf or a node. With this characterisation, one can define searching a data structure d as follows:

1. if d is the goal then return d ;
2. else if d is a compound data structure then traverse its components;
3. else stop.

For example, consider the problem of listing all the points in a data structure. Each point is represented by terms of the form $\text{Point } t$ where Point is a constructor used to represent points whose data is represented by t but the nature of the structure holding the points is not known. Let the syntax $[x_1, \dots, x_n]$ be used for the list whose entries are given by x_1, \dots, x_n and let $s@t$ be the result of appending s to the front of t . Now the solution can be given by a pattern-matching function defined by *cases*

```

letrec listPoints =
  Point y      → [Point y]
  | x y        → (listPoints x) @ (listPoints y)
  | y          → []

```

The most interesting case of the three is the second, whose pattern $x y$ is able to match against an *arbitrary* compound data structure. For example, when `listPoints` is applied to a pair `Pair s t` of points s and t we get

$$\begin{aligned}
 \text{listPoints (Pair } s \ t) &= ((\text{listPoints Pair}) @ (\text{listPoints } s)) @ (\text{listPoints } t) \\
 &= ([] @ [s]) @ [t] \\
 &= [s, t]
 \end{aligned}$$

This uniform approach to compound data structures supports *path polymorphism* in which all paths through a data structure can be traversed.

Another example of path polymorphism is the function that updates point data within an arbitrary data structure. It is given by

```

letrec updatePoint = f →
  Point y      → Point (f y)
  | z y        → (updatePoint f z) (updatePoint f y)
  | y          → y

```

Further generalisation is achieved by making `Point` a parameter to the *generic update function* defined by

```

letrec update = x → f →
  x y      →y x (f y)
  | z y    → (update x f z) (update x f y)
  | y      → y

```

This time the two variables in the pattern $x y$ above behave quite differently as x is a *free variable* ready to be substituted by, say, `Point` while y is a binding variable, as usual. To distinguish these alternatives, the arrow in the case is

decorated with a set of *binding variables*, in this case just y . Where no subscript is specified then all the free variables of the pattern are assumed bound.

The function **update** is *pattern polymorphic*, as it contains the free variable x in the pattern $x\ y$ whose instantiation can produce a variety of different update functions. For example, **update** Point reduces to **update**Point. Further, if **update** is applied to a case then the pattern must be reduced before matching can occur.

Such flexibility in the use of patterns leads to the following leitmotiv:

any term can be a pattern.

This complements the view in lambda-calculus that any term can be a function.

Hence, the *pure pattern calculus* has term syntax

$$t ::= x \mid \bullet \mid t\ t \mid t \rightarrow_{\theta} t$$

consisting of variables, a constructor, applications and cases $p \rightarrow_{\theta} s$ where θ is a set of variables, its *binding variables*. The sole reduction rule is motivated by the equation

$$(p \rightarrow_{\theta} s)\ u = \{p/u\}_{\theta} s \tag{1}$$

where $\{p/u\}_{\theta}$ is the *match* of p against u that produces either a substitution with domain θ or a failure.

A key challenge is to determine when a pattern is irreducible, and so is ready for matching. The difficulty arises from some pathological examples in which reduction of the pattern is blocked by binding variables of its own case. That resolved, the pure pattern calculus is fairly well behaved. In particular, every closed term of the form $(p \rightarrow_{\theta} s)\ u$ is reducible. Also, reduction is confluent.

The simplicity of the pure pattern calculus is best appreciated by comparing to previous approaches to pattern-matching. Popular functional programming languages such as Standard ML [SML], OCAML [Oca] and Haskell [Has] only support irreducible patterns which are either headed by a constructor or are a binding variable. Recent research has sought to augment the collection of patterns with new constructions [KPT96], reducible patterns [CK98] and free variables which do not bound occurrences in the body of the program [BCKL03], and patterns for compound data structures [Jay04c]. Only the last of these supports path polymorphism and none of them supports pattern polymorphic examples.

Moreover, the last of these underpinned the development of typed calculi supporting pattern polymorphism (e.g. [Jay04a]) which sought to allow more dynamic patterns (for us, polymorphism is about re-usability, which may be formalised by typing). These have been used to support the generic update, and its extension to handle arbitrary XML paths [HJS05a, HJS05b]. They have also been used to support an object model able to support central goals of object-orientation [Jay04b]. Again, they provide an account of *structure polymorphism* [JC94, JBM98, Jay04c] necessary to support operations such as mapping and folding in a uniform way, similar to *polytypic* or *generic functional programming* [Jan00, BdMH96, GJ03].

All these calculi attempted to control variable binding by restricting the class of patterns and their reduction. However, simplicity comes by treating binding

separately from the pattern itself. It is expected that all of the applications above can be re-engineered in the new, simpler, framework.

The structure of the rest of the paper is as follows. Section 2 introduces the terms. Section 3 defines reduction. Section 4 considers some examples. Section 5 shows that matching does not get stuck. Section 6 proves reduction is confluent. Section 7 draws conclusions and considers further work.

2 Terms

Fix a countable alphabet of variables (meta-variables $\dots x, y, z$). Let θ denote a finite set of variables. The term syntax of the *pure pattern calculus* is

$$\begin{array}{ll} \text{Terms } t ::= x & \text{(variable)} \\ & \bullet \quad \text{(constructor)} \\ & t \ t \quad \text{(application)} \\ & t \rightarrow_{\theta} t \quad \text{(case)} \end{array}$$

The variables are available for binding, matching and substitution. The constructor is used to build data structures. In practice, other primitive constructors could be considered but we do not include them for simplicity. The application $r \ u$ applies the *function* r to its *argument* u . The case $p \rightarrow_{\theta} s$ is formed of a *pattern* p and a *body* s linked by the set θ of *binding variables*. Application is left-associative and case is right-associative. Application binds tighter than case. For example $x \rightarrow x \ y \ z \rightarrow_y y$ is equal to $x \rightarrow ((x \ y) \ z) \rightarrow_y y$. Lambda-abstraction can be defined by setting $\lambda x. t$ to be $x \rightarrow_x t$.

Free variables of terms are defined by:

$$\begin{aligned} \text{fv}(x) &= \{x\} \\ \text{fv}(\bullet) &= \{\} \\ \text{fv}(r \ u) &= \text{fv}(r) \cup \text{fv}(u) \\ \text{fv}(p \rightarrow_{\theta} s) &= (\text{fv}(p) \cup \text{fv}(s)) \setminus \theta. \end{aligned}$$

Hence the binding variables of a case bind their free occurrences in both the pattern and body. A term is *closed* if it has no free variables.

The notation $p \rightarrow s$ stands for $p \rightarrow_{\text{fv}(p)} s$. Hence programmers need never actually mention binding variables explicitly unless they require free variables in the pattern.

2.1 Matches

A *substitution* σ is a partial function from variables to terms. The notation $\{x_1/u_1, \dots, x_n/u_n\}$ represents the substitution that maps x_i to u_i for $i = 1 \dots n$ and $\{\}$ denotes the empty substitution. A *match* (meta-variable m) is either a *successful match*, given by a substitution, or a *failure*, denoted by **none**. The usual concepts and notation associated with substitutions will be defined for arbitrary matches.

The *domain* of σ is denoted $\text{dom}(\sigma)$. The domain of **none** is the empty set. The set of *free variables* of σ is given by the union of the sets $\text{fv}(\sigma x)$ where $x \in \text{dom}(\sigma)$. Also, **none** has no free variables. Define the *variables* of m to be $\text{var}(m) = \text{dom}(m) \cup \text{fv}(m)$.

The *application* of a substitution σ to a term is defined by

$$\begin{aligned} \sigma x &= \sigma x && \text{if } x \in \text{dom}(\sigma) \\ \sigma x &= x && \text{if } x \notin \text{dom}(\sigma) \\ \sigma \bullet &= \bullet \\ \sigma(r \ u) &= (\sigma r) (\sigma u) \\ \sigma(p \rightarrow_\theta s) &= \sigma p \rightarrow_\theta \sigma s \text{ if } \text{var}(\sigma) \cap \theta = \{\} \end{aligned}$$

The restriction on the definition of $\sigma(p \rightarrow_\theta s)$ is necessary to avoid a *variable clash* which would change the semantics of the term. Variable clashes will be handled by α -conversion.

If matching fails in Equation (1) then **none** will be applied to the body of the case, which should be discarded. One possibility is to introduce a special error term, but match failure provides a natural branching mechanism which can be used to underpin the definitions of conditionals and pattern-matching functions. Hence, we define

$$\text{none } t = x \rightarrow_x x.$$

Given matches m_1 and m_2 then $m_1 \uplus m_2$ is the match defined as follows. If m_1 and m_2 are substitutions σ_1 and σ_2 whose domains are disjoint then $\sigma_1 \uplus \sigma_2$ is defined by

$$(\sigma_1 \uplus \sigma_2)x = \begin{cases} \sigma_1 x & \text{if } x \in \text{dom}(\sigma_1) \\ \sigma_2 x & \text{if } x \in \text{dom}(\sigma_2) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

In all other circumstances $m_1 \uplus m_2 = \text{none}$. Disjoint domains will be used to ensure that matching is deterministic.

The *composition* $\sigma_2 \circ \sigma_1$ of two substitutions σ_1 and σ_2 is defined by $(\sigma_2 \circ \sigma_1)x = \sigma_2(\sigma_1 x)$. Further, if m_1 and m_2 are matches of which at least one is **none** then $m_2 \circ m_1$ is defined to be **none**.

The *check* m_θ of a match m on a set of variables θ is m if m is a substitution whose domain is exactly θ and is **none** otherwise.

2.2 Alpha Conversion

Let θ be a set of variables and x and y be variables. Then $\{x/y\}\theta$ is defined to be the set obtained by replacing x by y in θ if $x \in \theta$ and $y \notin \theta$, and to be undefined otherwise.

Alpha conversion is the congruence relation generated by the following axiom

$$p \rightarrow_\theta s =_\alpha \{x/y\}p \rightarrow_{\{x/y\}\theta} \{x/y\}s \quad \text{if } y \notin \text{fv}(p) \cup \text{fv}(s).$$

For example, $x \ y \rightarrow_y x \ (f \ y) =_\alpha x \ z \rightarrow_z x \ (f \ z)$ if z is not free in f .

Lemma 1. *For every substitution σ and term t there is an α -equivalent term t' such that $\sigma t'$ is defined. If t_1 and t_2 are α -equivalent terms then $\text{fv}(t_1) = \text{fv}(t_2)$ and if $u_1 = \sigma t_1$ and $u_2 = \sigma t_2$ are both defined then $u_1 =_\alpha u_2$.*

Proof. The proofs are by straightforward inductions.

From now on, a *term* is an α -equivalence class in the term syntax.

3 Reduction

Reduction proceeds in two stages: first generate a match and then apply it. The richness of the class of patterns introduces some complexity to the matching process, even in the handling of variables.

The most common situation is that the free variables of a pattern are binding variables and so ready to be matched, as in $(x \rightarrow_x x) \bullet$ or $(x y \rightarrow_{\{x,y\}} y) (\bullet \bullet)$ where x and y both bind to \bullet . In pattern polymorphic examples such as the generic update a pattern may contain a free variable that is awaiting substitution. Then matching of the pattern must be delayed until the value of the variable is known, as in $x \rightarrow (x y \rightarrow_y y) (\bullet \bullet)$. There is, however, a third possibility which is illustrated by the following closed term

$$t = ((x \rightarrow_{\{\}} x) x \rightarrow_x x) (\bullet \bullet). \quad (2)$$

The pattern p given by $(x \rightarrow_{\{\}} x) x$ contains a free variable x which cannot be replaced by substitution, as it is a binding variable of t . Hence there is no way that p can ever be reduced and so it is natural to treat it as a compound data structure in order to match against its parts. Then the match of $x \rightarrow_{\{\}} x$ against \bullet will fail so that matching fails rather than gets stuck, as desired.

The difficulty with this approach is in determining that p is irreducible within t . Of course, this depends upon the status of x , so the notion of data structure must now be parametrised by a set of “fixed” variables φ and some means found of characterising the irreducible applications such as p . The easiest way we have found is to define the φ -data structures in terms of irreducibility of their parts, which in turn depends upon the data structures within them, in a virtuous cycle.

One can also consider matching of cases. Although feasible, it would require that matching be parametrised by yet another set of variables, representing those which are bound within the pattern itself, so is left for another occasion. In this paper, case matching will always fail.

Finally, there is a substantial literature concerning the appropriate treatment of non-linear patterns, i.e. those in which a binding variable x appears twice, such as $x x$. One approach would be to allow $x x$ to match with terms of the form $u u$. However, this may cause a loss of confluence, as in [FK03, Kah03], for reasons grounded in Klop’s observation that the combination of untyped λ -calculus with *non left-linear* first-order rewriting systems breaks confluence [Klo80]. Since non-linear patterns cannot be banned (any term can be a pattern) our solution is

to prevent $x x$ from matching anything, even a term of the form $u u$. This is handled implicitly by the requirement that the union of substitutions is only defined when their domains are disjoint.

3.1 Matching

The definitions of data structures, matchable forms, matching and reduction that follow are all mutually recursive. That is, the definition of reduction depends upon that of data structures which in turn depends on the irreducibility of some sub-terms.

Let φ be a set of variables. Define the φ -data structures (meta-variable d) by

$$\begin{aligned} d ::= & x && \text{if } x \in \varphi \\ & \bullet && \\ & (p \rightarrow_{\theta} s) u && \text{if } (p \rightarrow_{\theta} s) u \text{ is irreducible and all its free variables are in } \varphi \\ & d u. && \end{aligned}$$

For example, $\bullet u$ is a φ -data structure for any term u . Also, $(x \rightarrow_{\{\}} x) x$ will prove to be irreducible, and so to be a $\{x\}$ -data structure.

Define the *data structures* to be the $\{\}$ -data structures. The φ -matchable forms are the φ -data structures and all cases. The *matchable forms* are the $\{\}$ -matchable forms.

The *basic matching* $\{p//u\}_{\theta}$ of a term p (called the *pattern*) against a term u (called the *argument*) relative to a set θ of *binding variables* is the partial operation defined by the following equations

$$\begin{aligned} \{x//u\}_{\theta} &= \{x/u\} && \text{if } x \in \theta \\ \{\bullet//\bullet\}_{\theta} &= \{\} \\ \{q p//v u\}_{\theta} &= \{q//v\}_{\theta} \uplus \{p//u\}_{\theta} && \text{if } q p \text{ is a } \theta\text{-matchable form} \\ &&& \text{and } v u \text{ is a matchable form} \\ \{p \rightarrow_{\psi} s//u\}_{\theta} &= \text{none} \\ \{p//u\}_{\theta} &= \text{none} && \text{if } p \text{ is a } \theta\text{-matchable form} \\ &&& \text{and } u \text{ is a matchable form} \\ \{p//u\} &= \text{undefined} && \text{otherwise} \end{aligned}$$

where the equations are to be applied in order. That is, matching is always defined if the pattern is a θ -matchable form and the argument is a matchable form, and match failure can only arise if rules for successful matching do not apply. A binding variable matches anything. The constructor matches itself. Matching of compound data structures is component-wise, using (disjoint) union. Note that the ordering of the equations can be avoided by expanding the definition into an induction on the structure of the pattern.

For example, evaluation of t in (2) will use the match $m = \{(x \rightarrow_{\{\}} x) x // \bullet\bullet\}_{\{x\}}$. Now $\{x//x\}_{\{\}}$ is undefined and so $(x \rightarrow_{\{\}} x) x$ turns out to be irreducible and thus is an $\{x\}$ -data structure. Hence m matches $x \rightarrow_{\{\}} x$ against \bullet but fails, and so m is **none**.

Let p and u be terms and let θ be a set of variables. Define the *matching* $\{p//u\}_{\theta}$ of p against u with respect to binding variables θ to be the check of $\{p//u\}_{\theta}$

on θ , where the check of a match is the function defined in Section 2. The check is necessary to ensure that reduction does not allow bound variables to become free. For example, $\{x/\bullet\}_{\{x,y\}} = \text{none}$ since $\{x/\bullet\}_{\{x,y\}}$ is not defined on y .

The pure pattern calculus has a *match rule* given by

$$(p \rightarrow_{\theta} s) u \mapsto \{p/u\}_{\theta} s. \quad (3)$$

That is, if matching of the pattern against the argument produces a substitution whose domain is the binding variables then apply this to the body. If the matching fails then return the identity function. Of course, if $\{p/u\}_{\theta}$ is undefined (e.g. because p or u needs to be reduced) then the match rule does not apply.

The *one-step reduction relation* \longrightarrow is defined by applying the match rule to a sub-term. The *reduction relation* \longrightarrow^* is the reflexive-transitive closure of \longrightarrow . A term t is *irreducible* if there is no reduction of the form $t \longrightarrow t'$.

4 Examples

λ -Calculus. There is a simple embedding of the pure λ -calculus into the pure pattern calculus obtained by identifying the λ -abstraction $\lambda x.s$ with $x \rightarrow_x s$ or $x \rightarrow s$. Pattern-matching for these terms is exactly the β -reduction of the λ -calculus. For example, the *fixpoint* term

$$\text{fix} = (x \rightarrow f \rightarrow f (x x f)) (x \rightarrow f \rightarrow f (x x f))$$

can be used to define recursive functions. A term definition of the form $\text{letrec } x = t$ will be interpreted as giving f the value $\text{fix } (x \rightarrow t)$ in the usual way.

Branching Constructs. Let $\text{False} = \bullet$ and $\text{True} = \bullet \bullet$. Define conditionals by

$$\text{if } b \text{ then } s \text{ else } r = (\text{True} \rightarrow x \rightarrow s) b r$$

where $x \notin \text{fv}(s)$. Thus, if $\text{True then } s \text{ else } r$ reduces to $(x \rightarrow s) r$ and then to s while if $\text{False then } s \text{ else } r$ reduces to $(y \rightarrow y) r$ and then to r . More generally, the *extension* $p \rightarrow_{\theta} s \mid r$ extends the case $p \rightarrow_{\theta} s$ with a *default* r by

$$p \rightarrow_{\theta} s \mid r = x \rightarrow (p \rightarrow_{\theta} y \rightarrow s) x (r x)$$

where $x \notin \text{fv}(p \rightarrow_{\theta} s) \cup \text{fv}(r)$ and $y \notin \text{fv}(s)$. When applied to some term u it reduces to $\{p/u\}_{\theta}(y \rightarrow s) (r u)$. Now if $\{p/u\}_{\theta}$ is some substitution σ then this reduces to $\sigma(y \rightarrow s) (r u) = (y \rightarrow \sigma s) (r u)$ and then to σs as desired. Alternatively, if $\{p/u\}_{\theta} = \text{none}$ then the term reduces to $(\text{none } (y \rightarrow s)) (r u) = (z \rightarrow z) (r u)$ and then to $r u$ as desired.

Extensions can be iterated to produce pattern-matching functions out of a sequence of many cases. Make \mid right-associative so that

$$\begin{array}{l} p_1 \rightarrow s_1 \\ \mid p_2 \rightarrow s_2 \\ \vdots \\ \mid p_n \rightarrow s_n \end{array}$$

is $p_1 \rightarrow s_1 \mid (p_2 \rightarrow s_2 \mid (\dots \mid p_n \rightarrow s_n))$. For example, the function `listPoints` in the introduction is defined in this way.

Arithmetic. The natural numbers can be defined as data structures built from constructors `Zero` = \bullet and `Successor` = \bullet . Then recursive functions can be defined using `fix`. This compares favourably with the representation of numbers as the iterators used to define the Church numerals.

Constructors. In practice, it is natural to add a family of constructors c to the calculus, to represent truth values, numbers, lists etc. Each constructor is an atomic data structure that can be applied to an arbitrary number of arguments to produce compound data structures. The match rule $\{c//c\}_\theta = \{\}$ generalises that for \bullet .

Generic Equality. Now let us consider some novel programs. A generic equality is defined by

$$\text{equal} = x \rightarrow (x \rightarrow_{\{\}} \text{True} \mid y \rightarrow \text{False})$$

where the first argument is used as the pattern for matching against the second. For example, `equal` $(\bullet (\bullet \bullet))$ $(\bullet (\bullet \bullet))$ reduces to `True`. This is a simple example of *pattern polymorphism* where the pattern is created dynamically.

The Generic Eliminator. The generic eliminator is given by

$$\text{elim} = x \rightarrow x \ y \rightarrow_y y$$

For example, `elim` `Successor` reduces to `Successor` $y \rightarrow y$. Again, suppose that the list constructors `Nil` and `Cons` are defined to be some data structures built from \bullet . Given `singleton` = $x \rightarrow \text{Cons } x \ \text{Nil}$ then `elim` `singleton` reduces to `Cons` $y \ \text{Nil} \rightarrow y$ by reduction of the pattern.

Generic Updating. Patterns of the form $x \ y$ are used to access data along arbitrary paths through a data structure, i.e. to support *path polymorphism*. Combining the use of pattern and path polymorphism yields the generic update function defined in the introduction. When applied to a constructor c , and a function f and a data structure d it replaces sub-terms of d of the form $c \ t$ by $c \ (f \ t)$. For example, `update` $c \ f \ ((c \ u) \ (c \ v))$ reduces to $(c \ (f \ u)) \ (c \ (f \ v))$. In general, `update` can be applied to cases. For example, `update` `singleton` f reduces to

$$\begin{array}{ll} \text{Cons } y \ \text{Nil} & \rightarrow \text{Cons } (f \ y) \ \text{Nil} \\ \mid z \ y & \rightarrow (\text{update } \text{singleton} \ f \ z) \ (\text{update } \text{singleton} \ f \ y) \\ \mid y & \rightarrow y \end{array}$$

Also, updating can be iterated to give finer control. For example, given constructors `Salary`, `Employee` and `Department` and a function f then the program

$$\text{update } \text{Department} \ (\text{update } \text{Employee} \ (\text{update } \text{Salary} \ f))$$

updates departmental employee salaries. Note that it is not necessary to know how employees are represented within departments for this to work, so that a

new level of abstraction arises, similar to that which XML is intended to support. The full range of XML paths can be handled by defining an appropriate abstract data type, similar to that of *signposts* given in [HJS05a, HJS05b].

Wild-Cards. It is interesting to add a new constant denoted $?$ to the pure pattern calculus, the *wild-card*. It has no free variables and is unaffected by substitution. It is a data structure, is compatible with anything, and has the matching rule

$$\{?/u\}_\theta = \{\}.$$

That is, it behaves like a fresh binding variable in a pattern but like a constructor in a body. For example, the first and second projections from a pair can be encoded as $\text{elim}(\text{Pair } ?)$ and $\text{elim}(x \rightarrow \text{Pair } x ?)$.

The following example uses recursion in the pattern. Define the function for the extracting list entries by

```

letrec entrypattern =
  Succ  $n \rightarrow x \rightarrow \text{Cons } ? (\text{entrypattern } n \ x)$ 
  | Zero   $\rightarrow x \rightarrow \text{Cons } x \ ?$ 

entry =  $n \rightarrow \text{elim}(\text{entrypattern } n)$ 

```

For example, $\text{entry}(\text{Succ}(\text{Succ } \text{Zero}))$ reduces to $\text{Cons } ? (\text{Cons } ? (\text{Cons } x \ ?)) \rightarrow x$ which recovers the second entry from a list. Note the standard approach, in which each occurrence of the wild-card represents a distinct binding variable, cannot support such recursion.

5 Progress

Lemma 2. *An irreducible term t is a $\text{fv}(t)$ -matchable form.*

Proof. The proof is by a straightforward induction on the structure of t . If t is of the form $(p \rightarrow_\theta s) \ u$ then it is either reducible or a $\text{fv}(t)$ -data structure.

Theorem 1. *Pattern-matching cannot get stuck. That is, a closed term of the form $(p \rightarrow_\theta s) \ u$ is always reducible.*

Proof. Assume that p and u are irreducible. Then p is a θ -matchable form and u is a matchable form by Lemma 2. Hence $\{p/u\}_\theta$ is defined and the whole term can be reduced.

Corollary 1. *The data structures are those terms headed by the constructor.*

6 Confluence

Confluence of reduction is established using the parallel reduction technique due to Tait and Martin-Löf [Bar84] which can be summarised in four steps: define a parallel reduction relation denoted \gg ; prove that \gg^* and \longrightarrow^* are the same relation (Lemma 3); show that \gg has the diamond property (Theorem 2); and use this to prove confluence. The *parallel reduction relation* is given in Figure 1.

$\frac{}{t \gg t} \qquad \frac{r \gg r' \quad u \gg u'}{r \ u \gg r' \ u'} \qquad \frac{p \gg p' \quad s \gg s'}{p \rightarrow_{\theta} s \gg p' \rightarrow_{\theta} s'} \qquad \frac{p \gg p' \quad s \gg s' \quad u \gg u'}{(p \rightarrow_{\theta} s) \ u \gg \{p'/u'\}_{\theta} s'}$

Fig. 1. Parallel reduction

Lemma 3. *Every one-step reduction is a parallel reduction. Also, every parallel reduction is a reduction. Hence the reflexive-transitive closure \gg^* of \gg is the reduction relation \longrightarrow^* .*

Proof. The proofs are by straightforward induction on the definitions.

The *parallel reduction relation* \gg between matches is defined as follows. Given two substitutions σ and σ' then $\sigma \gg \sigma'$ if $\text{dom}(\sigma) = \text{dom}(\sigma')$ and $\sigma x \gg \sigma' x$ for every $x \in \text{dom}(\sigma)$. Also $\text{none} \gg \text{none}$. Substitutions and none are not related.

Lemma 4. *If t is a term and m is a match then $\text{fv}(mt) \subseteq \text{fv}(m) \cup (\text{fv}(t) \setminus \text{dom}(m))$.*

Proof. If m is none then the result is immediate so assume that m is a substitution σ . The proof is by induction on the structure of t . If t is $p \rightarrow_{\theta} s$ where $\theta \cap \text{var}(\sigma) = \{\}$ then

$$\begin{aligned} \text{fv}(\sigma p \rightarrow_{\theta} \sigma s) &= (\text{fv}(\sigma p) \cup \text{fv}(\sigma s)) \setminus \theta \\ &\subseteq (\text{fv}(\sigma) \cup ((\text{fv}(p) \cup \text{fv}(s)) \setminus \text{dom}(\sigma))) \setminus \theta \quad (\text{by induction}) \\ &= \text{fv}(\sigma) \cup (\text{fv}(t) \setminus \text{dom}(\sigma)). \end{aligned}$$

The other cases are straightforward.

Lemma 5. *If $m = \{p/u\}_{\theta}$ for some terms p and u and set of variables θ then $\text{fv}(m) \subseteq \text{fv}(u)$.*

Proof. If $m = \text{none}$ then there is nothing to prove. Otherwise the proof is by a straightforward induction on the structure of p .

Lemma 6. *If $t \gg t'$ is a parallel reduction then $\text{fv}(t') \subseteq \text{fv}(t)$. Hence, if $m \gg m'$ is a parallel match reduction then $\text{var}(m') \subseteq \text{var}(m)$.*

Proof. By Lemma 3 it suffices to prove the result for the reduction rule (3). Then

$$\begin{aligned} \text{fv}(t') &= \text{fv}(\{p/u\}_{\theta} s) \\ &\subseteq \text{fv}(\{p/u\}_{\theta}) \cup (\text{fv}(s) \setminus \text{dom}(\{p/u\}_{\theta})) \quad (\text{Lemma 4}) \\ &\subseteq \text{fv}(\{p/u\}_{\theta}) \cup (\text{fv}(s) \setminus \theta) \\ &\subseteq \text{fv}(u) \cup (\text{fv}(p \rightarrow_{\theta} s)) \quad (\text{Lemma 5}) \\ &= \text{fv}(t). \end{aligned}$$

Lemma 7. *Let m be a match and let θ be a set of variables such that $\text{var}(m) \cap \theta = \{\}$. If p and u are terms such that $\{p/u\}_{\theta}$ is defined then so is $\{m \ p//m \ u\}_{\theta}$ and $\{m \ p//m \ u\}_{\theta} \circ m = m \circ \{p/u\}_{\theta}$. Hence*

$$\{m \ p//m \ u\}_{\theta} \circ m = m \circ \{p/u\}_{\theta}.$$

Proof. The second statement follows directly from the first. If m is **none** then both sides are **none**. So without loss of generality, assume that m is a substitution σ . The proof is by induction on the structure of p . If p is a variable $x \in \theta$ then both sides map x to σu and behave as σ on all other variables. If p is the constructor and u is too then both sides are m . If p and u are compatible applications then apply induction twice.

If $\{p//u\}_\theta = \text{none}$ then $\{\sigma p//\sigma u\}_\theta = \text{none}$ (since $\text{dom}(\sigma) \cap \theta = \{\}$) and so both sides of the match equation are **none**.

Lemma 8. *If $p \gg p'$ and $u \gg u'$ are parallel reductions on terms and $\{p/u\}_\theta$ is defined then so is $\{p'/u'\}_\theta$ and $\{p/u\}_\theta \gg \{p'/u'\}_\theta$.*

Proof. The proof is by induction on the structure of p . If p is a variable then p' is the same variable and it must be in θ so that the result follows directly. If p is the constructor then u is the constructor if and only if u' is. If p is a case then both matches will fail. Otherwise p must be a θ -matchable form $p_1 p_2$ and u must be a matchable form. If u is not an application then it must be the constructor or a case: either way, both matchings will fail. Alternatively, if $u = u_1 u_2$ then Corollary 1 implies that u_1 is also a data structure and thus $u' = u'_1 u'_2$ where $u_1 \gg u'_1$ and $u_2 \gg u'_2$. We then have two possibilities for p_1 . If p_1 is a θ -data structure then $p' = p'_1 p'_2$ where $p_1 \gg p'_1$ and $p_2 \gg p'_2$. Hence induction applies. Otherwise, p_1 is a case so that p is irreducible and both matches against it will fail.

Lemma 9. *If $m \gg m'$ and $t \gg t'$ are parallel reductions of matches and terms respectively then $m t \gg m' t'$.*

Proof. If m is **none** then m' is **none** and so the result is immediate. So assume that m and m' are substitutions σ and σ' respectively. The proof is by induction on the derivation of $t \gg t'$. The only non-trivial case is when $t = (p \rightarrow_\theta s) u \gg \{p'/u'\}_\theta s'$ where $p \gg p'$ and $u \gg u'$ and $s \gg s'$. Without loss of generality, assume $\text{var}(\sigma) \cap \theta = \{\}$. By Lemma 6 and by the assumptions we have $\text{var}(\sigma') \cap \theta \subseteq \text{var}(\sigma) \cap \theta = \{\}$. Thus, $\sigma'(\{p'/u'\}_\theta s')$ is equal to $\{\sigma' p'/\sigma' u'\}_\theta(\sigma' s')$ by Lemma 7 and so $\sigma((p \rightarrow_\theta s)u) = (\sigma p \rightarrow_\theta \sigma s) (\sigma u) \gg \{\sigma' p'/\sigma' u'\}_\theta(\sigma' s') = \sigma'(\{p'/u'\}_\theta s')$.

Theorem 2. *The relation \gg has the diamond property. That is, $t \gg t_1$ and $t \gg t_2$ then there is t_3 such that $t_1 \gg t_3$ and $t_2 \gg t_3$.*

Proof. The proof is by induction on the definition of parallel reduction. Suppose

$$(p_2 \rightarrow_\theta s_2) u_2 \ll (p \rightarrow_\theta s) u \gg \{p_1/u_1\}_\theta s_1$$

where $p \gg p_1$ and $p \gg p_2$ and $s \gg s_1$ and $s \gg s_2$ and $u \gg u_1$ and $u \gg u_2$. By induction, there are terms p_3, s_3 and u_3 such that $p_1 \gg p_3$ and $p_2 \gg p_3$ and $s_1 \gg s_3$ and $s_2 \gg s_3$ and $u_1 \gg u_3$ and $u_2 \gg u_3$. Now $\{p_1/u_1\}_\theta \gg \{p_3/u_3\}_\theta$ by Lemma 8 and so $\{p_1/u_1\}_\theta s_1 \gg \{p_3/u_3\}_\theta s_3$ by Lemma 9. Hence, the diamond is completed by $\{p_3/u_3\}_\theta s_3$.

Again, suppose $(p \rightarrow_\theta s) u \gg \{p_2/u_2\}_\theta s_2$ and $(p \rightarrow_\theta s) u \gg \{p_1/u_1\}_\theta s_1$ where $p \gg p_1$ and $p \gg p_2$ and $s \gg s_1$ and $s \gg s_2$ and $u \gg u_1$ and $u \gg u_2$. By induction there are terms p_3, s_3 and u_3 such that $p_1 \gg p_3$ and $p_2 \gg p_3$ and $s_1 \gg s_3$ and $s_2 \gg s_3$ and $u_1 \gg u_3$ and $u_2 \gg u_3$. Now $\{p_1/u_1\}_\theta$ and $\{p_2/u_2\}_\theta$ both parallel reduce to $\{p_3/u_3\}_\theta$ by Lemma 8 and so Lemma 9 implies the diamond is completed by $\{p_3/u_3\}_\theta s_3$. The other cases are straightforward.

Corollary 2 (Confluence). *The reduction relation is confluent.*

Proof. Theorem 2 shows that \gg has the diamond property and so the reflexive-transitive closure of \gg is confluent. Now apply Lemma 3.

7 Conclusion and Further Work

Pattern-matching provides a natural mechanism for computing with data structures; its expressive power is determined by the nature of the patterns that are allowed. The pure pattern calculus maximises this expressive power by allowing any term to be a pattern. The resulting language supports patterns that are able to match with arbitrary compound data structures (path polymorphism), and patterns that can be assembled dynamically (using free variables to represent patterns) and simplified into a matchable form (pattern polymorphism). Such patterns will prove useful when manipulating remote data whose structure is only partially known, as illustrated by the example of updating.

There are a number of open questions concerning the pure pattern calculus itself, and its connections to rewriting, logic, type theory and category theory.

The matching process may extend to consider matching of cases as well as of data structures, provided the binding variables of cases are treated appropriately. We have not pursued this here as the complexity of the development was not justified by any new forms of polymorphism. However, it may prove useful in program analysis and transformation.

It is not yet clear what extensional equality should be for the pattern calculus, as earlier work on extensionally for pattern-matching [Kes97] does not take full account of data structures. For example, the η -equality rule $f = \lambda x. f x$ does *not* apply in our setting since a data structure is not a case.

Another issue concerns higher-order rewriting within a formalism with patterns [FK03]. It seems natural to extend such languages to capture the rich dynamics of the patterns presented here.

In the spirit of [KPT96] it would be interesting to explore a Curry-Howard interpretation for the pure pattern calculus in order to recognise, or develop, the corresponding logic. For example, matching against arbitrary compounds appears to model structural induction [Bur69] in a uniform way.

The calculus presented here uses a meta-level (or implicit) pattern-matching operation. One could also consider explicit pattern-matching, where the match equations become themselves rewriting rules which can then be interleaved with other reductions [CK99, For02, Kah03, Jay04c].

It is straightforward to provide simple types and indeed to support parametric polymorphism. Of more interest will be the addition of type specialisations [Jay04c] necessary to type the more complex examples. The calculus will then provide a clean foundation for a typed account of XML paths, as described in [HJS05a, HJS05b] and a platform upon which to model object-orientation, along the lines proposed in [Jay04b].

The denotational semantics of the pattern calculus also awaits exploration. It is not clear how to represent a case in a domain-theoretic setting. As a lambda-abstraction is an arrow in a category then perhaps a case is a *span* in a category or, rather, the internalisation of a span.

In conclusion, the pure pattern calculus provides a compact setting in which to handle both functions and data structures in a uniform manner, and so support new forms of polymorphism.

Acknowledgements. We would like to thank the anonymous referees and Eugenio Moggi for their constructive criticism.

References

- [Bar84] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. Revised Edition.
- [BCKL03] Gilles Barthe, Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Pure Pattern Type Systems. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 250–261. ACM, 2003.
- [BdMH96] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.
- [Bur69] Rod M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 1969.
- [CK98] Horatiu Cirstea and Claude Kirchner. ρ -calculus, the rewriting calculus. In *5th International Workshop on Constraints in Computational Logics (CCL)*, 1998.
- [CK99] Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. In Giuseppe Longo, editor, *14th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 98–108. IEEE Computer Society Press, 1999.
- [FK03] Julien Forest and Delia Kesner. Expression reduction systems with patterns. In Robert Nieuwenhuis, editor, *14th International Conference on Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 107–122. Springer-Verlag, 2003.
- [For02] Julien Forest. A weak calculus with explicit operators for pattern matching and substitution. In Sophie Tison, editor, *13th International Conference on Rewriting Techniques and Applications (RTA)*, volume 2378 of *Lecture Notes in Computer Science*, pages 174–191. Springer-Verlag, 2002.
- [GJ03] Jeremy Gibbons and Johan Jeuring, editors. *Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Programming July 11–12, 2002, Dagstuhl, Germany*. Kluwer Academic Publishers, 2003.

- [Has] The Haskell language. <http://www.haskell.org/>.
- [HJS05a] Freeman Yufei Huang, C. Barry Jay, and David B. Skillicorn. Dealing with complex patterns in XML processing. Technical Report 2005-497, Queen's University School of Computing, 2005.
- [HJS05b] Freeman Yufei Huang, C. Barry Jay, and David B. Skillicorn. Programming with heterogeneous structures: Manipulating XML data using `bondi`. Technical Report 2005-494, Queen's University School of Computing, 2005. To appear in ACSW'06.
- [Jan00] Patrick Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000.
- [Jay04a] C. Barry Jay. Higher-order patterns. Available as [www-staff.it.uts.edu.au/~cbj/Publications/higher_order_patterns.pdf](http://www.staff.it.uts.edu.au/~cbj/Publications/higher_order_patterns.pdf), 2004.
- [Jay04b] C. Barry Jay. Methods as pattern-matching functions. In Sophia Drossopoulou, editor, *The 11th International Workshop on Foundations of Object-Oriented Languages*, 2004. Proc. available as <http://www.doc.ic.ac.uk/~scd/F00.pdf>.
- [Jay04c] C. Barry Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):911–937, November 2004.
- [JBM98] C. Barry Jay, Gianna Bellè, and Eugenio Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.
- [JC94] C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming, Edinburgh, U.K., April 1994, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316. Springer Verlag, 1994.
- [Kah03] Wolfram Kahl. Basic pattern matching calculi: Syntax, reduction, confluence, and normalisation. Technical Report 16, Software Quality Research Laboratory, McMaster Univ., 2003.
- [Kes97] Delia Kesner. Reasoning about redundant patterns. *Journal of Functional and Logic Programming*, 1997(4), June 1997.
- [Klo80] Jan-Willem Klop. *Combinatory Reduction Systems*. PhD thesis, Mathematical Centre Tracts 127, CWI, Amsterdam, 1980.
- [KPT96] Delia Kesner, Laurence Puel, and Val Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, 1996.
- [Oca] The Objective Caml language. <http://caml.inria.fr/>.
- [SML] STANDARD ML OF NEW JERSEY. <http://cm.bell-labs.com/cm/cs/what/smlnj/>.

A Verification Methodology for Model Fields

K. Rustan M. Leino¹ and Peter Müller²

¹ Microsoft Research
leino@microsoft.com

² ETH Zürich
peter.mueller@inf.ethz.ch

Abstract. Model fields are specification-only fields that encode abstractions of the concrete state of a data structure. They allow specifications to describe the behavior of object-oriented programs without exposing implementation details.

This paper presents a sound verification methodology for model fields that handles object-oriented features, supports data abstraction, and can be applied to a variety of realistic programs. The key innovation of the methodology is a novel encoding of model fields, where updates of the concrete state do not automatically change the values of model fields. Model fields are updated only by a special pack statement. The methodology guarantees that the specified relation between a model field and the concrete state of an object holds whenever the object is valid, that is, is known to satisfy its invariant.

The methodology also improves on previous work in three significant ways: First, the formalization of model fields prevents unsoundness, even if an interface specification is inconsistent. Second, the methodology fully supports inheritance. Third, the methodology enables modular reasoning about frame properties without using explicit dependencies, which are not handled well by automatic theorem provers.

1 Introduction

The development of object-oriented programs makes use of mutable objects, aliasing, subtyping, and modularity. We are interested in verifying such programs. To do that, we need specifications with data abstraction and a systematic way (a methodology) of reasoning. Existing methodologies either do not address these characteristics of object-oriented programming or do not support data abstraction in a satisfactory way. In this paper, we present a methodology that addresses these problems and that can be applied to a wide variety of realistic programs.

Specifications that are visible to client code must be expressed in an implementation-independent way to support information hiding. This can be achieved by using data abstraction [11], that is, by mapping the concrete state of a data structure to an abstract value. A standard example is to map the state of a singly-linked list to a mathematical sequence. The behavior of the list class can be expressed in terms of the abstract value of list objects, that is, in terms of the sequence. A convenient way to support data abstraction in specification languages for object-oriented programs is by *model fields* [5, 14, 15, 19]. In contrast to ordinary (*concrete*) fields, a program cannot directly assign to model fields. Instead, model fields are specification-only fields whose values are determined by mappings from an object's concrete state.

Class *Rectangle* in Fig. 1 illustrates how model fields are used in specifications. A *Rectangle* object stores the coordinates of two opposite corners, as expressed by the invariant. The model field *width* is used to refer to the width of the rectangle in specifications. The value of *width* is the difference between the x-coordinates of the corners, *x2* and *x1*. This relation between the model field *width* and the concrete fields *x1* and *x2*—the so-called *constraint* for *width*—is expressed by the **constrained_by** part of the model field declaration. The declaration of the model field *height* is analogous.

Method *ScaleH* scales the rectangle horizontally by a given percentage. The *ensures* clause uses the model field *width* to express the functionality of *ScaleH* without referring to the concrete implementation. The *modifies* clause allows method *ScaleH* to change the values of the fields *width* and *x2*. The second *requires* clause as well as the *unpack* and *pack* statements are required by the methodology for object invariants we build on and will be explained in Sec. 4.

```

class Rectangle {
  int x1, y1, x2, y2 ;    // lower left and upper right corner
  invariant x1 ≤ x2 ∧ y1 ≤ y2 ;
  model int width constrained_by width = x2 - x1 ;
  model int height constrained_by height = y2 - y1 ;
  Rectangle() {
    x1 := 0 ; y1 := 0 ;
    x2 := 1 ; y2 := 1 ;
    pack this as Rectangle ;
  }
  void ScaleH(int factor)
    requires 0 ≤ factor ;
    requires inv = Rectangle ∧ ¬committed ;
    modifies width, x2 ;
    ensures width = old(width) * factor / 100 ;
    {
      unpack this from Rectangle ;
      x2 := (x2 - x1) * factor / 100 + x1 ;
      pack this as Rectangle ;
    }
}

```

Fig. 1. A specification with model fields

The basic concept of data abstraction is well-understood. However, existing verification techniques for model fields (including our own previous work) suffer from soundness, modularity, expressiveness, or practicality problems. Our contribution in this paper is a verification methodology that solves these problems. The key innovation is to treat model fields as if they were stored in the heap and updated automatically in a systematic way. This treatment reduces reasoning about model fields to simpler concepts.

We illustrate the problems any verification methodology for model fields has to address in the next section and explain our approach to their solution in Sec. 3. Our

approach is based on the Boogie methodology for object invariants [1, 17], which we summarize in Sec. 4. We present the details of our methodology for model fields in Sec. 5. The rest of the paper discusses related work and offers some conclusions.

2 Problems

A verification methodology for model fields has to address two major issues: (a) the meaning of model fields and their constraints, and (b) the meaning of frame properties in the presence of model fields. We discuss these issues in the following.

2.1 Meaning of Model Fields

In existing methodologies [4, 15, 18, 19, 20], the meaning of a model field is defined by an *abstraction function* that maps a receiver object and a heap to the model field's value. This abstraction function is specified by a programmer-provided constraint. The problem with this meaning is that if a programmer specifies inconsistent constraints, then the abstraction functions are not well-defined, which lends itself to unsound reasoning. Inconsistent constraints occur in two situations.

First, constraints can be unsatisfiable. For instance, consider the abstraction function abs_{len} of an integer model field len of a class *List*. If the model field len is constrained by $len = len + 1$, then abs_{len} has to satisfy the unsatisfiable $abs_{len}(l, H) = abs_{len}(l, H) + 1$ for any *List* object l and heap H . In practice, such ill-formed specifications are far less obvious than this example, because they typically involve strengthening of inherited constraints or cyclic dependencies among several model fields.

Second, abstraction functions of model fields are typically well-defined only for objects that satisfy their invariants. Applying an abstraction function to an object whose invariant is temporarily violated can lead to inconsistencies. For instance, consider a linked list implementation, where the invariant requires the list to be acyclic. In such an implementation, the model field len for the length of the list could be constrained by a conditional expression such as $len = (next = \text{null}) ? 1 : next.len + 1$, where $next$ is the field that stores the next node of the list. This, too, is inconsistent if applied to a cyclic list. For instance, if $l = l.next$ for a list l , then abs_{len} again has to satisfy the unsatisfiable $abs_{len}(l, H) = abs_{len}(l, H) + 1$.

Both kinds of inconsistent constraints can be avoided in carefully written specifications. However, specifications found in practice contain flaws. A verification methodology has to ensure that these flaws are detected during verification and do not lead to unsound reasoning.

2.2 Meaning of Frame Properties

The *frame properties* of a method limit the effects the method may have on the program state. This is crucial when reasoning about calls. Frame properties are typically specified using *modifies* clauses. Roughly speaking, a *modifies* clause lists the concrete and model locations a method is allowed to modify.

Any update of a field $x.f$ potentially affects each model field that depends on $x.f$. A model field $o.m$ *depends* on a field $x.f$ if the value of $x.f$ constrains the value

of $o.m$. For instance, for a *Rectangle* object o , $o.width$ depends on $o.x1$ and $o.x2$ because these fields are mentioned in the constraint for *width*.

When the meaning of a model field is given by an abstraction function, any modification of the heap, for instance, by an update of a field $x.f$, will have an *instant effect* on all dependent model fields. That is, the value of these model fields is changed simultaneously with the update of $x.f$.

Instant effects lead to a modularity problem, which is illustrated by class *Legend* in Fig. 2. *Legend* objects display some text within a bounding box. The box is represented by a *Rectangle* object. The **rep** modifier in the declaration of the field *box* is used to express ownership and will be explained in Sec. 4. The model field *maxChars* yields the maximum number of characters that fit into one line in the box at a given font size.

The value of *maxChars* depends on the width of the *Rectangle* object *box*. Therefore, if method *ScaleH* is executed on a *Rectangle* object r , it potentially modifies *maxChars* for any *Legend* object that uses r . However, we cannot require *ScaleH* to declare this potential modification in its *modifies* clause, since the implementor of *ScaleH* need not be aware of class *Legend* (in fact, *Legend* might have been implemented long after *Rectangle*).

```

class Legend {
  rep Rectangle box ;
  int fontSize ;
  invariant box ≠ null ∧ fontSize > 0 ;
  model int maxChars constrained_by maxChars = box.width/fontSize ;
  void Reset()
    requires inv = Legend ∧ ¬committed ;
    modifies maxChars ;
  {
    unpack this from Legend ;
    box.ScaleH(0) ;
    pack this as Legend ;
  }
  // constructors and other methods omitted.
}

```

Fig. 2. A client of class *Rectangle*

An analogous modularity problem occurs when model field constraints refer to inherited fields. For instance, if a subclass *MyRectangle* of class *Rectangle* declares a model field *area* that depends on the inherited field *x2*, method *ScaleH* potentially modifies *area* without listing the field in its *modifies* clause.

A useful verification methodology for model fields must address the modularity problem of frame properties for aggregate objects (such as *Legend* objects) and subclasses (such as *MyRectangle*).

3 Approach

In this section, we explain the general ideas that allow us to solve the problems described in the previous section. To focus on the essentials, we ignore subtyping in this overview, but we will include it in Sec. 5 when we explain our methodology in detail.

Our methodology for model fields builds on the Boogie methodology for object invariants [1, 17]. In the Boogie methodology, an object is either in a *valid* or a *mutable* state. Only when in a valid state, an object is guaranteed to satisfy its invariant, and only fields of objects in a mutable state can be assigned. The transition from valid to mutable and back is performed by two special statements, **unpack** and **pack**.

Principles. Our methodology is based on the following three principles:

1. *Validity principle:* The declared constraint for a model field m constrains the value of $o.m$ only if the object o is valid, that is, if o 's invariant is known to hold.
2. *Decoupling principle:* The change of the value of a model field is decoupled from the updates of the fields it depends on. Instead of applying an abstraction function to obtain the value of a model field $o.m$, a stored value is used. The stored value of $o.m$ is not updated instantly when a dependee field is modified, but only at the point when o is being packed.
3. *Mutable dependent principle:* If a model field $o.m$ depends on a field $x.f$, then the dependent object o must be mutable whenever x is mutable.

The validity principle defines the meaning of model fields and avoids inconsistencies due to temporarily broken invariants. Inconsistencies due to unsatisfiable constraints (e.g., $len = len + 1$) are avoided by assertions, as we explain in Sec. 5.2.

The decoupling principle solves the modularity problems of frame properties. Consider a method M that updates a field $x.f$. Because of decoupling, this update does not have an instant effect on a dependent model field $o.m$. That is, $o.m$ remains unchanged and, therefore, need not be mentioned in M 's modifies clause (as long as M does not pack o).

The mutable dependent principle and the validity principle are prerequisites for decoupling to be sound. Consider a model field $o.m$ that depends on a field $x.f$. Updating $x.f$ potentially causes $o.m$ not to satisfy its constraint any more. However, the Boogie methodology requires that x be mutable when $x.f$ is updated. Therefore, the mutable dependent principle implies that o is also mutable, and the validity principle allows $o.m$ not to satisfy its constraint.

There are several ways to enforce the mutable dependent principle. The one we use in this paper is to organize objects in an ownership hierarchy [17]. A model field $o.m$ is allowed to depend on fields of an object x only if x is (transitively) owned by o . The Boogie methodology guarantees that the (transitive) owner objects of a mutable object are themselves mutable.

Example. To illustrate how these principles work, we revisit the *Legend* example (Fig. 2). Let l be a *Legend* object and let r be the *Rectangle* object stored in $l.box$. The modifier **rep** in the declaration of box indicates that l owns r . Thus, the Boogie methodology guarantees that l is mutable whenever r is mutable. Since the model field

$l.maxChars$ depends on $r.width$, this ownership relation is required by the mutable dependent principle.

Consider the execution of method *ScaleH* invoked from *Legend*'s method *Reset*. The first statement of *ScaleH* unpacks the receiver object (that is, r) to permit updates of its fields. By the decoupling principle, the subsequent update of $x2$ does not change the value of the model field $r.width$, even though $width$ depends on $x2$. In the state after the update, the value of $r.width$ in general does not satisfy the specified constraint because the concrete state has changed, but the value of the model field has not (yet) been adapted. This discrepancy is permitted by the validity principle since r is mutable.

The value of $r.width$ is brought up to date when r is packed. Again, by the decoupling principle, this update does not instantly affect the value of $l.maxChars$. Consequently, this model field does not have to be mentioned in *ScaleH*'s modifies clause, which shows the modularity of the approach.

Updating $r.width$ potentially causes $l.maxChars$ not to satisfy its constraint. However, since l is mutable, this discrepancy is permitted by the validity principle. It will be resolved when l is packed in method *Reset*.

4 Background: The Boogie Methodology for Object Invariants

In this section, we summarize those parts of the Boogie methodology for object invariants [1] that are needed in the rest of this paper. The motivation for the design and the technical details are presented in our earlier paper [17].

Explicit Representation of When Invariants Have to Hold. To handle temporary violations of object invariants and reentrant method calls, the Boogie methodology represents explicitly in every object's state whether the object invariant is required to hold or allowed to be violated. For this purpose, it introduces for every object a concrete field *inv* that ranges over class names. If $o.inv <: T$ for a T object o (where $<:$ denotes the subtype relation), then o 's invariants declared in class T and its superclasses must hold and we say o is *valid for* T . If o is not valid for T then the invariant of o declared in T are allowed to be temporarily violated and we say o is *mutable for* T .

The *inv* field can be used in method specifications, but cannot be assigned directly by the program. Instead, the Boogie methodology provides two special statements: **unpack** o **from** T and **pack** o **as** T change $o.inv$ from T to T 's direct superclass and back, respectively. Before setting *inv* to T , the pack statement checks that the object invariant declared in class T holds for o .

Since the update of a field $o.f$ potentially breaks the invariant of o , $o.f$ is allowed to be assigned only at times when o is mutable for the class F that declares f . To enforce this policy, each update of $o.f$ is guarded by an assertion $F \not\leq o.inv$. This assertion is crucial for the soundness of our methodology, see Sec. 5.4.

Ownership. The Boogie methodology handles aggregate objects by guaranteeing that the validity of an object implies the validity of its component objects. Providing this guarantee requires some form of *aliasing control*, a discipline on the use of object references. The Boogie methodology uses the notion of *ownership* for aliasing control,

associating with every object a unique owner object. That is, an aggregate object is the owner of its component objects. Objects outside the aggregate are allowed to reference component objects, but these references are only of limited use.

To encode ownership, the Boogie methodology introduces two additional concrete fields for every object: a field *owner* that ranges over pairs $\langle o, T \rangle$, where o is the owner object and T is a superclass of the dynamic type of o at which the ownership is established, and a boolean field *committed*. Like *inv*, these fields can be used in method specifications, but cannot directly be assigned by the program. The owner of an object is set when the object is created. Because it would be a distraction in this paper, we omit a program statement for changing the *owner* field (but see [17]).

Let p be an object that is owned by $\langle o, T \rangle$. The fact that p is committed (that is, $p.\text{committed} = \text{true}$) expresses that p is valid for its dynamic type, and o is valid for T . The *committed* field is used to implement a protocol that enforces that an owner object is unpacked before the owned object is unpacked. Packing is done in the reverse order. More precisely, this protocol ensures that the owner object o is mutable for the owner type T whenever p is mutable.

In connection with the fact that field updates are allowed only for mutable objects, this protocol guarantees that the following two program invariants hold in each reachable execution state of a program: If an object o is valid for a class T , then the object invariants declared in T hold for o and all objects owned by $\langle o, T \rangle$ are committed (see J1 below). Committed objects are valid for their dynamic type (see J2 below). (Here and throughout the paper, quantifications over object references range over non-null references to allocated objects.)

$$\text{J1: } (\forall o, T \bullet o.\text{inv} <: T \Rightarrow \text{Inv}_T(o) \wedge (\forall \text{object } p \bullet p.\text{owner} = \langle o, T \rangle \Rightarrow p.\text{committed}))$$

$$\text{J2: } (\forall o \bullet o.\text{committed} \Rightarrow o.\text{inv} = \text{typeof}(o))$$

The protocol is implemented by the *unpack* and *pack* statements. The act of packing an object o for a class T also commits the objects owned by $\langle o, T \rangle$ by setting their *committed* fields to *true*. This operation requires these owned objects to be previously uncommitted and valid for their dynamic types. Unpacking an object o from a class T requires o to be uncommitted and sets the *committed* field of the objects owned by $\langle o, T \rangle$ to *false*. We formalize these statements by the pseudo code shown in Fig. 3. $\text{Inv}_T(o)$ denotes the expression that says that o satisfies the object invariant declared in class T , $\text{typeof}(o)$ is the dynamic type of object o , and $\text{Super}(T)$ denotes the direct superclass of T .

To simplify the specification of aggregate objects, we allow the use of the modifier *rep*. Applied to the declaration of a field f in class T , it gives rise to the *implicit object invariant* $f \neq \text{null} \Rightarrow f.\text{owner} = \langle \text{this}, T \rangle$. This keyword also allows us to prescribe syntactic checking of admissible model fields, as we shall see in Sec. 5.1.

Static Verification. The proof rules of the Boogie methodology are formulated in terms of assertions, which cause the program execution to abort if evaluated to *false*. Assertions appear in the following places: (a) before method calls for the *requires* clauses

```

unpack  $o$  from  $T \equiv$ 
  assert  $o \neq \text{null} \wedge o.\text{inv} = T \wedge \neg o.\text{committed}$  ;
   $o.\text{inv} := \text{Super}(T)$  ;
  #foreach object  $p$  such that  $p.\text{owner} = \langle o, T \rangle \{ p.\text{committed} := \text{false} \}$ 
pack  $o$  as  $T \equiv$ 
  assert  $o \neq \text{null} \wedge o.\text{inv} = \text{Super}(T)$  ;
  assert  $(\forall \text{object } p \bullet p.\text{owner} = \langle o, T \rangle \Rightarrow p.\text{inv} = \text{typeof}(p) \wedge \neg p.\text{committed})$  ;
  assert  $\text{Inv}_T(o)$  ;
  #foreach object  $p$  such that  $p.\text{owner} = \langle o, T \rangle \{ p.\text{committed} := \text{true} \}$ 
   $o.\text{inv} := T$ 

```

Fig. 3. Pseudo code for **unpack** and **pack**

of the called method, (b) at the end of a method body for the method's ensures and modifies clauses, (c) in the pseudo code for **unpack** and **pack**, and (d) before field updates. Proving the correctness of a program amounts to statically verifying that the program does not abort due to a violated assertion. To do that, each assertion is turned into a proof obligation. One can then use an appropriate program logic to show that the assertions hold. All of the proof obligations can be generated and shown modularly. That is, a class C can be verified based on the specifications of the classes used by C , but without knowing the complete program in which C will be used.

For the proof, one may assume that the program invariants J1 and J2 hold. This assumption is justified by a soundness theorem for the Boogie methodology presented in earlier work [17, 21].

5 Model Fields

In this section, we present the technical details of our methodology. We define which model field declarations are admissible, present a novel encoding of model fields that builds on the validity principle and enables decoupling, discuss how frame properties are specified and proved in our methodology, and prove soundness. In the following, we assume a programming language similar to the sequential subset of Java.

5.1 Declaration of Model Fields

The declaration of a model field m has the following form:

model T m **constrained_by** E ;

where T is the type of the model field. The expression E specifies a constraint for **this**. m . It is a boolean expression of the programming language, which is also allowed to mention model fields. For simplicity, we disallow method calls in model field constraints, but an extension is possible.

A model field constraint may specify a unique value for the model field, as for instance shown in Fig. 1. It is also allowed to underspecify the value of a model field, which is useful to express abstraction relations and to constrain model fields in abstract

classes and interfaces. For instance, an abstract superclass *Shape* of *Rectangle* might constrain *width* by $0 \leq \text{width}$.

A subclass can strengthen the constraint for an inherited model field by giving a declaration of the above form that repeats the name of the model field and supplies a further constraint. The *effective constraint* of a model field *m* in type *T* is the conjunction of the constraints for *m* in *T* and *T*'s supertypes.

The mutable dependent principle (see Sec. 3) limits what fields can be mentioned in the constraint for a model field. The admissible model fields are summarized by the following definition.

Definition 1. *A model field m declared in type T is admissible if the constraint given in m 's declaration typechecks according to the rules of the programming language and if each of the field access expressions in the constraint has one of the following forms:*

1. **this.m**
2. **this.f**, where *f* is a concrete field
3. **this.p.f**, where *p* is a concrete rep field and *f* is a model or concrete field

*The fields f and p must not be one of the predefined fields *inv* and *committed* (but we allow f to be *owner*).*

Field accesses of Form 2 occur when the constraint for a model field refers to concrete fields of the same object, for instance, in the constraint for *width* (Fig. 1). The standard type rules require *f* to be declared in *T* or a superclass of *T*. That is, the constraint is allowed to refer to inherited fields. The requirement that *f* be concrete is not strictly necessary, but simplifies the formalization; dependencies between different model fields of the same object could be permitted as long as they are not cyclic.

Field accesses of Form 3 are used for aggregate objects, for instance, in the constraint for *maxChars* (Fig. 2). The requirement that *p* be a rep field together with the implicit object invariant for rep fields guarantees that the object referenced by **this.p** is owned by $\langle \text{this}, T \rangle$ when **this** is valid for *T*. It is imposed to adhere to the mutable dependent principle. The field *p* is allowed to be an inherited field.

5.2 Encoding and Automatic Updates of Model Fields

Following the validity principle explained in Sec. 3, our methodology guarantees that a model field *o.m* satisfies the effective constraint for *m* in a class *T* if *o* is valid for *T*. That is, the following property is a program invariant:

$$\text{J3: } (\forall o, T, m \bullet o.\text{inv} <: T \Rightarrow R_m^T(o, o.m))$$

$R_m^T(o, r)$ denotes the effective constraint for *m* in *T*, where **this.m** is replaced by *r* and **this** is then replaced by *o*. For instance, $R_{\text{width}}^{\text{Rectangle}}(o, r)$ denotes $r = o.x2 - o.x1$.

To achieve decoupling, we store the value of a model field in the heap as if it were an extra field of the class. Whenever a model field is read, that is, whenever a specification refers to a model field, the stored value is used. With the value of a model field being stored in the heap, any update of the values of a model field's dependees may

cause the stored value to become out-of-date. We arrange for the stored value to be updated automatically, but we do so only at select times—eagerly updating the stored value whenever a dependee is changed would not just be inefficient and clumsy, but it would also retain the instant effect problems of using abstraction functions, that is the modularity problems of frame properties.

Specifically, we include an automatic update of a model field in the pack operation by inserting the following statements between the second and third assert statement of the pseudo code for **pack** o as T (see Fig. 3):

```
#foreach  $m$  such that  $m$  is declared in or inherited by  $T$  {
  if  $\neg R_m^T(o, o.m)$  then
    assert  $(\exists r \bullet R_m^T(o, r))$  ;
     $o.m :=$  choose  $r$  such that  $R_m^T(o, r)$ 
  end
}
```

The automatic updates nondeterministically assign to $o.m$ any value of m 's declared type that satisfies the effective constraint for m in T . If no such value exists, the assert statement will cause program execution to abort. This assertion allows us to detect unsatisfiable constraints such as the $len = len + 1$ example from Sec. 2. The guard $\neg R_m^T(o, o.m)$ simply avoids updates that are not necessary.

5.3 Frame Properties

As explained in Sec. 2.2, methodologies for model fields based on abstraction functions lead to difficult problems for the verification of frame properties. In our methodology, a model field behaves essentially like a concrete field that is updated automatically by pack statements. Therefore, model fields do not introduce additional complexity for the verification of frame properties. In particular, the semantics of modifies clauses used in the Boogie methodology [1] works also in the presence of model fields.

The modifies clause of a method M lists access expressions that, evaluated in the method's pre-state, give a set of locations that the method is allowed to modify. We denote this set by $mod(M)$. In addition to the locations in $mod(M)$, method M is allowed to modify fields of objects allocated during the execution of M as well as fields of objects that are committed in M 's pre-state. The latter policy lets the method modify the internal representation of valid aggregate objects without explicitly mentioning these fields in the modifies clause, which enables information hiding. Clients of an aggregate object should not access the internal representation directly. Therefore, they do not have to know whether or not these fields are modified by the method. In summary, M is allowed to modify a field $o.f$ if at least one of the following conditions applies:

1. $o.f$ is contained in $mod(M)$
2. o is not allocated in the pre-state of M
3. o is committed in the pre-state of M

Note that this interpretation of modifies clauses sometimes requires hidden fields to be mentioned in modifies clauses. For instance, the modifies clause of method *ScaleH*

of class *Rectangle* (Fig. 1) has to mention the concrete field $x2$ because **this** is allocated and uncommitted in the pre-state of the method. We do not address this information hiding problem in this paper, because existing solutions such as static data groups [16] or more coarse-grained wildcards [1] can be combined with our methodology.

In our example, method *ScaleH* potentially modifies $x2$ and *width*. Both modifications are permitted by Case 1 because $x2$ and *width* are mentioned in the modifies clause. We show that *height* is not modified as follows. By program invariant J3, we have $height = y2 - y1$ in the pre-state of the method. Since *ScaleH* does not assign to $y1$ and $y2$, this property still holds before the pack statement. Therefore, *height* is not updated when **this** is being packed.

Method *Reset* of class *Legend* (Fig. 2) potentially modifies fields of the *Rectangle* object *box* by the call *box.ScaleH*(0) as well as *maxChars* by packing **this**. Since *box* is a rep field and **this** is valid for *Legend* in the pre-state of *Reset*, program invariant J1 and the implicit object invariant for rep fields imply that the object referenced by **this.box** is owned by $\langle \mathbf{this}, Legend \rangle$ and committed in the pre-state of *Reset*. Therefore, modification of its fields is permitted by Case 3. The modification of *maxChars* is permitted by Case 1.

5.4 Soundness

As explained in Sec. 4, soundness of our methodology means that it is justified to assume certain program invariants when proving the assertions introduced by the methodology. Program invariants J1 and J2 are guaranteed by the Boogie methodology. To ensure that the proofs of these program invariants remain valid, we disallow model fields in object invariants. Our methodology is sound without this restriction, but we do not have the space to present the required soundness proof here, nor does the proof give additional insights. We now proceed with the proof of program invariant J3.

The proof runs by induction over the sequence of states of an execution of a program **P**. The induction base is trivial since there are no allocated objects in the initial program state.

For the induction step, we assume that the program invariant holds before the next statement s to be executed, and show that s preserves it by proving that the following property holds after the execution of s for any object o , type T , and model field m .

$$o.inv <: T \Rightarrow R_m^T(o, o.m) \quad (1)$$

We continue by case distinction on s . Only the statements that manipulate fields of objects are interesting; we omit all other cases for brevity.

Concrete field update. Let f be a concrete field declared in a class F and consider the effect of an update $x.f := e$. We show that if $R_m^T(o, o.m)$ contains an access expression that denotes $x.f$, then o is sufficiently unpacked: $T \not\leqslant o.inv$ (that is, the left-hand side of implication 1 is *false*). We follow the cases of Def. 1.

Form 1: Since f is a concrete field, $R_m^T(o, o.m)$ does not refer to $x.f$ by access expressions of this form.

Form 2: $R_m^T(o, o.m)$ refers to $o.f$ and $x = o$. The precondition of the field update requires $F \not\leqslant o.inv$. Since T is a subclass of F (otherwise the expression $o.f$ would not typecheck), we get $T \not\leqslant o.inv$.

Form 3: $R_m^T(o, o.m)$ refers to $o.p.f$, where p is a rep field declared in a (not necessarily proper) superclass S of T , and $o.p = x$. From the precondition of the update of $x.f$ and from J2, we know that x is not committed. If o were valid for S , then J1, and the fact that p is a rep field, which translates into an implicit object invariant, gives us $o.p.owner = \langle o, S \rangle$, and therefore $o.p.committed$ —a contradiction, so we conclude that o is mutable for S : $S \not\leqslant o.inv$. Since T is a subclass of S , we have $T <: S \not\leqslant o.inv$.

Unpack. Consider the statement **unpack** x **from** S . This statement changes the *inv* field of x as well as the *committed* fields of objects directly owned by x , but nothing else. Since model fields must not refer to *inv* or *committed* fields (see Def. 1), the value of $R_m^T(o, o.m)$ cannot be changed by the unpack statement.

If $x = o$, the value of $o.inv$ after the statement is the direct superclass of S . Thus, the value of $o.inv <: T$ might only be changed from *true* to *false*. That is, Property 1 still holds after the unpack statement.

Pack. Consider the statement **pack** x **as** S . The only concrete fields that are changed by a pack statement are *inv* and *committed*. Since model fields must not refer to these fields, these updates do not have an effect on $R_m^T(o, o.m)$.

One way the program can abort is if the implicit object invariants for **rep** modifiers (see Sec. 4) do not hold in the pre-state of a pack statement. This behavior is independent of the automatic updates and the checking of the model field constraints. Therefore, we may assume in the rest of the proof that these invariants hold.

For $x \neq o$, we can prove, analogously to the case for Form 3 of concrete field updates, that the update of a model field $x.f$ preserves $R_m^T(o, o.m)$. Also, $o.inv$ is not changed by the pack statement. Consequently, the pack statement preserves Property 1.

For $x = o$, Property 1 holds trivially if $T \not\leqslant S$ because the pack statement sets $o.inv$ to S . For $S <: T$, we have to consider two cases:

1. $R_m^S(o, o.m)$ holds before the pack statement. In this case, $o.m$ is not updated. Since effective constraints include the constraints of supertypes, the implication $R_m^S(o, o.m) \Rightarrow R_m^T(o, o.m)$ holds.
2. $R_m^S(o, o.m)$ does not hold before the pack statement. By the assert statement, we know that $R_m^S(o, o.m)$ is satisfiable, that is, there is a value to choose for the update of $o.m$. Consequently, the update establishes $R_m^S(o, o.m)$. Again, since effective constraints include the constraints of supertypes, we have $R_m^S(o, o.m) \Rightarrow R_m^T(o, o.m)$.

The automatic update of a model field $o.m$ establishes $R_m^T(o, o.m)$. It remains to show that the subsequent update of any other model field $o.n$ does not invalidate $R_m^T(o, o.m)$. This property follows from the fact that during the automatic updates, $R_m^T(o, o.m)$ does not depend on $o.n$. By the definition of admissible model

fields (Def. 1), $R_m^T(o, o.m)$ can only mention three forms of field access expressions. Forms 1 and 2 cannot refer to $o.n$, since n is a model field distinct from m .

Form 3 could refer to $o.n$ if there was a rep field p declared in T and $o.p = o$. However, for any such p , we show that $o.p \neq o$:

- (i) By the implicit invariant for rep fields, we have $o.p.owner = \langle o, T \rangle$;
- (ii) By (i) and the second assert statement of **pack**, we have $o.p.inv = \mathbf{typeof}(o.p)$;
- (iii) By the first assert statement of **pack**, we have $o.inv = S$, where S is a proper superclass of T ;
- (iv) By type safety, we have $\mathbf{typeof}(o) <: T$ (otherwise, **pack** o as T would not type check);
- (v) By (iii) and (iv), we have $o.inv \neq \mathbf{typeof}(o)$;
- (vi) By (ii) and (v), we have $o.p \neq o$. □

6 Related Work

JML [5, 14] requires model fields to satisfy their constraints even for objects whose invariants are temporarily violated. Therefore, programmers are supposed to provide constraints that are satisfiable in all execution states. In our methodology, constraints express properties of valid objects, which makes specifications more concise. JML and ESC/Java2 [6] allow strengthening of constraints for inherited model fields, but do not enforce consistency. This can lead to unsoundness.

Breunese and Poll [4] address the soundness problem due to unsatisfiable constraints. They propose two solutions. Like ours, their first solution requires verifiers to provide a witness to ensure that the constraint for a model field is satisfiable. However, their desugaring of model fields does not support recursive constraints, which are often useful to handle recursive data structures. Our methodology supports recursive constraints, provided that the pivot field in the recursive model field access is a rep field. Breunese and Poll's second solution transforms model fields into parameterless pure methods (that is, methods without side effects). However, they do not show how to specify and prove frame properties in this solution.

The work closest to ours is the earlier work by Müller *et al.* [19, 20]. Like the methodology presented here, that work uses ownership to solve the modularity problem of frame properties for aggregate objects. Ownership is expressed and enforced by the Universe type system [9], which is more restrictive than the ownership encoding of the Boogie methodology. Müller *et al.*'s work encodes model fields as abstraction functions, which leads to the instant effect problem described earlier. Our methodology avoids this problem by the decoupling principle.

Leino and Nelson [15, 18] require programmers to declare explicitly which fields a model field constraint is allowed to depend on. They use these explicit dependencies for three purposes: (a) to permit methods to modify certain model fields of aggregate objects without mentioning these model fields explicitly in the modifies clause. A method is allowed to modify model fields that depend on a field listed in the modifies clause. This solves the modularity problem of frame properties for aggregate objects. (b) as an abstraction mechanism to permit methods to modify the components of aggregate

objects without declaring these modifications explicitly. A method is allowed to modify all dependee fields of a model field listed in the `modifies` clause. (c) to determine whether the modification of a field potentially affects a model field.

Explicit dependencies are not well suited for automatic program verifiers such as ESC/Java [6, 10] and Boogie [2] because automatic theorem provers such as Simplify [7] cannot easily determine how often the recursive predicate for the (transitive) depends relation should be unfolded [8]. Our methodology avoids explicit dependencies as follows: (a) Due to the decoupling principle, model fields of aggregate objects do not change instantly when their dependees are modified. Avoiding these instant changes solves the modularity problem of frame properties. (b) We allow methods to modify fields of committed objects without mentioning these fields in the `modifies` clause. If an aggregate object is valid, its components are committed. (c) Again due to the decoupling principle, the modification of a field never changes the value of a model field. Whether a dependent model field $o.m$ will be updated by the next `pack o as T` statement can be determined using the constraint for m in T .

Both Müller *et al.*'s and Leino and Nelson's work [15, 18] need a strong authenticity requirement for soundness. This requirement prevents model fields from depending on inherited fields (such as *MyRectangle* in Sec. 2.2) and, therefore, limits the support for inheritance. Moreover, they do not allow classes to strengthen inherited constraints. By freeing model fields of mutable objects from the obligation to satisfy their constraints (validity principle) and by (un-)packing objects for each superclass of their dynamic type individually, these restrictions are not necessary in our methodology.

Other recent work use abstraction functions for model fields by exploring different encodings of the programming logic [13, 12].

In this paper, we have used ownership to adhere to the mutable dependent principle. There are extensions of the Boogie methodology that use alternatives to ownership. For example, our visibility-based approach [17] adheres to the mutable dependent principle. The *update guards* of Barnett and Naumann [3] adhere to a slightly weaker mutable dependent principle, which we could have used here instead. These extensions allow model fields to depend on non-owned state, which is useful in some implementations.

Separation logic uses new logical connectives to express that a predicate depends only on certain objects in the heap. It has been used successfully to modularly verify an invariant of a single class with a single instance [22]. Parkinson and Bierman [23] extend separation logic to an object-oriented language, introducing abstract predicate families to encapsulate an object's state. They do not show, however, how to express abstractions of aggregate objects such as the *maxChars* field in Fig. 2. Our methodology treats model fields like ordinary fields with automatic updates, which are both handled by separation logic. Therefore, we hope that this contribution will help to improve the support for data abstraction in separation logic.

7 Conclusions

We have presented a sound and modular verification methodology for reasoning about model fields. Since our methodology supports subtyping, aggregate objects, and recursive object structures, it can be applied to realistic programs. Our methodology is significantly simpler and more expressive than previous approaches. These improvements

are achieved by not making any guarantees about model fields of mutable objects (validity principle), by inserting automatic updates of model fields (decoupling principle), and by imposing an ownership structure (mutable dependent principle).

Model fields are used to express abstractions of the concrete states of objects. However, in our encoding, we have fully reduced the concept of a model field to other concepts that are well-understood and well-behaved, namely fields with (automatic) updates. Therefore, our treatment of model fields can be readily adopted by a variety of programming logics.

As future work, we plan to implement our methodology as part of the .NET program checker Boogie, which is part of the Spec# programming system [2].

Acknowledgments. We are grateful to Mike Barnett, David Cok, Ádám Darvas, Sophia Drossopoulou, Manuel Fähndrich, Bart Jacobs, Yannis Kassios, Gary Leavens, David Naumann, Arnd Poetzsch-Heffter, Wolfram Schulte, and the anonymous reviewers for helpful discussions and suggestions.

References

1. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6), 2004.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2004.
3. Mike Barnett and David Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, *LNCS*. Springer-Verlag, 2004.
4. Cees-Bart Breunese and Erik Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs*, pages 51–60, 2003. Tech. Rep. 408, ETH Zurich.
5. Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software—Practice & Experience*, 35(6):583–599, 2005.
6. David Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2004.
7. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Tech. Rep. HPL-2003-148, HP Labs, July 2003.
8. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
9. Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *JOT*, 4(8), 2005.
10. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, 2002.
11. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Inf.*, 1:271–281, 1972.
12. Bart Jacobs and Frank Piessens. Verifying programs using inspector methods for state abstraction. Tech. Rep. CW 432, Dept. of Comp. Sci., K. U. Leuven, December 2005.
13. Yannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. Tech. Rep. CSRG-528, U. of Toronto, Comp. Sys. Research Group, July 2005.
14. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Iowa State University, Department of Computer Science, 2003. See www.jmlspecs.org.

15. K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.
16. K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOP-SLA*, volume 33, number 10 in *SIGPLAN Notices*, pages 144–153. ACM, 1998.
17. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
18. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
19. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.
20. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency & Computation: Practice & Experience*, 15:117–154, 2003.
21. David Naumann and Mike Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *LICS*, pages 313–323. IEEE, 2004.
22. Peter W. O’Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL*, pages 268–280. ACM, 2004.
23. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM, 2005.

ILC: A Foundation for Automated Reasoning About Pointer Programs

Limin Jia and David Walker

Princeton University, Princeton, NJ 08544, USA
{ljia, dpw}@cs.princeton.edu

Abstract. This paper shows how to use Girard’s intuitionistic linear logic extended with a classical sublogic to reason about pointer programs. More specifically, first, the paper defines the proof theory for ILC (Intuitionistic Linear logic with Constraints) and shows it is well-defined via a proof of cut elimination. Second, inspired by prior work of O’Hearn, Reynolds, and Yang, the paper explains how to interpret linear logical formulas as descriptions of a program store. Third, this paper defines a simple imperative programming language with mutable references and arrays and gives verification condition generation rules that produce assertions in ILC. Finally, we identify a fragment of ILC, ILC^- , that is both decidable and closed under generation of verification conditions. Since verification condition generation is syntax-directed, we obtain a decidable procedure for checking properties of pointer programs.

1 Introduction

In the eighties and early nineties, formal program specification and verification was left for dead: it was too difficult, too costly, too time-consuming, and completely unscalable. Amazingly, in 2005, Microsoft is using verification technology in many of their internal projects and is currently planning to include a logical specification and checking language in their next version of Visual C [1]. This remarkable turnaround was made possible in part by moving away from complete program verification to verification of a smaller selection of simple but useful program properties, and in part by great improvements in abstract interpretation and theorem proving technologies.

Some of the most successful recent verification projects include the Microsoft assertion language mentioned above, Leino et al.’s extended static checking project and its successors [2, 3, 4], and Nacula and Lee’s proof-carrying code [5]. These tools have used conventional classical logic to specify and check program properties. These conventional logics work exceptionally well for specifying arithmetic conditions and verifying that array accesses are in bounds. One place where there remains room for improvement is in specification and verification of programs that manipulate pointers and manage resources. To better support verification of pointer programs, O’Hearn, Reynolds, and Yang [6, 7] have advocated using separation logic, which is the classical logic of bunched implications

extended with a collection of domain-specific axioms about storage. The crucial insight in this research is that the multiplicative connectives of the logic of bunched implications encapsulate “separation” invariants commonly used when reasoning about storage.

Inspired by the work of O’Hearn et al., we have begun to develop a new program logic in which the proof theory used to reason about state and resources is based on Girard’s linear logic as opposed to the logic of bunched implication. There are several reasons why we decided to focus on linear logic as opposed to bunched implications as a foundation for verifying programs. First, from a practical standpoint, there are a number of tools available for our use including logic programming engines Lolli [8] and Lollimon [9], theorem provers [10] and logical frameworks such as Forum [11], LLF [12], and CLF [13]. Second, since linear logic is older than BI, more is known about it. In particular, we have been able to use known results on the complexity of various fragments of linear logic to devise a useful decidable fragment of our logic. Third, we have recently looked at generating proof-carrying code for programs with rich memory management invariants [14, 15], and while we found encoding “single-pointer” invariants in separation logic highly effective, we were unable to find a simple encoding for general-purpose (typed) shared mutable references. Consequently, we fell back on older ideas from the work on alias types [16], which implicitly, and in newer work [17], explicitly, use linear logic’s unrestricted modality as part of the encoding. Though we do not focus on this issue in this paper, it is clear that ILC can easily accommodate these encodings.

In addition, this paper is a starting point from which we can begin to study the relative strengths and weaknesses of using the proof theory of intuitionistic linear logic, which is based on sequents with a flat context for assumptions, as opposed to the proof theory of BI, which is based on sequents with “bunched” or tree-like contexts, as the foundation for verification of pointer programs.

To summarize, there are four central contributions of this paper. First (Section 2), we propose ILC as opposed to bunched logic as a foundation for checking safety properties of pointer programs. We outline the proof theory for ILC as a sequent calculus and prove a cut-elimination theorem to show that it is well defined. The proof theory is sound with respect to the storage model, but not complete. As any automated program analysis will run up against incompleteness somewhere, this lack of completeness is not an immediate practical concern for us. However, an important element of future work will be understanding the sources of the incompleteness. Logic of bunched implications does have certain completeness properties and therefore has an advantage over linear logic in this respect. Our second contribution (Section 3) is to define a simple imperative language with references and to give syntax-directed verification condition generation rules that use ILC as the assertion language. We prove that our verification condition generation is sound with respect to our memory model. The third main contribution (Section 2.6) is in the definition of a useful, and decidable fragment of the logic, ILC^- . The key property of ILC^- is that it is closed under verification condition generation: if loop invariants and pre- and post-conditions fall

into ILC^- then the generated verification conditions also fall into ILC^- . The decidable logic plus the syntax-directed verification condition generation give rise to a terminating algorithm for verification of pointer programs. Fourth, we have implemented a prototype verifier for our language. Our prototype generates verification conditions in ILC. We then prove the validity of linear logic formulas in MetaPRL [18], a manual process at this point, and discharge the constraints using the CVC Lite [19] theorem prover. The examples in this paper have been verified using our implementation. Due to space considerations, we have omitted many technical details. Please see our technical report [20] for complete formal rules and additional metatheory.

2 Intuitionistic Linear Logic with Constraints

In this section we introduce ILC, Intuitionistic Linear logic with Constraints. After introducing the syntax, semantics, proof theory, and properties of ILC, we will present a decidable fragment, ILC^- .

2.1 Syntax

ILC formulas F include all of the first-order formulas present in multiplicative and additive intuitionistic linear logic. In addition, a modality $\bigcirc A$ encapsulates a language of classical constraints as a sublogic within ILC. For the purposes of this paper, the constraint language involves arrays and Presburger arithmetic.

The basic predicates for reasoning about program state include $(E_1 \Rightarrow E_2)$, which describes a heap containing only one location, E_1 , and its contents E_2 ; and $\text{Array}(E_1, E_2, \alpha)$, which describes an array that has a starting address E_1 , number of elements E_2 , and list of elements α . The classical constraints use E to range over integer terms and α to range over array terms. The empty array is denoted by Nil , $\text{sel}(\alpha, E)$ accesses the E_{th} element of α , and $\text{upd}(\alpha, E_1, E_2)$ generates a new array with the element indexed by E_1 replaced by E_2 .

<i>Integer Terms</i>	$E ::= n \mid x \mid E_1 + E_2 \mid -E \mid \text{sel}(\alpha, E)$
<i>Array Terms</i>	$\alpha ::= \text{Nil} \mid x \mid \text{upd}(\alpha, E_1, E_2)$
<i>Arithmetic Predicates</i>	$Pa ::= E_1 = E_2 \mid E_1 < E_2$
<i>Classical Formulas</i>	$A ::= \text{true} \mid \text{false} \mid Pa \mid A_1 \wedge A_2 \mid \neg A \mid A_1 \vee A_2$
<i>State Predicates</i>	$Ps ::= (E_1 \Rightarrow E_2) \mid \text{Array}(E_1, E_2, \alpha)$
<i>Intuitionistic Formulas</i>	$F ::= Ps \mid \mathbf{1} \mid F_1 \otimes F_2 \mid F_1 \multimap F_2 \mid \top \mid F_1 \& F_2 \mid \mathbf{0} \mid F_1 \oplus F_2 \mid !F \mid \exists b.F \mid \forall b.F \mid \bigcirc A$

2.2 Basic Concepts

We informally discuss the semantics of the connectives and highlight the key ideas for reasoning about program states. All the examples in this section refer to Figure 1, which shows a heap h containing two disjoint parts: h_1 and h_2 . The first part h_1 contains location x , which contains integer 3; the second part h_2 contains location y , which contains integer 4.

Emptiness. The connective $\mathbf{1}$ describes an empty heap. The counterpart in separation logic is usually written emp .

Separation. Multiplicative conjunction \otimes separates a linear state into two disjoint parts. For example, the heap h can be described by formula $(x \Rightarrow 3) \otimes (y \Rightarrow 4)$. Multiplicative conjunction does not allow weakening or contraction. Therefore, we can uniquely identify each part in the heap and track its state changes. The multiplicative conjunction $(*)$ in separation logic has the same properties.

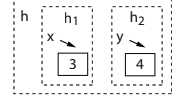


Fig. 1. A sample heap

Update. Multiplicative implication \multimap is similar to the multiplicative implication \multimap in separation logic. Formula $F_1 \multimap F_2$ describes a heap h waiting for another piece; if given another heap h' that is described by F_1 , and if h' is disjoint from h , then the union of h and h' can be described by F_2 . For example, h_2 can be described by $(x \Rightarrow 3) \multimap ((x \Rightarrow 3) \otimes (y \Rightarrow 4))$. A more interesting example is that h satisfies formula $F = (x \Rightarrow 3) \otimes ((x \Rightarrow 5) \multimap ((x \Rightarrow 5) \otimes (y \Rightarrow 4)))$. This example brings out the idea of describing store updates using multiplicative conjunction and implication.

No Information. The unit of additive conjunction \top describes any linear state, so it does not contain any specific information about the linear state it describes. The counterpart of \top in separation logic is usually written **true**.

Sharing. Formula $F_1 \& F_2$ represents a state that can be described by (shared between) both F_1 and F_2 . For example, h is described by $((x \Rightarrow 3) \otimes \top) \& ((y \Rightarrow 4) \otimes \top)$. The additive conjunction in separation logic is written \wedge . The basic sharing properties of these two connectives are the same. But the behavior of \wedge is closely connected to the additive implication \rightarrow and the bunched contexts, which our logic does not have.

Heap Free Conditions. The unrestricted modality $!F$ describes an empty heap and asserts F is true. For instance, $!((x \Rightarrow 3) \multimap \exists y.(x \Rightarrow y))$ says that given no initial resources, if we add a heap in which location x holds 3 then we end up with a heap in which location x holds some y . On the other hand, $!(x \Rightarrow 3)$ cannot be satisfied. Note that $!F$ is semantically equivalent to $F \& \mathbf{1}$. However, as we will see in the next section, the two formulas have different proof-theoretic properties. Formula $!F$ satisfies weakening and contraction and therefore can be used multiple times; $F \& \mathbf{1}$ does not satisfy these properties. Hence $!$ is used as a simple syntactic marker that informs the theorem prover of the structural properties to apply to the underlying formula. The equivalent idea in separation logic is that of a “pure formula.” Rather than using a connective to mark the purity attribute, a theorem prover analyzes the syntax of the formula to determine its status. Pure formulas are specially axiomatized in separation logic.

Classical Reasoning. In separation logic, the law of excluded middle holds in the classical semantics. For instance, formula $(x \Rightarrow 3) \vee \neg(x \Rightarrow 3)$ is valid.

However, negations of “heapful” conditions, such as $\neg(x \Rightarrow 3)$, appear very rarely, but classical reasoning about constraints is ubiquitous. Consequently, we add a classical sublogic to what we have already presented. The classical formulas describe constraints and are confined under the modality \bigcirc . For example, heap h satisfies $\exists e_1. \exists e_2. (x \Rightarrow e_1) \otimes (y \Rightarrow e_2) \otimes !(\bigcirc(\neg(e_1 = e_2)))$. In separation logic we would write $\exists e_1. \exists e_2. ((x \Rightarrow e_1) * (y \Rightarrow e_2)) \wedge (\neg(e_1 = e_2))$. The modality \bigcirc separates the classical reasoning about arithmetic or other constraints from the intuitionistic linear reasoning making it possible to use an off-the-shelf theorem prover or decision procedure for the constraints.

2.3 Semantics

Our logical formulas describe program stores that map locations to values. All values are integers or integer tuples; some integers (an infinite collection of them) are considered heap locations. We use metavariable n when referring to integers, ℓ when referring to locations, and v when referring to values.

We use $\text{dom}(h)$ to denote the domain of store h , $h(\ell)$ to denote the value stored at location ℓ , $h[\ell := v]$ to denote a store h' in which ℓ maps to v but is otherwise the same as h . We write $h_1 \uplus h_2$ to denote the union of disjoint stores. The \uplus operation is undefined if the stores are not disjoint. We use $|v|$ to denote the number of elements in tuple v , $v|_i$ to denote the i_{th} elements of v if $0 \leq i < |v|$, and $v[i \mapsto v']$ to denote the result of updating the i_{th} element of v with v' , if $0 \leq i < |v|$.

There are three semantic judgments:

$\mathcal{M} \models A$	Classical formula A is valid in model \mathcal{M}
$\mathcal{M}; h \models F$	Store h together with model \mathcal{M} satisfies formula F
$h \models F$	Store h satisfies formula F (exists a model \mathcal{M} such that $\mathcal{M}; h \models F$)

\mathcal{M} is a model for the first-order theories we consider. We write $\llbracket E \rrbracket$ for the integer value that the closed expression E denotes. The denotation of an array term $\llbracket \alpha \rrbracket_n$ is an integer tuple of length n . The semantics of classical formulas is standard, and we omit it in this paper. The formal definition of $\mathcal{M}; h \models F$ is given in Figure 2.

- $\mathcal{M}; h \models (E_1 \Rightarrow E_2)$ iff $\text{dom}(h) = \{\llbracket E_1 \rrbracket\}$, $h(\llbracket E_1 \rrbracket) = \llbracket E_2 \rrbracket$.
- $\mathcal{M}; h \models \text{Array}(E_1, E_2, Y)$ iff $\{\llbracket E_1 \rrbracket\} = \text{dom}(h)$ and $h(\llbracket E_1 \rrbracket) = \llbracket Y \rrbracket_n$, where $n = \llbracket E_2 \rrbracket$.
- $\mathcal{M}; h \models \mathbf{1}$ iff $\text{dom}(h) = \emptyset$
- $\mathcal{M}; h \models F_1 \otimes F_2$ iff $h = h_1 \uplus h_2$, and $\mathcal{M}; h_1 \models F_1$, and $\mathcal{M}; h_2 \models F_2$.
- $\mathcal{M}; h \models F_1 \multimap F_2$ iff for all stores h' , $\mathcal{M}; h' \models F_1$ implies $\mathcal{M}; h \uplus h' \models F_2$.
- $\mathcal{M}; h \models \top$ is true for all stores.
- $\mathcal{M}; h \models F_1 \& F_2$ iff $\mathcal{M}; h \models F_1$, and $\mathcal{M}; h \models F_2$.
- $\mathcal{M}; h \models \mathbf{0}$ is false for all stores.
- $\mathcal{M}; h \models F_1 \oplus F_2$ iff $\mathcal{M}; h \models F_1$, or $\mathcal{M}; h \models F_2$.
- $\mathcal{M}; h \models !F$ iff $\text{dom}(h) = \emptyset$, and $\mathcal{M}; h \models F$.
- $\mathcal{M}; h \models \exists x. F$ iff there exists some value a such that $\mathcal{M}; h \models F[a/x]$.
- $\mathcal{M}; h \models \forall x. F$ iff for all values a , $\mathcal{M}; h \models F[a/x]$.
- $\mathcal{M}; h \models \bigcirc A$ iff $\text{dom}(h) = \emptyset$, and $\mathcal{M} \models A$.

Fig. 2. The semantics of formulas

2.4 Proof Theory

Our logical judgments make use of an unrestricted context Γ for classical constraints, an unrestricted context Θ for intuitionistic formulas, and a linear context Δ , also for intuitionistic formulas. The first two contexts have contraction, weakening, and exchange properties, while the last has only exchange. The context Ω contains the set of variables free in the rest of the sequent.

Our logic has two sequent judgments.

$$\begin{array}{ll} \Omega \mid \Gamma \# \Gamma' & \text{classical sequent rules} \\ \Omega \mid \Gamma; \Theta; \Delta \Longrightarrow F & \text{intuitionistic sequent rules} \end{array}$$

The sequent rules for classical logic follow the LK formalization [21]. An intuitive reading of the intuitionistic sequent is that if a state is described by unrestricted assumptions in Θ , linear assumptions Δ , and satisfies all the classical constraints in Γ , then this state can also be described by F .

Our logic has the same sequent rules as those in intuitionistic linear logic except that the classical context Γ is carried around. The interesting rules are the left and right rule for the new modality \circ and the *absurdity* rule listed below. These rules illustrate the interaction between the classical and the intuitionistic part of the logic. The right rule for \circ says that if Γ contradicts the assertion “ A false” (which means A is true) then we can derive $\circ A$ without using any linear resources. If we read the left rule for \circ bottom up, it says that whenever we have $\circ A$, we can put A together with other classical assumptions in Γ . The absurdity rule is a peculiar one. The justification for this rule is that since Γ is not consistent, no state can meet the constraints imposed by Γ ; therefore, any statement based on the assumption that a state satisfies those constraints is simply true.

$$\frac{\Omega \mid \Gamma \# A}{\Omega \mid \Gamma; \Theta; \cdot \Longrightarrow \circ A} \circ R \quad \frac{\Omega \mid \Gamma, A; \Theta; \Delta \Longrightarrow F}{\Omega \mid \Gamma; \Theta; \Delta, \circ A \Longrightarrow F} \circ L \quad \frac{\Omega \mid \Gamma \# \cdot}{\Omega \mid \Gamma; \Theta; \Delta \Longrightarrow F} \text{Absurdity}$$

Interesting Theorems. The following axioms, all of which are provable in our sequent calculus, illustrate some of the interactions between the classical and intuitionistic connectives.

$$\begin{array}{lll} \circ \text{true} \Longleftrightarrow \mathbf{1} & \circ A \otimes \circ B \Longleftrightarrow \circ(A \wedge B) & \circ(A \wedge B) \Longrightarrow \circ A \& \circ B \\ \circ \text{false} \Longleftrightarrow \mathbf{0} & & \circ A \oplus \circ B \Longrightarrow \circ(A \vee B) \end{array}$$

It is also interesting to consider the proof theory for heap-free formulas, which we represent using Girard’s unrestricted modality. The critical axioms here are the structural properties of contraction and weakening: $!F \Longrightarrow \mathbf{1}$, $!F \Longrightarrow !F \otimes !F$. In separation logic, Reynolds [22] adds specialized axioms for relating the additive conjunction of pure facts to the multiplicative conjunction of them:

$$P \wedge Q \Longrightarrow P * Q \text{ when } P \text{ or } Q \text{ is pure} \quad P * Q \Longrightarrow P \wedge Q \text{ when } P \text{ and } Q \text{ is pure}$$

In our logic, we can prove $!P \otimes !Q \Longrightarrow !P \& !Q$ but not the reverse. We forgo these additional axioms for practical reasons: we wish to reuse a theorem prover for first-order intuitionistic linear logic rather than building a new prover from

scratch. One consequence of this choice is that programmers must write invariants consistently in the form $!P \otimes !Q$ instead of $!P \& !Q$. So far, we have seen no practical consequences of omitting this axiom.

2.5 Properties of ILC

We have proven a cut elimination theorem of our logic (Thm 1). We also proved that the proof theory of our logic is sound with regard to its semantics (Thm 2).

We use the notion of semantics for logical contexts (written $h \models \Gamma; \Theta; \Delta$) in Theorem 2. It means that store h satisfies all the constraints in Γ and h contains all the unrestricted resources in Θ and all the linear resources in Δ .

Theorem 1 (Cut Elimination).

1. If $\Omega \mid \Gamma \# A$ and $\Omega \mid \Gamma, A; \Theta; \Delta \Rightarrow F$ then $\Omega \mid \Gamma; \Theta; \Delta \Rightarrow F$.
2. If $\Omega \mid \Gamma; \Theta; \cdot \Rightarrow F$ and $\Omega \mid \Gamma; \Theta, F; \Delta \Rightarrow F'$ then $\Omega \mid \Gamma; \Theta; \Delta \Rightarrow F'$.
3. If $\Omega \mid \Gamma; \Theta; \Delta \Rightarrow F$ and $\Omega \mid \Gamma; \Theta; \Delta', F \Rightarrow F'$ then $\Omega \mid \Gamma; \Theta; \Delta, \Delta' \Rightarrow F'$.

Theorem 2 (Soundness of Logic Deduction).

If $\Omega \mid \Gamma; \Theta; \Delta \Rightarrow F$ and σ is a grounding substitution for all the variables Ω , and $h \models \Gamma[\sigma]; \Delta[\sigma]; \Delta[\sigma]$, then $h \models F[\sigma]$.

2.6 A Decidable Fragment: ILC^-

We have identified a fragment of our logic, ILC^- , which is decidable and sufficient to encode many pre- and post-conditions for programs. One important property of ILC^- is that it is closed under the verification condition generation, which we will present in the next section. In other words, if all the programmer supplied program annotations fall into this fragment, then the whole process of program verification is decidable.

The factors that contribute to the undecidability of ILC are that 1) it contains Intuitionistic Linear Logic as a sub-logic which is undecidable, and 2) the validity of arbitrarily quantified first-order classical formulas of equality and array theory is undecidable. In order to obtain a decidable fragment of ILC, first we replace the *copy* rule with the *U-Init* and $! \circ L$ rules (we use \Rightarrow for sequents in ILC^-):

$$\begin{array}{c}
 \frac{\Omega \mid \Gamma; \Theta, F; \Delta, F \Rightarrow F'}{\Omega \mid \Gamma; \Theta, F; \Delta \Rightarrow F'} \text{ Copy} \\
 \\
 \frac{}{\Omega \mid \Gamma; \Theta, P; \cdot \Rightarrow P} \text{ U-Init} \quad \frac{\Omega \mid \Gamma, A; \Theta; \Delta \Rightarrow F}{\Omega \mid \Gamma; \Theta; \Delta, ! \circ A \Rightarrow F} ! \circ L
 \end{array}$$

Second, we syntactically restrict the logical formulas so that we don't need to decompose connectives in the unrestricted context. Now the two new rules have the same power as the old *copy* rule. Furthermore, we only consider Presburger Arithmetic to guarantee the decidability in classical reasoning part (any decidable system of constraints will do). Each syntactic class in this decidable fragment is defined as follows:

<i>Forms in Intuit. Unrestricted Ctx</i>	$D_u ::= Ps$
<i>Forms in Intuit. Linear Ctx</i>	$D_l ::= Ps \mid !Ps \mid !\circ A \mid \mathbf{1} \mid D_l \otimes D'_l \mid \top \mid D_l \& D'_l$
	$\mid \mathbf{0} \mid D_l \oplus D'_l \mid \exists x. D_l \mid \forall x. D_l$
<i>Goal Forms</i>	$G ::= Ps \mid \mathbf{1} \mid G_1 \otimes G_2 \mid D_l \multimap G \mid \top \mid G_1 \& G_2$
	$\mid \mathbf{0} \mid G_1 \oplus G_2 \mid !G \mid \exists b. G \mid \forall b. G \mid \circ A$

We have proven that in the above fragment the sequent rules with *U-Init* and $!\circ L$ are sound and complete with regard to the original sequent rules in Section 2.4.

Theorem 3 (Soundness & Completeness of \Rightarrow). $\Omega \mid \Gamma; \Theta; \Delta \Rightarrow G$ iff $\Omega \mid \Gamma; \Theta; \Delta \Rightarrow G$, provided that all the formulas in Γ are in A , all the formulas in Θ are in D_u , and all the formulas in Δ are in D_l .

The proof of the decidability of ILC^- can be found in the technical report [20]. Informally, any proof search in ILC^- can be reduced to two procedures: first, a proof search in the sequent calculus of intuitionistic linear logic without the *copy* rule, and second, the validity checking of Presburger Arithmetic formulas with equality. The first part is decidable since every premise of each sequent rule is strictly smaller than its consequent (by smaller we mean that the number of connectives in the sequent decreases [23]). The second part is also decidable. Therefore, the whole process is decidable.

Theorem 4 (Decidability). ILC^- is decidable.

Discussion. Notice that we only consider the decidable Presburger Arithmetic constraints in ILC^- . More generally, the intuitionistic linear logic part of ILC^- is always decidable, and the decidability is sustained when we extend ILC^- with any decidable constraint domain.

3 Verifying Pointer Programs

In this section, we show how to verify an imperative language with pointer operations using our logic. We present syntax-directed verification condition generation rules and give examples to show how they are used to verify programs.

3.1 Syntax and Operational Semantics

Now we introduce the syntax and operational semantics of an imperative language that includes control flow, mutable references, and arrays.

Syntax. The syntactic constructs of our language are listed below. We use E to range over integer expressions and B to range over boolean expressions. The language has commands for allocation, deallocation, variable binding, dereference, assignment, array operations, sequencing, while loop, if branching, and skip. The while loop expression **while**_[I] R **do** C is annotated with loop invariant I . The condition expression R computes a boolean that determines while termination. These condition expressions have special structure and scoping

rules to both simplify pre-condition generation and to provide ample expressive power. The variables in the loop body of the while loop are bound by the *let* expression in the condition R . For example, in the following command $\text{while}_{[\top]} \text{let } x = !y \text{ in } x > 0 \text{ end do } y := x - 1$, variable x in the loop body is bound by the *let* expression. In order to generate verification conditions properly from expressions, we require them to be in A-Normal form. Naturally, an implementation would allow programmers to write ordinary expressions and then unwind them to A-Normal form for verification.

<i>Int Exps</i>	$E ::= n \mid x \mid E + E \mid -E$
<i>Boolean Exps</i>	$B ::= \text{true} \mid \text{false} \mid E_1 = E_2 \mid E_1 < E_2 \mid B_1 \wedge B_2 \mid \neg B \mid B_1 \vee B_2$
<i>Condition Exps</i>	$R ::= B \mid \text{let } x = !E \text{ in } R \text{ end}$
<i>Command</i>	$C ::= \text{let } x = \text{new}(E) \text{ in } C \text{ end} \mid \text{free}(E)$ $\quad \mid \text{let } x = E \text{ in } C \text{ end} \mid \text{let } x = !E \text{ in } C \text{ end} \mid E_1 := E_2$ $\quad \mid \text{let } x = \text{newArray}(E) \text{ in } C \text{ end} \mid \text{let } x = E_1[E_2] \text{ in } C \text{ end}$ $\quad \mid E_1[E_2] := E_3 \mid \text{let } x = \text{Len}(E) \text{ in } C \text{ end} \mid C_1 ; C_2$ $\quad \mid \text{while}_{[I]} R \text{ do } C \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{skip}$

Operational Semantics. A program state consists of a control stack S , a store (or a heap) h , and an instruction ι being evaluated. An instruction ι can be a command, a loop guard R followed by a command C , or the special instruction \bullet , which indicates the termination of certain commands. In our language, variables are bound, and there is no imperative assignment to variables. We therefore do not need a stack to map variables to values. We use $(S, h, \iota) \mapsto (S', h', \iota')$ to denote the small step operational semantics. The control stack S is a list of evaluation contexts and is not crucial for the understanding of this paper, so we omit its definition.

3.2 Verification Condition Generation

The program to be verified is annotated with pre- and post-condition and loop invariants by the programmer. The verification condition generation scans the program bottom-up and generates a formula (from the postcondition) such that if the initial program states satisfy this formula then the program will execute safely, and if it terminates then the ending state will satisfy the specified postcondition. The computed verification condition will be satisfied by the initial state if it is logically entailed by the programmer provided precondition, which we assume will hold before the execution of the program. We have not tackled the question of whether or not our verification conditions are weakest preconditions.

There are two judgments involved in verification condition generation.

$\Delta \vdash (\exists x_1 \dots \exists x_n, F, A, \rho) R$	There exist values for variables $x_1 \dots x_n$ such that the precondition of executing R is F , the core boolean expression in R is A , and ρ is the substitution of logical terms for the variables in R bound by <i>let</i> .
$\Delta \vdash \{P\} C \{Q\}$	The precondition of C is P , and the postcondition is Q .

$$\boxed{\Delta \vdash \{P\} C \{Q\}}$$

$$\frac{\Delta, x \vdash \{P\} C \{Q\} \quad x \notin FV(Q)}{\Delta \vdash \{\forall y. (y \Rightarrow E) \multimap P[y/x]\} \text{ let } x = \text{new}(E) \text{ in } C \text{ end } \{Q\}} \text{ New}$$

$$\frac{\Delta \vdash \{\exists y. (E \Rightarrow y) \otimes Q\} \text{ free}(E) \{Q\}}{\Delta \vdash \{P\} C \{Q\}} \text{ Free}$$

$$\frac{\Delta, x \vdash \{P\} C \{Q\} \quad x \notin FV(Q)}{\Delta \vdash \{\exists y. ((E \Rightarrow y) \otimes \top) \& P[y/x]\} \text{ let } x = !E \text{ in } C \text{ end } \{Q\}} \text{ Deref}$$

$$\frac{\Delta \vdash \{\exists x. (E_1 \Rightarrow x) \otimes ((E_1 \Rightarrow E_2) \multimap Q)\} E_1 := E_2 \{Q\}}{\Delta \vdash \{P\} C \{Q\}} \text{ Assignment}$$

$$\frac{\Delta, x \vdash \{P\} C \{Q\} \quad x \notin FV(Q)}{\Delta \vdash \{! \circ (\neg(E < 0)) \otimes \forall y. (\text{Array}(y, E, \text{Nil}) \multimap P[y/x])\} \text{ let } x = \text{newArray}(E) \text{ in } C \text{ end } \{Q\}} \text{ New Array}$$

$$\frac{\Delta, x \vdash \{P\} C \{Q\} \quad x \notin FV(Q)}{\Delta \vdash \{\exists \text{size}. \exists \alpha. (\text{Array}(E_1, \text{size}, \alpha) \otimes ! \circ (\neg(E_2 < 0))) \otimes ! \circ (E_2 < \text{size}) \otimes \top\} \& P[\text{sel}(\alpha, E_2)/x] \text{ let } x = E_1[E_2] \text{ in } C \text{ end } \{Q\}} \text{ Subscript}$$

$$\frac{\Delta \vdash \{\exists \text{size}. \exists \alpha. \text{Array}(E_1, \text{size}, \alpha) \otimes ! \circ (\neg(E_2 < 0)) \otimes ! \circ (E_2 < \text{size})\} \& P[\text{sel}(\alpha, E_2, E_3) \multimap Q] \quad E_1[E_2] := E_3 \{Q\}}{\Delta \vdash \{P\} C \{Q\}} \text{ Array Upd}$$

$$\frac{\Delta \vdash \{P_1\} C_1 \{Q\} \quad \Delta \vdash \{P_2\} C_2 \{Q\}}{\Delta \vdash \{(! \circ B \multimap P_1) \& (! \circ \neg B \multimap P_2)\} \text{ if } B \text{ then } C_1 \text{ else } C_2 \{Q\}} \text{ If}$$

$$\frac{\Delta \vdash \{P\} C \{I\} \quad \Delta \vdash (\exists x_1 \dots \exists x_n. F, B, \rho) \ R}{\Delta \vdash \{\otimes ! (I \multimap (\exists x_1 \dots \exists x_n. F \& (! \circ (\neg B) \multimap Q) \& (! \circ (B) \multimap P[\rho]))) \& (! \circ (\neg B) \multimap Q))\} \text{ while}_{[I]} R \text{ do } C \{Q\}} \text{ While}$$

Fig. 3. Selected Rules for Verification Condition Generation

Commands. The verification condition generation rules are backward-reasoning rules, and are syntax directed. Most of the rules are identical to O’Hearn’s weakest precondition generation [6] except that $*$ is replaced by \otimes , \multimap by \multimap and \wedge by $\&$. We explain a few key rules here. The set of selected rules is shown in Figure 3.

The assignment command updates the cell at address E_1 with the value of E_2 . The precondition of this command asserts that the heap comprises two parts: one that contains cell E_1 , and another that waits for the update.

The precondition of the array allocation command first asserts that the size of the array is legal; the second part describes a heap that is waiting for the new piece described by $\text{Array}(y, E, \text{Nil})$. After merging with the newly allocated array, the heap satisfies the precondition of C with x substituted with the address of the new array. The precondition of the array update command first checks that E_1 indeed points to an array on the heap ($\text{Array}(E_1, \text{size}, \alpha)$). Then it checks that the index is in bounds ($! \circ (\neg(E_2 < 0)) \otimes ! \circ (E_2 < \text{size})$). The last part in the precondition describes a heap that requires the updated array to satisfy Q .

The *if* instruction branches on boolean expression B . The precondition for *if* says that if B is true then the precondition of the true branch holds; otherwise the precondition of the false branch holds. The additive conjunction is used to give two possible descriptions to the same heap. Note that the precondition of

the branch that is not taken will be proven using the absurdity rule. We will give a concrete example in Section 3.3.

While loops are annotated with loop invariants. A while loop either executes the loop body or exits the loop depending on the condition expression R . There are two parts to the precondition of a while loop. The first part $(\exists x_1 \dots \exists x_n. F \& (!\odot (\neg B) \multimap Q) \& (!\odot (B) \multimap P[\rho]))$ asserts that when we execute the loop for the first time, the precondition F for evaluating the condition expression must hold; if the condition is not true then the postcondition Q must hold, otherwise the precondition P for the loop body C must hold. The second part $(!(I \multimap (\exists x_1 \dots \exists x_n. F \& (!\odot (B) \multimap P[\rho]) \& (!\odot (\neg B) \multimap Q))))$ asserts that each time we re-enter the loop, the condition for entering the loop holds. Notice that the second formula is wrapped by an unrestricted connective ($!$). This implies that this invariant cannot depend upon the current heap state. This is a critical criterion as the heap state may be different each time around the loop.

3.3 Examples

In this section, we give two examples to demonstrate how we verify programs using the verification condition generation rules defined in the previous section. We prove the validity of ILC formulas in MetaPRL [18] and discharge the constraints using the CVC Lite [19] theorem prover. We do not have an automated theorem prover for ILC yet, but it is technically feasible to develop one and we are working with Frank Pfenning and Kaustuv Chaudhuri to develop the theorem prover we need.

If Branching. In this example, the store is shown in Figure 4. Location a contains 0. Depending on the contents of location x , there are two possibilities for the remainder of the store. If x contains 0, then the location next to x contains 3; if x contains an integer other than 0, then the location next to x contains another location y , and y contains 3. The first case is illustrated in the figure above the dashed line, and the second case is illustrated below the dashed line. Formula F describes the store h . We use additive disjunction to describe the two cases.

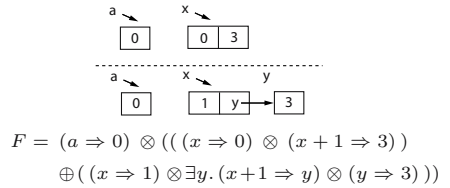


Fig. 4. Example

The following piece of code branches on the contents of x . The true branch looks up the value stored in location $x + 1$ and stores it into a ; the false branch looks up the value stored in location y and stores the value into a . At the merge point of the branch, a should contain 3.

```
{F = (a => 0) ⊗ (((x => 0) ⊗ (x + 1 => 3)) ⊕ ((x => 1) ⊗ ∃y. (x + 1 => y) ⊗ (y => 3)))}
let t = !x in
if (t = 0)
then let s = !(x + 1) in a := s end
else let s = !(x + 1) in
    let r = !s in
    a := r end end end
{(a => 3) ⊗ ⊤}
```

To show this program is memory safe and will store 3 into a in the end $((a \Rightarrow 3) \otimes \top)$, we first generate a verification condition. Next we prove that the precondition describing the initial state entails the verification condition we generated $(Pre): x, a \mid \cdot; \cdot; F \Longrightarrow Pre$. According to our sequent rules, one of the subgoals we need to prove is:

$$x, a \mid \cdot; \cdot; (a \Rightarrow 0), (x \Rightarrow 0), (x + 1 \Rightarrow 3) \Longrightarrow (! \circ \neg(0 = 0)) \multimap P_2$$

where $P_2 = \exists u.((x + 1 \Rightarrow u) \otimes \top) \& \exists v.((u \Rightarrow v) \otimes \top)$
 $\& \exists w.(a \Rightarrow w) \otimes ((a \Rightarrow v) \multimap ((a \Rightarrow 3) \otimes \top))$

After applying $! \circ L$ rule, we have

$$x, a \mid \neg(0 = 0); \cdot; (a \Rightarrow 0), (x \Rightarrow 0), (x + 1 \Rightarrow 3) \Longrightarrow P_2$$

Obviously, the resources in the linear context are not sufficient to prove P_2 , which requires $x + 1$ to contain another location. However, we have a contradiction in the classical context $(\neg(0 = 0))$, so we prove P_2 using the absurdity rule. This is the situation where we cannot establish the precondition required by the branch that is not taken. Instead, we prove it by contradiction.

Array Copying. In this example, we prove the correctness of an array copying program. The code is shown below. At the beginning of this program, there is an array x with at least 2 elements. We will allocate a new array y that has exactly one fewer element than x and copy the elements from x to y using while loop. The postcondition specifies that at the end of the program we have two arrays, that one is one element shorter than the other, and that their elements are the same up to the length of the shorter array. The loop invariant says that the loop induction variable is always between 0 and the length of the longer array, and that from the first element up to the element indexed by the loop induction variable the two arrays have the same elements.

```
{ $\exists a$ Array( $x, n, a$ ) $\otimes ! \circ (\neg(n < 2))$ }
let len = arrayLen x in
let y = newArray[len - 1] in
let i = newPtr(0) in
while let j = !i in j < len - 1 end
  [ $\exists v. \exists n_2. \exists a. \exists b. \text{Array}(x, n, a) \otimes \text{Array}(y, n_2, b)$ 
    $\otimes ! \circ (n = n_2 + 1) \otimes (i \Rightarrow v) \otimes ! \circ (\neg(v < 0)) \otimes ! \circ (v < n)$ 
    $\otimes \forall j. ! \circ ((\neg(j < 0) \wedge (j < v)) \supset (\text{sel}(a, j) = \text{sel}(b, j)))$ ]
do
  let z = x[j] in
  y[j] := z;
  i := j + 1
end ;
free(i) end end end
{ $\exists m_1. \exists m_2. \exists c. \exists d. \exists l_x \exists l_y. \text{Array}(l_x, m_1, c) \otimes \text{Array}(l_y, m_2, d)$ 
  $\otimes ! \circ (m_1 = m_2 + 1) \otimes \forall i. ! \circ ((\neg(i < 0) \wedge (i < m_2)) \supset (\text{sel}(c, i) = \text{sel}(d, i)))$ }
```

We remark that the proof obligations generated from this program involve existentially quantified formulas of the array theory and are not in the obviously decidable quantifier free fragment. However, CVC Lite handles them fine.

3.4 Soundness of Verification Generation

Finally, we proved that the rules for verification generation are sound with regard to the semantics of the language.

Theorem 5 (Soundness of VC Gen). *If $\Delta \vdash \{P\} C \{Q\}$, and σ is a grounding substitution for all the variables in Δ , and $h \models P[\sigma]$, then*

- *either for all $n \geq 0$, there exist S' , h' , and ι such that $(\cdot, h, C[\sigma]) \mapsto^n (S', h', \iota)$.*
- *or there exists $k \geq 0$ such that $(\cdot, h, C[\sigma]) \mapsto^k (\cdot, h', \bullet)$, and $h' \models Q[\sigma]$.*

4 Related Work

The most closely related work to our own is O’Hearn, Reynolds, and Yang’s separation logic [6, 7]. Their key insight was the fact that a substructural logic, when used as the assertion language in a program logic, facilitates local reasoning about state. Recently, Berdine, Calcagno, and O’Hearn [24, 25] have investigated a decidable fragment of separation logic with equality, separating conjunction, and lists. An important advantage of their proof theory is that it is complete with respect to their model whereas our proof theory is incomplete. For us, this means that programmers must reason syntactically using linear logic proof rules as opposed to semantically. On the other hand, we consider a more extensive logic that, unlike Berdine et al., includes additives ($\&$, \top), first-order quantifiers, and an arbitrary classical sublogic, which we have instantiated with a theory of arrays and arithmetic. If the sublogic used in loop invariants is decidable then the verification conditions we generate and the overall program verification procedure is also decidable.

As researchers have been investigating new program logics, the designers of advanced type systems have been using similar techniques to check programs for safety [16, 26, 27, 28, 17]. For instance, DeLine and Fähndrich’s Vault programming language [26] uses a variation of alias types [16] to reason about memory management and software protocols for device drivers. Alias types very much resemble the fragment of separation logic containing the empty formula, the points-to predicate, and separating conjunction. In addition, alias types have a second points-to predicate that can be used to represent shared parts of the heap, an idea that is not directly present in separation logic. We believe it is straightforward to add this second form of points-to predicate to ILC and include it under Girard’s modality. The main difference between the program logics and the type systems is that type systems, particularly Vault, support better inferences while the logics include a wider variety of connectives and more sophisticated constraint systems, and therefore, are much more expressive.

More recently, Zhu and Xi [29] have shown how to blend the idea of alias types with Xi’s previous work on Dependent ML [30] to produce a type system with “stateful views.” The common link between this work and our own is that they both allow a mixture of linear and unrestricted reasoning. There are also many differences. Zhu and Xi define a type system to check for safety whereas we define a program logic with verification condition generation. Zhu and Xi’s type checking algorithm appears to require quite a number of annotations — in general, when a programmer gets or sets a reference, they must bind a new proof variable, though in some cases these annotations can be inferred. On the

other hand, Zhu and Xi define facilities for handling recursive data structures, something we do not attempt in this paper.

5 Conclusions

We have developed a sequent calculus for ILC, linear logic with constraints, and proved a cut elimination theorem. We have also defined a collection of sound, syntax-directed verification condition generation rules for a simple imperative language that produce assertions in ILC. Lastly, we have identified a fragment of ILC, ILC^- , that is both decidable and closed under generation of verification conditions. If loop invariants and pre-/post-conditions are specified in ILC^- , then the resulting verification conditions are also in ILC^- . Since verification condition generation is syntax-directed, we obtain a decidable procedure for checking properties of pointer programs.

Acknowledgements. We would like to thank Kaustuv Chaudhuri, Manuel Fähndrich, Peter O'Hearn, and Frank Pfenning for fruitful discussions about this research. We are grateful to Jason Hickey and Aleksey Nogin for helping us with MetaPrl, the proof assistant we used to check our linear logical proofs.

References

1. Yang, Z.: Putting program analysis to work at Microsoft (2005) Princeton Computer Science Department Colloquium.
2. Detlefs, D.L.: An overview of the extended static checking system. In: The First Workshop on Formal Methods in Software Practice. (1996)
3. Flanagan, C., Leino, R., Lillibridge, M., Nelson, G., Saxes, J., Stata, R.: Extended static checking for java. In: ACM Conference on Programming Language Design and Implementation. (2002)
4. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: CASSIS 2004. Number 3362 in LNCS (2004) 49–69
5. Nacula, G.: Proof-carrying code. In: Twenty-Fourth ACM Symposium on Principles of Programming Languages, Paris (1997) 106–119
6. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: 28th ACM Symposium on Principles of Programming Languages. (2001)
7. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Computer Science Logic. Number 2142 in LNCS (2001)
8. Hodas, J., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. *Information and Computation* **110** (1994)
9. Lopez, P., Pfenning, F., Polakow, J., Watkins, K.: Monadic concurrent linear logic programming. In: PPDP. (2005)
10. Chaudhuri, K., Pfenning, F.: A focusing inverse method prover for first-order linear logic. In: CADE-20. (2005)
11. Miller, D.: A multiple-conclusion meta-logic. In: Ninth Annual Symposium on Logic in Computer Science, IEEE Computer Society Press (1994) 272–281
12. Cervesato, I., Pfenning, F.: A linear logical framework. In: Information and Computation. (2000)

13. Watkins, K., Cervesato, I., Pfenning, F., Walker, D.: A concurrent logical framework: The propositional fragment. In: *Types for Proofs and Programs*. (2004)
14. Ahmed, A., Jia, L., Walker, D.: Reasoning about hierarchical storage. In: *IEEE Symposium on Logic in Computer Science*. (2003)
15. Jia, L., Spalding, F., Walker, D., Glew, N.: Certifying compilation for a language with stack allocation. In: *IEEE Symposium on Logic in Computer Science*. (2005)
16. Smith, F., Walker, D., Morrisett, G.: Alias types. In: *European Symposium on Programming, Berlin* (2000) 366–381
17. Morrisett, G., Ahmed, A., Fluet, M.: L^3 : A linear language with locations. In: *7th International Conference on Typed Lambda Calculi and Applications*. (2005)
18. Hickey, Nogin, Constable, Aydemir, Barzilay, Bryukhov, Eaton, Granicz, Kopylov, Kreitz, Krupski, Lorigo, Schmitt, Witty, Yu: MetaPRL – A modular logical environment. In: *IWHOLTP, LNCS* (2003)
19. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: *Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04)*. (2004)
20. Jia, L., Walker, D.: ILC: A foundation for automated reasoning about pointer programs. Technical Report TR-738-05, Princeton University (2005)
21. Gentzen, G.: *The Collected Papers of Gerhard Gentzen*. North Holland (1969) Edited by M. E. Szabo.
22. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: *LICS*. (2002)
23. Lincoln, P., Mitchell, J., Scedrov, A., Shankar, N.: Decision problems for propositional linear logic. *Annals of Pure and Applied Logic* **56** (1992) 239–311
24. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: *FST TCS 2004. Number 3328 in LNCS* (2004)
25. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: *Asian Symposium on Programming Languages and Systems. Number 3780 in LNCS* (2005) 52–68
26. Deline, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: *ACM Conference on Programming Language Design and Implementation*. (2001)
27. Foster, J., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: *ACM Conference on Programming Language Design and Implementation*. (2002)
28. Mandelbaum, Y., Walker, D., Harper, R.: An effective theory of type refinements. In: *International conference on functional programming*. (2003)
29. Zhu, D., Xi, H.: Safe Programming with Pointers through Stateful Views. In: *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages, Springer-Verlag LNCS vol. 3350* (2005)
30. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: *ACM Conference on Programming Language Design and Implementation, Montreal* (1998) 249–257

Bisimulations for Untyped Imperative Objects

Vasileios Koutavas and Mitchell Wand

Northeastern University
{vkoutav, wand}@ccs.neu.edu

Abstract. We present a sound and complete method for reasoning about contextual equivalence in the untyped, imperative object calculus of Abadi and Cardelli [1]. Our method is based on bisimulations, following the work of Sumii and Pierce [25, 26] and our own [14]. Using our method we were able to prove equivalence in more complex examples than the ones of Gordon, Hankin and Lassen [7] and Gordon and Rees [8]. We can also write bisimulations in closed form in cases where similar bisimulation methods [26] require an inductive specification. To derive our bisimulations we follow the same technique as we did in [14], thus indicating the extensibility of this method.

1 Introduction

Contextual equivalence, attributed to Morris in 1968 [19], is the standard relation used to prove that two terms are operationally identical. Terms a and a' are contextually equivalent if and only if for any program context C , $C[a]$ and $C[a']$ co-terminate. *CIU theorems* [16] try to ease the quantification over all contexts by examining only a subset of them (usually the reduction contexts).

In the presence of a store, though, an inductive proof of equivalence must reason not only about the possible contexts under empty, or even equal stores, but also under *related* stores. This is something that neither the standard definition of contextual equivalence, nor CIU theorems take into account. Thus using them to prove the equivalence of expressions that manipulate the store in a sufficiently different way can become cumbersome.

Denotational approaches addressed this problem by translating terms to more structured mathematical models (e.g. [18]). If two terms have the same denotations in some model then they are equivalent. Unfortunately even some evident equivalences do not hold in naive models [17], and finding fully-abstract models, in which all contextual equivalences hold, is generally difficult. For example consider the two different implementations of a cell class shown in Figure 1. The first is the usual implementation; the second stores the object in two fields and its `get` method returns one of them, depending on the value of a counter. These two programs have different denotations in most models, and thus can't be proven equivalent by a straightforward denotational method.

A more appropriate method to deal with such equivalences is by using *bisimulations*. Bisimulations were introduced in process calculi by Hennessy and Milner [9, 10], and adapted later to sequential calculi by Abramsky [2]. They are

<pre> class Cell { private Object y; Cell (Object x) {y = x;} public void set (Object z) {y = z;} public int get () {return y;} } </pre>	<pre> class Cell { private Object y1, y2; private int p; Cell (Object x) {p = 0; y1 = x; y2 = x;} public void set (Object z) {p = p+1; y1 = z; y2 = z;} public int get () {if ((p % 2) == 0) then return y1; else return y2;} } </pre>
---	---

Fig. 1. Cell Example

relations between entire program configurations, and thus the main difficulty of using them as equivalence relations is to apply them to terms and show that they are a congruence. Moreover bisimulations are often hard to write down explicitly because they are usually infinite sets with little structure.

Sumii and Pierce in [25, 26] greatly simplified the use of bisimulations in sequential calculi. Their main innovation was to group the related pairs of configurations according to their conditions of knowledge (e.g. the type environment). In this way they gave more structure to their bisimulations, thus making their concrete definition easier. Similar ideas have also been used in process calculi (eg. [5]).

In [14] we improved their method and provided a proof technique for equivalence in an imperative, untyped λ -calculus. Our improvements aimed mainly to reduce the size of bisimulations, and make possible a constructive proof even in the presence of higher-order procedures and a general store, where previous methods had shown limitations [3, 20, 26]. We achieved this by applying “up-to” techniques usually used in process calculi [22, 21, 23], and by analyzing a direct proof of equivalence to unveil weaker conditions for our bisimulations.

In this paper we follow the same methodology to create a bisimulation proof technique for the imperative, untyped object calculus of Abadi and Cardelli [1]. In contrast to [14], the values of this calculus do not have significant structure, and thus an “up-to context” technique is not useful here. Instead we use an “up-to store” technique to deal with the complex structure of the store. Using our framework we are able to construct bisimulations to prove known examples [7, 8], as well as more complex ones, which are hard to prove using the method in [7].

The rest of the paper is structured as follows: In Section 2 we review the untyped, imperative object calculus $\text{imp}\mathcal{O}$. In Section 3 we define a notion of contextual equivalence for values, and we connect it with the standard notion of contextual equivalence. In Section 4 we attempt a proof of a set being included in contextual equivalence, from which we derive the necessary conditions for the elements of that set. In Section 5 we gather these conditions into a definition for a bisimulation which we simplify in Section 6 by introducing an “up-to store”

closure on sets. In Section 7 we use bisimulations to prove the equivalence of some examples. Sections 8 and 9 summarize the related work and conclusions.

2 The Language $\text{imp}\varsigma$

We develop our theory based on $\text{imp}\varsigma$ [1], an untyped, imperative object calculus. The syntactic domains and the big-step environment semantics of the language are shown in Fig. 2 and 3, respectively.

As in [14] we use an overbar notation to denote a syntactic sequence:

$$\overline{s} = s_1, \dots, s_n$$

where s is a syntax fragment and s_i is the same fragment with i -subscripts on all meta-identifiers it contains. Thus we write $\overline{[l = \varsigma(x)b]}$, instead of $[l_1 = \varsigma(x_1)b_1, \dots, l_n = \varsigma(x_n)b_n]$, and $\overline{(v, v')} \in R$ instead of $(v_1, v'_1), \dots, (v_n, v'_n) \in R$. The size of the sequence in the overbar notation is arbitrary, or implicitly defined by the context.

In $\text{imp}\varsigma$, objects are defined by the syntactic construct $\overline{[l = \varsigma(x)b]}$, where l_i and b_i are the label and the body of the i -th method, respectively. The variable x_i is bound to the entire object when b_i is evaluated. During the evaluation bindings are kept in environments (called “stacks” in [1]).

Values in $\text{imp}\varsigma$ are objects that map method names to locations in the store; they are denoted by $\overline{[l = \iota]}$. Locations range over an infinite, countable set. The store maps locations to method closures, which consist of a method and the appropriate environment, e.g. $\langle \varsigma(x_i)b_i, \rho_i \rangle$.

We use the operational semantics of $\text{imp}\varsigma$ given in [1], with the addition of an extra condition in the ENV x rule which guarantees that there are no dangling pointers in an environment. Furthermore, there is an implicit α -conversion in the RED SELECT and RED LET rule, so that the identifiers added to the environments are unique.

EXPRESSIONS:	$a, b ::= x$	variables
	$\overline{[l = \varsigma(x)b]}$	Objects
	$a.l$	Method Invocation
	$a.l \leftarrow \varsigma(x)b$	Method update
	$\text{clone}(a)$	Cloning
	$\text{let } x = a \text{ in } b$	Let
LOCATIONS:	ι	
VALUES:	$v, u ::= \overline{[l = \iota]}$	
ENVIRONMENTS:	$\rho ::= \overline{(x \mapsto v)}$	
STORES:	$\sigma ::= \overline{(\iota \mapsto \langle \varsigma(x)b, \rho \rangle)}$	

Fig. 2. The $\text{imp}\varsigma$ Language

Judgments of the form $\sigma; \rho \vdash a \Downarrow v; \sigma_1$ represent a big-step evaluation, where expression a , under store σ and environment ρ , evaluates to value v and a new store σ_1 . We also use the form $\sigma; \rho \vdash a \Downarrow^{<k} v; \sigma_1$ to denote that the evaluation tree has height less than k .

Store and environment extensions are written as $(\sigma, \iota \mapsto \langle \varsigma(x)b, \rho \rangle)$ and $(\rho, x \mapsto \iota)$, respectively. Store update is written as $(\sigma. \iota \leftarrow \langle \varsigma(x)b, \rho \rangle)$; the contents of the location ι of store σ is given by $\sigma(\iota)$. The domain of a store or environ-

$\sigma \vdash_{\text{wf}} \diamond$		
$\emptyset \vdash_{\text{wf}} \diamond$	STORE \emptyset	STORE ι
	$\sigma; \rho \vdash_{\text{wf}} \diamond$	$\iota \notin \text{Dom}(\sigma) \quad FV(b) \subseteq \text{Dom}(\rho) \cup \{x\}$ $(\sigma, \iota \mapsto \langle \varsigma(x)b, \rho \rangle) \vdash_{\text{wf}} \diamond$
$\sigma; \rho \vdash_{\text{wf}} \diamond$		
$\frac{\sigma \vdash_{\text{wf}} \diamond}{\sigma; \emptyset \vdash_{\text{wf}} \diamond}$	ENV \emptyset	ENV x
	$\sigma; \rho \vdash_{\text{wf}} \diamond$	$x \notin \text{Dom}(\rho) \quad \{\bar{\iota}\} \subseteq \text{Dom}(\sigma)$ $\sigma; (\rho, x \mapsto [\bar{\iota}]) \vdash_{\text{wf}} \diamond$
$\sigma; \rho \vdash a \Downarrow v; \sigma'$		
	$\frac{\sigma; (\rho', x \mapsto v, \rho'') \vdash_{\text{wf}} \diamond}{\sigma; (\rho', x \mapsto v, \rho'') \vdash x \Downarrow v; \sigma}$	RED X
	$\frac{\sigma; \rho \vdash_{\text{wf}} \diamond \quad \{\bar{\iota}\} \cap \text{Dom}(\sigma) = \emptyset}{\sigma; \rho \vdash [\bar{\iota} = \varsigma(x)b] \Downarrow [\bar{\iota} = \iota]; (\sigma, \iota \mapsto \langle \varsigma(x)b, \rho \rangle)}$	RED OBJECT
$\sigma; \rho \vdash a \Downarrow [\bar{\iota} = \iota]; \sigma'$	$\frac{x_j \notin \text{Dom}(\rho') \quad \sigma'(\iota_j) = \langle \varsigma(x_j)b_j, \rho' \rangle}{\sigma'; (\rho', x_j \mapsto [\bar{\iota} = \iota]) \vdash b_j \Downarrow v; \sigma''}$	RED SELECT
	$\sigma; \rho \vdash a.l_j \Downarrow v; \sigma''$	
	$\frac{\sigma; \rho \vdash a \Downarrow [\bar{\iota} = \iota]; \sigma' \quad l_j \in \{\bar{\iota}\}}{\sigma; \rho \vdash a.l_j \Leftarrow \varsigma(x)b \Downarrow [\bar{\iota} = \iota]; (\sigma'.l_j \leftarrow \langle \varsigma(x)b, \rho \rangle)}$	RED UPDATE
	$\frac{\sigma; \rho \vdash a \Downarrow [\bar{\iota} = \iota]; \sigma' \quad \{\bar{\iota}'\} \notin \text{Dom}(\sigma')}{\sigma; \rho \vdash \text{clone}(a) \Downarrow [\bar{\iota} = \iota']; (\sigma', \iota' \mapsto \sigma'(\iota))}$	RED CLONE
$\sigma; \rho \vdash a \Downarrow v'; \sigma'$	$\frac{x \notin \text{Dom}(\rho) \quad \sigma'; (\rho, x \mapsto v') \vdash b \Downarrow v''; \sigma''}{\sigma; \rho \vdash \text{let } x = a \text{ in } b \Downarrow v''; \sigma''}$	RED LET

Fig. 3. Operational Semantics

ment is given by $Dom(\sigma)$ or $Dom(\rho)$, and the locations of a value v are given by $Locs(v)$.

Judgments of the form $\sigma \vdash_{\text{wf}} \diamond$ define the *well-formed stores*, while judgments $\sigma; \rho \vdash_{\text{wf}} \diamond$ define the *well-formed environments* under some store. We also call the form $\sigma; \rho \vdash a$ a *configuration*, and define the notion of a *well-formed configuration* as follows:

Definition 1. *We say that $\sigma; \rho \vdash a$ is a well-formed configuration, and we write $\sigma; \rho \vdash_{\text{wf}} a$, iff $\sigma; \rho \vdash_{\text{wf}} \diamond$ and $FV(a) \subseteq Dom(\rho)$.*

The following lemma states that the final value of a well-formed configuration can create only well-formed configurations.

Lemma 1. *If $\sigma; \rho \vdash_{\text{wf}} a$ and $\sigma; \rho \vdash a \Downarrow v; \sigma_1$, then for any environment ρ_1 , and identifier $x \notin Dom(\rho_1)$, such that $\sigma_1; \rho_1 \vdash_{\text{wf}} \diamond$, we have $\sigma_1; (\rho_1, x \mapsto v) \vdash_{\text{wf}} \diamond$.*

Proof. By straightforward induction on the height of $\sigma; \rho \vdash a \Downarrow v; \sigma_1$.

3 Contextual Equivalence

The first relation we define is the standard contextual equivalence for this calculus.

Definition 2 (Standard Contextual Equivalence (\equiv_{std})). *$(a, a') \in \equiv_{\text{std}}$ if and only if for all contexts C such that $\emptyset; \emptyset \vdash_{\text{wf}} C[a]$ and $\emptyset; \emptyset \vdash_{\text{wf}} C[a']$, we have:*

$$\emptyset; \emptyset \vdash C[a] \Downarrow \iff \emptyset; \emptyset \vdash C[a'] \Downarrow$$

Proving equivalence of two expressions using this definition is hard because carrying out the proof will require us to reason about equivalent expressions not just under empty or even equal stores and environments, but also under *equivalent* stores and environments.

To address this complication we first define a different notion of contextual equivalence (\equiv) as a set of *bisimulation states*. A bisimulation state consists of a pair of stores, and a set of pairs of values that are to be considered equivalent in these states. Then we extend this equivalence to any expression and we show that it coincides with standard contextual equivalence.

We reach this new definition of contextual equivalence by building on the following two definitions of simpler relations.

Definition 3 (Value Relation). *A value relation R is a set of pairs of values.*

Value relations hold the related values, and implicitly the related store locations, in some state of the equivalence. These objects and locations are also the only ones that a context can access.

Definition 4 (Environment Relation (R^ϵ)). If R is a value relation, then R^ϵ is a relation on environments, defined by:

$$R^\epsilon = \{(\rho, \rho') \mid \text{Dom}(\rho) = \text{Dom}(\rho') \ \& \ \forall x \in \text{Dom}(\rho). (\rho(x), \rho'(x)) \in R\}$$

Since environments related by R^ϵ contain objects from R , the locations in the environments are in the domain of the stores of a state iff R contains locations only in the domain of these stores. Therefore we define the notion of well-formed bisimulation states:

Definition 5 (Well-Formed State). We call a state (σ, σ', R) well-formed, and we write $\sigma, \sigma' \vdash_{\text{wf}} R$, iff the following holds:

$$\begin{aligned} \forall (v, v') \in R \ . \quad & (\text{Locs}(v) \subseteq \text{Dom}(\sigma)) \ \wedge \ (\sigma \vdash_{\text{wf}} \diamond) \\ & \wedge \ (\text{Locs}(v') \subseteq \text{Dom}(\sigma')) \wedge (\sigma' \vdash_{\text{wf}} \diamond) \end{aligned}$$

We now give the definition of contextual equivalence:

Definition 6 (Contextual Equivalence (\equiv)). Contextual equivalence is the set of all states (σ, σ', R) such that σ, σ' are stores and R is a value relation, and:

1. $\sigma, \sigma' \vdash_{\text{wf}} R$,
2. if $(\rho, \rho') \in R^\epsilon$, and a is any expression such that $FV(a) \subseteq \text{Dom}(\rho)$ then:

$$\sigma; \rho \vdash a \Downarrow \quad \text{iff} \quad \sigma'; \rho' \vdash a \Downarrow$$

To extend \equiv to any pair of expressions, we create a one-method object for each expression, such that the expression is the body of the method, and we relate these objects in an appropriate state of \equiv .

Definition 7 (Extension of (\equiv) to Any Expression). Two expressions a, a' are related by \equiv , and we write $a \equiv a'$, iff:

$$\begin{aligned} \forall \sigma, \rho. \exists l, \iota: \text{if} \quad & \sigma_1 = (\sigma, \iota \mapsto \langle \varsigma(_)a, \rho \rangle) \\ & \wedge \sigma'_1 = (\sigma, \iota \mapsto \langle \varsigma(_)a', \rho \rangle) \\ \text{then} \quad & \sigma_1, \sigma'_1 \vdash_{\text{wf}} \text{Id}(\sigma) \\ & \wedge (\sigma_1, \sigma'_1, \text{Id}(\sigma)) \in \equiv \end{aligned}$$

where:

$$\text{Id}(\sigma) = \{(\overline{[l = \iota]}, \overline{[l = \iota]}) \mid \iota \in \text{Dom}(\sigma)\}$$

The extended \equiv coincides with \equiv_{std} :

Theorem 1. $a \equiv a'$ if and only if $a \equiv_{\text{std}} a'$.

4 Deriving Obligations for an Equivalence Proof

As in [14] we attempt to prove that a set of triples (σ, σ', R) , namely \mathcal{X} , is included in \equiv . In this proof we encounter sub-cases where \mathcal{X} must satisfy specific conditions in order for the proof to go through. These conditions will become the necessary conditions of our bisimulations in the next section.

For \mathcal{X} to be included in \equiv , one must prove that for all $(\sigma, \sigma', R) \in \mathcal{X}$:

1. $\sigma, \sigma' \vdash_{\text{wf}} R$,
2. if $(\rho, \rho') \in R^\epsilon$ and a is an expression such that $FV(a) \subseteq \text{Dom}(\rho)$ then:

$$\sigma; \rho \vdash a \Downarrow \quad \text{iff} \quad \sigma'; \rho' \vdash a \Downarrow$$

The proof of the first condition is straightforward. It suffices to inspect that the environments of all stored methods and the values in R refer to locations in the domains of σ and σ' . This condition will become the first condition of our bisimulations.

Proving the second part requires an induction on the height of the derivations of $\sigma; \rho \vdash a \Downarrow$ in the forward direction, and $\sigma'; \rho' \vdash a \Downarrow$ in the reverse direction. We show only the forward direction; the other is symmetric.

To carry out the induction we strengthen the induction hypothesis by relating the final configurations under some state of \mathcal{X} .

$$\begin{aligned} IH(k) = & \forall \sigma, \sigma', R, \rho, \rho', a, v, \sigma_1. \\ & ((\sigma, \sigma', R) \in \mathcal{X}) \wedge ((\rho, \rho') \in R^\epsilon) \wedge (FV(a) \subseteq \text{Dom}(\rho)) \\ & \wedge (\sigma; \rho \vdash a \Downarrow^{<k} v; \sigma_1) \\ \implies & \exists v', \sigma'_1, Q : (\sigma'; \rho' \vdash a \Downarrow v'; \sigma'_1) \\ & \wedge ((v, v') \in Q) \wedge (Q \supseteq R) \\ & \wedge ((\sigma_1, \sigma'_1, Q) \in \mathcal{X}) \end{aligned} \tag{1}$$

Using (1) as the induction hypothesis we proceed by induction on k , considering the cases of a . Most of the cases follow immediately by the induction hypothesis; the rest will become the proof obligations for \mathcal{X} , and furthermore the conditions in the definition of bisimulations.

To demonstrate this we consider the case of method invocation ($a = a_1.l$). We assume (1) for k and we prove it for $k+1$.

Let $(\sigma, \sigma', R) \in \mathcal{X}$, $(\rho, \rho') \in R^\epsilon$, $FV(a) = FV(a_1) \subseteq \text{Dom}(\rho)$, and $\sigma; \rho \vdash a_1.l_j \Downarrow^{<k+1} v; \sigma_2$. We have to show that:

$$\begin{aligned} \exists v', \sigma'_1, Q : & (\sigma'; \rho' \vdash a_1.l \Downarrow v'; \sigma'_1) \\ & \wedge ((v, v') \in Q) \wedge (Q \supseteq R) \\ & \wedge ((\sigma_1, \sigma'_1, Q) \in \mathcal{X}) \end{aligned}$$

The RED SELECT evaluation rule for the left-hand side gives:

$$\frac{\sigma; \rho \vdash a_1 \Downarrow^{<k} [\overline{l = \iota}]; \sigma_1 \quad l_j \in \{\bar{l}\} \quad \sigma_1(\iota_j) = \langle \varsigma(x)b_j, \rho_1 \rangle \quad x \notin \text{Dom}(\rho_1) \quad \sigma_1; (\rho_1, x \mapsto [\overline{l = \iota}]) \vdash b_j \Downarrow^{<k} v; \sigma_2}{\sigma; \rho \vdash a_1.l_j \Downarrow^{<k+1} v; \sigma_2}$$

To show that $\sigma'; \rho' \vdash a_1.l \Downarrow v'; \sigma'_1$ we must establish the premises of RED SELECT for the right-hand side as well.

By the induction hypothesis at a_1 we get that there exist $\sigma'_1, \overline{l'}, \iota'$, Q , such that:

$$\sigma'; \rho' \vdash a_1 \Downarrow [\overline{l' = \iota'}]; \sigma'_1, \quad Q \supseteq R, \quad ([\overline{l = \iota}], [\overline{l' = \iota'}]) \in Q, \quad (\sigma_1, \sigma'_1, Q) \in \mathcal{X} \tag{2}$$

We also need to show that $\{\bar{l}'\} \subseteq \{\bar{l}\}$. This does not follow from the induction hypothesis and therefore we formulate it as a condition on \mathcal{X} :

If $(\sigma, \sigma', R) \in \mathcal{X}$ and $([\bar{l} = \iota], [\bar{l}' = \iota']) \in R$, then $\{\bar{l}\} \subseteq \{\bar{l}'\}$.

By Lemma 1 and the first formula of (2) we get that $\sigma'_1; (x \mapsto [\bar{l}' = \iota']) \vdash_{\text{wf}} \diamond$, and therefore there exist b'_j, ρ'_j , such that $\sigma'_1(\iota'_j) = \langle \varsigma(x)b'_j, \rho'_j \rangle$.

Finally, to show that the right-hand side terminates and prove the inductive step, we require the following condition to hold for \mathcal{X} :

If $(\sigma, \sigma', R) \in \mathcal{X}$ and $([\bar{l} = \iota], [\bar{l}' = \iota']) \in R$, then for all $\iota_j \in \{\bar{l}\}$, with $\sigma(\iota_j) = \langle \varsigma(x)b, \rho \rangle$ and $\sigma'(\iota'_j) = \langle \varsigma(x)b', \rho' \rangle$, the following must be true:

If $IH(k)$ holds, then:

$$\begin{aligned} & \sigma; (\rho, x \mapsto [\bar{l} = \iota]) \vdash b \Downarrow^{<k} v; \sigma_1 \\ \implies & \exists v', \sigma'_1, Q : \begin{aligned} & \sigma'; (\rho', x \mapsto [\bar{l}' = \iota']) \vdash b' \Downarrow v'; \sigma'_1 \\ & \wedge ((v, v') \in Q) \wedge (Q \supseteq R) \\ & \wedge ((\sigma_1, \sigma'_1, Q) \in \mathcal{X}) \end{aligned} \end{aligned}$$

With a similar treatment for the rest of the cases we discover all the proof obligations of \mathcal{X} , which we will formulate as the conditions of bisimulations in the following section.

5 Small Bisimulations

We first define some new notation to make the transfer of the induction hypothesis from the direct proof into the definition of bisimulations easier.

Definition 8 (k -Approximation). We write $(\sigma, \sigma', R) \vdash_{\mathcal{X}} a|_{\rho} \sqsubseteq_k a'|_{\rho'}$ to mean:

$$\begin{aligned} & \forall v, \sigma_1. \sigma; \rho \vdash a \Downarrow^{<k} v; \sigma_1 \\ \implies & \exists v', \sigma'_1, Q : \begin{aligned} & \sigma'; \rho' \vdash a' \Downarrow v'; \sigma'_1 \\ & \wedge ((v, v') \in Q) \wedge (Q \supseteq R) \\ & \wedge ((\sigma_1, \sigma'_1, Q) \in \mathcal{X}) \end{aligned} \end{aligned}$$

Similarly, in the other direction, we write $(\sigma, \sigma', R) \vdash_{\mathcal{X}} a|_{\rho} \sqsupseteq_k a'|_{\rho'}$ to mean:

$$\begin{aligned} & \forall v', \sigma'_1. \sigma'; \rho' \vdash a' \Downarrow^{<k} v'; \sigma'_1 \\ \implies & \exists v, \sigma_1, Q : \begin{aligned} & \sigma; \rho \vdash a \Downarrow v; \sigma_1 \\ & \wedge ((v, v') \in Q) \wedge (Q \supseteq R) \\ & \wedge ((\sigma_1, \sigma'_1, Q) \in \mathcal{X}) \end{aligned} \end{aligned}$$

Note that the two directions are not converse, since they both contain $(\sigma_1, \sigma'_1, Q) \in \mathcal{X}$ and $(v, v') \in Q$. Similarly we give two versions of the full induction hypothesis, one for each direction:

Definition 9 (Induction Hypotheses).

$$\begin{array}{ll}
IH_{\mathcal{X}}^L(k) \triangleq \forall (\sigma, \sigma', R) \in \mathcal{X}. & IH_{\mathcal{X}}^R(k) \triangleq \forall (\sigma, \sigma', R) \in \mathcal{X}. \\
\quad \forall (\rho, \rho') \in R^\epsilon. & \quad \forall (\rho, \rho') \in R^\epsilon. \\
\quad \forall a: FV(a) \subseteq Dom(\rho). & \quad \forall a: FV(a) \subseteq Dom(\rho). \\
\quad (\sigma, \sigma', R) \vdash_{\mathcal{X}} a|_{\rho} \sqsubseteq_k a|_{\rho'} & \quad (\sigma, \sigma', R) \vdash_{\mathcal{X}} a|_{\rho} \sqsupseteq_k a|_{\rho'}
\end{array}$$

The induction hypotheses involve the k -approximation of the same terms a under related stores and environments. The definition of bisimulations follows:

Definition 10 (Bisimulation). *A set \mathcal{X} of states (σ, σ', R) is called a bisimulation if and only if, for any $(\sigma, \sigma', R) \in \mathcal{X}$, the following conditions are satisfied:*

1. $\sigma, \sigma' \vdash_{\text{wf}} R$
2. For all $\bar{\iota} \notin Dom(\sigma)$, $\bar{\iota}' \notin Dom(\sigma')$, labels \bar{l} , $(\rho, \rho') \in R^\epsilon$, $x \notin Dom(\rho)$, and expressions b such that $FV(b) \subseteq \rho \cup \{x\}$ there exists $Q \supseteq R$ with

$$([\bar{l} = \bar{\iota}], [\bar{l} = \bar{\iota}']) \in Q \quad \text{and} \quad ((\sigma, \bar{\iota} \mapsto \langle \varsigma(x)b, \rho \rangle), (\sigma', \bar{\iota}' \mapsto \langle \varsigma(x)b, \rho' \rangle), Q) \in \mathcal{X}$$

3. If $([\bar{l} = \bar{\iota}], [\bar{l}' = \bar{\iota}']) \in R$, then $\{\bar{l}\} = \{\bar{l}'\}$.
4. If $([\bar{l} = \bar{\iota}], [\bar{l} = \bar{\iota}']) \in R$, then, for all $l_j \in \{\bar{l}\}$, $(\rho, \rho') \in R^\epsilon$, $x \notin Dom(\rho)$, and expressions b such that $FV(b) \subseteq Dom(\rho) \cup \{x\}$, there exists $Q \supseteq R$, with

$$((\sigma, \iota_j \leftarrow \langle \varsigma(x)b, \rho \rangle), (\sigma', \iota'_j \leftarrow \langle \varsigma(x)b, \rho' \rangle), Q) \in \mathcal{X}$$

5. If $([\bar{l} = \bar{\iota}], [\bar{l} = \bar{\iota}']) \in R$, then, for all $\bar{\iota}_1 \notin Dom(\sigma)$, $\bar{\iota}'_1 \notin Dom(\sigma')$, there exists $Q \supseteq R$ with

$$([\bar{l} = \bar{\iota}_1], [\bar{l} = \bar{\iota}'_1]) \in Q \quad \text{and} \quad ((\sigma, \bar{\iota}_1 \mapsto \sigma(\bar{\iota})), (\sigma', \bar{\iota}'_1 \mapsto \sigma'(\bar{\iota}')), Q) \in \mathcal{X}$$

6. If $([\bar{l} = \bar{\iota}], [\bar{l} = \bar{\iota}']) \in R$, then, for any $l_j \in \{\bar{l}\}$ with $\sigma(\iota_j) = \langle \varsigma(x)b, \rho \rangle$ and $\sigma'(\iota'_j) = \langle \varsigma(x)b', \rho' \rangle$, let $\rho_1 = (\rho, x \mapsto [\bar{l} = \bar{\iota}])$ and $\rho'_1 = (\rho', x \mapsto [\bar{l} = \bar{\iota}'])$, we have:

$$\begin{array}{l}
IH_{\mathcal{X}}^L(k) \implies (\sigma, \sigma', R) \vdash_{\mathcal{X}} b|_{\rho_1} \sqsubseteq_k b'|_{\rho'_1} \\
IH_{\mathcal{X}}^R(k) \implies (\sigma, \sigma', R) \vdash_{\mathcal{X}} b|_{\rho_1} \sqsupseteq_k b'|_{\rho'_1}
\end{array}$$

The first condition of the definition allows only well-formed states in the bisimulations. The second condition addresses the proof obligation for the case of evaluating an object. Conditions 3 and 6 are the proof obligations for the case of method invocation, as explained in Section 4. Conditions 4 and 5 address the proof obligations for the cases of method update and cloning, respectively. The case of **let** follows immediately from the induction hypothesis and generates no condition for the bisimulations.

Next we show that our method is sound and complete, and that a maximal bisimulation exists.

Theorem 2 (Completeness). *Contextual equivalence is a bisimulation.*

Theorem 3 (Soundness). *Any bisimulation \mathcal{X} is included in Contextual Equivalence.*

Proof. This proof recapitulates the derivation of Section 4.

Theorem 4 (Bisimilarity). *A maximal bisimulation, called Bisimilarity (\sim), exists and coincides with Contextual Equivalence.*

Proof. By the definition of (\equiv) and Theorems 2 and 3, we get that (\equiv) is itself the largest sound bisimulation. Thus bisimilarity coincides with contextual equivalence.

6 Up-to Store Closure

Conditions 2 and 4 of Definition 10 close bisimulations under any possible extensions of the store with new object methods and any possible update of existing methods. Writing down sets to satisfy these conditions can become cumbersome.

To eliminate the need of satisfying these conditions when one writes a bisimulation in closed form, we introduce an *up-to store* closure operator on sets. Then we give a new set of necessary conditions on the (smaller) sets, such that their up-to store closure is a bisimulation.

Definition 11 (Up-to Store Extension of States). *The state $(\sigma_1, \sigma'_1, R_1)$ is an up to store extension of state $(\sigma_0, \sigma'_0, R_0)$, written $(\sigma_0, \sigma'_0, R_0) \sqsubseteq (\sigma_1, \sigma'_1, R_1)$, iff it satisfies the rules of Figure 4.*

The second rule of Figure 4 states that extending a bisimulation state with arbitrary new pairs of related objects, and also extending the stores accordingly to keep their methods, is a valid up-to store extension. The third rule states that updating some known locations with related methods is also a valid up-to store extension.

$$\begin{array}{c}
 \hline
 \overline{(\sigma, \sigma', R) \sqsubseteq (\sigma, \sigma', R)} \\
 \\
 \frac{(\sigma, \sigma', R) \sqsubseteq (\sigma_1, \sigma'_1, R_1) \quad \overline{\iota_e, \iota'_e \text{ fresh}} \quad (\rho, \rho') \in R^e}{(\sigma, \sigma', R) \sqsubseteq ((\sigma_1, \iota_e \mapsto \langle \zeta(x)b, \rho \rangle), (\sigma'_1, \iota'_e \mapsto \langle \zeta(x)b, \rho' \rangle), R_1 \cup \{([l = \iota_e], [l = \iota'_e])\})} \\
 \\
 \frac{\overline{(\sigma, \sigma', R) \sqsubseteq (\sigma_1, \sigma'_1, R_1)} \quad ([l = \iota], [l = \iota']) \in R_1 \quad (\rho, \rho') \in R^e \quad \iota_u \in \{\bar{\iota}\} \quad \iota'_u \in \{\bar{\iota}'\}}{(\sigma, \sigma', R) \sqsubseteq ((\sigma_1. \iota_u \leftarrow \langle \zeta(x)b, \rho \rangle), (\sigma'_1. \iota'_u \leftarrow \langle \zeta(x)b, \rho' \rangle), R_1)} \\
 \hline
 \end{array}$$

Fig. 4. Up-to Store Extension of States

We now give the definition of an up-to store closure operator on sets, and the necessary conditions for a set \mathcal{X} such that its up-to store closure is a bisimulation:

Definition 12 (Up-to Store Closure of Sets).

$$\mathcal{X}^* = \{(\sigma, \sigma', R) \mid \exists (\sigma_0, \sigma'_0, R_0) \in \mathcal{X} : (\sigma_0, \sigma'_0, R_0) \sqsubseteq (\sigma, \sigma', R)\}$$

Theorem 5. \mathcal{X}^* is a bisimulation if for all $(\sigma, \sigma', R) \in \mathcal{X}$, we have:

1. $\sigma, \sigma' \vdash_{\text{wf}} R$
2. If $([\overline{l = \iota}], [\overline{l' = \iota'}]) \in R$, then $\{\overline{l}\} = \{\overline{l'}\}$.
3. If $([\overline{l = \iota}], [\overline{l' = \iota'}]) \in R$, then, for all $\overline{\iota_1} \notin \text{Dom}(\sigma)$, $\overline{\iota'_1} \notin \text{Dom}(\sigma')$, there exists $Q \supseteq R$ with

$$([\overline{l = \iota_1}], [\overline{l' = \iota'_1}]) \in Q \quad \text{and} \quad ((\sigma, \overline{\iota_1 \mapsto \sigma(\iota)}), (\sigma', \overline{\iota'_1 \mapsto \sigma'(\iota')}), Q) \in \mathcal{X}^*$$

4. If $([\overline{l = \iota}], [\overline{l' = \iota'}]) \in R$, then, for any $l_j \in \{\overline{l}\}$ with $\sigma(\iota_j) = \langle \varsigma(x)b, \rho \rangle$ and $\sigma'(\iota'_j) = \langle \varsigma(x')b', \rho' \rangle$, and for all $(\sigma_1, \sigma'_1, R_1)$ with $(\sigma, \sigma', R) \sqsubseteq (\sigma_1, \sigma'_1, R_1)$, let $\rho_1 = (\rho, x \mapsto [\overline{l = \iota}])$ and $\rho'_1 = (\rho', x \mapsto [\overline{l' = \iota'}])$, we have:

$$\begin{aligned} IH_{\mathcal{X}^*}^L(k) &\Longrightarrow (\sigma_1, \sigma'_1, R_1) \vdash_{\mathcal{X}^*} b|_{\rho_1} \sqsubseteq_k b'|_{\rho'_1} \\ IH_{\mathcal{X}^*}^R(k) &\Longrightarrow (\sigma_1, \sigma'_1, R_1) \vdash_{\mathcal{X}^*} b|_{\rho_1} \sqsupseteq_k b'|_{\rho'_1} \end{aligned}$$

Proof. Straightforward by inspecting that \mathcal{X}^* satisfies the conditions of Definition 10.

7 Example

Using our bisimulations and Theorem 5 we were able to prove the equivalences in [7] and the untyped, imperative equivalent of the example in [8]. Due to space limitations, we omit these examples here. Instead we prove equivalence in a more interesting example that demonstrates the ability of our method to deal with hidden imperative fields, different store manipulation, and higher-order methods.

Consider the two classes shown in Figure 1. Objects of these classes are indistinguishable in any context (in the absence of reflection). To prove this we encode the objects in `imp ς` . We simplify the encoding by extending `imp ς` in the usual way with integers, arithmetic operators, and a conditional statement, and we encode methods containing λ -abstractions as follows:

$$[\cdots, f = \varsigma(s)\lambda y.e, \cdots] \triangleq [\arg = \varsigma(s)s.\arg, \cdots, f = \varsigma(s)e[s.\arg/y], \cdots]$$

The context passes an argument to the body of method f by updating the *arg* label and then selecting f . Because the *arg* label may be updated with an arbitrary method, every argument is potentially arbitrarily complicated. The induction hypothesis in the last condition of Definition 10 and Theorem 5 is crucial for reasoning about these arguments.

The objects of the **Cell** classes are encoded as:

$$\begin{aligned} M &= \text{let } o = [y = \varsigma(-)0] \\ &\quad \text{in } [arg = \varsigma(s)s.arg, set = set_M, get = get_M] \\ N &= \text{let } o = [y_1 = \varsigma(-)0, y_2 = \varsigma(-)0, c = \varsigma(-)0] \\ &\quad \text{in } [arg = \varsigma(s)s.arg, set = set_N, get = get_N] \end{aligned}$$

where:

$$\begin{aligned} set_M &\triangleq \varsigma(s)\text{let } z = s.arg \\ &\quad \text{in } o.y \Leftarrow \varsigma(-)z \\ set_N &\triangleq \varsigma(s)\text{let } n = o.c + 1 \\ &\quad \quad z = s.arg \\ &\quad \text{in } (o.c \Leftarrow \varsigma(-)n; \\ &\quad \quad o.y_1 \Leftarrow \varsigma(-)z; \\ &\quad \quad o.y_2 \Leftarrow \varsigma(-)z) \\ get_M &\triangleq \varsigma(-)o.y \\ get_N &\triangleq \varsigma(-)\text{let } x = \text{even?}(o.c) \\ &\quad \text{in if } x \text{ then } o.y_1 \text{ else } o.y_2 \end{aligned}$$

By Theorem 1, to prove $M \equiv_{\text{std}} N$ it is sufficient to show that $M \equiv N$. Thus we have to construct a bisimulation that contains $((\sigma, \iota_0 \mapsto \langle \varsigma(-)M, \rho \rangle), (\sigma, \iota_0 \mapsto \langle \varsigma(-)N, \rho \rangle), R)$, for all σ, ρ , and for some ι_0, l_0 and R , such that $([l_0 = \iota_0], [l_0 = \iota_0]) \in R$.

To do this we define the parameterized value relation:

$$\begin{aligned} &\overline{Q(\iota_s, \iota_g, \iota_a, \iota'_s, \iota'_g, \iota'_a, \iota_0)} \\ &= \overline{\{([\arg = \iota_a, set = \iota_s, get = \iota_g], [\arg = \iota'_a, set = \iota'_s, get = \iota'_g]), \\ &\quad ([l_0 = \iota_0], [l_0 = \iota_0])\}} \end{aligned}$$

the parameterized stores:

$$\begin{aligned} &\overline{\sigma(\iota_s, \iota_g, \iota_a, \iota_y, \rho_M, \rho_1, \iota_0, \rho_0)} \\ &= \overline{(\iota_0 \mapsto \langle \varsigma(-)M, \rho_0 \rangle, \iota_a \mapsto \langle \varsigma(s)s.arg, \rho_M \rangle, \iota_s \mapsto \langle set_M, \rho_M \rangle, \iota_g \mapsto \langle get_M, \rho_M \rangle, \\ &\quad \iota_y \mapsto \langle \varsigma(-)x, \rho_1 \rangle)} \\ &\overline{\sigma'(\iota'_s, \iota'_g, \iota'_a, \iota_{y_1}, \iota_{y_2}, \iota_c, \rho_N, \rho'_1, n, \iota'_0, \rho'_0)} \\ &= \overline{(\iota_0 \mapsto \langle \varsigma(-)N, \rho'_0 \rangle, \iota'_a \mapsto \langle \varsigma(s)s.arg, \rho_N \rangle, \iota'_s \mapsto \langle set_N, \rho_N \rangle, \iota'_g \mapsto \langle get_N, \rho_N \rangle, \\ &\quad \iota_{y_1} \mapsto \langle \varsigma(-)x, \rho'_1 \rangle, \iota_{y_2} \mapsto \langle \varsigma(-)x, \rho'_1 \rangle, \iota_c \mapsto \langle \varsigma(-)n, \rho'_1 \rangle)} \end{aligned}$$

and the set:

$$\begin{aligned} \mathcal{X} &= \{(\sigma, \sigma', R) \mid \overline{\exists \iota_s, \iota_g, \iota_a, \iota'_s, \iota'_g, \iota'_a, \iota_y, \iota_{y_1}, \iota_{y_2}, \iota_c, \rho_M, \rho_N, \rho_1, \rho'_1, n, \iota_0, \iota'_0, \rho_0, \rho'_0} \\ &\quad : R = \overline{Q(\iota_s, \iota_g, \iota_a, \iota'_s, \iota'_g, \iota'_a, \iota_0)} \\ &\quad \wedge \sigma = \overline{\sigma(\iota_s, \iota_g, \iota_a, \iota_y, \rho_M, \rho_1, \iota_0, \rho_0)} \\ &\quad \wedge \sigma' = \overline{\sigma'(\iota'_s, \iota'_g, \iota'_a, \iota_{y_1}, \iota_{y_2}, \iota_c, \rho_N, \rho'_1, n, \iota'_0, \rho'_0)} \\ &\quad \wedge \overline{((\rho_0, \rho'_0) \in R^\epsilon) \wedge ((\rho_1, \rho'_1) \in R^\epsilon)} \\ &\quad \wedge \overline{\rho_M = (\rho_1, y \mapsto \iota_y)} \\ &\quad \wedge \overline{\rho_N = (\rho'_1, y_1 \mapsto \iota_{y_1}, y_2 \mapsto \iota_{y_2}, c \mapsto \iota_c)}\} \end{aligned}$$

We have to show that \mathcal{X} satisfies the conditions of Theorem 5, and thus \mathcal{X}^* is a bisimulation. It is easy to check that conditions 1, 2, and 3 are satisfied. It remains to prove Condition 4 for $(\iota_{0_k}, \iota'_{0_k})$, $(\iota_{s_k}, \iota'_{s_k})$, and $(\iota_{g_k}, \iota'_{g_k})$, for any k .

We consider $(\iota_{s_k}, \iota'_{s_k})$. Let $(\sigma_1, \sigma'_1, R_1) \in \mathcal{X}^*$, and:

$$([arg = \iota_{a_k}, set = \iota_{s_k}, get = \iota_{g_k}], [arg = \iota'_{a_k}, set = \iota'_{s_k}, get = \iota'_{g_k}]) \in R_1$$

By the definition of up-to store extension for states, we observe that some of the labels of the above objects may have been updated by the context. Thus we have two cases:

Case 1. The labels ι_{s_k} and ι'_{s_k} have been updated: $\sigma_1(\iota_{s_k}) = \langle \varsigma(x)b, \rho' \rangle$, $\sigma'_1(\iota'_{s_k}) = \langle \varsigma(x)b, \rho' \rangle$, $(\rho, \rho') \in R^\epsilon$.

We have to show that if $\rho_1 = (\rho, x \mapsto [arg = \iota_{a_k}, set = \iota_{s_k}, get = \iota_{g_k}])$ and $\rho'_1 = (\rho', x \mapsto [arg = \iota'_{a_k}, set = \iota'_{s_k}, get = \iota'_{g_k}])$, then:

$$\begin{aligned} IH_{\mathcal{X}^*}^L(k) &\Longrightarrow (\sigma_1, \sigma'_1, R_1) \vdash_{\mathcal{X}^*} b|_{\rho_1} \sqsubseteq_k b|_{\rho'_1} \\ IH_{\mathcal{X}^*}^R(k) &\Longrightarrow (\sigma_1, \sigma'_1, R_1) \vdash_{\mathcal{X}^*} b|_{\rho_1} \sqsupseteq_k b|_{\rho'_1} \end{aligned}$$

But these are immediately satisfied by $IH_{\mathcal{X}^*}^L(k)$ and $IH_{\mathcal{X}^*}^R(k)$, since $(\rho_1, \rho'_1) \in R^\epsilon$.

Case 2. The labels ι_{s_k} and ι'_{s_k} have not been updated: $\sigma_1(\iota_{s_k}) = \langle \varsigma(_)set_M, \rho_{M_k} \rangle$, $\sigma'_1(\iota'_{s_k}) = \langle \varsigma(_)set_N, \rho_{N_k} \rangle$. We will show only the forward direction:

$$IH_{\mathcal{X}^*}^L(k) \Longrightarrow (\sigma_1, \sigma'_1, R_1) \vdash_{\mathcal{X}^*} set_M|_{\rho_{M_k}} \sqsubseteq_k set_N|_{\rho_{N_k}}$$

Let:

$$\sigma_1; \rho_{M_k} \vdash set_M \Downarrow^{<k} [arg = \iota_{a_k}, set = \iota_{s_k}, get = \iota_{g_k}]; \sigma_2$$

This implies that $(\sigma_1; \rho_{M_k} \vdash s.arg \Downarrow^{<k-1} v; \sigma_3)$, $\sigma_2 = (\sigma_3.\iota_{y_k} \leftarrow \langle \varsigma(_)z, \rho_1 \rangle)$, and $\rho_1(z) = v$. From $IH_{\mathcal{X}^*}^L(k)$ we get:

$$\begin{aligned} \exists v', \sigma'_3, R_3 : & \sigma'_1; \rho_{N_k} \vdash s.arg \Downarrow v'; \sigma'_3 \\ & \wedge ((v, v') \in R_3) \wedge (R_3 \supseteq R_1) \\ & \wedge ((\sigma_3, \sigma'_3, R_3) \in \mathcal{X}^*) \end{aligned}$$

thus:

$$\begin{aligned} & \sigma'_1; \rho_{N_k} \vdash set_N \Downarrow [arg = \iota'_{a_k}, set = \iota'_{s_k}, get = \iota'_{g_k}]; \sigma'_2 \\ & \sigma'_2 = (\sigma'_3.\iota_{y_{1_k}} \leftarrow \langle \varsigma(_)z, \rho'_1 \rangle, \iota_{y_{2_k}} \leftarrow \langle \varsigma(_)z, \rho'_1 \rangle, \iota_{c_k} \leftarrow \langle \varsigma(_)m, \rho'_1 \rangle) \end{aligned}$$

and by the definition of \mathcal{X} and \mathcal{X}^* we get that there exists $R_2 \supseteq R_1$ such that $(\sigma_2, \sigma'_2, R_2) \in \mathcal{X}^*$.

Similarly we prove Condition 4 for $(\iota_{g_k}, \iota'_{g_k})$, and $(\iota_{0_k}, \iota'_{0_k})$.

8 Related Work

Gordon, Hankin and Lassen in [7] gave an *operational equivalence* for the same calculus that we consider here, and they showed that it coincides with contextual equivalence. This equivalence is a CIU theorem for this language. Their relation does not provide a technique to prove difficult equivalences. For example, proving the `Cell` example with their CIU theorem would require an induction over all reduction contexts. Such an induction is not obvious because the stores and the environments of the two sides may change in a different way in some of the cases. Such a proof would be at least as difficult as the induction discussed in Section 4. On the other hand, using our bisimulations we were able to prove equivalence for all of their examples, as well as more complex ones, by a constructive proof.

Gordon and Rees in [8] proved for one of the stateless, typed, object calculi of Abadi and Cardelli that bisimilarity coincides with contextual equivalence. This was the first study of contextual equivalence in an object calculus. Their method was quite different than ours, being closer to the original operational bisimulations of Abramsky [2] and Howe's proof of congruence [11]. Reasoning about higher-order programs is easier with our method because of the use of the induction hypotheses in the definition of bisimulation.

A different approach to studying equivalence in object calculi is by translating them to π -calculus. Kleist and Sangiorgi have done this in [13] for the typed version of the calculus we study here, and Sangiorgi in [24] for the functional version of this calculus. Similar work has been done for parallel object oriented languages (e.g. [12, 15, 27]). This approach is in essence denotational and all of these translations were not fully abstract, so none of these methods is complete.

9 Conclusions and Future Work

We have presented a method of deriving bisimulations for the untyped, imperative object calculus of Abadi and Cardelli. To our knowledge this is the first sound and complete method that uses bisimulations to prove equivalence for this calculus, and successfully handles complex examples.

We hope to use our method to investigate contextual equivalence in more realistic imperative object languages [4, 6]. We also plan to investigate better ways to express stylized bisimulations, like those in the example of Section 7.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, Berlin, Heidelberg, and New York, 1996.
2. Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.
3. Nick Benton and Benjamin Leperchey. Relational reasoning in a nominal semantics for storage. In *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2005.

4. Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, April 2003.
5. Yuxin Deng and Davide Sangiorgi. Towards an algebraic theory of typed mobile processes. In *ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 445–456. Springer-Verlag, 2004.
6. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, London, UK, 1999. Springer-Verlag.
7. Andrew D. Gordon, Paul D. Hankin, and Søren B. Lassen. Compilation and equivalence of imperative objects. In *Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 74–87, London, UK, 1997. Springer-Verlag.
8. Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 386–395, New York, NY, USA, 1996. ACM Press.
9. Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *ICALP*, pages 299–309, 1980.
10. Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.
11. Douglas J. Howe. Equality in lazy computation systems. In *Proc. 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.
12. Cliff B. Jones. A pi-calculus semantics for an object-based design notation. In Eike Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1993.
13. Josva Kleist and Davide Sangiorgi. Imperative objects as mobile processes. *Sci. Comput. Program.*, 44(3):293–342, 2002.
14. Vasileios Koutavas and Mitchell Wand. Smaller bisimulations for reasoning about higher-order imperative programs. In *Proceedings 33rd Annual ACM Symposium on Principles of Programming Languages*, New York, NY, USA, 2006. ACM Press.
15. Xinxin Liu and David Walker. Partial confluence of processes and systems of objects. *Theor. Comput. Sci.*, 206(1-2):127–162, 1998.
16. Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.
17. Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Proceedings 15th Annual ACM Symposium on Principles of Programming Languages*, pages 191–203, 1988.
18. Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, 1976. Also Wiley, New York.
19. James H. Morris, Jr. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, Cambridge, MA, 1968.
20. Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In Andrew Gordon and Andrew Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Publications of the Newton Institute, Cambridge University Press, 1998.
21. D. Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 155:39–83, 1996.
22. D. Sangiorgi. On the bisimulation proof method. *Mathematical. Structures in Comp. Sci.*, 8(5):447–479, 1998.

23. D. Sangiorgi and R. Milner. The problem of “Weak Bisimulation up to”. In W.R. Cleveland, editor, *Proc. CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 1992.
24. Davide Sangiorgi. An interpretation of typed objects into typed π -calculus. *Inf. Comput.*, 143(1):34–73, 1998.
25. Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *Proceedings 31st Annual ACM Symposium on Principles of Programming Languages*, pages 161–172, New York, NY, USA, 2004. ACM Press.
26. Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. In *Proceedings 32nd Annual ACM Symposium on Principles of Programming Languages*, pages 63–74, New York, NY, USA, 2005. ACM Press.
27. David Walker. Objects in the π -calculus. *Inf. Comput.*, 116(2):253–271, 1995.

A Typed Assembly Language for Confidentiality

Dachuan Yu and Nayeem Islam

DoCoMo Communications Laboratories, USA
{yu, nayeem}@docomolabs-usa.com

Abstract. Language-based information-flow analysis is promising in protecting data confidentiality. Although much work has been carried out in this area, relatively little has been done for assembly code. Source-level techniques do not easily generalize to assembly code, because assembly code does not readily present certain abstraction about the program structure that is crucial to information-flow analysis. Nonetheless, low-level information-flow analysis is desirable, because it yields a small trusted computing base. Furthermore, many (untrusted) applications are distributed in native code; their verification should not be overlooked.

We present a simple yet effective solution for this problem. Our observation is that the missing abstraction in assembly code can be restored using annotations. Following the philosophy of certifying compilation, these annotations are generated by a compiler, used for static validation, and erased before execution. In particular, we propose a type system for low-level information-flow analysis. Our system is compatible with Typed Assembly Language, and models key features including a call stack, memory tuples and first-class code pointers. A noninterference theorem articulates that well-typed programs respect confidentiality.

1 Introduction

With the growing reliance on networked information systems, the protection of confidential data becomes increasingly important. The problem is especially subtle for a computing system which both manipulates sensitive data and requires access to public information channels. Simple policies that restrict the access to either the sensitive data or the public channels (or a combination) often prove too restrictive. A more desirable policy is that no information about the sensitive data can be inferred from observing the public channels, even though a computing system is granted access to both. Such a regulation of the flow of information is often referred to as *information flow*, and the policy that sensitive data should not affect public data is often called *noninterference*.

Whereas it is relatively easy to detect and prevent naive violations that directly give out sensitive data, it is much more difficult to prevent applications from sending out information that is sophisticatedly encoded. Conventional security mechanisms such as access control, firewalls, encryption and anti-virus fall short on enforcing the noninterference policy [14]. On the one hand, noninterference posts seemingly conflicting requirements: it allows the access to sensitive information, but restricts the flow of it. On the other hand, the violation of noninterference cannot be observed from monitoring a single execution of the program [16], yet such execution monitoring is the basis of many conventional mechanisms.

In recent years, much effort has been put on enforcing noninterference using techniques based on programming language theory and implementation. These techniques are promising, because they directly inspect or instrument the program code, and hence have the potential of learning all possible run-time behavior of the program. Unfortunately, the vast amount of language-based research on information flow [14] does not address well the problem for assembly code. The challenge there, as we will elaborate later, largely lies in working with the lack of high-level abstractions and managing the extreme flexibility offered by assembly code.

Nonetheless, it is desirable to enforce noninterference directly at a low-level. On the one hand, high-level programs must be translated into low-level code before executed on a real machine; compilation or optimization bugs may invalidate the security guarantee established for the source programs. On the other hand, some applications are distributed (*e.g.*, native code for mobile computation) or even directly written (*e.g.*, core libraries for embedded systems) in assembly; enforcement at a low-level is a must.

This paper presents some important steps of a project tackling information flow at the assembly level. The contributions are:

- We propose a Typed Assembly Language for Confidentiality (TAL_C) and present its proof of noninterference. Our abstract machine is generic and close to real architectures. To reuse existing results on low-level verification, our system is designed to be compatible with Typed Assembly Language (TAL) [11]. It thus approaches a unified framework for conventional type safety and security.
- Our system models key features of an assembly language, including heap, call stack and register file, memory tuples (aliasing), and first-class code pointers (higher-order functions). Because assembly code is often arduous to work with, we present our formal result with a core language supporting the above features for ease of understanding, but also informally discuss other extensions.
- Although desirable to directly verify at an assembly level, it is more practical to develop programs in high-level languages. We briefly discuss a translation from a system of linear continuations [24] to TAL_C . A companion technical report [23] presents a translation from a security-typed imperative source language with first-order procedures to TAL_C , illustrating certifying compilation for noninterference.

This paper does not address covert channels (*e.g.*, termination [19, 1] and timing [21, 2]) or abstract-violation attacks (*e.g.*, cache [3]). Section 2 provides background on language-based approaches for information flow, places our work in the context of existing researches, and points out the extra difficulties for noninterference at an assembly level. An informal overview of our approach is given in Section 3. Section 4 presents the TAL_C system, focusing on core features that illustrate ideas pertinent to information flow. Section 5 helps better understand TAL_C in comparison with work on linear continuations. Orthogonal issues and practical extensions are discussed in Section 6.

2 Background

2.1 Information Flow

The problem of information flow can be abstracted as a program that operates on data of different security levels, *e.g.*, *low* and *high*. Low data are public data that may be

observed by all principals; high data are secret data whose access is restricted. An information-flow policy requires that no information about high inputs can be inferred from observing low outputs. The security levels can also be generalized to a lattice [20].

Such a policy concerns tracking the flow of information inside a target system. Whereas it is easy to detect explicit flows (*e.g.*, through an assignment from a secret h to a public l with $l=h$), it is much harder to detect various forms of implicit flow. For example, the statement $l=0; \text{ if } h \text{ then } l=1$ involves an implicit flow from h to l . At run-time, if the *then* branch is not taken, a conventional security mechanism based on execution monitoring will not detect any violation. However, information about h can indeed be inferred from the result of l .

Instead of observing a single execution, language-based techniques derive assurance about a program's behavior by examining, and possibly instrumenting, the program code. In the above example, the information essentially leaks through the program counter (often referred to as *pc*)—the fact that a branch is taken reflects information about the guard of the conditional. In response, a security-type system typically tags the *pc* with a security label. If the guard of a conditional concerns high data, then the branches are verified under a *pc* with a high security label. Furthermore, assignments to low variables are prohibited under such a high *pc*.

2.2 Related Work

Although there has been much work applying language-based techniques to information flow [14], most of it focused on high-level languages. Many high-level abstractions have been formally studied, including functions [8], exceptions [13], objects [5], and concurrency [18, 1], and practical implementation is within reach [12]. Nonetheless, enforcing information flow at only a high level puts the compiler into the trusted computing base (TCB) [15]. Furthermore, we should not overlook the verification of software distributed (or written) directly in low-level code.

Barthe *et al.* [6] presented a security-type system for a bytecode language and a translation that preserves security types. Their stack-based language is much different from the RISC architecture that we model. More importantly, their verification circumvents a main difficulty—the lack of program structures at a low-level—by introducing a trusted component that computes the dependence regions and postdominators [4] for conditionals. This component is inside the TCB and trusted.

Zdancewic and Myers [24] used linear continuations to enforce noninterference at a low-level. Their language is based on variables and still much different from assembly language. In particular, linear continuations, although useful in enforcing a stack discipline that helps information-flow analysis, are absent from conventional assembly code. Hence further (trusted) compilation to native code is required. Nonetheless, we borrowed some ideas from Zdancewic and Myers, including the handling of memory aliasing and code pointers, although these features are modeled as different constructs in our system. A more thorough discussion of the connection between linear continuations and our solution is given in Section 5, after presenting our system.

Bonelli *et al.* [7] explored the realization of linear continuations in an assembly language SIFTAL. Two new instructions are introduced in correspondence with the operations on linear continuations as proposed by Zdancewic and Myers [24]. These

two instructions enforce structured control flows that are missing from normal assembly code with the help of a continuation stack (this stack is different from the one for function calls). One instruction pushes a linear continuation onto the stack, the other pops a linear continuation off the stack and transfers the control to it. Such a mechanism maintains structured control flow in assembly code, thus helps information-flow analysis. Unfortunately, conventional assembly programming and machine models do not contain such a special continuation stack and the instructions manipulating it.

Recently, Medel *et al.* [9] improved SIFTAL to SIF, using a stack of labels to simplify the above approximation of linear continuations. Unlike SIFTAL, SIF resorts to static type annotations to enforce noninterference, and no longer requires a stack of linear continuations during execution. This is in spirit similar to our solution of TAL_C . However, SIF supports only a minimal set of language features (arithmetic, memory update, branching and direct jumps), and does not address how the type annotations can be produced. In contrast, our language TAL_C further supports code pointers and a call stack. The companion technical report [23] presents a type-preserving translation to TAL_C from a security-typed source language, where the support for procedure calls introduces extra subtleties for noninterference. Section 5 provides more details.

This paper targets RISC-style assembly code. We introduce a type system to verify the unstructured control flow, which in turn helps information-flow analysis. Type annotations are used to recover information about high-level program structures, and no trusted component is required for computing postdominators. This contrasts with the above work on bytecode languages. Furthermore, we do not rely on extra constructs such as linear continuations or a continuation stack. An erasure semantics trivially reduces programs in our language to normal assembly code.

2.3 Assembly Code

There are several challenges for enforcing information flow for assembly code.

First, high-level languages make use of a virtually infinite number of variables, each of which can be assigned a fixed security label. In assembly code, the use of memory cells is similar. However, a finite number of registers are reused for different source-level variables. As a result, one cannot assign a fixed security label to a register.

Second, the control flow of an assembly program is not as structured. The body of a conditional is often not obvious, and generally undecidable, from the program code. Hence the idea of using a security context to prevent implicit flow through conditionals cannot be easily carried out.

Third, assembly languages are very expressive. Aliasing between memory cells are difficult to reason about [17]. The support for first-class code pointers (the reflection of higher-order functions at the assembly level) is very subtle. A code pointer may direct a program to different execution paths, even though no branching instruction is present.

Fourth, since it is not practical to always directly program in an assembly language, a low-level type system must be designed so that the type annotations can be generated automatically, *e.g.*, through certifying compilation. The type system must be at least as expressive as a high-level type system, so that any well-typed source program can be translated into well-typed assembly code.

Finally, it is desirable to achieve an erasure semantics where type annotations have no effect at runtime. A security mechanism can not be generally applied in practice if it incurs too much overhead. Similarly, it is also undesirable to change the programming model for accommodating the verification needs. Such a model change indicates either a trusted compilation process or a different target machine.

3 Our Approach

3.1 Explicit Assignment

An obvious kind of information flow is through assignment. In a high-level language, variables can be “tagged” with security labels; the security-type system prevents label mismatch for assignments. At an assembly level, memory cells can be tagged similarly. When storing into a memory cell, a typing rule ensures that the security label of the source matches that of the target.

Registers need to be regulated differently, because they can be reused for different variables with different security labels (registers cannot be aliased, which makes it safe to update their types). Since variable and liveness information is not available at an assembly level, one can not easily base the enforcement upon that.

In fact, a similar problem arises even for normal type safety. A register in TAL can have different types at different program points. These types are essentially inferred from the computation itself. For instance, in an addition instruction $\text{add } r_d, r_s, r_t$, the register r_d is given the type `int`, because only `int` can be valid here. Similarly, when loading from a memory cell, the target register is given the type of the source memory cell. Adapting such inference for security labels is straightforward. In the addition $\text{add } r_d, r_s, r_t$, the label of r_d is obtained by joining the labels of r_s and r_t , because the result in r_d reflects information from both r_s and r_t . Moving and memory reading instructions are handled similarly.

3.2 Program Structure

A conditional statement in a high-level program can be verified so that both sub-commands respect the security level of the guard expression. Such verification becomes difficult in assembly code, where the “flattened” control flow provides little help in identifying the program structure. A conditional is typically translated into a branching instruction and some code blocks, where the postdominator of the two branches are no longer apparent.

We use annotations to restore the program structure by pointing out the postdominators whenever they are needed. Note that high-level programs provide sufficient information for deciding the postdominators, and these postdominators can always be statically determined. For instance, the end of a conditional command is the postdominator of the two branches. Hence a compiler can generate the annotations automatically based on a securely typed source program. In our system, our postdominator annotation is essentially a static code label paired with a security label.

Since branching instructions ($\text{bnz } r, l$) are the only instructions that could directly result in different execution paths, it would appear that one should enhance branching

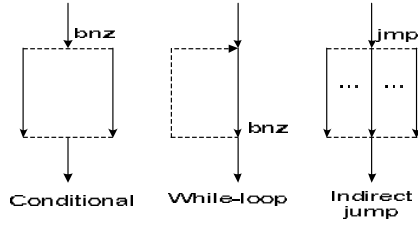


Fig. 1. Flow through program structure

instructions with postdominators. The typing rule then checks both branches under a proper security context that takes into account the guard expression. Such a security context terminates when the postdominator is reached.

Although plausible, this approach is awkward. Figure 1 demonstrates three scenarios. Besides the conditional scenario, branching instructions are also used to implement while-loops, where the postdominator is exactly the beginning of one of the branches. In this case, only the other branch should be checked under a new security context. If we directly annotate the branching instruction, the corresponding typing rule would be “overloaded.” More importantly, an assembly program may contain “implicit branches” where no branching instruction is present. The third scenario illustrates that an indirect jump may lead the program to different paths based on the value of its operand register. A concrete example will appear in Section 3.5.

Inspiration of a better solution lies in high-level security-type systems [23], which typically use a subsumption rule to outline a region of computation where the security level is raised from low to high. The end of the region is exactly a postdominator. Following this, our approach is to mimic the high-level subsumption rule with two low-level *raising* and *lowering* operations that explicitly manipulate the security context and mark the beginning and the end of the secured region.

3.3 Memory Aliasing

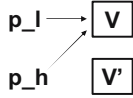
Aliasing of memory cells present another channel for information transfer. In Figure 2, a low pointer p_l and a high pointer p_h are aliases of the same cell. This is useful if a high principal wishes to observe a low computation. The code in this figure may change the aliasing relation based on some high variable h by letting p_h point to another cell. Further modification through p_h may or may not change the value in the original cell. As a result, observing through the low pointer p_l reveals information about the high variable h .

The problem lies in the assignment through the high pointer p_h , because it reveals information about the aliasing relation. The solution, following Zdancewicz and Myers [24], is to tag pointers with two security labels. One is for the pointer itself, and the other is for the data being referenced. Assignments to low data through high pointers are not allowed. This is a conservative approach—all pointers are considered as potential aliases.

```

(* suppose p_h alias p_l *)
if (h=0) then p_h=new cell;
*p_h=1;
(* now *p_l reveals h *)

```


Fig. 2. Flow through aliasing

```

fun f0 () = (l:=0; ())
fun f1 () = (l:=1; ())
let f = (if h then f1 else f0) in f()

```

Fig. 3. Flow through code pointer

3.4 Code Pointers

Code pointers further complicate information flow. Figure 3 shows a piece of functional code where f represents different functions based on a high variable h . In its reflection at an assembly level, different code blocks will be executed based on the value of h . Naturally, f contains sensitive information and should be labeled high. However, the actual functions f_0 and f_1 can only be executed under a low context, because they modify a low variable l . In this case, the invocation to f should be prohibited.

In our system, similar to data pointers, code pointers are also given two security labels. The typing rules ensure that no low function is called through a high code pointer.

3.5 Security Context Coercion

Finally, Figure 4 shows a piece of code where a mutable code pointer complicates the flow analysis. Functions f_0 and f_1 only modify high data. A reference cell f is assigned different code pointers within a high conditional. Later, f is dereferenced and invoked in a low context.

```

fun f0 () = (h' := 0; ())
fun f1 () = (h' := 1; ()) ...
if h then f := f1 else f := f0;
l := 1; !f(); l := l * 2; ...

```

Fig. 4. Context coercion without branching

This code is safe with respect to information flow. At a high level, a subsumption rule allows calling the high function $!f()$ in a low context. However, in its assembly counterparts, both the calling to f and the returning from f are implemented as indirect jumps. The calling sequence transfers the control from a low context to a high context, whereas the returning sequence does the opposite. Since the function invocation is no longer atomic at an assembly level, one cannot directly devise a subsumption rule. Furthermore, there is no explicit branching instruction present when f is dereferenced and invoked (the third scenario of Figure 1).

In our system, the raising and lowering operations explicitly mark the boundary of the subsumption rule. During certifying compilation, the source-level typing and program structure provide sufficient information for generating the target-level annotations. When a subsumption rule is applied in the source code, the corresponding target code is generated within a pair of raising and lowering operations.

4 TAL_C

4.1 Abstract Machine

Our language TAL_C is designed to resemble TAL [11] for ease of understanding. We introduce some new constructs for confidentiality, and accommodate a stack following STAL [10] for supporting procedure calls. For simplicity, we omitted from TAL and STAL some features (*e.g.*, polymorphism, existential types and heap allocation) that are orthogonal to our proposed security operations.

We assume that security labels form a lattice \mathcal{L} . We use θ to range over elements of \mathcal{L} . We use \perp and \top as the bottom and top of the lattice, \cup and \cap as the lattice join and meet operations, and \subseteq as the lattice ordering. The syntactic constructs of TAL_C can be understood in three steps as follows.

Type Constructs. The top portion of Figure 5 presents the type constructs. Security contexts κ follow the idea of Section 3.2. An empty security context (\bullet) represents an program counter with the lowest security label. A concrete context ($\theta \triangleright w$) is made up of a security label θ (the current security level) and a postdominator w . The postdominator w has the syntax of a word value, but its use is restricted by the semantics to be

(contexts)	$\kappa ::= \bullet \mid \theta \triangleright w$
(pre-type)	$\tau ::= \text{int} \mid \langle \sigma_1, \dots, \sigma_n \rangle \mid \forall[\Delta].\langle \kappa \rangle \Gamma$
(types)	$\sigma ::= \tau_\theta \mid ns$
(stack ty)	$\Sigma ::= \rho \mid nil \mid \sigma :: \Sigma$
(var env)	$\Delta ::= \circ \mid \rho \Delta \mid \alpha \Delta$
(type arg)	$\psi ::= \Sigma \mid w$
(heap ty)	$\Psi ::= \{l_1 : \sigma_1, \dots, l_n : \sigma_n\}$
(reg file ty)	$\Gamma ::= \{r_1 : \sigma_1, \dots, r_n : \sigma_n, \text{sp} : \Sigma\}$
(registers)	$r ::= r_1 \mid r_2 \mid \dots$
(word val)	$w ::= \alpha \mid l \mid i \mid ns \mid w[\psi]$
(small val)	$v ::= r \mid w \mid v[\psi]$
(heap val)	$h ::= \langle w_1, \dots, w_n \rangle \mid \text{code}[\Delta]\langle \kappa \rangle \Gamma.I$
(heaps)	$H ::= \{l_1 \mapsto h_1, \dots, l_n \mapsto h_n\}$
(reg files)	$R ::= \{r_1 \mapsto w_1, \dots, r_n \mapsto w_n, \text{sp} \mapsto S\}$
(stacks)	$S ::= nil \mid w :: S$
(instr)	$\iota ::= \text{add } r_d, r_s, v \mid \text{ld } r_d, r_s(i) \mid \text{st } r_d(i), r_s \mid \text{mov } r_d, v \mid \text{bnz } r, v$ $\mid \text{salloc } i \mid \text{sfree } i \mid \text{sld } r_d, \text{sp}(i) \mid \text{sst } \text{sp}(i), r_s \mid \text{raise } \kappa$
(instr seq)	$I ::= \iota; I \mid \text{lower } w \mid \text{jmp } v \mid \text{halt } [\sigma]$
(prog)	$P ::= (H, R, I)_\kappa$

Fig. 5. Syntax of TAL_C

eventually an instantiated code label, *i.e.*, the ending point of the current security level. The postdominator w could also be a variable α ; this is useful for compiling procedures, which can be called in different contexts with different postdominators.

Pre-types (τ) reflect the normal types as seen in TAL, including integer types, tuple types, and code types. In comparison with TAL, our code type requires an extra security context (κ) as part of the interface. A type (σ) is either a pre-type tagged with a security label or a nonsense type (ns) for uninitialized stack slots. A stack type (Σ) is either a variable (ρ), or a (possibly empty) sequence of types. We sometimes use nature numbers to refer to stack slots; slot 0 refers to the bottom-most element. The variable context (Δ) is used for typing polymorphic code; it documents stack type variables (ρ) and postdominator variables (α). Stack types and postdominators are also generally referred to as type arguments ψ . Finally, heap types (Ψ) or register file types (Γ) are mappings from heap labels or registers to types; the sp in the register file represents the stack.

Note that the type constructs provide two layers of security labels for a data pointer (*e.g.*, $\langle \text{int}_{\theta_2} \rangle_{\theta_1}$; Section 3.3) or a code pointer (*e.g.*, $(\forall[\circ].(\theta_2 \triangleright l) \Gamma)_{\theta_1}$; Section 3.4)—one (θ_1) for the pointer itself, the other (θ_2) for the data or code being referenced.

Value Constructs. The middle portion of Figure 5 shows the value constructs. A word value w is either a variable, a heap label l , an immediate integer i , a nonsense value for an uninitialized stack slot, or another word value instantiated with a type argument. Small values v serve as the operands of some instructions; they are either registers r , word values w , or instantiated small values. Heap values h are either tuples or typed code sequences; they are the building blocks of the heap H . Note that a value does not carry a security label. This is consistent with the philosophy that a value is never intrinsically sensitive—it is sensitive only if it comes from a sensitive location [20], which is documented in the corresponding types (Ψ and Γ). Finally, a register file R stores the contents of all registers and the stack, where the stack is a (possibly empty) sequence of word values.

Code Constructs. Code constructs are given in the bottom portion of Figure 5. We retain a minimal set of instructions from TAL and STAL, and introduce two new instructions (`raise κ` and `lower l`) for manipulating the security context as discussed in Section 3. A program is the usual triple tagged with a security context. The security context facilitates the formal soundness proof, but does not affect the computation.

In the operational semantics (available in the technical report [23]), there are only two cases that modify the security context: `raise κ'` updates the security context to κ' , and `lower w` picks up a new security context from the interface of the target code w . In all other cases, the security context remains the same, and the semantics is standard. It is easy to see that this operational semantics mimics the behavior of a real machine, and does not prohibit bad flows. One can obtain a conventional machine by removing the security contexts and `raise κ` instructions, and replacing `lower w` with `jmp w` .

4.2 Typing Rules

The static semantics consists of judgment forms summarized in Figure 6. A security context appears in the judgment of a valid instruction sequence. Heap and register file types are made explicit in the judgment of a valid program for facilitating the noninterference theorem. All other judgment forms closely resemble those of TAL and STAL.

Judgment	Meaning
$\Delta \vdash \kappa$	κ is a valid context
$\Delta \vdash \tau$	τ is a valid pre-type
$\Delta \vdash \sigma$	σ is a valid type
$\Delta \vdash \Sigma$	Σ is a valid stack type
$\vdash \Psi$	Ψ is a valid heap type
$\Delta \vdash \Gamma$	Γ is a valid register file type
$\Delta \vdash \Gamma_1 \subseteq \Gamma_2$	Register file type Γ_1 weakens Γ_2
$\vdash H : \Psi$	Heap H has type Ψ
$\Psi \vdash S : \Sigma$	Stack S has type Σ
$\Psi \vdash R : \Gamma$	Register file R has type Γ
$\Psi \vdash h : \sigma$	Heap value h has type σ
$\Psi; \Delta \vdash w : \sigma$	Word value w has type σ
$\Psi; \Delta; \Gamma \vdash v : \sigma$	Small value v has type σ
$\Psi; \Delta; \Gamma; \kappa \vdash I$	I is a valid sequence of instructions
$\Psi; \Gamma \vdash P$	P is a valid program

Fig. 6. TAL_C typing judgments

The typing rules are available in the technical report [23]. Selected typing rules are given in Figure 7. A type construct is valid (top six judgment forms in Figure 6) if all free type variables are documented in the type environment. Heap values and integers may have any security label. The types of heap labels and registers are as described in the heap type and the register file type respectively. All other rules for non-instructions are straightforward extensions of those in TAL and STAL.

We use $SL(\kappa)$ to refer to the security label component of κ . $SL(\bullet)$ is defined to be \perp . The typing rules for `add`, `ld` and `mov` instructions infer the security labels for the destination registers; they take into account the security labels of the source and target operands and the current security context.

The rule for `bnz` first checks that the guard register r is an integer and the target value v is a code label. It then checks that the current security context is high enough to cover the security levels of the guard (preventing flows through program structures; Section 3.2) and the target code (preventing flows through code pointers; Section 3.4). Lastly, the checks on the register file and the remainder instruction sequence make sure that both branches are secure to execute.

The rule for `st` concerns four security labels. The label of the target cell must be higher than or equal to those of the context (Section 3.2), the containing tuple (Section 3.3), and the source value (Section 3.1).

The rules for the stack instructions follow similar ideas. In essence, the stack can be viewed as an infinite number of registers. Instruction `salloc` or `sfree` add new slots to or remove existing slots from the slot, so the rules check the remainder instruction sequence under an updated stack type. The rule for instruction `sld` or `sst` can be understood following that of the `mov` instruction.

The rule for `raise` checks that the new security context is higher than the current one. Moreover, it looks at the postdominator w' of the new context, and makes sure that

$$\begin{array}{c}
\frac{\circ \vdash \kappa \quad \vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi; \circ; \Gamma; \kappa \vdash I}{\Psi; \Gamma \vdash (H, R, I)_\kappa} \\
\\
\frac{SL(\kappa) = \theta \quad \Gamma(r_s) = \langle \sigma_1, \dots, \sigma_n \rangle_{\theta_1} \quad \sigma_i = \tau_{\theta_2} \quad \Psi; \Delta; \Gamma\{r_d : \tau_{\theta \cup \theta_1 \cup \theta_2}\}; \kappa \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \text{ld } r_d, r_s(i); I} \\
\\
\frac{SL(\kappa) = \theta \quad \Gamma(r) = \text{int}_{\theta_1} \quad \Psi; \Delta; \Gamma \vdash v : (\forall[\circ]. \langle \kappa \rangle \Gamma')_{\theta_2} \quad \theta_1 \cup \theta_2 \subseteq \theta \quad \Delta \vdash \Gamma' \subseteq \Gamma \quad \Psi; \Delta; \Gamma; \kappa \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \text{bnz } r, v; I} \\
\\
\frac{SL(\kappa) = \theta \quad \Gamma(r_d) = \langle \sigma_1, \dots, \sigma_n \rangle_{\theta_1} \quad \sigma_i = \tau_{\theta'} \quad \Gamma(r_s) = \tau_{\theta_2} \quad \theta \cup \theta_1 \cup \theta_2 \subseteq \theta' \quad \Psi; \Delta; \Gamma; \kappa \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \text{st } r_d(i), r_s; I} \\
\\
\frac{\Gamma(\text{sp}) = \Sigma \quad \Psi; \Delta; \Gamma\{\text{sp} : \overbrace{ns :: \dots :: ns}^i :: \Sigma\}; \kappa \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \text{salloc } i; I} \\
\\
\frac{SL(\kappa) = \theta \quad \Gamma(\text{sp}) = \sigma_0 :: \dots :: \sigma_i :: \Sigma \quad \Gamma(r_s) = \tau_{\theta'} \quad \Psi; \Delta; \Gamma\{\text{sp} : \sigma_0 :: \dots :: \sigma_{i-1} :: \tau_{\theta \cup \theta'} :: \Sigma\}; \kappa \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \text{sst sp}(i), r_s; I} \\
\\
\frac{\kappa = \theta \triangleright w \quad \kappa' = \theta' \triangleright w' \quad \theta \subseteq \theta' \quad \Psi; \Delta \vdash w' : (\forall[\circ]. \langle \kappa \rangle \Gamma')_{\theta_1} \quad \Psi; \Delta; \Gamma; \kappa' \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \text{raise } \kappa'; I} \\
\\
\frac{\kappa = \theta \triangleright w \quad \Psi; \Delta \vdash w : (\forall[\circ]. \langle \kappa' \rangle \Gamma')_{\theta_1} \quad \theta_1 \subseteq SL(\kappa') \quad \Delta \vdash \Gamma' \subseteq \Gamma}{\Psi; \Delta; \Gamma; \kappa \vdash \text{lower } w}
\end{array}$$

Fig. 7. Selected typing rules of TAL_C

the security context at w' matches the current one. The remainder instruction sequence is checked under the new context.

Since the rule for **raise** already checked the validity of the ending label of a secured region, the task for ending the region is relatively simple. The rule for **lower** checks that its operand label matches that dictated by the security context. This guarantees that a secured region be enclosed within a **raise-lower** pair. The rule also makes sure that the code at w is safe to execute, which involves checking the security labels (Section 3.4) and the register file types.

The rule for **jmp** checks that the target code is safe to execute. Similar checks also appeared in the rule for **bnz**. In these two rules, the security context of the target code is always the same as the current one. This is because context changes are separated from conventional instructions in our system. For example, one may enclose high target code within **raise** and **lower** before calling it in a low context.

Finally, halting is valid only if the security context is empty, and the value in r_1 has the expected type σ .

We note that supporting explicit heap allocations does not introduce new difficulties. Although different program branches may exhibit different allocation behaviors under

a high security context, they must agree at a common heap type when joining at a corresponding `lower` instruction. Furthermore, we define the equivalence of heaps (stacks, register files) based only on their elements of low security types (the next section).

4.3 Soundness

TAL_C enjoys conventional type safety (memory and control-flow safety), which can be established following the preservation and progress lemmas.

Before presenting the noninterference theorem, we define the equivalence of two programs with respect to a security level θ . Intuitively, two programs (heaps, stacks, register files) are equivalent if and only if they agree on their low-security contents.

Definition 1 (Heap Equivalence). $\Psi \vdash H_1 \approx_\theta H_2 \iff$ for every $l \in \text{dom}(\Psi)$, if $\Psi(l) = \tau_{\theta'}$ and $\theta' \subseteq \theta$ then $H_1(l) = H_2(l)$.

Definition 2 (Stack Equivalence). $\Sigma \vdash S_1 \approx_\theta S_2 \iff$ for every stack slot i , if $\Sigma(i) = \tau_{\theta'}$ and $\theta' \subseteq \theta$ then $S_1(i) = S_2(i)$.

Definition 3 (Register File Equivalence). $\Gamma \vdash R_1 \approx_\theta R_2 \iff$ (1) $\Gamma(sp) \vdash R_1(sp) \approx_\theta R_2(sp)$, and (2) for every $r \in \text{dom}(\Gamma)$, if $\Gamma(r) = \tau_{\theta'}$ and $\theta' \subseteq \theta$, then $R_1(r) = R_2(r)$.

Definition 4 (Program Equivalence). $\Psi; \Gamma \vdash P_1 \approx_\theta P_2 \iff P_1 = (H_1, R_1, I_1)_{\kappa_1}$, $P_2 = (H_2, R_2, I_2)_{\kappa_2}$, $\Psi \vdash H_1 \approx_\theta H_2$, $\Gamma \vdash R_1 \approx_\theta R_2$, and either: (1) $\kappa_1 = \kappa_2$, $SL(\kappa_1) \subseteq \theta$, and $I_1 = I_2$, or (2) $SL(\kappa_1) \not\subseteq \theta$, $SL(\kappa_2) \not\subseteq \theta$.

It is easy to see that the above relations are all reflexive, symmetrical, and transitive. Our noninterference theorem relates the executions of two equivalent programs that both start in a low security context (relative to the security level of concern). If both executions terminate, then the result programs must also be equivalent.

The idea of the proof is intuitive. Given a security level of concern, the executions can be phased into “low steps” and “high steps.” It is easy to relate the two executions under a low step, because they involve the same instructions. Under a high step, the two executions are no longer in lock step. Recall that `raise` and `lower` mark the beginning and the end of a secured region. We relate the program states before the `raise` and after the `lower`, circumventing directly relating two executions under high steps. Interested readers are referred to the technical report [23] for more details.

Theorem 1 (Noninterference). If $P = (H, R, I)_\kappa$, $SL(\kappa) \subseteq \theta$, $\Psi; \Gamma \vdash P$, $\Psi; \Gamma \vdash Q$, $\Psi; \Gamma \vdash P \approx_\theta Q$, $P \mapsto^* (H_p, R_p, \text{halt}[\sigma_p])_\bullet$, and $Q \mapsto^* (H_q, R_q, \text{halt}[\sigma_q])_\bullet$, then exists Γ' such that $\Psi; \Gamma' \vdash (H_p, R_p, \text{halt}[\sigma_p])_\bullet \approx_\theta (H_q, R_q, \text{halt}[\sigma_q])_\bullet$.

4.4 Example

Figure 8 gives a simple example to demonstrate the use of security labels and contexts. The high-level pseudo-code program involves a low variable `a` and two high variables `b` and `c`. In a corresponding TAL_C program, we use heap cells labeled l_a , l_b and l_c to represent these variables. The TAL_C program starts from the code labeled l_0 in a low security context. After the initial setup, it raises the security context to $\top \triangleright l_3$. The

A pseudo-code program: $a = 0; \text{if } (b <> 0) \text{ then } c = 1 \text{ else } c = 0; a = 1$

A corresponding TAL_C program: $(H, \{\text{sp} : \text{nil}\}, \text{jmp } l_0) \bullet$ where $H = \{(l_a, l_b, l_c \text{ omitted})\}$

```

 $l_0 \mapsto$   code[o]( $\bullet$ ){ $\text{sp} : \text{nil}$ }.
           mov  $r_0, 0$ ;                                %  $r_0 \leftarrow 0$ 
           mov  $r_1, l_a$ ;                                %  $r_1 \leftarrow l_a$ 
           mov  $r_2, l_b$ ;                                %  $r_2 \leftarrow l_b$ 
           mov  $r_3, l_c$ ;                                %  $r_3 \leftarrow l_c$ 
           st  $r_1(0), r_0$ ;                              %  $l_a \leftarrow 0$ 
           raise  $\top \triangleright l_3$ ;                          % raise security context
           jmp  $l_1$ 

 $l_1 \mapsto$   code[o]( $\top \triangleright l_3$ ){ $r_0 : \langle \text{int}_\perp \rangle_\perp, r_1 : \langle \text{int}_\perp \rangle_\perp, r_2 : \langle \text{int}_\top \rangle_\perp, r_3 : \langle \text{int}_\top \rangle_\perp, \text{sp} : \text{nil}$ }.
           ld  $r_4, r_2(0)$ ;                                % go to  $l_2$  if content of  $l_b$  is not zero
           bnz  $r_4, l_2$ ;                                % the else branch:  $l_c \leftarrow 0$ 
           st  $r_3(0), r_0$ ;                                % restore security context and go to  $l_3$ 
           lower  $l_3$ 

 $l_2 \mapsto$   code[o]( $\top \triangleright l_3$ ){ $r_0 : \langle \text{int}_\perp \rangle_\perp, r_1 : \langle \text{int}_\perp \rangle_\perp, r_2 : \langle \text{int}_\top \rangle_\perp, r_3 : \langle \text{int}_\top \rangle_\perp, \text{sp} : \text{nil}$ }.
           mov  $r_0, 1$ ;
           st  $r_3(0), r_0$ ;                                % the then branch:  $l_c \leftarrow 1$ 
           lower  $l_3$                                 % restore security context and go to  $l_3$ 

 $l_3 \mapsto$   code[o]( $\bullet$ ){ $r_1 : \text{int}_\perp, \text{sp} : \text{nil}$ }.
           mov  $r_0, 1$ ;
           st  $r_1(0), r_0$ ;                                %  $l_a \leftarrow 1$ 
           halt [int $_\perp$ ]

```

Fig. 8. TAL_C example

control is then transferred to the code labeled l_1 , which contains a test on the high variable b and directs the execution to two separate branches. In either branch of the conditional, the high variable c is updated, and the security context is restored with `lower l_3` . The code at l_3 is then free to update the low variable a again.

A closer look at the code labeled l_1 reveals several interesting issues. When checking the first load instruction (`ld $r_4, r_2(0)$`), the security level for r_4 is inferred to be high (\top). The following branching instruction (`bnz r_4, l_2`) type-checks because the current security context ($\top \triangleright l_3$) is high enough to cover the security level of r_4 . The next store instruction (`st $r_3(0), r_0$`) is also valid, because it is ok to update a high variable in a high context. In comparison, the store instruction would fail to type-check if c was a low variable. Finally, the high security context is ended with a lower instruction (`lower l_3`) that directs the control flow to the postdominator of the conditional.

5 Discussions

Linear Continuations. Zdancewic and Myers [24] introduced a notion of ordered linear continuations to facilitate the information-flow analysis at a low level (we use ZM to refer to their system). An important requirement of such analysis is that one needs to allow a high-security conditional to be surrounded by low-security computation. In ZM, before the conditional statement, a linear continuation is created to capture the

computation after the conditional. Such a linear continuation must be called exactly once at the end of either branch of the conditional. Furthermore, the linear continuation records the security context in which it is created, allowing the security context to be reset properly when the branches meet.

As a higher-order analog to postdominators in a control-flow graph, ordered linear continuations enforce a stack discipline that allows security contexts to be reset at the join points of program branches. The static semantics ensures that the linear continuations are properly nested, and at any time only the top continuation on the (virtual) continuation stack is available. The linearity is enforced because the continuation is essentially popped off the stack when used. In particular, every value in ZM is tagged with a security label. The operational semantics keeps track of the security context during the execution, and ensures that security labels of the values are propagated correctly.

It may help to view our solution as an adaptation of linear continuations for the RISC architecture. A postdominator of program branches is essentially expressed as a static code label. The security operations `raise` and `lower` correspond to the creation and elimination of linear continuations. At any program point, our static semantics keeps track of only the top element of the (virtual) continuation stack. The typing rule for `raise` ensures that the security context at the postdominator matches the current one, thus enforcing the stack discipline.

We wish to point out, nonetheless, that such an adaptation yields a simple, practical and well-grounded solution to the identified problem of information-flow analysis for assembly code. In particular, it bridges the gap between the functional abstraction of linear continuations and the raw assembly code running on actual machines. In comparison with ZM, our system TAL_C models the use of registers and assembly instructions, and hence is closer to the actual RISC architecture. We do not attach security labels to values; this makes it trivial to see that security annotations do not affect computation. In fact, the enforcement of noninterference in TAL_C is cleanly separated from normal program execution. It is also obvious that security operations in TAL_C are orthogonal from conventional instructions (e.g., branching and jumping) and mechanisms (e.g., call stack), which allows our approach to be carried further with other language extensions. Consequently, we consider TAL_C as a good first step toward a scaled-up typed assembly language for noninterference.

Translating Linear Continuations. It may appear that TAL_C is not as expressive as the language of Zdancewic and Myers' [24] (ZM), because the security context of TAL_C uses static labels. Nonetheless, these static labels are only used to refer to code (e.g., that of linear continuations in ZM) whose locations can be statically determined. Indeed, their source level counterparts are the ending points of conditional structures, which are always statically known. Therefore, there is not a loss of expressiveness. We demonstrate this by speculating a translation from ZM to TAL_C .

In ZM, there are two expressions manipulating linear continuations: creation and elimination. The creation of a linear continuation essentially has the form $\text{letlin } y = \lambda\langle pc \rangle(x:\sigma).e \text{ in } e'$. A corresponding elimination has the form $\text{lgoto } y \ v$.

The translation can be carried out following Morrisett *et al.* [11]. The step of CPS conversion is not needed because ZM is already in CPS. During closure conversion, the abstraction $\lambda\langle pc \rangle(x:\sigma).e$ (which corresponds to the code at a postdominator) will be

assigned a static code label. This code label is exactly the static postdominator needed for raising the security context in TAL_C . In a formal translation, this label can be used to generate a `raise` instruction when a corresponding branching point is reached. The typing (in particular, the security labels) of a ZM program is sufficient for identifying the branching point.

The elimination of linear continuation (`lgoto y v`) is relatively straightforward. Suppose the code of y (the lambda abstraction) declared using `letlin` is assigned the heap label l_y during closure conversion, the elimination expression `lgoto y v` can be translated as a `lower l_y` preceded with appropriate code computing the argument v .

For better understanding the relationship between linear continuations in ZM and security contexts in TAL_C , we further look into an example of nested `letlin` declarations: `letlin $y_1 = lv_1$ in letlin $y_2 = lv_2$ in e` . Once the second `letlin` is declared, the first linear continuation y_1 should be accessible only from inside lv_2 . Therefore, ZM requires that e type checks under y_2 , and lv_2 type checks under y_1 . This essentially enforces a stack discipline.

TAL_C has a similar mechanism. Suppose the current security context is $\theta_1 \triangleright l_1$ and the current instruction sequence is `raise $\theta_2 \triangleright l_2$; I` . The type system of TAL_C checks I under $\theta_2 \triangleright l_2$, and checks that the code type at l_2 respects $\theta_1 \triangleright l_1$. This enforces a similar stack discipline as in ZM; note that only the top stack element is apparent at any time.

SIF. SIF [9] is developed independently from TAL_C . These two systems are similar in spirit—both use static types for information-flow analysis. However, SIF is based on a minimal language where relatively simple annotations, namely a stack of static code labels, suffice. In a more realistic language, a single function (even if monomorphic with respect to security levels) can be called at different program points. The security contexts of these program points may be different with respect to (1) the postdominator of the current context (SIF tracks this with the top stack element), and (2) the “enclosing contexts” (SIF tracks these with the stack tail). Since the label stack of SIF is made up of static code labels, one cannot reuse the same code at different program points with different contexts.

TAL_C only maintains the current security context at any program point, and we show that it suffices for establishing noninterference. With such a treatment, the code types are naturally polymorphic with respect to enclosing contexts. We also allow postdominators to be polymorphic. The certifying compilation scheme in the technical report [23] further demonstrates the expressiveness of TAL_C .

6 Extensions and Future Work

Orthogonal Features. For ease of understanding, TAL_C focuses on a minimal set of language features. Nonetheless, polymorphic and existential types, as seen in TAL, are orthogonal and can be introduced with little difficulty. Furthermore, since TAL_C is compatible with TAL, it is also possible to accommodate other features of the TAL family. For instance, alias types [17] may provide a more accurate alias analysis, improving the current conservative approach that considers every pointer as a potential alias. In the following, we will also discuss the use of singleton types [22].

Security Polymorphism. TAL_C relies on a security context $\theta \triangleright w$ to identify the current security level θ and its ending point w . It is monomorphic with respect to security, because the security level of a code block is fixed. In practice, security-polymorphic code can also be useful.

```
int double(x:int) { x=x*2; } ...
if h<10 then double(h);
double(1); ...
```

Fig. 9. Security-polymorphic function

Figure 9 gives an example. The function `double` can be invoked with either low or high input. It is safe to invoke `double` in a context if only the security level of the input matches that of the context. In a security polymorphic TAL_C -like type system, `double` can be given the type $(\forall[\theta, \alpha]. \langle \theta \triangleright \alpha \rangle \{r_1 : \text{int}_\theta, r_0 : (\forall[], \langle \theta \triangleright \alpha \rangle \{r_1 : \text{int}_\theta\})_\perp\})_\perp$. Here r_1 is the argument register, r_0 stores the return pointer, and the meta-variable θ is reused as a variable.

It is straightforward to support this kind of polymorphism. In fact, most of the required constructs are already present in TAL_C . We omitted such polymorphism simply because it complicates the presentation without providing additional insights. Nonetheless, the expressiveness of such polymorphism is still limited. Since the label α is not known until instantiated, the code of `double` has no knowledge about α . Hence the security context $\theta \triangleright \alpha$ cannot be discharged within the body of `double`.

It is not obvious why one would wish to discharge the security context within a polymorphic function. Indeed, it is always possible to wrap a function call inside a secured region by symmetric `raise` and `lower` operations from the caller’s side. However, the asymmetric discharging of security context may be desirable for *certifying optimization*. For instance, in Figure 9, `double` is called as the last statement of the body of a high conditional. In this case, directly discharging the security context when `double` returns would remove a superfluous `lower` from the caller’s side. Such a discharging requires `lower` to operate on small values—since the return label is not statically fixed, it must be passed in through a register.

It may require singleton and intersection types to support such a `lower` operation. For example, a `double` function that discharges its security context can have type

$$\left(\forall[\theta, \alpha]. \langle \theta \triangleright \alpha \rangle \left\{ r_1 : \text{int}_\theta, r_0 : \text{sing}(\alpha)_\perp \wedge (\forall[], \langle \bullet \rangle \{r_1 : \text{int}_\theta\})_\perp \right\} \right)_\perp.$$

At the end of the function, `lower` r_0 discharges the security context and transfers the control to the return code. For type checking, the singleton integer type $\text{sing}(\alpha)$ matches the register r_0 with the label in the security context, and the code type ensures that the control flow to the return point is safe.

Full Erasure. With the powerful type constructs above, one can achieve a full erasure for the `lower` operation. Instead of treating `lower` as an instruction, one can treat it as a transformation on small values. This is in spirit similar to the *pack* operation of

existential types in TAL. Such a `lower` transformation bridges the gap between the current security context and the security level of the target label. The actual control flow transfer is then completed with a conventional jump instruction (*e.g.*, `jmp (lower r_0)`).

One can also achieve a full erasure for `lower` even without singleton types. The idea is to separate the jump instruction into direct jump and indirect jump. This is also consistent with real machine architectures. The `lower` operation transforms word values (eventually, direct labels). Lowered labels, similar to packed values, may serve as the operand of direct jump. Indirect jump, on the other hand, takes normal small values. This is expressive enough for certifying compilation, yet may not be sufficient for certifying optimization as discussed above.

7 Conclusion

We have presented a language TAL_C for enforcing data confidentiality in assembly code. The main idea is to use type annotations to restore high-level abstractions that are crucial to information-flow analysis. In TAL_C , operations related to security are kept orthogonal from other language features. As a result, it is possible to accommodate existing results on low-level verification, such as the TAL family. Our technical report presents a translation from a high-level security language with first-order procedures to TAL_C . A soundness theorem shows that the translation preserves security types. We consider this as a useful step toward a certifying compiler for noninterference.

Acknowledgments. We thank Eduardo Bonelli, Adriana Compagnoni, Ricardo Medel, Greg Morrisett, Steve Zdancewic and the anonymous referees for helpful comments on previous drafts.

References

1. M. Abadi, A. Banerjee, H. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Prin. of Prog. Lang.*, pages 147–160, San Antonio, TX, Jan. 1999.
2. J. Agat. Transforming out timing leaks. In *Proc. 27th ACM Symposium on Principles of Programming Languages*, pages 40–53, Boston, MA, Jan. 2000.
3. J. Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Chalmers Univ. of Tech. and Gothenburg Univ., Gothenburg, Sweden, Dec. 2000.
4. T. Ball. What’s in a region? Or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Prog. Lang. and Syst.*, 2(1-4):1–16, Mar.-Dec. 1993.
5. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. 15th IEEE CSFW Workshop*, pages 253–267, June 2002.
6. G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In *Proc. 5th International Conf. on VMCAI*, volume 2937 of *LNCS*, pages 2–15, Venice, Italy, Jan. 2004.
7. E. Bonelli, A. Compagnoni, and R. Medel. SIFTAL: A typed assembly language for secure information flow analysis. Technical report, Stevens Inst. of Tech., Hoboken, NJ, July 2004.
8. N. Heintze and J. G. Riecke. The SLam calculus: Programming with security and integrity. In *Proc. 25th ACM Symp. on Prin. of Prog. Lang.*, pages 365–377, San Diego, CA, Jan. 1998.
9. R. Medel, A. Compagnoni, and E. Bonelli. Non-interference for a typed assembly language. In *Proc. 2005 Workshop on Foundations of Computer Security*, Chicago, IL, June 2005.

10. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, Jan. 2002.
11. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, Nov. 1999.
12. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Prin. of Prog. Lang.*, pages 228–241, San Antonio, TX, 1999.
13. F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003.
14. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
15. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9), Sept. 1975.
16. F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 86–101, 2001.
17. F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proc. 9th European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381, Berlin, Germany, Apr. 2000.
18. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Prin. of Prog. Lang.*, pages 355–364, San Diego, CA, Jan. 1998.
19. D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *10th IEEE Computer Security Foundations Workshop*, pages 156–169, Washington, DC, June 1997.
20. D. Volpano and G. Smith. A type-based approach to program security. In *Proc. 7th Inter. Joint Conf. CAAP/FASE TAPSOFT*, LNCS, pages 607–621, Lille, France, Apr. 1997.
21. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proc. 11th IEEE CSFW Workshop*, pages 34–43, Washington, DC, June 1998.
22. H. Xi and R. Harper. A dependently typed assembly language. In *Proc. 6th ACM International Conference on Functional Programming*, pages 169–180, Florence, Italy, Sept. 2001.
23. D. Yu and N. Islam. A typed assembly language for confidentiality. Technical Report DCL-TR-2005-0002, DoCoMo Communications Laboratories USA, San Jose, CA, Mar. 2005. <http://www.docomolabsresearchers-usa.com/~dyu/talc-tr.pdf>.
24. S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2–3):209–234, Sept. 2002.

Flow Locks: Towards a Core Calculus for Dynamic Flow Policies

Niklas Broberg and David Sands

Chalmers University of Technology and Göteborg University

Abstract. Security is rarely a static notion. What is considered to be confidential or untrusted data varies over time according to changing events and states. The static verification of secure information flow has been a popular theme in recent programming language research, but information flow policies considered are based on multilevel security which presents a static view of security levels. In this paper we introduce a very simple mechanism for specifying dynamic information flow policies, flow locks, which specify conditions under which data may be read by a certain actor. The interface between the policy and the code is via instructions which open and close flow locks. We present a type and effect system for an ML-like language with references which permits the completely static verification of flow lock policies, and prove that the system satisfies a semantic security property generalising noninterference. We show that this simple mechanism can represent a number of recently proposed information flow paradigms for declassification.

1 Introduction

Unlike access control policies, enforcing an information flow policy at run time is difficult because information flow is not a runtime property; we cannot in general characterise when an information leak is about to take place by simply observing the actions of a running system. From this perspective, statically determining the information-flow properties of a program is an appealing approach to ensuring secure information flow. However, security *policies*, in practice, are rarely static: a piece of data might only be untrusted until its signature has been verified; an activation key might be secret only until it has been paid for.

This paper introduces a simple policy specification mechanism based on the idea that the reading of storage location ℓ by certain actors (principals, levels) is guarded by boolean flags, which we call *flow locks*. For example, the policy $\ell_{\{High; paid \Rightarrow Low\}}$ says that ℓ can always be read by an actor with a high clearance level, and also by an actor with a low clearance level providing the “paid” lock is open.

The interface between the flow lock policies and the security relevant parts of the program is provided by simple instructions for opening and closing locks. The program itself does not depend on the lock state, and the intention is that by statically verifying that the dynamic flow policy will not be violated, the lock state does not need to be computed at run time.¹

¹ The term *dynamic* flow policy could have different interpretations. We use it in the sense that the flow policies vary over time, but they are still statically known at compile time.

In addition to the introduction of flow locks, the main contributions of this paper are:

- The definition of a type system for an ML-like language with references which permits the completely static verification of flow lock policies.
- A formulation of the semantics of secure information flow for flow locks, and a proof that well typed programs are flow-lock secure (the reader is referred to the extended version of this article for the details).
- The demonstration that flow lock policies can represent a number of recently proposed information flow paradigms.

Regarding the last point, the work presented here can be viewed as a study of *declassification* mechanisms. In a recent study by Sabelfeld and Sands [18], declassification mechanisms are classified along four dimensions: *what* information is released, *who* releases information, *where* in the system information is released, and *when* information can be released. One of the key challenges stated in that work is to *combine* these dimensions. In fact, combination is perhaps not difficult; the real challenge is to combine these dimensions without simply amassing the combined complexities of the contributing approaches. Later in this paper we argue that flow locks can encode a number of recently proposed “declassification” paradigms, including the lexically scoped flow policies introduced by Almeida Matos and Boudol [2], Chong and Myers’ notion of *noninterference until declassification* [5], and Zdancewic and Myers *robust declassification* [22, 13]. These examples, represent the “where”, “when” and “who” dimensions of declassification, respectively, suggesting that flow locks have the potential to provide a core calculus of dynamic information flow policies.

The remainder of the paper is organised as follows. Section 2 gives an informal introduction to flow locks by showing a few motivating examples. In Section 3 we then present the system formally, and outline a semantic security condition in Section 4. Section 5 discusses related systems, with an emphasis on how we can use flow locks to encode them. Finally Section 6 concludes.

2 Motivating Examples

First let us assume we have a simple imperative language without any security control mechanisms of any kind. Borrowing an example from Chong and Myers [5], suppose we want to implement a system for online auctions with hidden bids in this language. We could write part of this system as the code on the right.

This surely works, but there is nothing in the language that prevents us from committing a serious security error. We could for instance accidentally switch the lines 2 and 3, resulting in *A*’s bid being made public before *B* places her bid, giving *B* the chance to tailor her bid after *A*’s.

```

1 int aBid = getABid();
2 int bBid = getBBid();
3 makePublic(aBid);
4 makePublic(bBid);
5 ...decide winner + sell item
```

Flow locks are a mechanism to ensure that these and other kinds of programming errors are caught and reported in a static check of the code.

The basic idea is very similar to what many other systems offer. To deny the flow of data to places where it was not meant to go, we annotate variables with policies that govern how the data held by those variables may be used. Looking back on our example, a proper policy annotation on the variable `aBid` could be $\{A; \text{BBid} \Rightarrow B\}$. The intuitive interpretation of this policy is that the data held by variable `aBid` may always be accessed by *A*, and may also be accessed by *B* whenever the condition `BBid`, that *B* has placed a bid, is fulfilled. `BBid` here is a *flow lock* — only if the lock is *open* can the data held by this variable flow to *B*. To know whether the lock is open or not we must look at how the functions for getting the bids could be implemented.

The function shown on the right first fetches the bid sent by *A*. We model the incoming channel as a global variable that can be read from, one with the same policy as `aBid`. When the bid has been read, the function signals this by opening the `ABid` lock—*A* has now placed a bid and the program can act accordingly. The implementation of `getBBid` follows the same pattern, and will result in `BBid` being open.

```
function getABid() {
  int {A;BBid ⇒ B} x
    = bidChanFromA;
  open ABid;
  return x;
}
```

```
function makePublic(bid) {
  publicChannel = bid;
}
```

Now both bids have been placed and can thus be released. The `makePublic` function would be implemented as shown on the left. The outgoing `publicChannel` is also modelled as a global

variable that can be written to. This one has the policy $\{A; B\}$ attached to it, denoting that both *A* and *B* will be able to access any data written into it. At the points in the program where `makePublic` is applied, both *A* and *B* will have placed their bids, the locks `ABid` and `BBid` will both be open, and the flows to the public channel will both be allowed. However, if the lines 2 and 3 were now accidentally switched, it would be a different story. Then we would attempt to release *A*'s bid, guarded by the policy $\{A; \text{BBid} \Rightarrow B\}$, onto the public channel with policy $\{A; B\}$. Since the flow lock `BBid` will then not yet be opened, this flow is illegal and the program can be rejected.

Taking the example one step further, assume that we have two items up for auction, one after the other. We can implement this rather naively as the program to the right. The locks `ABid` and `BBid` will both be opened on the first calls to the `getXBid` functions. But unless we have some means to reset them, there is again nothing to stop us from accidentally switching lines to make our program insecure, this time lines 9 and 10. The same problem could also be seen from a different angle: what if the locks were already open when we got to

```
1 auctionItem(firstItem);
2 aBid = getABid();
3 bBid = getBBid();
4 makePublic(aBid);
5 makePublic(bBid);
6 ... decide winner + sell item
7 auctionItem(secondItem);
8 aBid = getABid();
9 bBid = getBBid();
10 makePublic(aBid);
11 makePublic(bBid);
12 ... decide winner + sell item
```

this part of the program? Clearly we need a closing mechanism to go with the open. The function `auctionItem` could then be implemented as shown here. By closing the locks when an auction is initiated, we can rest assured that both *A* and *B* must place new bids for the new item before either bid is made public.


```
function auctionItem(item){
  close ABid, BBid;
  ... present item ... }
```

It should be fairly easy to see that what we have here is a kind of state machine. The state at any program point is the set of locks that are open at that point, and the open and close state-

ments form the state transitions. A clause $\sigma \Rightarrow A$ in a policy means that A may access any data guarded by that policy in any state where σ is open.

Our lock-based policies also give us an easy way to separate truly secret data from data that is currently secret, but that may be released to other actors under certain circumstances. Assume for instance that payment for auctioned items is done by credit card, and that the server stores credit card numbers in memory locations `aCCNum` and `bCCNum` respectively. Assume further that the line `aBId := aCCNum;` is inserted, either by sheer mistake or through malicious injection, just before where `aBId` is made public. This would release A 's credit card number to B , however, the natural policy on `aCCNum` would be $\{A\}$, meaning only A may view this data, ever. Thus when we attempt the assignment above, it will be statically rejected since the policy on `aBId` is too permissive.

All the above are examples of policies to track confidentiality. The dual of confidentiality is integrity, i.e. deciding to what extent data can be trusted, and it should come as no surprise that flow locks can handle both kinds.

Returning to the example with the credit card, we assume that when A gives her credit card number, it must be validated (in some unspecified way) before we can trust it. To this end we introduce a “pseudo” actor T (for “trusted”) who should only be allowed to read data that is fully trusted. We then use an intermediate location `tmpACCNum` to hold the credit card number when it is submitted by A . This location is given the policy $\{A; ACCVal \Rightarrow T\}$, stating that this data is trusted only if the lock `ACCVal` is open, which is done when the submitted number has been validated. Once validated we can transfer the value to `aCCNum`, which now has the policy $\{A; T\}$ stating that this data is trusted.²

3 A Secure Type and Effect System

In the previous section we used a simple imperative language to give an easy introduction to the concept of flow locks. In this section we define the type system for flow locks in the more general context of an ML-like language with recursion and references (but without polymorphism).

3.1 The Language λ_{FL}

The terms and types of our language, dubbed λ_{FL} , are listed in Figure 1.

The policy language is worth some extra attention. The flow lock policies with which we work assumes a set of *actors* (or *levels*, *principals*) ranged over by A, B , and a set of flow locks ranged over by σ , with Σ for sets of locks. Both actors and flow locks are global in a program. A *policy* is a set of *clauses*, where each clause of the form

² In order to prevent overwriting this data with a new number that hasn't been validated, we should also be sure to close the lock `ACCVal` once the assignment is done.

Policies:	$p ::= \{c_1; \dots; c_n\} \quad c ::= \{\sigma_1, \dots, \sigma_k\} \Rightarrow A$
Values and types:	$v ::= n \mid b \mid () \mid \lambda x.M \mid \ell_{p,\tau}$ $\tau ::= int \mid bool \mid unit \mid (\tau, p) \xrightarrow{\Sigma, p, p, \Sigma} \tau \mid ref_p \tau$
Terms:	$M ::= v \mid x \mid MM \mid \text{if } M \text{ then } M \text{ else } M \mid \text{rec } x.M$ $\mid ref_{p,\tau} M \mid !M \mid M := M \mid \text{open } \sigma \mid \text{close } \sigma$
Derived forms:	$\text{let } x = M_1 \text{ in } M_2 \equiv (\lambda x.M_2)M_1 \quad M_1; M_2 \equiv (\lambda_.M_2)M_1$

Fig. 1. The λ_{FL} language

$\Sigma \Rightarrow A$ states the circumstances (Σ) under which A may view the data governed by this policy. Σ is a set of locks which we name the *guard* of the clause, and interpret it as a conjunction. Thus for the guard to be fulfilled, all the locks in Σ must be open. We can however have more than one clause for the same A , in which case the separate clauses also form a conjunction — A may read the data if either of the guards are fulfilled. In the special case where the guard contains no locks, signifying that the corresponding actor A may always view the data, we write the clause as only A instead of $\{\} \Rightarrow A$. From a logical perspective a policy is just a conjunction of definite Horn clauses, i.e. $\bigwedge_i \{\sigma_{i1} \wedge \dots \wedge \sigma_{in} \Rightarrow A_i\}$. We implicitly identify policies up to logical equivalence.³

Now we can continue with the language itself. Apart from the terms from standard λ calculus with recursion, λ_{FL} has constructs for creating (ref), dereferencing (!) and assigning to ($:=$) memory locations ($\ell_{p,\tau}$) through references. In addition to the core terms, we can also derive a few useful language constructs as is also shown in Figure 1.

The reference creation construct takes an extra parameter p which is the policy that the contents should be governed by. The same parameter also shows up on the memory locations themselves, together with the base type τ of the contents. In many cases this τ is irrelevant, or clear from the context, and in those cases we omit it and just write ℓ_p . Function types are annotated with read and write policies, and start and end states, and arguments are annotated with a reading policy. We discuss the meaning of these when we define the type system. There are also the open and close terms for manipulation flow locks, thereby changing the state of the program.

The semantics of the language is standard, but apart from the term M and a memory μ , the configurations include the current state Σ . This state is the set of currently open locks, which are effected by the execution of **open** and **close** expressions. The small-step semantics of these are simply:

$$\langle \Sigma, \text{open } \sigma, \mu \rangle \rightarrow \langle \Sigma \cup \{\sigma\}, (), \mu \rangle \quad \langle \Sigma, \text{close } \sigma, \mu \rangle \rightarrow \langle \Sigma \setminus \{\sigma\}, (), \mu \rangle$$

It is important to note that the only interaction between a program and the lock state is via the open and close instructions. This is because we are aiming for a completely static verification — we include the lock state in the semantics only to be able to prove

³ It is worth noting that we do not allow negative flow policies. Our policy language is monotonic, i.e. the more locks that are open, the more flows are allowed.

properties about flows, but the state is not actually represented at runtime. For this reason we also do not need to consider potential covert channels introduced by the flow lock state.

3.2 Some Intuitions About Flow-Lock Security

Before we define our type system, it is useful to get some intuitions about which programs we deem secure/insecure. At this point we only concern ourselves with information leaks arising from direct or indirect data flows. In particular we will not consider timing or termination sensitivity.

A few small example programs are presented on the right. All of these contain insecure direct data flows, except (3). In (1) the contents of $m_{\{B\}}$ may only be read by B, but we are attempting to leak them into a location readable by A. Same thing goes for (2) — even though B can read the contents of the target location, we are still leaking the contents of $m_{\{B\}}$ to A. The simple pattern is that we may not write data to a memory location if that location may be read by someone who cannot already access the data. What's more, this should hold for future time as well. Thus if a reader could access the data from the location we are writing to in some future state, that reader must also have access to the data that is being written, in that same state. Thus the example $m_{\{\sigma \Rightarrow A\}} := !\ell_{\{\sigma \Rightarrow A\}}$ is secure while program (4) is not. In program (5) we attempt to take data not yet readable by A, and put it in a location where A could read it right away. This should clearly not be allowed for the same reasons as for (4).

- (1) $\ell_{\{A\}} := !m_{\{B\}}$
- (2) $\ell_{\{A;B\}} := !m_{\{B\}}$
- (3) $\ell_{\{A\}} := !m_{\{A;B\}}$
- (4) $\ell_{\{\sigma \Rightarrow A;B\}} := !m_{\{B\}}$
- (5) $\ell_{\{A\}} := !m_{\{\sigma \Rightarrow A\}}$

The lock state in effect at the point of the assignment determines its validity, so the programs (6) and (7) are secure. However, we also want a program like (8) below to be considered secure, so we should take the policy of data read from some memory location to be the policy on the location, but taking into account the current state.

- (6) **open** σ ; $\ell_{\{A\}} := !m_{\{\sigma \Rightarrow A\}}$
- (7) $\ell_{\{A\}} := (\mathbf{open} \ \sigma; !m_{\{\sigma \Rightarrow A\}})$

$$(8) \quad \ell_{\{A\}} := \mathbf{let} \ x = (\mathbf{open} \ \sigma; !m_{\{\sigma \Rightarrow A\}}) \ \mathbf{in} \ (\mathbf{close} \ \sigma; x)$$

In program (8) above, the data read from the reference will thus have the policy $\{A\}$ and not $\{\sigma \Rightarrow A\}$, since it is read in a state where σ is open.

Putting all this slightly more formally, data may be written to a memory location if and only if the policy on the location is at least as restrictive as the one on the data, with respect to the state in effect at the point of the assignment. We give a formal definition of this in the next section.

We must also handle indirect flows that arise from various branching situations. A very simple example program containing an invalid indirect flow is

$$(9) \quad \mathbf{if} \ !\ell_{\{A\}} \ \mathbf{then} \ m_{\{B\}} := \mathbf{true} \ \mathbf{else} \ m_{\{B\}} := \mathbf{false}$$

This program is obviously insecure since it will leak the value of $\ell_{\{A\}}$ into $m_{\{B\}}$, but for some programs it is not so easy to tell. Consider the three programs

- (10) **if** $!\ell_{\{\sigma \Rightarrow A\}}$ **then** (**open** σ ; $m_{\{A\}} := \text{true}$) **else** (**open** σ ; $m_{\{A\}} := \text{false}$)
 (11) **if** $!\ell_{\{\sigma \Rightarrow A\}}$ **then** (**open** σ ; $m_{\{A\}} := \text{true}$; **close** σ) **else** ()
 (12) **if** (**open** σ ; $!\ell_{\{\sigma \Rightarrow A\}}$) **then** (**close** σ ; $m_{\{A\}} := \text{true}$) **else** ()

Program (10) could be argued correct since at the points where we leak the information to A , i.e. the assignments, the state allows A to access the result of the branching conditional directly, and hence the leak is secure.

However, as program (11) shows it is not that simple. If the second branch in (11) is chosen, the value of the condition is still leaked to A by the absence of a write, but at no point does the state allow the flow. The leaks come from knowing which of the two branches is taken, which suggests that the leak actually occurs at the branch point. Thus it is the policy of the condition, taken in the state in effect at the branch point, that decides what writes the branches may perform. This means that (9), (10) and (11) are all insecure, while (12) is secure even though the lock is closed again before the write.

Another possible source of indirect leaks is function application. If the function itself is secret, an attacker could still get information about what that function is by observing its effects, just like he could know which branch was taken by observing the effects of a conditional expression. Thus in a sense we can view function application as a kind of branching.

Consider the programs (13) – (19). In the program (13) we must ensure that the function read from the reference does not write to locations visible by anyone other than A , otherwise we could leak information about which function that was used. As an example, if the function read from $\ell_{\{A\}}$ in (13) is $(\lambda x. m_{\{B\}} := 1)$ or $(\lambda x. m_{\{B\}} := 2)$, B can determine which of the two that was used by reading $m_{\{B\}}$. We treat the application point in the same way as the branch point of a conditional, so in program (14) the body of the function must not write to a location directly visible to A , even if it first opens σ . However, since we have a call-by-value semantics, in program (15) the function body may perform writes to locations directly visible to A , even if it first closes σ , since σ will be open at the application point.

- (13) $(!\ell_{\{A\}}) ()$
 (14) $(!\ell_{\{\sigma \Rightarrow A\}}) ()$
 (15) $(!\ell_{\{\sigma \Rightarrow A\}}) (\text{open } \sigma; ())$
 (16) $(!\ell_{\{A\}}) := 0$
 (17) $(!\ell_{\{\sigma \Rightarrow A\}}) := (\text{open } \sigma; 0)$
 (18) $(\lambda x. \ell_{\{B\}} := x) (!\ell_{\{A\}})$
 (19) $(\lambda x. \ell_{\{B\}} := 0) (!\ell_{\{A\}})$

A similar situation is assignment to a reference that in turn has been read from a reference, as illustrated in program (16) which should be disallowed if the reference read from $\ell_{\{A\}}$ is visible to anyone other than A . In particular, the contents of $\ell_{\{A\}}$ could be $m_{\{B\}}$ or $n_{\{B\}}$, in which case B can determine the contents of $\ell_{\{A\}}$ by checking which of the two latter locations that contain the value 0. However, just as for application, program (17) is secure if the reference assigned to has policy $\{A\}$, or any policy that is more restrictive than $\{A\}$, since σ is opened before the assignment takes place.

We also need to look at how functions handle the values passed to them as arguments. Clearly we want to rule out a direct leak in the function body, as the one in example (18). One solution attempt could be to rule out all functions that write to “low” memory, i.e. locations with less restrictive policies than the one placed on the argument. But this also rules out perfectly secure programs such as (19) which in particular would

mean that we could not derive a sequential composition form as in figure 1 without placing too heavy restrictions on the writing capabilities of the second sub-program. Thus we want our type system to treat these two programs differently — (18) should be deemed insecure, but not (19).

Other issues such as whether our system is termination sensitive or timing sensitive (see [16] for an overview of these concepts) are orthogonal to the above discussion. We choose to develop a type system and semantics for termination and timing insensitive security. Termination insensitivity makes the type system simpler but the semantics more complex.

3.3 The Type System

Now we have all the intuition needed to construct the type system. We choose to model our system as a type and effect system in the style of Almeida Matos and Boudol [2]. This means in particular that all expressions will be given a *reading effect* and a *writing effect*. In our system the reading effect of an expression is a policy which states who may read the result of that expression, and in what lock states they may do so. The writing effect is also a policy, which records which actors and in what lock states they can see the memory effect of the expression's execution. Type judgments then have the form

$$\Gamma; \Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'$$

- Γ is a typing environment for variables giving a type and policy for each variable.
- Σ is the state, i.e. the set of locks currently open.
- τ is the type of the term.
- (r, w) are the reading and writing effects of the term, both on the form of policies
- Σ' is the state the program will be in after evaluating the term

First we need to define a few operators on policies that we will use in the typing rules. The aforementioned ordering of how restrictive policies are is defined as

$$p_1 \preceq p_2 \equiv \forall (\Sigma_2 \Rightarrow A) \in p_2. \exists (\Sigma_1 \Rightarrow A) \in p_1. \Sigma_1 \subseteq \Sigma_2$$

Read out, we say that p_1 is less restrictive than p_2 if and only if every clause in p_2 is matched by a clause in p_1 for the same A with a less restrictive guard (one with no additional locks). From the logical perspective, this ordering corresponds directly to implication. The most restrictive policy is $\{\}$, also written \top , and data with this policy can never be accessed by anyone. On the other end of the spectrum is \perp , defined as the set of all actors in the system. In other words, data marked with \perp can be read by everyone at all times.

To join two policies means combining their respective clauses, thereby forming the logical disjunction. We define

$$p_1 \sqcup p_2 \equiv \{ \Sigma_1 \cup \Sigma_2 \Rightarrow A \mid \Sigma_1 \Rightarrow A \in p_1, \Sigma_2 \Rightarrow A \in p_2 \}$$

It should be intuitively clear that the join of two policies is at least as restrictive as each of the two operands, i.e. $p \preceq p \sqcup p'$ for all p, p' . In contrast, forming the union of two policies, i.e. the meet, corresponding to \sqcap or logical conjunction, makes the result less

$$\begin{array}{c}
\frac{}{\Gamma; \Sigma \vdash n : \text{int}, (\perp, \top) \Rightarrow \Sigma} \quad \frac{}{\Gamma; \Sigma \vdash b : \text{bool}, (\perp, \top) \Rightarrow \Sigma} \\
\\
\frac{}{\Gamma; \Sigma \vdash u_{p,\tau} : \text{ref}_p \tau, (\perp, \top) \Rightarrow \Sigma} \quad \frac{}{\Gamma; \Sigma \vdash () : \text{unit}, (\perp, \top) \Rightarrow \Sigma} \\
\\
\frac{\Gamma, x : (\tau, r_\alpha); \Delta \vdash M : \tau', (r, w) \Rightarrow \Delta'}{\Gamma; \Sigma \vdash \lambda x. M : (\tau, r_\alpha) \xrightarrow{\Delta, r, w, \Delta'} \tau', (\perp, \top) \Rightarrow \Sigma} \quad \frac{x : (\tau, r) \in \Gamma}{\Gamma; \Sigma \vdash x : \tau, (r(\Sigma), \top) \Rightarrow \Sigma} \\
\\
\frac{}{\Gamma; \Sigma \vdash \text{open } \sigma : \text{unit}, (\perp, \top) \Rightarrow \Sigma \cup \{\sigma\}} \quad \frac{}{\Gamma; \Sigma \vdash \text{close } \sigma : \text{unit}, (\perp, \top) \Rightarrow \Sigma \setminus \{\sigma\}} \\
\\
\frac{\Gamma, x : (\tau, r); \Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma}{\Gamma; \Sigma \vdash \text{rec } x. M : \tau, (r, w) \Rightarrow \Sigma} \\
\\
\frac{\Gamma; \Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'}{\Gamma; \Sigma \vdash \text{ref}_p M : \text{ref}_p \tau, (r, w) \Rightarrow \Sigma'} \quad \frac{\Gamma; \Sigma \vdash M : \text{ref}_p \tau, (r, w) \Rightarrow \Sigma'}{\Gamma; \Sigma \vdash !M : \tau, (r \sqcup p(\Sigma'), w) \Rightarrow \Sigma'} \\
\\
\frac{\Gamma; \Sigma \vdash M_1 : \text{ref}_p \tau, (r_1, w_1) \Rightarrow \Sigma' \quad \Gamma; \Sigma' \vdash M_2 : \tau, (r_2, w_2) \Rightarrow \Sigma''}{\Gamma; \Sigma \vdash M_1 := M_2 : \text{unit}, (\perp, w_1 \sqcap w_2 \sqcap p) \Rightarrow \Sigma''} \quad r_1(\Sigma'') \sqcup r_2(\Sigma'') \preceq p \\
\\
\frac{\Gamma; \Sigma \vdash M_0 : \text{bool}, (r_0, w_0) \Rightarrow \Sigma' \quad \Gamma; \Sigma' \vdash M_i : \tau, (r_i, w_i) \Rightarrow \Sigma_i \quad r_0(\Sigma') \preceq w_1 \sqcap w_2}{\Gamma; \Sigma \vdash \text{if } M_0 \text{ then } M_1 \text{ else } M_2 : \tau, (r_0 \sqcup r_1 \sqcup r_2, w_0 \sqcap w_1 \sqcap w_2) \Rightarrow \Sigma_1 \sqcap \Sigma_2} \\
\\
\frac{\Gamma; \Sigma \vdash M_1 : (\tau, r_2) \xrightarrow{\Sigma_2, r_f, w_f, \Sigma_3} \tau', (r_1, w_1) \Rightarrow \Sigma_1 \quad \Gamma; \Sigma_1 \vdash M_2 : \tau, (r_2, w_2) \Rightarrow \Sigma_2}{\Gamma; \Sigma \vdash M_1 M_2 : \tau', (r_1 \sqcup r_f, w_1 \sqcap w_2 \sqcap w_f) \Rightarrow \Sigma_3} \quad r_1(\Sigma_2) \preceq w_f
\end{array}$$

Fig. 2. Type and Effect system

restrictive, so we have $p \sqcap p' \preceq p$ for all p, p' . Both \sqcap and \sqcup are clearly commutative and associative.

Finally we need to define using a policy with respect to a particular state, or normalising to a state. We say that policy p normalised at state Σ is

$$p(\Sigma) \equiv \{\Sigma' \setminus \Sigma \Rightarrow A \mid \Sigma' \Rightarrow A \in p\}$$

Informally, we remove all open locks from all guards in p , since these no longer restrict data governed by p . This function is antimonotonic, so $\Sigma \subseteq \Sigma' \implies p(\Sigma') \preceq p(\Sigma)$, and in particular $p(\Sigma) \preceq p$ for all Σ . Logically this operation is a partial evaluation, where all variables (locks) that appear in Σ are set to *true* in p .

The type and effect system is presented in Figure 2. The rules for literal values are straight-forward, giving all such values the reading effect bottom. However, from the variable rule we see that variables are given a reading policy. This is used to keep track of the reading policies of function arguments, as can be seen from the rules for abstraction and application, and the purpose is to disallow programs like (18) while still allowing (19). It is important to note that we do *not* check that $r_2(\Sigma_2) \preceq w_f$ in the application rule, since doing so would invalidate program (19). Instead we rely on the

type checking of the body of the function to find any leaks inside it, with the help of the annotation on its parameter.

In the rule for abstractions, we annotate the function arrow with the latent read and write effects that will be accurate for the function body once it is applied. We also annotate the arrow with the state that the program will be in at the application point, and the state the program will be in after evaluating the body. The interpretation of a function with type $(\tau, r_\alpha) \xrightarrow{\Delta, r, w, \Delta'} \tau'$ is thus that when applied in state Δ on an argument of type τ and with reading policy r_α , it will produce a result of type τ' with reading policy r . The writing policy w states who could see that the function has been applied, and the whole program will be in state Δ' afterwards. This is all mirrored by the appropriate states in the application rule.

Direct leaks, like the ones in programs (1), (2), (4) and (5), are handled by the check $r_2(\Sigma'') \preceq p$ in the rule for assignment. Since we normalise the policy r_2 of the assignee to the state in effect at the point of the assignment, program (5) would be secure if run in a state where σ is open, which is exactly what happens in programs (6) and (7). Also the normalisation to the current state in the dereferencing rule, i.e. $p(\Sigma')$ in the reading effect of the conclusion, means that program (8) will be deemed secure. The same kind of normalisation also appears in the variable rule.

The check $r_0(\Sigma') \preceq w_1 \sqcap w_2$ in the conditional rule will ensure that an indirect leak like the one in (9) will not be allowed. The normalisation of r_0 to Σ' means that it is the state at the branch point that is important, which disallows (10) and (11) but lets (12) through. The branches may open and close different locks, so the end states can differ. Since policies are monotonic, we can use the intersection of the end states as a safe approximation for the following program.

The checks $r_1(\Sigma'') \preceq p$ in the assignment rule, and the corresponding $r_1(\Sigma_2) \preceq w_f$ in the application rule handle indirect flows like in (13), (14) and (16), but allow (15) and (17).

In the assignment rule, the reading effect in the conclusion is \perp . The reason is that the result of an assignment is always $()$, independent of the result values of the two expressions M_1 and M_2 , so no information is leaked by making the $()$ result public. For similar reasons, r_2 does not show up in the reading effect in the conclusion of the application rule. Since function arguments are annotated with their reading effects, if the result of M_2 has any effect on the result of the whole application expression, this fact will be seen through r_f .⁴

4 Semantic Security Properties

For reasons of space this section gives only a brief outline of the semantic definitions and results about flow-lock security. For details the reader is referred to the full version of the paper. The main development is the definition of a notion of *flow lock security* which

⁴ The rules involving functions are fairly restrictive as they are formulated here. One could easily imagine various forms of subsumption, both for lock states and argument policies, that would make the system less restrictive. However, adding subsumption would complicate the overall formulation of the type system, so we leave it for the full version of the paper.

- generalises a standard notion of noninterference, since amongst other things it guarantees that a noninterference property holds for computation between any changes in the flow lock state.
- holds whenever a flow lock program is well-typed – i.e. well-typed programs are flow-lock secure.

The two main challenges in generalising the notion of noninterference to the flow lock setting are (i) dealing with policy change, in particular when a policy become less liberal (i.e. when locks are closed), and (ii) coping with the “latency” of a language with higher-order functions and state.

Before we can deal with policy change we must understand the underlying notion of noninterference that we build upon in the definition of flow lock security. Suppose in a computation that the set of locks Σ are open. This means that an actor A is permitted, at that point, to read the contents of a location with policy c providing $(\Delta \Rightarrow A) \in c$ for some $\Delta \subseteq \Sigma$. If the set of open locks Σ does not change, then the basic noninterference property that we expect is the following: given two memories which agree on all A -readable locations, the results of computing with these two respective memories should also agree on A -readable locations. This means that the actor A does not learn anything about the memory locations that were not visible initially.

Now to deal with change in the set of open locks we follow the “self-bisimulation” approach from [17], whereby security is characterised by a more general property of two programs being bisimilar with respect to the observable parts of memory. One particular feature of the definition from [17] is that the bisimulation is defined over programs and not configurations (program-memory pairs). The idea is that at each step of the bisimulation the pair of programs under comparison are inspected in all pairs of memory states which are indistinguishable to the attacker. This very strong requirement was needed to make the definition of security compositional with respect to parallel composition. But this approach of “resetting” the store at each step has another very useful property: it enables us to reset the state in the event of a policy change. For example, one particular difficulty is that when the current policy becomes *more* restrictive — in our case when locks are closed — then we need a way to reestablish a stronger security requirement at that point in the execution. It is notable that two previous semantic accounts of temporary policy weakening mechanisms, Mantel and Sands’s language based intransitive noninterference condition [8], and Almeida Matos and Boudol’s *nondisclosure* policy [2], both rely on such a “resetting” bisimulation not only to deal with threads, but more importantly to provide a semantics to local policy change mechanisms. Of these two earlier definitions, our definition is close in spirit to Almeida Matos and Boudol’s — although we refer to the full paper for details.

A straightforward “resetting” bisimulation is not enough to define flow lock security; it is not enough to consider just the locations which are currently visible to an actor A . Consider a program such as $\ell_{\{\sigma \Rightarrow A\}} := !m_{\{\sigma' \Rightarrow A\}}$ in a state where neither σ nor σ' are open. Since this assignment deals with locations not currently visible to A then a simple resetting bisimulation would allow it. However it is clearly insecure with respect to possible *future* states which may open σ . In order to detect the insecure flow that might be revealed at some future time we must check the equivalence of the two memories in a state where σ is open but σ' is not. More generally, the definition

of flow-lock security therefore takes into account all possible (more permissive) future lock states.

For the full details of these developments the reader is referred to the extended version of this paper.

5 Relating to Other Systems and Idioms

Standard Noninterference. As a first example of the expressiveness of our system, consider a standard termination insensitive noninterference property for a lattice-based security model in the standard Denning style [6].

In this setting we have a lattice of security levels $\langle \mathcal{L}, \sqsubseteq, \sqcup \rangle$, and a policy level: $\text{Loc} \rightarrow \mathcal{L}$ that fixes the intended security level of the storage locations in the program. Given such a policy we can define noninterference. For simplicity we consider closed programs of unit type which do not perform any storage allocation or lock open/close operations. In what follows let metavariables P and Q range over such programs.

Definition 1 (Noninterference). *Given two stores μ and ν , and a level $k \in \mathcal{L}$, define μ and ν to be indistinguishable at level k , written $\mu =_k \nu$, iff for all ℓ such that $\text{level}(\ell) \sqsubseteq k$ we have $\mu(\ell) = \nu(\ell)$.*

Then we say that P is noninterfering if for all k , whenever $\langle P, \mu \rangle \rightarrow^ \langle (), \mu' \rangle$ and $\langle P, \nu \rangle \rightarrow^* \langle (), \nu' \rangle$, then $\mu =_k \nu$ implies $\mu' =_k \nu'$.*

To represent a lattice policy we do not need any locks; we represent the reading level of a variable by the set of levels at which it may be read. Thus the policy for a storage location ℓ is the upwards closure of its lattice level, written $\uparrow \text{level}(\ell)$, where $\uparrow j = \{\{\} \Rightarrow k \mid k \sqsupseteq j\}$. Given this, we have the following:

Theorem 1. *If P is flow lock secure then P is noninterfering.*

Thus whenever we show that P is secure in the flow lock setting then it is also noninterfering. But it is perhaps not too surprising that our security specification is stronger than standard noninterference. A reasonable concern might be that the definition, or indeed the type system, is too strong to be useful. Here we show that despite being stronger, we are still able to type just as much as “typical” systems for regular noninterference.

Figure 3 presents a simple type system for a while language which can be seen as a straightforward reformulation of the typing system presented by Volpano, Irvine and Smith [21].

$$\begin{array}{c}
 \frac{p = \bigsqcup_{\ell \in E} \text{level}(\ell).}{\vdash_{NI} E : p} \quad \frac{\vdash_{NI} E : q \quad p \sqcup q \sqsubseteq \text{level}(\ell)}{p \vdash_{NI} u := E} \quad \frac{p \vdash_{NI} C_1 \quad p \vdash_{NI} C_2}{p \vdash_{NI} C_1; C_2} \\
 \frac{\vdash_{NI} E : q \quad p \sqcup q \vdash_{NI} C_i \quad i = 1, 2}{p \vdash_{NI} \text{if } E \text{ then } C_1 \text{ else } C_2} \quad \frac{\vdash_{NI} E : q \quad p \sqcup q \vdash_{NI} C}{p \vdash_{NI} \text{while } (E) C}
 \end{array}$$

Fig. 3. Standard Noninterference Type System

Define the following translation $\lceil \cdot \rceil$ from terms in the while language to λ_{FL} :

$$\begin{aligned}
\lceil \text{while } (E) C \rceil &= \text{rec } x. \text{if } \lceil E \rceil \text{ then } \lceil C \rceil; x \text{ else } () \\
\lceil \text{if } E \text{ then } C_1 \text{ else } C_2 \rceil &= \text{if } \lceil E \rceil \text{ then } \lceil C_1 \rceil \text{ else } \lceil C_2 \rceil \\
\lceil C_1; C_2 \rceil &= \lceil C_1 \rceil; \lceil C_2 \rceil \\
\lceil \ell := E \rceil &= \ell_p := \lceil E \rceil \quad \text{where } p = \uparrow \text{level}(\ell) \\
\lceil E \rceil &= E' \quad \text{where } E' \text{ is the result of replacing} \\
&\quad \text{each location } \ell \text{ in } E \text{ with } \ell_{\uparrow \text{level}(\ell)}.
\end{aligned}$$

To make our formulations easier, let us restrict the language of expressions to booleans (so we do not have to consider typing issues). Now we can state that whenever something is typeable in the simple noninterference system, a corresponding derivation holds for the flow locks system:

Theorem 2. *Let Γ_0 be the type environment that maps every storage location to *bool*. Then*

1. *If $\vdash_{NI} E : k$ then $\Gamma_0; \emptyset \vdash \lceil E \rceil : \text{bool}, (r, \top) \Rightarrow \emptyset$ where $r = \uparrow k$*
2. *If $pc \vdash_{NI} C$ then $\Gamma_0; \emptyset \vdash \lceil C \rceil : \text{unit}, (r, w) \Rightarrow \emptyset$ where $w \subseteq \uparrow pc$*

We also expect that a similar theorem holds for some suitable termination-insensitive version of DCC [1], although we have not attempted to show this formally.

Simple Declassification. We can encode a simple declassification mechanism in the same Denning-style setting as used in the previous example. The needed extra step is to extend all policies with clauses to allow declassification. For each level j not in the policy already, we introduce a flow lock σ_j representing a declassification to that level. The new policies then look like

$$\{k \mid k \sqsupseteq \text{level}(\ell)\} \cup \{\sigma_j \Rightarrow k \mid j \not\sqsupseteq \text{level}(\ell), k \sqsupseteq j\}$$

We can now define a declassification operator to level j as

$$\text{declassify}_j \equiv (\lambda v. \text{let } x = (\text{open } \sigma_j; v) \text{ in } (\text{close } \sigma_j; x))$$

It is easy to verify from the type system that the only effect of applying this function to some value is that the value will then be readable also at level j , as was our intention.

Lexically Scoped Flows. In the setting of a multilevel security model, Almeida Matos and Boudol describe how to build a system with lexically scoped dynamic flow policies [2]. They start from a λ -calculus with recursion and references like we do, and introduce a construct “*flow $\alpha \prec \beta$ in M* ” that extends the current global flow policy to also allow flows from level α to β in the scope of M . These flows are transitive, so if the current policy already allows flows from say β to γ , flows from α to γ would also be allowed in M .

Modelling scoped flows using flow locks is easy, but the global nature of policies in Almeida Matos and Boudol’s system, as opposed to our local policies on memory locations, needs special treatment. We introduce a lock $\sigma_{\alpha \prec \beta}$ for each pair of levels α and β that data could flow between. Each policy on some data must record the fact that a future flow declaration could allow that data to flow to many new locations due to the transitive nature of flows. Thus if a location in Almeida Matos and Boudol’s system would have level A , we could represent that as

$$A \cup \{ \sigma_{\alpha \prec \beta_0}, \sigma_{\beta_0 \prec \beta_1}, \dots, \sigma_{\beta_{k-1} \prec \beta_k} \Rightarrow \beta_k \mid \alpha \in A, \beta_i \notin A \}$$

where the \notin is taken with respect to some universal set of levels. In effect, each location records all possible future transitive flows from it. We then derive our representation of the “flow” construct that opens a lock in the scope of some subprogram:

$$\text{flow } \sigma \text{ in } M \equiv \text{let } x = (\text{open } \sigma; M) \text{ in } (\text{close } \sigma; x)$$

Almeida Matos and Boudol also include parallel execution in their system, and as a consequence make their type system and semantic security definition, called *non-disclosure*, sensitive to possible non-termination. Our system has no parallel execution so we cannot model their full system, only the sequential subset.

Intransitive Noninterference. Flow locks represent a lower level abstraction than lattice-based information flow models in the sense that the lattice ordering is not “built in” but must be represented explicitly. One advantage of such a lower level view is that it can also represent *intransitive noninterference* policies [15, 14] — i.e. ones in which the flow relation is intentionally not transitive. Since intransitive policies are the default case for flow locks, it is straightforward to represent simple language-based intransitive policies such as the one described by Mantel and Sands [8].

Noninterference Until Declassification. Chong and Myers’ [5] introduce a class of temporal declassification policies. This is achieved by annotating variables with types of the form $k_0 \xrightarrow{c_1} \dots \xrightarrow{c_n} k_n$, which intuitively means that a variable with such an annotation may be successively declassified to the levels k_1, \dots, k_n , and that the conditions c_1, \dots, c_n will hold at the execution of the corresponding declassification points. The exact nature of the conditions are left unspecified, and it is assumed in the type system that these conditions are verified at certain key program points by some external tool.

We can achieve a similar effect fairly naturally using flow locks, where we would use a distinct lock C_i for each condition c_i . One should then insert **open** C_i constructs in the program at points where the intended declassification takes place, and verify (with an external tool) that the corresponding condition c_i does indeed hold at these points, and that lock C_{i-1} has been opened (we assume that locks are never closed in this encoding). The policy above could then be represented as

$$\{k_0; \{C_1\} \Rightarrow k_1; \dots; \{C_1, \dots, C_n\} \Rightarrow k_n\}.$$

Robust Declassification. Information flow may be used to verify integrity properties, to ensure that untrusted (low integrity) data does not influence the values of trusted

(high integrity) data. Since flow lock policies are neutral with respect to whether we are dealing with confidentiality or integrity properties it is no problem to add such integrity policies to data, and we can easily have clauses for integrity and confidentiality in the same policy. The interesting case, however, is the interaction between confidentiality and integrity in the presence of dynamic policies.

Zdancewic and Myers [22] introduced the concept of *robust declassification* to characterise the property that an attacker (who controls low integrity data) cannot influence what is declassified. This guarantees that the attacker cannot manipulate the amount of information which is released through declassification.

In the setting of flow lock policies, “declassification” can be thought of as the process of opening locks, since whenever a lock is opened more flows are enabled. Thus we can interpret robust declassification as the question of whether low integrity data can influence the decision to open locks.⁵

One possible way of enforcing robust declassification using flow locks is to observe the following: since we cannot perform any computation with locks, the only way that an open operation can be influenced by low integrity data is via indirect information flow from low integrity data. Suppose that our policies use an indexed set of locks $\sigma_i, i \in I$ to control confidentiality. These are unguarded (i.e. we ignore *endorsement*). Let us assume that in addition to the actors of the system we have the pseudo-actor *trusted* used to track integrity information, just as we did in Section 2.

In order to prevent indirect flow from low integrity data to the opening of locks, we will log each use of an open operation by writing to a variable *log*. An obvious way to enforce this is to define a “robust” version of open:

$$\mathbf{ropen} \sigma_i \equiv \mathbf{open} \sigma_i; \mathit{log} := i$$

Now we give *log* the policy $\{\mathit{trusted}\}$. This ensures that the assignment is always safe from a confidentiality perspective (since normal actors can never read it anyway), and that the open operation can never have taken place in a low integrity context (since otherwise the assignment would cause information to flow from untrusted to trusted data). Finally, to additionally prevent the declassification of low integrity data we can syntactically enforce that lock-guarded policies are only used on high integrity data.

The Decentralized Label Model. In the Decentralized Label Model (DLM) [10, 11, 12], data is said to be *owned* by a set of principals. These principals may allow other principals to read the data, and the effective reader set is those principals that all owners agree may read the data. Allowing a new reader roughly corresponds to declassification, and we can model it similarly. The DLM also defines a global principal hierarchy, where one principal may allow another principal to *act for* it, which means it may read all the same things. This is very similar in spirit to introducing a new flow in the system by Almeida Matos and Boudol, including transitivity, and we can model it in the same way. Apart from clauses for declassification and hierarchic flows, the policies must also include clauses for the combination of the two, e.g. *A* can read the data if *B* owns it, has declassified it for *C* to read it, and *A* acts for *C*.

⁵ If we also take the view from [13], then we extend this concept with the requirement that we should not be able to declassify low integrity data.

A common extension of the DLM [22, 20, 19] deals with integrity and trust. The interesting part for us is the integration with the principal hierarchy, where if A trusts some data and A acts for B , then B also trusts that data. This can be modelled as the reverse of the normal clauses for transitive flows, and the clauses will be very similar to those for forward flows.

The complete general policy for a DLM variable encoded with flow locks would be fairly large and awkward, so we do not show it here.

Other Related Work. The JFlow language [9], as well as several recent papers [19, 23, 7], supports runtime mechanisms to enforce security in situations where this cannot be determined statically, e.g. permissions on a file that cannot be known at compile time. Our flow locks is a static, compile-time mechanism only, and thus cannot handle these issues.

Banerjee and Naumann [4] describe a combination of stack-based access control and information flow types to allow the static checking of policies such as “the method returns a result at level L unless the caller has permission p ”. It may be possible to encode these kinds of policies in a straightforward way using flow locks, but this remains a topic for future work.

6 Conclusions and Future Work

Flow locks are a very simple mechanism that generalises many existing systems and idioms for dynamic information flow policies. We have only just started looking at flow locks however, and much remains to be done.

To really establish flow locks as a core calculus, we need to show more formally how to embed other systems and idioms, and prove that our semantic condition is sufficiently strong compared to the semantic conditions of these other systems. It would also be worthwhile to look at extensions of our core system, in order to handle systems that we definitely cannot model at this point. Examples of such systems include the parallel execution of Almeida Matos and Boudol [2], and also systems that use various runtime mechanisms [19, 23, 7].

Furthermore, we would need to investigate how to implement the flow locks system as a programming language, and to determine what kinds of inference would be needed for policies and locks. Also, flow locks are fairly low-level in nature, being a raw mechanism for controlling data flows in a program. As such it is nontrivial to write and maintain correct flow lock programs. It would therefore be useful to look at what higher-level abstractions and design patterns that could be used together with flow locks. There exists some work specifically targeting the question of patterns, for instance the *seal* pattern by Askarov and Sabelfeld [3].

Acknowledgements. Thanks to Ulf Norell and our colleagues in the ProSec group for helpful comments, and to the anonymous referees for numerous helpful comments and suggestions. This work was partly supported by the Swedish research agencies SSF, VR and Vinnova, and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, Jan. 1999.
2. A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005.
3. A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. European Symp. on Research in Computer Security*, volume 3679 of *LNCS*, 2005.
4. A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, Mar. 2005.
5. S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, Oct. 2004.
6. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
7. M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Proc. Foundations of Computer Security Workshop*, 2005.
8. H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, Nov. 2004.
9. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, Jan. 1999.
10. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, Oct. 1997.
11. A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.
12. A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
13. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004.
14. S. Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symp. on Security and Privacy*, pages 102–113, May 1995.
15. J. M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, 1992.
16. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
17. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.
18. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, 2005.
19. S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *Proc. Symposium on Security and Privacy*, 2004.
20. S. Tse and S. Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. European Symp. on Programming*, volume 3444 of *LNCS*, pages 279–294. Springer-Verlag, Apr. 2005.
21. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
22. S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.
23. L. Zheng and A. Myers. Dynamic security labels and noninterference. In *Proc. Workshop on Formal Aspects in Security and Trust*, 2004.

A Basic Contract Language for Web Services^{*}

Samuele Carpineti and Cosimo Laneve

Department of Computer Science, University of Bologna, Italy
{carpineti, laneve}@cs.unibo.it

Abstract. We design a schema language that includes channel schemas with capabilities of input, output, and input-output. These schemas may describe documents containing references to operations of remote services on the web. In this language, the subschema relation turns out to have an exponential cost. We therefore discuss a language restriction that admits a subschema relation with a polynomial cost.

1 Introduction

Several schema languages have been recently proposed for describing the tree-structure of XML documents. We recall DTD [12], XML-Schema [9], RELAX NG [5], and XDuce types [7] and we refer to [13] for an analysis of their expressiveness. These schema languages are used in WSDL [11, 10] documents that are interfaces of web-services describing the messages sent and/or received by the services and the informations for reaching the services (location, transport protocol, etc.). For example, the one-way operation in WSDL (we are omitting some details)

```
<portType name="op-one-way">
  <operation name="one-way">
    <input message="Real"/>
  </operation>
</portType>
<service name="one-way-service">
  <port name="op-one-way">
    <address location="http://example.com/op-one-way"/>
  </port>
</service>
```

is expressing that the reference at `http://example.com/op-one-way` may be invoked with documents of schema `Real`. WSDL documents are also used in repositories for selecting appropriate references. In this context, “`http://example.com/op-one-way`” may be returned to queries asking for references that can be invoked with `Integer` (because integers are also reals). A client receiving “`http://example.com/op-one-way`”, besides invoking it, might forward the reference to a third party that, in turn, could invoke `op-one-way` with `Natural`.

^{*} Aspects of this investigation were supported in part by a Microsoft initiative in concurrent computing and web services.

Yet, web-services technologies also require the possibility to express and communicate references to operations of remote services [15] and to verify that the receiver uses the service according to its contract (sending proper data and performing the permitted operations). In facts, these requirements are recognized in the new specification of WSDL [10], which extends the schemas with references to interfaces of web-services (called **portTypes**). However this extension is by no means satisfactory because no mechanism for comparing schemas with references is provided at all.

We therefore design a basic schema language with references $\langle S \rangle^i$, $\langle S \rangle^o$, and $\langle S \rangle^{io}$, called *channel schemas*, that collect references of schema S and being respectively used to receive notifications, to invoke services, and for both. In our notation, the channel `http://example.com/op-one-way` has schema $\langle \text{Real} \rangle^o$. The assessment that (channel) schemas are used according to the WSDL description is given by a subschema relation $<:$. Following [2, 14], $<:$ is the largest relation satisfying the closure property “if $S <: T$ then every branch of the syntax tree of S is matched by those of T yielding pairs that are still in $<:$ ”. This matching is actually weakened for tag-labelled branches because, in our schema language, union schemas may be nondeterministic. To illustrate the problem, let $S = a[\text{Int} + \text{String}], c[\text{Int}]$ and $T = a[\text{Int}], c[\text{Int}] + a[\text{String}], c[\text{Int}]$. It turns out that $S <: T$ however, to demonstrate this, one has to pick one addend of T , let it be $T' = a[\text{Int}], c[\text{Int}]$, compute the difference of S and T' , and show that this difference is still in T . In this case the difference is $a[\text{String}], c[\text{Int}]$, which is clearly contained in T .

The relation $<:$ turns out to be computationally expensive – it has an exponential cost with respect to the sizes of the schemas [8]. This is an issue in web-services, where data coming from untrusted parties, such as WSDL documents, might be validated at run-time before processing. While validation has a polynomial cost with respect to the size of the datum in current schema languages, this is not so when data carry references. In these cases, validation has to verify that the schema of the reference conforms with some expected schema, thus reducing itself to the subschema relation. (In **XDuce** run-time subschema checks are avoided because programs are strictly coupled and typechecking guarantees that invalid values cannot be produced.)

To avoid significative run-time degradations of web-services technologies, we impose a language restriction to diminish the cost of the subschema relation. Specifically, following XML Schema, we constrain schemas to retain a deterministic model as regards tag-labelled transitions. The model is still nondeterministic with respect to channel-labelled transitions. The resulting schemas, called *labelled-determined*, are equipped with a subschema relation defined as a set of syntax-directed rules. We prove the equivalence of this subschema relation with $<:$ and we demonstrate that it has a polynomial cost with respect to the sizes of the schemas. This result extends to channel schemas the computational complexity of language difference for deterministic tree automata (and XML Schemas) computed in [8].

Related Works. The schema language studied in this article is similar to those introduced in languages extending π -calculus with XML datatypes [3, 1, 4]. The design of the schema language of [3] has been strongly affected by this study. As a minor difference, channel schemas in [3] only have output capabilities. The schema language of [1] is simpler than the one in this paper. In particular labelled schemas have singleton labels and the subschema relation seems not powerful enough (for example $a[b[]] + c[] \not\leq a[b[]] + a[c[]]$ does not hold in [1]).

The types in [4] include channels with capabilities, union, product, intersection and negation. The definition of subschema is semantic, by means of a set-inclusion on a set-theoretic model. Our schema language is simpler than [4] and the notion of subschema is quite different. For example, in our case, top and bottom are derived schemas and channel schemas may be nested at wish, while this is problematic in presence of recursion and intersection. The contribution [4] overlooks the restrictions for reducing the computational complexity of the subschema relation that turns out to be hyperexponential.

Structure of the Paper. We proceed as follows. Section 2 reviews WSDL and describes how operations may be encoded in our schema language. The schema language with channels is formally described in Section 3. In Section 4 we define the subschema relation \leq and analyze some of its properties. In Section 5 we discuss the constraint of labelled-determinedness and design the alternative syntax-directed subschema definition. We also analyze its algorithmic cost. The appendix is devoted to the proof of equivalence of \leq and the syntax-directed subschema.

2 Encoding WSDL Interfaces

WSDL documents are XML documents that consist of several parts. Among these parts, *ports* are logical groupings of operations that are defined by a name, an interaction pattern, and the schema of messages for invoking the operations and receiving back the answers. Operations may use four interaction patterns: *one-way*, *notification*, *request-response*, and *solicit-response*. The former two model asynchronous unidirectional communications and require a single schema: in one-way, the schema describes the messages to invoke the operation; in notification, the schema describes the messages returned by the invocation. Request-response and solicit-response operations model two communication actions. Therefore they require two schemas. In request-response, the two schemas describe the messages to invoke the operation and to receive the answer, respectively; in solicit-response, schemas are in the other way around.

The one-way operation in WSDL1.1 (we are omitting some details of the WSDL document)

```
<portType name="op-one-way">
  <operation name="one-way">
    <input message="InvokeScm"/>
  </operation>
```

```

</portType>
<service name="one-way-service">
  <port name="op-one-way">
    <address location="http://example.com/op-one-way"/>
  </port>
</service>

```

is expressing that the reference at `http://example.com/op-one-way` may be invoked with documents of schema `InvokeScm`. Technically, the tag `<input message="S"/>` in the WSDL must be interpreted as a schema constructor collecting references that may be invoked with values of schema S , or with subsets of such values. Said otherwise, the constructor `<input message="...">` behaves contravariantly with respect to the argument schema. In our notation, introduced in the next section, the operation `one-way` has schema $\langle \text{InvokeScm} \rangle^o$.

The notification operation is defined by

```

<operation name="notification">
  <output message="ReturnScm"/>
</operation>

```

The intended meaning of this pattern is that the remote service is communicating the schema of the messages it will send back. To receive this message, the client service has to create a reference whose schema in our notation is (greater than) $\langle \text{ReturnScm} \rangle^i$. It is worth to remark that, operationally, the notification is equivalent to delivering a fresh reference of schema $\langle \text{ReturnScm} \rangle^i$ to the client. The capability “ i ” constrains the client to use the reference for receiving messages.

The request-response operation is defined by (as usual, some details of the WSDL document are omitted)

```

<portType name="op-request-response">
  <operation name="request-response">
    <input message="InvokeScm"/>
    <output message="ReturnScm"/>
  </operation>
</portType>
<service name="op-request-response">
  <port name="op-request-response">
    <address location="http://example.com/request-response"/>
  </port>
</service>

```

In this case, the connection with the service at `http://example.com/request-response` is *bidirectional*, that is two references are created: one for invoking the service and the other for receiving the return value. The two have schemas $\langle \text{InvokeScm} \rangle^o$ and $\langle \text{ReturnScm} \rangle^i$, respectively.

Finally, the solicit-response operation is described by

```

<operation name="solicit-response">
  <output message="ReturnScm"/>
  <input message="InvokeScm"/>
</operation>

```

Also in this case two references are created during the connection. The first reference is for receiving solicitations and is described by the schema $\langle \text{ReturnScm} \rangle^i$; the second reference is for responses and is described by $\langle \text{InvokeScm} \rangle^o$.

3 Schemas with Channels

We use two disjoint countably infinite sets: the *tags*, ranged over by a, b, \dots , and the *schema names*, ranged over by $\mathbf{U}, \mathbf{V}, \dots$. The term κ is used to range over i , o , and io . The syntax of our language includes the categories of *labels* and *schemas* defined by the following rules

$L ::=$	label	$S ::=$	schema
a	(tag)	\perp	(empty schema)
\sim	(wildcard label)	$()$	(void schema)
$L + L$	(union)	$\langle S \rangle^\kappa$	(channel schema)
$L \setminus L$	(difference)	$L[S], S$	(labelled sequence schema)
		$S + S$	(union schema)
		\mathbf{U}	(schema name)

Labels. Labels specify collections of tags. The semantics of labels is defined by the following function $\hat{\cdot}$:

$$\hat{a} = \{a\} \quad \hat{\sim} = \{a, b, c, \dots\} \quad \widehat{L + L'} = \hat{L} \cup \hat{L'} \quad \widehat{L \setminus L'} = \hat{L} \setminus \hat{L'}$$

(\sim represents the whole sets of tags). We write $a \in L$ for $a \in \hat{L}$.

Schema. Schemas describe (XML) documents that are structurally similar. The schema \perp describes the empty set of documents; $()$ describes the empty document; $\langle S \rangle^\kappa$ describes references whose messages have schema S and that may be used with *capability* $\kappa \in \{i, o, io\}$. The capabilities i , o , io mean that the reference can be used for performing inputs, outputs, and both inputs and outputs, respectively. The schema $L[S], S'$ describes a sequence starting with a document having a tag in \hat{L} and a document of schema S as content, and followed by a document of schema S' . Finally $S + S'$ describes the set of documents belonging to S or S' . The schema name \mathbf{U} describes the set of documents such that $\mathbf{U} = \mathcal{E}(\mathbf{U})$, where \mathcal{E} is a fixed mapping from names to schemas that fulfills the following *finiteness* and *guardedness properties*. Let $\mathbf{names}(S)$ be the least set containing the schema names in S and such that if $\mathbf{U} \in \mathbf{names}(S)$ then $\mathbf{names}(\mathcal{E}(\mathbf{U})) \subseteq \mathbf{names}(S)$. A map \mathcal{E} is *finite* if, for every $\mathbf{U} \in \mathbf{dom}(\mathcal{E})$, the set $\mathbf{names}(\mathbf{U})$ is finite. A map \mathcal{E} is *guarded* if every occurrence of \mathbf{U} in $\mathcal{E}(\mathbf{U})$ is underneath a channel or labelled sequence schema constructor.

In the following, $L[()]$ and $L[S], ()$ are always abbreviated into $L[]$ and $L[S]$, respectively.

We illustrate the syntax by means of few sample schema name definitions. Let **Bool**, **Blist**, and **Btree** be such that

$$\begin{aligned}
\mathcal{E}(\text{Bool}) &= \text{true}[\] + \text{false}[\] \\
\mathcal{E}(\text{Blist}) &= () + \text{bool}[\text{Bool}], \text{Blist} \\
\mathcal{E}(\text{Btree}) &= () + \text{val}[\text{Bool}], \text{left}[\text{Btree}], \text{right}[\text{Btree}] \\
\mathcal{E}(\text{Empty}) &= a[\text{Empty}]
\end{aligned}$$

The name **Bool** defines booleans that are encoded as tags *true* and *false* with content (). The name **Blist** defines any flat sequence of labelled documents containing booleans; **Btree** defines documents that are binary trees of booleans. The name **Empty** defines an empty set of documents because this set is the least solution of the equation $\text{Empty} = a[\text{Empty}]$. As such **Empty** is equal to \perp .

As regards channel schemas, $\langle \text{Bool} \rangle^o$ describes references that may be invoked with booleans; $\langle \text{Bool} \rangle^{io}$ contains references that may be invoked with booleans *and* may receive notifications carrying booleans. The name **NCbool** defined as

$$\mathcal{E}(\text{NCbool}) = \langle \text{Bool} \rangle^o + \langle \text{NCbool} \rangle^o$$

describes the references to be invoked with booleans or with references to be invoked with booleans, etc., till some finite but not bound depth. (The nesting of channel constructors in [4] is always bound.) We observe that a service querying a repository for references of schema $\langle \text{Bool} \rangle^o$ may get back a service of schema $\langle \text{Bool} \rangle^{io}$ or of schema **NCbool**. Conversely, if the query is about references of schema $\langle \text{Bool} \rangle^{io}$ then the repository will never return references of schema $\langle \text{Bool} \rangle^o$ nor **NCbool**.

Remarks.

1. According to the above grammar, sequences are lists of labelled elements concluded either by the void schema (the empty sequence), or by a channel schema, or by a name (we ignore sequences with a tailing \perp because they are equivalent to \perp , see the forthcoming relation of subschema). Since schema names may only occur in tail position of sequences, it is not possible to define context-free schemas like $a[\]^n, b[\]^n$. Said otherwise, our grammars defines *tree regular schemas*, a class of languages that retain decision algorithms for language inclusion – the subschema relation [8].
2. The subschema language without channel schemas is closed under union, difference, and intersection [7]. Union closure is a consequence of the presence of union schemas; difference closure $S \setminus T$ follows by the fact that labels are represented as sets. For example $L[S], S' \setminus L'[T], T'$ is $(L \setminus L')[S], S' + L[S \setminus T], S' + L[S], S' \setminus T'$. Intersection $S \cap T$ may be defined in terms of difference as $(S + T) \setminus (S \setminus T) \setminus (T \setminus S)$. This sublanguage has a decidable algorithm testing the emptiness of a schema. Thereafter $S \prec T$ may be implemented as an emptiness test on $S \setminus T$. Channel schemas does not preserve the closures under difference and intersection. For this reason these operators are primitive in [4].

4 The Subschema Relation

The semantic definition of subschema in [6] does not adapt well to our language. In that paper, a language for *values* was introduced and a schema S was

considered a subschema of T if the set of values described by S was contained in the set of values described by T . In our case values should contain references that do not carry any “structural” information about their schema. Therefore, in order to verify that a reference belongs to a schema S , we should verify the schema of the reference is a subschema of S . To circumvent this circularity we use an “operational” definition – a *simulation* relation – in the style of [2, 14].

The subschema relation uses handles to manifest all the branches of the syntax tree of a schema. Let μ range over $()$, $\langle S \rangle^\kappa$, $L(S ; T)$ and let $S \downarrow \mu$, read S has a handle μ , be the least relation such that:

$$\begin{aligned} () &\downarrow () \\ \langle S \rangle^\kappa &\downarrow \langle S' \rangle^{\kappa'} \quad \text{if } \widehat{L} \neq \emptyset \text{ and there are } \mu, \mu' \text{ such that } S \downarrow \mu \text{ and } T \downarrow \mu' \\ L[S], T &\downarrow L(S ; T) \quad \text{if } S \downarrow \mu \text{ or } T \downarrow \mu \\ S + T &\downarrow \mu \\ \mathcal{U} &\downarrow \mu \end{aligned}$$

We observe that \perp has no handle. The schema $a[\]$, \perp has no handle as well; the reason is that a sequence has a handle provided that every element of the sequence has a handle. We also remark that a channel $\langle S \rangle^\kappa$ always retains a handle. A schema S is *not-empty* if and only if S has a handle; it is *empty* otherwise.

In the following definition we use the intersection operator on labels: $L \cap L' \stackrel{\text{def}}{=} \sim \setminus ((\sim \setminus L) + (\sim \setminus L'))$.

Definition 1. Let \leq be the least partial order on capabilities such that $io \leq i$ and $io \leq o$. The subschema relation $<:$ is the largest relation on schemas such that $S <: T$ implies:

1. if $S \downarrow ()$ then $T \downarrow ()$;
2. if $S \downarrow \langle S' \rangle^{\kappa'}$ then $T \downarrow \langle T' \rangle^{\kappa'}$ with $\kappa \leq \kappa'$ and one of the followings holds:
 - (a) $\kappa' = o$ and $T' <: S'$;
 - (b) $\kappa' = i$ and $S' <: T'$;
 - (c) $\kappa' = io$ and $S' <: T'$ and $T' <: S'$;
3. if $S \downarrow L(S' ; S'')$ then $T \downarrow L'(T' ; T'')$ with $\widehat{L} \cap \widehat{L}' \neq \emptyset$ and:
 - (a) either $\widehat{L} \subseteq \widehat{L}'$, $S' <: T'$, and $S'' <: T''$;
 - (b) or $(L \setminus L')[S'], S'' + (L \cap L')[R'], S'' + (L \cap L')[S'], R'' <: T$, for some R' and R'' such that $S' <: T' + R'$ and $S'' <: T'' + R''$.

The item 1 constraints greater schema to manifest a void handle if the smaller one retains such a handle. The item 2 reduces the subschema relation on channel schemas to the subschema of the arguments according to the capability. In case of output capability the relation is inverted on the arguments (contravariance), in case of input capability the relation is the same for the arguments (covariance), in case of input-output capability the relation reduces to check the equivalence of the arguments (invariance). The item 3.a allows one to reduce the subschema relation to the schema arguments of handles $-(-; -)$ when the labels of the smaller schema are contained into those of the greater schema. The item 3.b is the

problematic one: it weakens the item 3.a to those cases when the smaller schema shows up a handle $L(S' ; S'')$ and the greater one has no handle $L'(T' ; T'')$ with $L \subseteq L'$ and $S' \prec T'$ and $S'' \prec T''$. To explain item 3.a, we use a schema difference operator “ \setminus ”. (Contrary to [4], our schemas are not closed by difference. This operator is only used for the sake of explanation.) If $T \downarrow L'(T' ; T'')$, in order to prove $L[S'], S'' \prec T$ one may reduce to demonstrate that $(L[S'], S'') \setminus (L'[T'], T'') \prec T$. Such difference of labelled sequence schemas is equal to $(L \setminus L')[S'], S'' + (L \cap L')[S' \setminus T'], S'' + (L \cap L')[S'], (S'' \setminus T'')$. We observe that proving

$$(L \setminus L')[S'], S'' + (L \cap L')[S' \setminus T'], S'' + (L \cap L')[S'], (S'' \setminus T'') \prec T$$

is equivalent to proving

$$\begin{aligned} & (L \setminus L')[S'], S'' + (L \cap L')[R'], S'' + (L \cap L')[S'], R'' \prec T \\ & \text{and } S' \prec T' + R' \\ & \text{and } S'' \prec T'' + R'' \end{aligned}$$

that does not mention the difference operator. This is exactly what 3.b says. Let us illustrate 3.b for deriving $c[a[] + b[], (d[] + e[]) \prec T$, where $T = (c[a[]], d[]) + (c[b[]], (d[] + e[])) + (c[a[]], e[])$. Since $T \downarrow c[a[] ; d[]]$, by 3.b, one may reduce to verifying that $c[R'], (d[] + e[]) + c[a[] + b[], R'' \prec T$ with $R' = b[]$ and $R'' = e[]$. The relationship $c[b[]], (d[] + e[]) \prec T$ follows by 3.a because $c[b[]], (d[] + e[])$ is the second addend of T . As regards $c[a[] + b[], e[] \prec T$ we observe that $T \downarrow c(b[] ; d[] + e[])$. This reduces to $c[a[]], e[] \prec T$, which is true because $c[a[]], e[]$ is the third addend of T .

The schemas **Chan** and **Any** defined as:

$$\begin{aligned} \mathcal{E}(\text{Chan}) &= \langle \perp \rangle^o + \langle \text{Any} \rangle^i \\ \mathcal{E}(\text{Any}) &= () + \sim[\text{Any}], \text{Any} + \text{Chan} \end{aligned}$$

own relevant properties. **Chan** collects all the channel schemas, no matter what they can carry; **Any** collects all the documents, namely possibly empty sequences of documents, including channel schemas, no matter how they are labelled (the label “ \sim ”). We observe that $\langle \perp \rangle^o$ and $\langle \text{Any} \rangle^o$ are very different. $\langle \perp \rangle^o$ collects every reference with either capability “ o ” or “ io ”, $\langle \text{Any} \rangle^o$ refers only to references where that arbitrary data can be sent. For instance $\langle a[] \rangle^o$ is a subschema of $\langle \perp \rangle^o$ but not of $\langle \text{Any} \rangle^o$. The channel schemas $\langle \text{Any} \rangle^i$ and $\langle \perp \rangle^i$ are different as well. $\langle \text{Any} \rangle^i$ refers to references that may receive arbitrary data; $\langle \perp \rangle^i$ refers to a reference that cannot receive anything.

We also remark about differences between labelled schemas and channel schemas. Let $R = a[\text{Blist}] + a[\text{Btree}]$ and $R' = a[\text{Blist} + \text{Btree}]$. Then $R \prec R'$ and $R' \prec R$. However $Q = \langle \text{Blist} \rangle^\kappa + \langle \text{Btree} \rangle^\kappa$ is not subschema-equivalent to $Q' = \langle \text{Blist} + \text{Btree} \rangle^\kappa$. Let us discuss the case $\kappa = i$ that is similar to $L[\cdot]$ because covariant. It is possible to prove that $Q \prec Q'$. However the converse is false because references in Q may be invoked only with documents that are lists of booleans or only with documents that are trees of booleans. Channels in Q' may be invoked with documents belonging either to **Blist** or to **Btree**.

A few properties of \prec are in order.

Proposition 1.

1. \prec is reflexive and transitive;
2. (Contravariance of $\langle \cdot \rangle^o$) $S \prec T$ if and only if $\langle T \rangle^o \prec \langle S \rangle^o$;
3. (Covariance of $\langle \cdot \rangle^i$) $S \prec T$ if and only if $\langle S \rangle^i \prec \langle T \rangle^i$;
4. (Invariance of $\langle \cdot \rangle^{io}$) $S \prec T$ and $T \prec S$ if and only if $\langle S \rangle^{io} \prec \langle T \rangle^{io}$;
5. If S is empty then $S \prec \perp$;
6. For every S , $\perp \prec S \prec \mathbf{Any}$ and $\langle S \rangle^\kappa \prec \mathbf{Chan}$ and $\langle \mathbf{Any} \rangle^{io} \prec \langle S \rangle^o$ and $\langle \perp \rangle^{io} \prec \langle S \rangle^i$.

We discuss Proposition 1.5. The name \perp (as well as **Empty**) has no handle; thereafter it is a subschema of any other schema. To prove $S \prec \mathbf{Any}$, consider the relation $\mathcal{R} = \{(S, \mathbf{Any}) \mid S \text{ is a schema}\}$. It is easy to prove that $\langle \cdot \rangle \prec \mathbf{Any}$ and that $L[S], T \prec \mathbf{Any}$, for every L, S , and T . As regards channel schemas $\langle S \rangle^\kappa$, it suffices to demonstrate that $\langle S \rangle^\kappa \prec \mathbf{Chan}$. By definition of \prec , if $\kappa \leq o$ then $\langle S \rangle^\kappa \leq \langle S \rangle^o$. This fact, $\perp \prec S$, and Proposition 1.2 yield $\kappa \leq o$ implies $\langle S \rangle^\kappa \prec \langle \perp \rangle^o$. If $\kappa = i$ then $\langle S \rangle^i \downarrow \langle \cdot \rangle^i(S)$ and $\mathbf{Chan} \downarrow \langle \cdot \rangle^i \mathbf{Any}$, and we are reduced to $(S, \mathbf{Any}) \in \mathcal{R}$, which is true. We are left with $\langle \mathbf{Any} \rangle^{io} \prec \langle S \rangle^o$ and $\langle \perp \rangle^{io} \prec \langle S \rangle^i$. We detail the former, the last statement is similar. By Proposition 1.2 applied to $S \prec \mathbf{Any}$ we obtain $\langle \mathbf{Any} \rangle^o \prec \langle S \rangle^o$; then by Proposition 1.1 and definition of \prec , we derive $\langle \mathbf{Any} \rangle^{io} \prec \langle S \rangle^o$.

4.1 Primitive Types

The extension of our schema language with primitive types is not difficult. Consider the new syntax:

$T ::=$	primitive types	$S ::=$	schema
n	(integer constant)	\dots	
"s"	(string constant)	T	(primitive types)
Int	(integers)		
String	(strings)		

The primitive types **n**, **"s"**, **Int**, and **String** respectively describe a specific integer, a specific string, the set of integers, and the set of strings. For example, the schema that collects integers and strings is **Int + String**; the schema that collects references with integer messages is $\langle \mathbf{Int} \rangle^i + \langle \mathbf{Int} \rangle^o$. As in XML-Schema, sequences of primitive types are not allowed: in our language every sequence must be composed by labelled elements (except the tailing one).

As regards the subschema relation, the handles are extended with $T \downarrow T$. Let \leq_p be the least partial order on primitive types such that **n** \leq_p **Int** and **"s"** \leq_p **String**. To define the subschema relation for the new language it suffices to extend Definition 1 with

4. if $S \downarrow T$ then $T \downarrow T'$ and $T \leq_p T'$.

It follows that $1 + \mathbf{Int} \prec \mathbf{Int}$ and $a[1 + \mathbf{"bye"}] \prec a[1] + a[\mathbf{"bye"}]$ (the proofs are left to the reader).

5 Labelled-Determined Schema

The relation $<:$ can be verified in exponential time [8]. As we have discussed in the Introduction, this is problematic when $<:$ must be computed at run time, such as when received references must be validated. In this section we study a restriction of the schema language that bears a polynomial subschema algorithm (and validation program). The restriction prevents unions of schemas having a common starting tag and is similar to the restriction used in single-type tree grammars [13] such as XML-Schema. The restrictions also allows an alternative definition of subschema that, instead of examining the potentiality to produce handles, compares the syntactic structure of the schemas.

Definition 2. *The set **ldet** of labelled-determined schemas is the least set containing empty schemas and such that:*

1. $() \in \mathbf{ldet}$;
2. if $S \in \mathbf{ldet}$ then $\langle S \rangle^\kappa \in \mathbf{ldet}$;
3. if $S \in \mathbf{ldet}$ and $T \in \mathbf{ldet}$ then $L[S], T \in \mathbf{ldet}$;
4. if $S \in \mathbf{ldet}$ and $T \in \mathbf{ldet}$ and, for every $S \downarrow L(S' ; S'')$ and $T \downarrow L'(T' ; T'')$, $\widehat{L} \cap \widehat{L}' = \emptyset$ then $S + T \in \mathbf{ldet}$;
5. if $\mathcal{E}(U) \in \mathbf{ldet}$ then $U \in \mathbf{ldet}$.

For example, **Empty** $\in \mathbf{ldet}$ because **ldet** is closed by empty schemas; if $S \in \mathbf{ldet}$ and $T \in \mathbf{ldet}$ then $a[S] + (\sim \setminus a)[T] \in \mathbf{ldet}$ and $\sim[S] + \langle S \rangle^\kappa + \langle T \rangle^{\kappa'} \in \mathbf{ldet}$. The last example displays that union of channel schemas does not invalidate labelled-determinedness. The schemas $a[\] + (a + b)[\]$ and $\langle a[\] + \sim[\] \rangle^\kappa$ are not labelled-determined.

Of course, Definition 1 also holds for labelled-determined schemas. For these schemas $<:$ is much simpler. Item 3 of Definition 1 can be simplified to:

3. if $S \downarrow L(S' ; S'')$ then $T \downarrow L'(T' ; T'')$ with $\widehat{L} \cap \widehat{L}' \neq \emptyset$ and $S' <: T'$, $S'' <: T''$, and $(L \setminus L')[S'], S'' <: T$.

Alternatively, one may also consider the following formulation of item 3 (this is the one that is used in the proof of Theorem 1):

3. if $S \downarrow L(S' ; S'')$ then there is I such that, for every $i \in I$, $T \downarrow L_i(T'_i ; T''_i)$, $\widehat{L} \cap \widehat{L}_i \neq \emptyset$, $\widehat{L} \subseteq \bigcup_{i \in I} \widehat{L}_i$, $S' <: T'_i$, and $S'' <: T''_i$.

However, labelled-determined schemas retain a different, more algorithmic definition of subschema. This definition is presented below as a set of syntax-directed rules defining a relation $S \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}'$ where \mathbf{A} and \mathbf{A}' are sets of pairs (U, R) – the first element is always a schema name – that are used to detect termination. In what follows we abbreviate $S \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}'$ into $S \lesssim_{\mathbf{A}} T$ when we are not interested in \mathbf{A}' .

Let $\mathbf{first}(S) \stackrel{\text{def}}{=} \sum_{S \downarrow L(S' ; S'')} L$.

Definition 3. *The syntax-directed subschema relation $\lesssim_{\mathbf{A}}$ is the smallest relation closed under commutativity of unions and under the rules in Table 1.*

Table 1. The subschema relation \lesssim_A

$\begin{array}{c} \text{(VOID)} \\ () \lesssim_A () \Rightarrow \mathbf{A} \end{array}$	$\begin{array}{c} \text{(BOT)} \\ \perp \lesssim_A T \Rightarrow \mathbf{A} \end{array}$	$\begin{array}{c} \text{(LBOT)} \\ \frac{S \lesssim_A \perp \Rightarrow \mathbf{A}'}{L[S], S' \lesssim_A T \Rightarrow \mathbf{A}'} \end{array}$	$\begin{array}{c} \text{(SBOT)} \\ \frac{S' \lesssim_A \perp \Rightarrow \mathbf{A}'}{L[S], S' \lesssim_A T \Rightarrow \mathbf{A}'} \end{array}$
$\begin{array}{c} \text{(CHAN-I)} \\ \frac{\kappa \leq i \quad S \lesssim_A T \Rightarrow \mathbf{A}'}{\langle S \rangle^\kappa \lesssim_A \langle T \rangle^i \Rightarrow \mathbf{A}'} \end{array}$	$\begin{array}{c} \text{(CHAN-O)} \\ \frac{\kappa \leq o \quad T \lesssim_A S \Rightarrow \mathbf{A}'}{\langle S \rangle^\kappa \lesssim_A \langle T \rangle^o \Rightarrow \mathbf{A}'} \end{array}$	$\begin{array}{c} \text{(CHAN-IO)} \\ \frac{S \lesssim_A T \Rightarrow \mathbf{A}' \quad T \lesssim_{A'} S \Rightarrow \mathbf{A}''}{\langle S \rangle^{io} \lesssim_A \langle T \rangle^{io} \Rightarrow \mathbf{A}''} \end{array}$	
$\begin{array}{c} \text{(RSEQ)} \\ \frac{\hat{L} \subseteq \hat{L}' \quad S \lesssim_A T \Rightarrow \mathbf{A}' \quad S' \lesssim_{A'} T' \Rightarrow \mathbf{A}''}{L[S], S' \lesssim_A L'[T], T' \Rightarrow \mathbf{A}''} \end{array}$			
$\begin{array}{c} \text{(UNIONR)} \\ \frac{S \lesssim_A T \Rightarrow \mathbf{A}'}{S \lesssim_A T + T' \Rightarrow \mathbf{A}'} \end{array}$		$\begin{array}{c} \text{(UNIONL)} \\ \frac{S \lesssim_A T \Rightarrow \mathbf{A}' \quad S' \lesssim_{A'} T' \Rightarrow \mathbf{A}''}{S + S' \lesssim_A T \Rightarrow \mathbf{A}''} \end{array}$	
$\begin{array}{c} \text{(LSEQ)} \\ \frac{\begin{array}{l} L' = \mathbf{first}(T) \quad \emptyset \subsetneq \hat{L} \cap \hat{L}' \subsetneq \hat{L} \\ (L \cap L')[S], S' \lesssim_A T \Rightarrow \mathbf{A}' \quad (L \setminus L')[S], S' \lesssim_{A'} T' \Rightarrow \mathbf{A}'' \end{array}}{L[S], S' \lesssim_A T + T' \Rightarrow \mathbf{A}''} \end{array}$			
$\begin{array}{c} \text{(NAMEL)} \\ \frac{(\mathbf{U}, T) \in \mathbf{A}}{\mathbf{U} \lesssim_A T \Rightarrow \mathbf{A}} \end{array}$	$\begin{array}{c} \text{(NAMEH)} \\ \frac{\mathbf{A}' = \mathbf{A} \cup \{(\mathbf{U}, T)\} \quad \mathcal{E}(\mathbf{U}) \lesssim_{A'} T \Rightarrow \mathbf{A}''}{\mathbf{U} \lesssim_A T \Rightarrow \mathbf{A}''} \end{array}$		$\begin{array}{c} \text{(NAMER)} \\ \frac{S \lesssim_A \mathcal{E}(\mathbf{U}) \Rightarrow \mathbf{A}'}{S \lesssim_A \mathbf{U} \Rightarrow \mathbf{A}'} \end{array}$

The first four rules are simple and do not require any comment. Rules (CHAN-I), (CHAN-O), and (CHAN-IO) reduce subschema to the arguments of the channel constructors; they respectively establish covariant, contravariant, and invariant relationships on the arguments. Rules (RSEQ) and (LSEQ) define the subschema relation for sequences. The former applies if the arguments are already sequences. This rule, together with (UNIONR), permits to single out the sequence branch, if any, of the right argument. However, rules (RSEQ) and (UNIONR) do not suffice for proving that $\sim[()], () <: a[()], () + (\sim \setminus a)[()], ()$. In this case \sim needs to be partitioned and this operation is performed by (LSEQ). It is worth noticing that rule (LSEQ), due to labelled-determinedness, only requires that $(L \cap L')[S], S' \lesssim_A T$, not just $(L \cap L')[R], R' \lesssim_A T$ with $S \lesssim_A R$ or $S' \lesssim_A R'$ (see item 3.b of Definition 1). The last three rules are about schema names. Rule (NAMEL) derives a subschema $\mathbf{U} \lesssim_A T$ if the pair (\mathbf{U}, T) is in the (hypothesis) set \mathbf{A} . Rule (NAMER) unfolds the name \mathbf{U} when it is the right argument. Rule (NAMEH) is the unique one that uses an augmented set in the hypotheses. According to this rule, in order to prove that $\mathbf{U} \lesssim_A T$, one unfolds \mathbf{U} and, at the same time, it is reminded that $\mathbf{U} \lesssim_A T$ is being proved. This remind is stored in \mathbf{A}' . Such a machinery permits to avoid loops: if, during the proof of $\mathbf{U} \lesssim_A T$, one reduces to $\mathbf{U} \lesssim_{A'} T$ then it is possible to

terminate (by rule (NAMEL)). This is the case, for example, when $U \lesssim_{\emptyset} V$ must be proved, with $\mathcal{E}(U) = () + U$ and $\mathcal{E}(V) = () + V$.

The main result of this contribution is the equivalence between $<:$ and \lesssim_A . The proof is technical and detailed in the Appendix.

Theorem 1. *Let S and T be labelled-determined and, for every $(U, R) \in A$, let U and R be labelled-determined and $U <: R$. Then $S \lesssim_A T$ if and only if $S <: T$.*

5.1 The Code of the Syntax-Directed Subschema and Its Computational Complexity

Next we design an algorithm for the syntax-directed subschema relationship and discuss its computational complexity. The algorithm **Alg** is detailed in Table 2. It is a boolean function using two sets of assumptions **At** and **Af** that are implemented as bi-dimensional associative arrays. **At**, similarly to **A**, stores schemas whose subschema relation is either verified or is being verified. However, unlike **A**, **At** also stores generic pairs of schemas, not just pairs (U, T) . **Af** stores schemas whose subschema relation have been already verified to be false. The arrays **At** and **Af** improve the efficiency of **Alg** by preventing that the same subschema relation is verified twice.

Alg is initially invoked with every entry of the arrays **At** and **Af** set to **false** – **Alg** is computing $S \lesssim_{\emptyset} T$ –, with an environment **E** and with the two schemas S and T . **Alg** primarily verifies whether the subschema relation has been already computed – the checks on **At**[S][T] and on **Af**[S][T] at lines (2) and (3) –, and in case returns immediately. These checks implement rule (NAMEL). Otherwise, **Alg** sets **At**[S][T] to **true**, meaning that the pair (S, T) is being verified, and begins the syntax-directed case analysis of the schemas (line (5)). The alternatives of the case analysis from line (6) to line (15) respectively implement the rules (VOID), (BOT), (CHAN-I), (CHAN-O), (CHAN-IO), (RSEQ) and (LSEQ) and (UNIONR), (NAMEH), (UNIONR), and (NAMER).

Line (11) deserves to be spelled out. When S is a labelled schema $L[S'], S''$, the verification is delegated to the auxiliary boolean function **aux_Alg**. This function assumes that the label **L** is nonempty and is always contained into **first**(T), where T is the last argument of **aux_Alg**. Then **aux_Alg** verifies if T may be decomposed into $L'[T'], T'' + R$ such that $L \cap L' \neq \emptyset$, $S' <: T'$, and $S'' <: T''$. In case, **aux_Alg** is recursively invoked with $L \setminus L'$ (see instruction (3) of **aux_Alg**, lines 3 and 4), otherwise **aux_Alg** returns **false**. The assignment **At**[S][T] := **true** in line (4) guarantees the termination of the algorithm in case of nested recursive invocations of **Alg** (with same S and T). This is sound because, by the guardedness property of \mathcal{E} , the recursive invocations in lines (13) or (15) must reduce to execute an instruction from line (6) to (11).

We also remark that **Alg** has no instruction for rules (LBOT) and (SBOT). Indeed, these rules entangle the algorithm (in (11) we should verify that schemas are not empty) and are useless if we assume that every empty schema is rewritten to \perp . Therefore, for the sake of correctness of **Alg** and its computational complexity we assume that empty schemas are always \perp . Later on, we discuss how a schema can be rewritten in order to conform with this constraint.

Table 2. The syntax-directed subschema algorithm

```

bool Alg(At, Af, E, S, T) {
(1)  bool res:= false ;
(2)  if (At[S][T]) res:= true ;
(3)  else if (Af[S][T]) res:= false ;
(4)  else At[S][T]:= true ;
(5)  case S, T of
(6)    () , () : res:= true ;
(7)    ⊥ , ⊥ : res:= true ;
(8)    ⟨S'⟩κ , ⟨T'⟩i : res:= (κ==i or κ==io) and Alg(At,Af,E,S',T') ;
(9)    ⟨S'⟩κ , ⟨T'⟩o : res:= (κ==o or κ==io) and Alg(At,Af,E,T',S') ;
(10)   ⟨S'⟩io , ⟨T'⟩io : res:= Alg(At,Af,E,S',T') and Alg(At,Af,E,T',S') ;
(11)   L[S'],S'' , ⊥ : if (L ⊆ first(T)) then
                                res:= aux_Algo(At,Af,E,L,S',S'',T) ;
                                else res:= false ;
(12)   S' + S'' , ⊥ : res:= Alg(At,Af,E,S',T) and Alg(At,Af,E,S'',T) ;
(13)   U , ⊥ : res:= Alg(At,Af,E,E(U),T) ;
(14)   ⊥ , T' + T'' : res:= Alg(At,Af,E,E,S,T') or Alg(At,Af,E,S,T'') ;
(15)   ⊥ , U : res:= Alg(At,Af,E,S,E(U)) ;
(16)   if (res == false) At[S][T]:= false ; Af[S][T]:= true ;
(17) return res;
}

bool aux_Algo(At, Af, E, L, S', S'', T) {
(1)  case T is
(2)    L'[T'],T'': return(Alg(At,Af,E,S',T') and Alg(At,Af,E,S'',T'')) ;
(3)    T' + T'': if (L ⊆ first(T')) return(aux_Algo(At,Af,E,L,S',S'',T')) ;
                else if (L ∩ first(T') == ∅)
                    return(aux_Algo(At,Af,E,L,S',S'',T''));
                else return(aux_Algo(At,Af,E,L ∩ first(T'), S', S'', T')
                    and aux_Algo(At,Af,E,L \ first(T'), S', S'', T''));
(4)    U: return(aux_Algo(At,Af,E,L,S',S'',E(U)) ;
}

```

Proposition 2. *Alg terminates in polynomial time.*

Proof. Let $t(S)$ be the set of subterms of a schema S (see the Appendix for a formal definition) and let $|\cdot|$ be the cardinality function. The dimensions of the arrays At and Af is $|t(S) \cup t(T)| \times |t(S) \cup t(T)| = |t(S) \cup t(T)|^2$. The reason is due to the contravariance of $\langle \cdot \rangle^o$ that may reduce $S <: T$ to $T' <: S'$ where $T' \in t(T)$. Let $[\text{true}] = 1$ and $[\text{false}] = 0$.

Let At_i and Af_i denote the arrays At and Af when one of them has been modified exactly i times. The following invariants are preserved at the end of every line of Alg and aux_Alg :

1. for every S, T : $(\text{At}[S][T] == \text{false})$ or $(\text{Af}[S][T] == \text{false})$, that is true is never stored both in $\text{Af}[S][T]$ and in $\text{At}[S][T]$;

2. for every S, T : if $(\mathbf{Af}_i[S][T] == \text{true})$ then $(\mathbf{Af}_{i+1}[S][T] == \text{true})$, that is **true** is never deleted from **Af**;
3. $\sum_{S,T} [\mathbf{At}_i[S][T]] + [\mathbf{Af}_i[S][T]] \leq \sum_{S,T} [\mathbf{At}_{i+1}[S][T]] + [\mathbf{Af}_{i+1}[S][T]]$ (i.e. the total number of **true**s either grows or remains the same)
4. if $\sum_{S,T} [\mathbf{At}_i[S][T]] + [\mathbf{Af}_i[S][T]] = \sum_{S,T} [\mathbf{At}_{i+1}[S][T]] + [\mathbf{Af}_{i+1}[S][T]]$ then $\sum_{S,T} [\mathbf{Af}_i[S][T]] < [\mathbf{Af}_{i+1}[S][T]]$ (i.e. when the total number of **true**s remains the same then the **true**s in **Af** strictly increase).

We observe that, in the worst case, the algorithm terminates when $\sum_{S,T} [\mathbf{At}[S][T]] + [\mathbf{Af}[S][T]]$ is equal to $|\mathbf{t}(S) \cup \mathbf{t}(T)|^2$. Invariants 3 and 4 guarantee terminations (the number of **true**s either grows or remains the same for at most $|\mathbf{t}(S) \cup \mathbf{t}(T)|^2$ times before terminating). Invariants 1 and 2 state that **true** is never set in the same entry twice and it is never assigned to the same entry of the two arrays. Therefore, there may be at most $|\mathbf{t}(S) \cup \mathbf{t}(T)|^2$ stores of **true** into **At** and each **true** may be “moved” at most once into **Af**. The cost of this movement is proportional to $\max(|\mathbf{t}(S)|, |\mathbf{t}(T)|)$ because the body of **Alg** may parse the structure of one of the schema (function **auxAlg**). This means that the total cost of **Alg** is $O(\max(|\mathbf{t}(S)|, |\mathbf{t}(T)|) \times |\mathbf{t}(S) \cup \mathbf{t}(T)|^2)$. \square

To rewrite empty schemas to \perp we define an algorithm similar to **Alg**. The algorithm takes two associative boolean vectors of size $\mathbf{t}(S)$, **Et** and **Ef**, that are initialized to **false** at the beginning. At each step **true** is either added to **Et** or moved into **Ef**. Base cases are: \perp , $()$, $\langle S \rangle^\kappa$, and S when either **Et**[S] or **Ef**[S]. In case of \perp and in case of **Et**[S], **true** is returned and **Et** is set to **true**; in the other cases **false** is returned, **Et** is set to **false**, and **Ef** to **true**. The recursive cases are for sequences, unions, and schema names. In every case the corresponding value of **Et** is set to **true** and subterms are checked. If the recursive calls determine that the schema is not empty (i.e. the schema definition is not empty for schema names; one of the components is not empty for unions; both the components are not empty for sequences) **Et** is set to **false** and **Ef** is set to **true**, otherwise the schema is considered empty. The cost of this algorithm is $O(|\mathbf{t}(S)|)$. Once **Et** has been computed, the algorithm **Alg** may be modified to verify at every recursive call whether the arguments are empty or not, and in case replace them with \perp .

Acknowledgments. The authors thank Allen Brown for the interesting discussions about XML schema languages.

References

1. L. Acciai and M. Boreale. XPI: a typed process calculus for XML messaging. In *7th Formal Methods for Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *Lecture Notes in Computer Science*, pages 47 – 66. Springer-Verlag, 2005.
2. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

3. A. Brown, C. Laneve, and L. Meredith. **PiDuce**: A process calculus with native xml datatypes. In *2nd International Workshop on Web Services and Formal Methods*, LNCS, 2005.
4. G. Castagna, R. D. Nicola, and D. Varacca. Semantic subtyping for the π -calculus. In *20th IEEE Symposium on Logic in Computer Science (LICS'05)*. IEEE Computer Society, 2005.
5. J. Clark and M. Murata. Relax ng specification. Available on: <http://www.oasis-open.org/committees/relax-ng/spec-20011203.html>, 2001. December, 3rd 2001.
6. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.
7. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.
8. H. Comon et al. Tree automata techniques and applications. At www.grappa.univ-lille3.fr/tata, October, 2002.
9. W3C XML Schema Working Group. XML Schema Part 2: Datatypes Second Edition. Available on: <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html>. W3C Recommendation - October, 28th 2004.
10. Web Services Description Working Group. Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. Available on: <http://www.w3.org/TR/2005/WD-wsdl20-primer-20050510/>. W3C Working Draft 10 May 2005.
11. Web Services Description Working Group. Web Services Description Language (WSDL)1.1. Available on: <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>. W3C Note 15 March 2001.
12. XML Protocol Working Group. Extensible markup language (xml) 1.0 (third edition). Available on: <http://www.w3.org/TR/2004/REC-xml-20040204>, 2004. February, 4th 2004.
13. M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.
14. B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science*, Vol. 6, No. 5, 1996.
15. S. Vinosky. Web service notifications. *IEEE Internet Computing magazine*, 2004.

A The Equivalence of the Two Subschema Relationships

Let $\|S\|_\emptyset$, called the *size* of S , be the function inductively defined as: $\|\perp\|_x = 0$, $\|\mathcal{U}\|_x = 0$ if $\mathcal{U} \in X$, $\|\mathcal{U}\|_x = \|\mathcal{E}(\mathcal{U})\|_{x \cup \{\mathcal{U}\}}$ if $\mathcal{U} \notin X$, $\|\langle S \rangle^\kappa\| = 1 + \|S\|_x$, $\|S + S'\|_x = 1 + \|S\|_x + \|S'\|_x$, and $\|L[S], S'\|_x = 1 + \|S\|_x + \|S'\|_x$. It is easy to verify that $\|S\|_\emptyset = 0$ implies that S is empty. In the following $\|S\|_\emptyset$ will be shortened into $\|S\|$. Let also $\mathfrak{t}(S)$, called the *set of subterms* of S , be the smallest set satisfying the following equations: $\mathfrak{t}(\perp) = \{\perp\}$, $\mathfrak{t}(\mathcal{U}) = \{\mathcal{U}\} \cup \{\mathfrak{t}(\mathcal{E}(\mathcal{U}))\}$, $\mathfrak{t}(\langle S \rangle^\kappa) = \{\langle S \rangle^\kappa\} \cup \mathfrak{t}(S)$, $\mathfrak{t}(L[S], T) = \{L[S], T\} \cup \mathfrak{t}(S) \cup \mathfrak{t}(T)$, and $\mathfrak{t}(S + T) = \{S + T\} \cup \mathfrak{t}(S) \cup \mathfrak{t}(T)$. We note that $\|S\|$ and $\mathfrak{t}(S)$ are different. For instance, $\|S + S\| = 2 * \|S\| + 1$ whilst $\mathfrak{t}(S + S) = \mathfrak{t}(S) \cup \{S + S\}$. We also note that $\text{names}(S) = \{\mathcal{U} \mid \mathcal{U} \in \mathfrak{t}(S)\}$. Finally, let $\text{lsubt}(S, T)$ be the smallest set containing $\mathfrak{t}(S)$, $\mathfrak{t}(T)$, and closed under the following property: if $L[Q], Q' \in \text{lsubt}(S, T)$ and $L'[Q''], Q''' \in \text{lsubt}(S, T)$ and $\emptyset \subsetneq L \setminus L' \subsetneq L$ then $(L \setminus L')[Q], Q' \in \text{lsubt}(S, T)$

and $(L \cap L')[Q], Q' \in \mathbf{1subt}(S, T)$. We observe that $\|S\|$, $\mathbf{t}(S)$, $\mathbf{names}(S)$, and $\mathbf{1subt}(S, T)$ are always finite.

The following properties are immediate consequences of the definition of \lesssim .

Proposition 3. (1) Let $A \subseteq A'$. If $S \lesssim_A T$ then $S \lesssim_{A'} T$; (2) If $S \lesssim_A T \Rightarrow A'$ then $A \subseteq A'$.

Lemma 1. If S is empty then, for every A and T , $S \lesssim_A T \Rightarrow A'$

Proof. We construct a proof of $S \lesssim_A T \Rightarrow A'$. The proof is defined by induction on $\|S\| + |(\mathbf{names}(S) \times \|T\|) \setminus A|$.

The base cases are: $S = \perp$ and $S = \mathbf{U}$ with $(\mathbf{U}, T) \in A$. The first follows from (BOT), the second from (NAMEL). The inductive cases are $\|S\| + |(\mathbf{names}(S) \times \|T\|) \setminus A| = n + 1$ and either (1) $S = S' + S''$ where both S' and S'' are empty, or (2) $S = L[S'], S''$ where S' is empty, or (3) $S = L[S'], S''$ where S'' is empty, or (4) $S = \mathbf{U}$ where $\mathcal{E}(\mathbf{U})$ is empty. Case (1) follows from the inductive hypothesis and from the rule (UNIONL). Cases (2) and (3) follow from inductive hypothesis and from (LBOT) and (SBOT) respectively. As regards case (4), note that, by definition of handle, $\mathcal{E}(\mathbf{U})$ is empty. Then we use either (NAMEL) and we conclude, or (NAMEH) and we are reduced to prove $\mathcal{E}(\mathbf{U}) \lesssim_{A'} T$, with $A' = A \cup \{(\mathbf{U}, T)\}$. This relationship follows by inductive hypothesis because $\|S\| + |(\mathbf{names}(S) \times \|T\|) \setminus A| = n$. \square

Theorem 1. Let S and T be labelled-determined and, for every $(\mathbf{U}, R) \in A$, let \mathbf{U} and R be labelled-determined and $\mathbf{U} <: R$. Then $S \lesssim_A T \Rightarrow A'$ if and only if $S <: T$.

Proof. (\Rightarrow) To prove that $S \lesssim_A T \Rightarrow A'$ implies $S <: T$, we argue by induction on the proof of $S \lesssim_A T \Rightarrow A'$. We focus on the interesting cases.

(lbot) According to (LBOT), the conclusion $L[S'], S'' \lesssim_A T \Rightarrow A'$ has premise $S' \lesssim_A \perp \Rightarrow A'$. By inductive hypothesis applied to such, $S' <: \perp$. Thus $L[S'], S''$ is an empty schema and $L[S'], S'' <: T$ follows by Proposition 1 (items 1, 5 and 6).

(lseq) The conclusion $L[S'], S'' \lesssim_A T' + T''$ of the proof is obtained by the hypothesis

$$(L \cap L')[S], S' \lesssim_A T \Rightarrow A' \quad (1)$$

$$(L \setminus L')[S], S' \lesssim_{A'} T' \Rightarrow A'' \quad (2)$$

By inductive hypothesis applied to (1) and (2) we obtain $(L \cap L')[S]$, $S' <: T'$ and $(L \setminus L')[S], S' <: T''$. By definition of $<:$, the first subschema implies $T' \downarrow L_i(R_i; R'_i)$ for $i \in 1..h$ and $(\widehat{L} \cap \widehat{L}') \subseteq \bigcup_{i \in 1..h} \widehat{L}_i$, $S' <: R_i$ and $S'' <: R'_i$. The subschema $(L \setminus L')[S], S' <: T''$ implies $T'' \downarrow L_j(Q_j; Q'_j)$ for $j \in 1..h'$ and $\widehat{L \setminus L'} \subseteq \bigcup_{j \in 1..h'} \widehat{L}'_j$, $S' <: Q_i$ and $S'' <: Q'_i$. (The definition of $<:$ is reformulated in this way for labelled-determined schemas: see Section 5.) Therefore, we can conclude $L[S'], S'' <: T$.

(namer) According to (NAMER), the conclusion $S \lesssim_A U$ has premise $S \lesssim_A \mathcal{E}(U) \Rightarrow A'$. By the inductive hypothesis, $S \prec: \mathcal{E}(U)$. Being $\mathcal{E}(U) \prec: U$, we conclude $S \prec: U$ by transitivity.

(\Leftarrow) Let $S \prec: T$ and, for every $(U, R) \in A$: $U \prec: R$. To verify that $S \lesssim_A T \Rightarrow A'$ we construct a proof tree. The argument is by induction on the structure of the triple $(n, \|S\|, \|T\|)$, where n is $|\text{names}(S + T) \times \text{lsb}(S + T)| \setminus A|$. The base cases are: (1) $S = \perp$, then, we conclude by (BOT); (2) $T = \perp$, then, by Lemma 1, S is empty and $S \prec: T$ is immediate; (3) $S = U$ and $n = 0$, then $(U, T) \in A$ and we conclude by (NAMEL); (4) $\|S\| = 0$ then S is empty and lemma 1 applies. The inductive cases are discussed with a case analysis on the structure of S .

- If $S = ()$ then $T \downarrow ()$. The proof of $() \lesssim_A T \Rightarrow A'$ is constructed by induction on the derivation of $T \downarrow ()$. Every application of “ $T_1 + T_2 \downarrow ()$ if $T_1 \downarrow ()$ or $T_2 \downarrow ()$ ” corresponds to an instance of (UNIONR); every application of “ $U \downarrow ()$ if $\mathcal{E}(U) \downarrow ()$ ” corresponds to an instance of (NAMER).
- If $S = \langle S' \rangle^\kappa$ then $T \downarrow \diamond^{\kappa'}(T')$. The proof $\langle S' \rangle^\kappa \lesssim_A T$ distinguishes several sub-cases depending on the capabilities. When $\kappa = i$, $S' \prec: T'$ and, by inductive hypothesis, we obtain $S' \lesssim_A T' \Rightarrow A'$. The proof of $S' \lesssim_A T' \Rightarrow A'$ is extended to one of $\langle S' \rangle^\kappa \lesssim_A T \Rightarrow A'$ by arguing on the derivation of $T \downarrow \diamond^i(T')$. The details are similar to the case when the handle is $()$. Same arguments apply when $\kappa = o$ and $\kappa = io$.
- If $S = L[S'], S''$, we assume that both S' and S'' are not empty, otherwise we conclude by Lemma 1. There are two subcases: (1) $T \downarrow L'(T'; T'')$ with $\widehat{L} \subseteq \widehat{L}'$, $S' \prec: T'$, and $S'' \prec: T''$; (2) $T \downarrow L_i(T'_i; T''_i)$, $\widehat{L} \cap \widehat{L}'_i \neq \emptyset$, $L \subseteq \bigcup_{i \in I} \widehat{L}_i$ with $|I| > 1$, $S' \prec: T'_i$, and $S'' \prec: T''_i$. In case (1), $S' \lesssim_A T' \Rightarrow A'$ and of $S'' \lesssim_{A'} T'' \Rightarrow A''$ follow by inductive hypothesis. Then we use the derivation of $T \downarrow L'(T'; T'')$ to complete the proof as in the case of $()$.
In case (2), $T = T' + T''$ with $T' \downarrow L_i(T'_i; T''_i)$, $i \in I'$ and $T'' \downarrow L_i(T'_i; T''_i)$, $i \in I''$ where $I = I' \uplus I''$ and the labels are pairwise disjoint (because T is labelled-determined). We therefore have $S' \prec: T'_i$ and $S'' \prec: T''_i$ for every i . Let $\widehat{L}' = \bigcup_{i \in I'} \widehat{L}_i$, we may derive $(L \cap L')[S'], S'' \prec: T'$, $(L \setminus L')[S'], S'' \prec: T''$, $\emptyset \subsetneq \widehat{L} \cap \widehat{L}' \subsetneq \widehat{L}'$. We conclude by inductive hypothesis and (LSEQ).
- If $S = S' + S''$ then $S' \prec: T$ and $S'' \prec: T$. By inductive hypothesis it is possible to prove $S' \lesssim_A T \Rightarrow A'$ and of $S'' \lesssim_{A'} T \Rightarrow A''$. We conclude by (UNIONL).
- If $S = U$, we may use the rules (NAMEH) or (NAMEL). (NAMEH) allows us to close the branch of the proof tree, (NAMEL) allows us to reduce to one of the previous cases. (NAMEL) unfolds the schema U . Since there are finitely many constants in $\mathcal{E}(U)$ (because \mathcal{E} is finite) (NAMEL) may be used finitely many times in a single branch of the proof tree of $S \lesssim_A T \Rightarrow A'$ before saturating the set A . □

Types for Dynamic Reconfiguration

João Costa Seco and Luís Caires

Departamento de Informática,
Universidade Nova de Lisboa
{Joao.Seco, Luis.Caires}@di.fct.unl.pt

Abstract. We define a core language combining computational and architectural primitives, and study how static typing may be used to ensure safety properties of component composition and dynamic reconfiguration in object-based systems. We show how our language can model typed entities analogous of configuration scripts, makefiles, components, and component instances, where static typing combined with a dynamic type-directed test on the structure of objects can enforce consistency of compositions and atomicity of reconfiguration.

1 Introduction

In current object-oriented programming practice, composition-based modularization seems to have become the most common structuring mechanism, reflecting a shift of programming style from a pure, inheritance-based object-oriented style, towards the so-called “component-based programming” idioms, which favor blackbox composition.

Notwithstanding the proposal of many sophisticated type safe approaches to module and class composition [2, 5, 6, 10], the mechanism most frequently used to structure object-oriented applications in the “component-oriented” style is the ad-hoc assembly of webs of objects, where individual elements refer to each other through references. Since the code for construction of object structures is not distinguished at the programming language level from any other code, static checking of architectural consistency (of the kind found, for example, with ML functors or mix ins) is not performed during type checking, and may cause hard to correct errors to show up only at runtime. Moreover, the widespread use of sophisticated mechanisms such as dynamic loading, and mobile code, in mainstream programming frameworks adds relevance to the issue of finding expressive and safe programming constructs to dynamically build and reconfigure applications by aggregation and replacement of components and objects.

In previous work [13], we had presented a programming calculus with the aim to capture essential ingredients of object-oriented component programming styles, such as explicit context dependence, subtype polymorphism at the level of both components and objects, late composition, and avoidance of inheritance in favor of composition. A type system was also defined, with types assigned to (first-class) components and objects, thus ensuring runtime safety of compositions. However, although in such a model components may be dynamically composed, the structure of objects gets fixed once for all at instantiation time, thus excluding any possibility of dynamic reconfiguration.

In this paper, we present a new core component-oriented programming language, obtained by extending a λ -calculus with imperative records with a minimal set of architectural primitives. Moreover, we develop a type system that statically enforces, besides

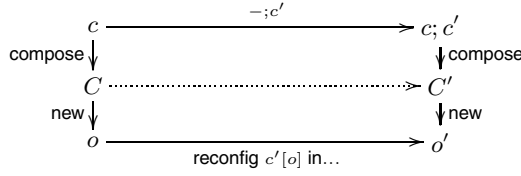


Fig. 1. Composition, Instantiation and Reconfiguration

the absence of more usual runtime errors, consistency of component compositions and atomicity of dynamic reconfiguration.

Our design is semantically motivated by considering a domain of *configurators*, *components*, and *objects*; all such entities are first-class in our model. Intuitively, configurators correspond (by analogy) to the usual notion of “makefile”. Essentially, each configurator contains a series of instructions (architectural primitives) about how to assemble a component. Thus, language expressions that evaluate to configurator values may be seen as counterparts of configuration scripts, the kind of programs used in software configuration management systems to dynamically generate makefiles. Configurators which do not refer to external entities may generate components, by means of a **compose** primitive. Components are linked pieces of code (*cf.*, a class or a module), that may be further composed with other components and scripting code, in configuration scripts, or instantiated, by means of a **new** primitive, to yield objects. Methods can then be called on the appropriate ports of an object, in order to invoke its services. Additionally, configurators may also be applied to objects, by means of a **reconfig** primitive, to dynamically reconfigure their internal structure. The relation between configurators, components and objects, w.r.t. is hinted to in Figure 1. Intuitively, the object o' obtained from instantiating a component constructed from a configurator c and afterwards reconfigured by the configurator c' , is *structurally* indistinguishable from the object instantiated from a component built from the composition of configurators $c; c'$ (although of course not *behaviorally* indistinguishable, since objects are stateful).

In our language, expressions denoting configurators, components and objects are distinguished at the level of typing, rather than at the syntax level, where they may be freely combined. For example, configurator types carry not only extensional but also intensional information, describing the internal architecture of the target component, while component (and object) types are purely extensional as usual, describing only the composition capabilities of a component in terms of required and provided service interfaces. Intensional information is needed to type configurator values, and ensure safety of component composition and dynamic reconfiguration.

It is expected that the soundness of any expressive notion of dynamic reconfiguration will turn out hard to ensure by purely static typing means, if one also wants to preserve object-level information hiding in the programming language. It is therefore important to explore the language design space involving combinations of static and dynamic checking, we believe to have isolated such an interesting combination. Thus, in our present proposal, type checking statically ensures good behavior of configurators, that is, that components built from well-typed configurators are architecturally consistent, and that objects instantiated from well-typed components are free from

runtime errors. Additionally, it is also ensured that objects reconfigured from well-typed configurators will always be architecturally consistent and free from runtime errors. These safety properties are crucial in our model, where both components and configurators are stateless first-class values that can be freely manipulated and composed in a language which is closed under abstraction and application. For example, it is conceivable for a software distribution system to export both a component set and a configuration script to a client, who will later on run the script, after composing it with local configuration information, to produce a certain subsystem. Such a scenario can be easily modeled in our language, in a typeful way.

We now illustrate the fundamental features of our language using a toy example; for the effect we assume to be given notions such as interfaces and method declarations with their standard meanings. Let `ICounter` be the interface type $\{\text{tick} : \text{int} \rightarrow \text{int}\}$, declaring a method `tick`, and consider the following definition of a component `Counter`

```
let Counter = compose(
  provides p:ICounter;
  x[s:int=0, tick:(int → int)=fun y:int → x.s:=s+y];
  plug x into p) in ...
```

As argument of the `compose` operation, we find a configuration expression, namely a sequence of operations each of which introduces a particular element of a counter's architecture. First, a provided port named `p`, then a block of methods named `x` (implementing the method `tick`) and a state variable `s`, and finally a connection between the two, using the `plug` operation. Hence, object instances of component `Counter` will implement a port `p` conforming to the interface type `ICounter`. The type of `Counter` is $\{\} \Rightarrow \{p:ICounter\}$, meaning that it has no required services to be instantiated, and that their instances implement, at port `p`, the interface `ICounter`. Component `Counter` can then be instantiated, yielding an object `o`, by the expression

```
let o = new Counter in (o.p.tick(1); o.p.tick(1))
```

Component `Counter` may also be used as an element to define other components, for example, a `ZeroCounter` component, whose instances will count all calls to `tick` performed with zero as argument.

```
let ZeroCounter = compose(
  provides p:{ tick:int → int };
  c[Counter:{ } ⇒ {p:ICounter }];
  x[tick:int → int =
    fun y → if y=0 then c.p.tick(1)];
  plug x into p) in
```

Here, component `Counter` is inserted in the architecture of `ZeroCounter` under the name `c` (in `c[...]`), and used in the composition context (in `c.p.tick(1)`). The component `ZeroCounter` may then be used to build other components, or instantiated as in

```
let zc = new ZeroCounter in ...
```

Now, suppose that a `ZeroCounter` object, such as `zc`, is running in a server application, and the need arises of extending it with a new service, to reset the inner counter, without shutting it down: clearly, this is a situation calling for a dynamic reconfiguration facility. Consider then the following (re)configuration script:

```

let AddReset =
  (provides r : { reset : unit → unit };
   y [ reset : unit → unit =
     if c.get() > 0 then c.p.tick(-1); y.reset() ];
   plug y into r) in ...

```

Configurator `AddReset` adds a provided port `r`, to expose the new `reset` method, implemented by the method block `y`. Notice that the architectural operations used in the definition of `AddReset` refer to elements (e.g., `c`) which are not declared in the static context of definition (and thus may be seen as white-box operations). However, the context of use is captured at the type level, with configurator `AddReset` being given configurator type

$$\{c \bullet \{p : \text{ICounter}\}\} \Longrightarrow \{c \bullet \{p : \text{ICounter}\}, r \triangleright \{\text{reset} : \text{unit} \rightarrow \text{unit}\}, y \bullet \{\text{reset} : \text{unit} \rightarrow \text{unit}\}\}$$

Configurator types are of the form $K \Longrightarrow K'$, where the two bags of “resources” K and K' describe the change of internal elements in a configuration; each resource is tagged with its (object or interface) type. The type of `AddReset` states that the configurator may be applied in every context where an element `c` of (object) type $\{p : \text{ICounter}\}$ is present (the \bullet resource on the left hand side). It also says that, after application, `c` remains available, alongside with a (new) provided port `r` (the \triangleright resource on the right hand side) and a (new) method block `y`. The following expression

```

reconfig zcr = AddReset[zc] in ... use of zcr... else ... use of zc ...

```

has then the effect of actually reconfigure the object `zc`, returning a properly typed reference `zcr` to the updated object that implements the `reset` service at a new port `r`. In general, a reconfiguration may not be possible, due to a mismatch between the internal structure of the object to be reconfigured (which is not visible to the type system) and the precondition of the configurator. In any case, the type system ensures the atomicity of reconfiguration, i.e. that either the reconfiguration is fully applied as specified by the configurator, and the resulting object is well defined (**in** branch), or the object is not modified (**else** branch). This property is a consequence of static typing, at the level of configurator values, and of a simple and efficient test on type information recorded inside objects, in the spirit of [1], realized at reconfiguration time.

Although not illustrated here, it is possible for a component’s implementation to depend, through a required port, on some external implementations of an interface. Whenever a component with required ports is instantiated or new required ports are added through reconfiguration then both `new` and `reconfig` expressions must provide compatible implementations to each required port. This is achieved by a special `with` clause containing multiple assignments.

Related Work. To the best of our knowledge, the calculus presented in [13] was a first proposal to integrate computation and (first-class) architectural definition in the context of a object-oriented strongly typed programming language. Programming languages supporting first-class components have been studied by several authors [2, 10, 17], although not considering dynamic reconfiguration of instances. More related to our model are the module calculi of [3, 11, 12, 19] which also introduce composition operations for first-class modules and mixins: in these approaches the module language is

stratified on top of a core language. While relying on a different choice of primitives, inherited from our early work [13], we believe that our approach is particularly suitable as a basis for defining component-based languages where computational and configuration / reconfiguration operations may be freely combined (modulo typing constraints) at the same level. More recently, the work in [9] has extended the approach of [3] with a form of dynamic reconfiguration that allows for the interleaved execution of the module manipulation operations and the core language expressions. This seems to correspond to some form of dynamic composition, while we consider the in-place modification of the internal structure of (potentially aliased) stateful objects.

From the perspective of software evolution, several works [4, 8, 16] have addressed the operational semantics and type structure of software systems that support dynamical change of modules. In these approaches, modules implement ADTs, and the focus is on version management of values of such abstract types. In our model, components do not usually represent ADTs but rather service providers, and we concentrate on dynamic reconfiguration of architectures, rather than on individual replacement of a module's implementations.

Forms of dynamic reconfiguration for object-oriented languages involving a fixed predetermined number of future configurations, have also been considered by [7]. In this work, we aimed to model unanticipated reconfiguration using first-class typed notions of (re)configuration scripts, thus following an approach that does not seem to have been explored before. In this context, the fundamental work of [18] on meta-programming and staged programming languages also appears to bear some relation to our development here, even if our focus is on isolating first class semantic entities related to software assembly, rather than on how to express and type source (meta)level program manipulations.

Outline. The remainder of the paper is organized as follows: Section 2 formally presents the language syntax. The operational semantics is introduced in Section 3. In Section 4 we present the type system, and state the main type safety results. Finally, we conclude with some remarks on this work, and suggest possible developments.

2 The Component Calculus

In this section, we introduce λ_χ , a component-based calculus aimed at capturing the programming model motivated above. The language is a simply typed λ -calculus with mutable records enriched with primitives to build and manipulate components. The types of λ_χ are shown in Figure 2. Besides standard functional types, we include types for interfaces and mutable records, components and configurators. Not all type expressions are meaningful, for example, in a component type $\tau \Rightarrow \sigma$, τ and σ are expected to be object types, expressing the required and provided services of the component: the type system presented below will only accept meaningful type expressions.

Configurator types describe the effects of configurators on compositions, expressed in the form of required and provided *resources* (do not confuse with required and provided service *ports*). A resource is represented by a combination of a tag, a name, and a type. The possible tags are: \circ (open), meaning that the resource is unsatisfied, for instance, that a provided port is not connected; \bullet , meaning that the resource is available

$\tau, \sigma ::=$	types
$\tau \rightarrow \sigma$	function
$\{\ell_i : \tau_i \mid i \in 1..n\}$	record
$\{\ell_i : \tau_i \mid i \in 1..n\}$	interface
$\tau \Rightarrow \sigma$	component
$\{r_i \mid i \in 1..n\} \Rightarrow \{r_i \mid i \in 1..m\}$	configurator
$r ::= \pi \circ \tau \mid \pi \bullet \tau \mid \pi \triangleright \tau \mid \pi \triangleleft \tau$	resources
$e ::= x \mid \lambda x : \tau. e \mid e(e) \mid [\ell_i : \tau_i = e_i \mid i \in 1..n] \mid e.\ell \mid e.\ell := e$	
$\mid \text{compose } e \mid \text{new } e \text{ with } \ell_i := e_i \mid i \in 1..n$	
$\mid \text{reconfig } x = e[e] \text{ with } \ell_i := e_i \mid i \in 1..n \text{ in } e \text{ else } e$	
$\mid c$	
$c ::= e; e \mid \text{requires } \ell : \tau \mid \text{provides } \ell : \tau \mid \text{plug } \pi : \tau \text{ into } \pi : \tau$	
$\mid x[e : \tau] \mid x_K[\ell_i : \tau_i = \lambda x_i : \tau_i. e_i \mid i \in 1..n]$	
$\pi ::= x \mid \ell \mid x.\ell$	

Fig. 2. Types and terms for λ_χ

for connection, for instance a certain method block or inner component is present; \triangleright denotes that a provided port is present, and \triangleleft denotes that a required port is present. Typically, at the level of typing, composition operation rewrites a bag of resources into another bag of resources, reflecting the internal change that takes place in the component architecture. In general, we use K to denote resource sets. We also define K_* to be the interface type containing all resources tagged with $*$ in K where $*$ may be \circ , \bullet , \triangleright , or \triangleleft . For example, K_\bullet is $\{\ell_i : \tau_i \mid i \in 1..n\}$ where $\ell_i \bullet \tau_i$ for $i = 1..n$ are all the \bullet -tagged elements in K . We use I, J for interface types and R, P for object types, i.e. interfaces of the form $\{\ell_i : I_i \mid i \in 1..n\}$. We denote by $- \oplus -$ the concatenation operation on interfaces, and by $- \# -$ the disjointness predicate for interfaces and resource sets.

We define the syntax of λ_χ on Figure 2, based on a standard formulation for an imperative λ -calculus, enriched with three new imperative expressions of interest, compose, new, and reconfig. Additionally, a set of primitive composition operations are defined (under syntactic category c), each being a canonical configuration script represented at runtime by a configurator. These configurators are typed stateless values programmed to produce a specific structural effect on an architecture, either in the construction of a component or in the reconfiguration of an instance. They are combined under a *white-box* discipline by the composition operation $(e_1; e_2)$, which means that any element introduced by e_1 can be referred and connected to elements introduced by e_2 .

The typed and named ports of a component are declared by $(\text{requires } \ell : \tau)$ to import a service and by $(\text{provides } \ell : \tau)$ to declare a port exporting a service; $(x[e : \tau])$ to introduce in the architecture a component, resulting from evaluating e . Such an element is referred in the composition context by the local name x . Basic building blocks containing method implementations are introduced by $(y_K[\ell_i : \tau_i = \lambda x_i : \tau_i. e_i \mid i \in 1..n])$ and referred by the local name y . Notice that the set of resources K declares explicit architectural dependencies from other elements at the same compositional level, allowing references to them to be made inside the expressions of the methods. Connections between elements in architectures are created by $(\text{plug } \pi_1 : \tau_1 \text{ into } \pi_2 : \tau_2)$ expressions, declaring that method invocations at port π_2 should be redirected to port π_1 .

Given an expression e denoting a configurator with no required resources, $\text{compose } e$ yields a component value which “freezes” the configurator’s architecture in a component value in such a way that it can only be further composed using black-box operations (by means of a composition operation $x[c : \tau]$). Component values can be instantiated, with new , to yield objects. Notice that these objects will be fully operational only if all of their required ports get actually linked to compatible implementations. Such dependencies may either be satisfied in a composition context, or by *plug-assignments* (with clause) in an instantiation expression.

In a reconfiguration expression $\text{reconfig } x = e_1[e_2] \text{ with } \ell_i := e'_i \text{ } i \in 1..n \text{ in } e_3 \text{ else } e_4$, the distinguished occurrence of x is binding, with scope e_3 and e_4 . If a reconfiguration is successful, the e_3 branch will be executed, with x denoting the reconfigured instance (at the “new” type), otherwise the fail branch e_4 will be chosen. Moreover, since the configurator e_1 may add new required ports to the instance new values must be assigned to them by *plug-assignments*.

3 Operational Semantics

In this section, we present the semantics of our language. Technically, this will be accomplished with big-step operational semantics, using judgments of the form $e; S \Downarrow v; S'$, where e is an expression and S a heap, v is the value of e and S' is the resulting heap. An heap S is an assignment of values v to locations l from a set of locations Loc , along standard lines. The values of λ_χ are listed in Figure 3.

$$\begin{aligned} v &::= \lambda x : \tau. e \mid r \mid \text{conf}(\tau, c) \mid \text{comp}(c) \mid (r, r, r)_\Gamma \mid l \mid \text{nil} \\ r &::= \{ \ell_i \mapsto l_i \text{ } i \in 1..n \} \end{aligned}$$

Fig. 3. Evaluation results

As expected, the more basic values are *abstractions*, and mutable *records*, which are in our current context finite mappings from labels to locations. A *configurator* value, of the form $\text{conf}(\tau, c)$, is a pair that packs the runtime representation of a sequence of instructions to construct or change the architecture of a component, with a configurator type τ that specifies a precondition on its application. Thus, configurators embed some type information at runtime, to be used in a dynamic check during the evaluation of reconfiguration expressions. A *component* value, of the form $\text{comp}(c)$, is the runtime representation of a sequence of instructions to construct or change the architecture of a component. Notice that configurator and component values are pure values, while object instances are (of course) stateful entities, constructed as specified by their generating component. An object value (component instance), is a triple of records of the form $(r, e, p)_\Gamma$ where the labels in r refer to its required ports, the labels in e refer to its inner elements, and the labels in p their provided ports. Γ is a local typing environment assigning types to the object’s internal elements, useful for checking the precondition of a configurator. For the sake of simplicity we sometimes refer to an object value s by the single record obtained by concatenating the three records r, e and p in s . Let $s = (r, e, p)_\Gamma$ be an object, we write s_Δ to denote the record r containing the required

ports in s , Γ_s to denote Γ in s and $s_{\triangleleft} \oplus s_{\bullet}$ to denote the concatenation of the records r and e . We write $\Gamma(s_{\triangleright}.\ell) = \tau$ for $s_{\triangleright} = \{\dots \ell \mapsto 1 \dots\}$ and $\Gamma(1) = \tau$.

We write $\{l_i \mapsto v_i \mid i \in 1..n\}$ to define a heap, $S(1)$ to denote the value associated to 1 in S , $S[l \mapsto v]$ to denote a heap S updated with a new relation, and $\text{Dom}(S)$ to denote the domain set of S . We say that a heap S is *closed* if all locations occurring in S are elements of $\text{Dom}(S)$. We define $\text{new}(S) \triangleq 1$ such that $1 \in \text{Loc}(\text{Dom}(S))$, and use it in the operational semantics to denote a fresh memory location. Notice that since locations are values, cyclic chains of locations may potentially exist in a heap, leading from a location to itself after a number of indirections. We say that a location participating in such a cycle is *undefined*. Such cyclic reference chains may only be introduced if components with vacuous connections, connecting a provided port to a required port, are defined.

The rules defining the operational semantics of λ_{χ} are listed in Figures 4, 6, and 7. The “main” judgment form is mutually dependent on a second judgment form, $s; c; S \Downarrow s'; S'$. This defines the application of a composition operation c to a

<div style="display: flex; justify-content: space-around;"> <div style="text-align: left;"> <p>(Eval Value)</p> $v; S \Downarrow v; S$ </div> <div style="text-align: left;"> <p>(Eval Call)</p> $\frac{e_1; S \Downarrow \lambda x : \tau.e; S' \ e_2; S' \Downarrow v; S'' \ e[x \leftarrow v]; S'' \Downarrow v'; S'''}{e_1(e_2); S \Downarrow v'; S'''}$ </div> </div>		
<p>(Eval Record)</p> $\frac{(1, l_i = \text{new}(S) \ \forall i \in 1..n) \quad e_i; S_{i-1} \Downarrow v_i; S_i \ \forall i \in 1..n}{[\ell_i = e_i \mid i \in 1..n]; S_0 \Downarrow 1; S^n[l \mapsto \{\ell_i \mapsto l_i \mid i \in 1..n\}][l_i \mapsto v_i \mid i \in 1..n]}$		
<p>(Eval Assign)</p> $\frac{e_1; S \Downarrow 1; S' \ l' = \text{deref}_{S'}(1) \quad S'(l') = \{\dots, \ell \mapsto 1'', \dots\} \quad e_2; S' \Downarrow v; S''}{e_1.l := e_2; S \Downarrow v; S''[l'' \mapsto v]}$	<p>(Eval Select)</p> $\frac{e; S \Downarrow 1; S' \ l' = \text{deref}_S(1) \quad S'(l') = \{\dots, \ell \mapsto 1'', \dots\}}{e.l; S \Downarrow S'(l''); S'}$	<p>(Eval Compose)</p> $\frac{e; S \Downarrow \text{conf}(\tau, c); S'}{\text{compose } e; S \Downarrow \text{comp}(c); S'}$
<p>(Eval New)</p> $\frac{(s_{\triangleleft} = \{\ell_i \mapsto l_i \mid i \in 1..n\}, \ 1 = \text{new}(S)) \quad e; S \Downarrow \text{comp}(c); S' \ \mathbf{0}; c; S' \Downarrow s; S_0 \quad e_i; S_{i-1} \Downarrow v_i; S_i \ \forall i \in 1..n}{\text{new } e \text{ with } \ell_i := e_i \mid i \in 1..n; S \Downarrow 1; S_n[l \mapsto s][l_i \mapsto v_i \mid i \in 1..n]}$		
<div style="display: flex; justify-content: space-around;"> <div style="text-align: left;"> <p>(Eval Reconfig)</p> $\frac{(s'_{\triangleleft} = s_{\triangleleft} \oplus \{\ell_i \mapsto l_i \mid i \in 1..n\}) \quad e_1; S \Downarrow \text{conf}(K \Rightarrow K', c); S' \quad e_2; S' \Downarrow 1; S_0 \quad s = S_0(1) \quad s // K \quad s; c; S_n \Downarrow s'; S_{n+1} \quad f_i; S_{i-1} \Downarrow v_i; S_i \ \forall i \in 1..n \quad e_3[x \leftarrow 1']; S_{n+1}[l' \mapsto s'][l_i \mapsto v_i \mid i \in 1..n] \Downarrow v; S'''}{\left(\begin{array}{l} \text{reconfig } x = e_1[e_2] \\ \text{with } \ell_i := f_i \mid i \in 1..n \\ \text{in } e_3 \text{ else } e_4 \end{array} \right); S \Downarrow v; S'''}$ </div> <div style="text-align: left;"> <p>(Eval Reconfig Else)</p> $\frac{e_1; S \Downarrow \text{conf}(K \Rightarrow K', c); S' \quad e_2; S' \Downarrow 1; S'' \quad s = S''(1) \quad \neg s // K \quad e_4[x \leftarrow 1]; S'' \Downarrow v; S'''}{\left(\begin{array}{l} \text{reconfig } x = e_1[e_2] \\ \text{with } \ell_i := f_i \mid i \in 1..n \\ \text{in } e_3 \text{ else } e_4 \end{array} \right); S \Downarrow v; S'''}$ </div> </div>		

Fig. 4. Evaluation of computational expressions

(Match Provides)	(Match Requires)	(Match Element)
$\frac{\Gamma_s(s_{\triangleright}.\ell) = \tau \quad s \parallel K}{s \parallel \ell \triangleright \tau, K}$	$\frac{\Gamma_s(s_{\triangleleft}.\ell) = \tau \quad s \parallel K}{s \parallel \ell \triangleleft \tau, K}$	$\frac{\Gamma_s((s_{\triangleleft} \oplus s_{\bullet}).\ell) = \tau \quad s \parallel K}{s \parallel \ell \bullet \tau, K}$
(Match Element Port)	(Match Unsatisfied)	(Match Unsatisfied Port)
$\frac{S(s_{\bullet}.x) = s' \quad \Gamma_{s'}(s'_{\triangleright}.\ell) = \tau \quad s \parallel K}{s \parallel x.\ell \bullet \tau, K}$	$\frac{\Gamma_s(s_{\triangleright}.\ell) = \tau \quad s \parallel K}{s \parallel \ell \circ \tau, K}$	$\frac{S(s_{\bullet}.x) = s' \quad \Gamma_{s'}(s'_{\triangleleft}.\ell) = \tau \quad s \parallel K}{s \parallel x.\ell \circ \tau, K}$

Fig. 5. Rules for matching

(Eval Requires)	(Eval Provides)
$(\sigma = \emptyset \implies \{\ell \bullet \tau, \ell \triangleleft \tau\})$ requires $\ell : \tau; S \downarrow \text{conf}(\sigma, \text{requires } \ell : \tau); S$	$(\sigma = \emptyset \implies \{\ell \circ \tau, \ell \triangleright \tau\})$ provides $\ell : \tau; S \downarrow \text{conf}(\sigma, \text{provides } \ell : \tau); S$
(Eval Plug)	
$(\sigma = \{\pi_2 \circ \tau, \pi_1 \bullet \tau\} \implies \{\pi_1 \bullet \tau\})$ plug $\pi_1 : \tau$ into $\pi_2 : \tau; S \downarrow \text{conf}(\sigma, \text{plug } \pi_1 : \tau \text{ into } \pi_2 : \tau); S$	
(Eval Sequence)	
$\frac{e_1; S \downarrow \text{conf}((K \implies K', K_c), c_1); S' \quad e_2; S \downarrow \text{conf}((K_c, K'' \implies K'''), c_2); S'}{(e_1; e_2); S \downarrow \text{conf}((K, K'' \implies K', K'''), (c_1; c_2)); S'}$	
(Eval Uses)	
$\frac{\left(\tau = \{\ell_i^r : \tau_i \mid i \in 1..n\}, \sigma = \{\ell_j^p : \sigma_j \mid j \in 1..m\} \right)}{K = \{x \bullet \sigma, x.\ell_i^r \circ \tau_i \mid i \in 1..n, x.\ell_j^p \bullet \sigma_j \mid j \in 1..m\}}$ $\frac{e; S \downarrow v; S'}{x[e : \tau \Rightarrow \sigma]; S \downarrow \text{conf}(\emptyset \implies K, x[v : \tau \Rightarrow \sigma]); S'}$	
(Eval Method Block)	
$(\sigma = K \implies K, x \bullet \{\ell_i : \tau_i \mid i \in 1..n\})$ $x_K[\ell_i : \tau_i = v_i \mid i \in 1..n]; S_0 \downarrow \text{conf}(\sigma, x_K[\ell_i : \tau_i = v_i \mid i \in 1..n]); S$	

Fig. 6. Evaluation of Composition Operations

composition context s with relation to a heap S , and resulting in a modified object instance s' and heap S' . s is a partially built instance where the effects of composition operations get accumulated during composition. To dereference a chains of locations in the heap to its target value we introduce the auxiliary function $\text{deref}_S(l)$ that denotes the last location of a chain starting with l . $\text{deref}_S(-)$ is useful to make the mapping from locations to values independent of the number of heap indirections, created during **plug** operations.

We now discuss some key aspects of the operational semantics. For simplicity, in the rules of Figure 6, the basic composition operations *provides*, *requires*, and *plug*, evaluate to themselves, and are directly stored in configurator values along with appropriate intensional type information. Since the fields of method blocks (code) are, by definition values of the language, they may also directly stored inside a configurator value. In the case of the introduction of an inner component ($x[e : \tau]$) the resulting value depends on the evaluation of the inner expression e to a component which then forms a composition operation ($x[v : \tau]$) where v is again a value, then stored in a configurator. The combination of two operations, Rule (Eval Sequence), produces a configurator containing the composition of the two operands. Notice that the new type information

in the sequential composition is obtained from the manifest information of both parts. In general, the type annotations in configurators are constructed in a mechanical way, we will later show that in well-typed programs this computations always succeed.

The evaluation of a `compose`, Rule (Eval Compose), simply tags a configurator as being a closed architecture by means of a `comp` constructor, enclosing its composition operation. Such a configurator may not be further combined by composition with other configurators, but only to instantiate objects.

The evaluation of the instantiation expression `new` uses the configuration instruction stored in the component value, applying them to an “empty” object instance, written `0`. This is expressed in Rule (Eval New) by the premise $0; c; S \Downarrow s; S'$. Required ports left open by the application are satisfied by the values given by the plug-assignments.

A reconfiguration depends on a (runtime) test, to check that a configurator is in fact compatible with the structure of a given instance. Formally, we specify that a configurator with precondition type K is applicable to s if the matching $s // K$ test (defined in Figure 5) holds. Intuitively, an instance $s = (r, e, p)_\Gamma$ matches a set of resources K if each one of the resources in K can be found, with compatible types, in r , e , or p .

The evaluation of `reconfig` expression, is thus defined by two rules (Eval Reconfig) and (Eval Reconfig Else), that consider the two possible outcomes of a matching test. Rule (Eval Reconfig) is applicable if the test $s // K$ succeeds and the composition operation c , taken from the configurator yield by e_1 is applied to s , the instance obtained from e_2 . The final result comes from evaluating e_3 . Rule (Eval Reconfig Else) is applicable otherwise, it skips the application of the composition application and follows by evaluating the `else` branch. Notice that only the required resources (in the precondition) in the runtime type information are used to test the instance. The added resources (in the type’s post condition) are nevertheless important in the process of building the type information (see (Eval Sequence)).

The rules in Figure 7, for the judgement form $s; c; S \Downarrow s'; S'$, interpret the application of a composition operation c to a partially built instance s , with relation to a heap S , and incrementally build an object instance.

As expected, Rule (App Sequence) sequentially applies the two parts of the operation thus causing the combined effect of both. (App Requires) and (App Provides) both create `nil` initialised references (empty placeholders) in the heap for ports, and establish the corresponding connections in the records r or p .

The integration of a inner component instance inside an instance depends on the (recursive) construction of the inner instance, and corresponding introduction as an inner element of the instance in record e . Similarly, (App Method Block) takes the field values and builds a record associated to its local name. Here, $v_i[(r, e, p)_\Gamma]$ denotes the substitution of the object’s labels by their locations in the fields and therefore give access to the elements already in the instance and that $[x \leftarrow \perp]$ introduces the “self” reference of the method block itself in the field expressions.

Finally, the application of a plug expression connects plug sources to target ports by simply forming a chain between the two locations, Rule (App Plug). We use the function $\text{select}_S(o, \pi)$ to denote the location corresponding to port π . Notice how the runtime type annotation of an object is progressively built on each rule, and added to the Γ component. The resulting object is then a structured web of ports, other objects, and

(App Requires)	($l = \text{new}(S)$)
$(r, e, p)_F; (\text{requires } \ell : \tau); S \Downarrow (r \oplus \{\ell \mapsto l\}, e, p)_{F, l; \tau}; S[l \mapsto \text{nil}]$	
(App Provides)	($l = \text{new}(S)$)
$(r, e, p)_F; (\text{provides } \ell : \tau); S \Downarrow (r, e, p \oplus \{\ell \mapsto l\})_{F, l; \tau}; S[l \mapsto \text{nil}]$	
(App Uses)	(App Sequence)
$\frac{\text{new } v; S \downarrow l; S'}{(r, e, p)_F; x[v : \tau]; S \Downarrow (r, e \oplus \{x \mapsto l\}, p)_{F, l; \tau}; S'}$	$\frac{s; c_1; S \Downarrow s'; S' \quad s'; c_2; S' \Downarrow s''; S''}{s; (c_1; c_2); S \Downarrow s''; S''}$
(App Method Block)	
$(r, e, p)_F; x_K[\ell_i : \tau_i = v_i^{i \in 1..n}]; S \Downarrow (r, e \oplus \{x \mapsto l\}, p)_{F, l; \tau_i^{i \in 1..n}}; S'$	
(App Plug)	
$s; \text{plug } \pi_1 : \tau_1 \text{ into } \pi_2 : \tau_2; S \Downarrow s; S[\text{select}_S(s, \pi_2) \mapsto \text{select}_S(s, \pi_1)]$	

Fig. 7. Application of Configurators

records containing variables and methods. Methods can be accessed through provided ports, that lead to appropriate implementations as specified by the object's architecture.

4 Type System

In this section we present a type system for λ_χ . Well-typed programs are ensured to be well behaved, in the sense motivated in the introduction, and made precise below.

Our type system includes rules for typing computational expressions (Figure 8), and rules for typing compositional expressions (Figure 9). Typing environments (Δ, Γ) assign types to variables, as usual, and also to locations (this is only useful for stating our subject reduction result). The rules for the λ -calculus and imperative records are standard. Rule (Val Interface) allows us to coerce a record type to an interface type.

The architectural soundness of configurators, components and instances is ensured by the combination of the typing of composition operations in Figure 9 together with the typing of the `compose`, `new` and `reconfig`. On one hand, the typing of composition operations intentionally describes and combines their effect, on the other hand, the typing of computational expressions uses that information in three levels of visibility. Rule (Val Compose) ensures that components are only produced given a completed architecture, i.e. from configurators that do not depend on any existing resource ($\emptyset \Rightarrow K$) and leave no unsatisfied resources left open ($K_\circ = \emptyset$). The resulting component type $K_\triangleleft \Rightarrow K_\triangleright$ reveals only the required and provided service types, hiding the remaining intensional information about the component internal structure. Thus, a component is indistinguishable from any other with the same type. Rule (Val New) types a new instance with the object type containing the provided ports of its generator component and checks for the proper satisfaction of all required ports, if there are any. Notice that once again some type information gets hidden: here, the existing required ports are not

(Val Var)	(Val Abstraction)	(Val Application)	(Val Interface) ($m \leq n$)
$\frac{x : \tau \in \Delta}{\Delta \vdash x : \tau}$	$\frac{\Delta, x : \tau \vdash e : \sigma}{\Delta \vdash \lambda x : \tau. e : \tau \rightarrow \sigma}$	$\frac{\Delta \vdash e_1 : \tau \rightarrow \sigma \quad \Delta \vdash e_2 : \tau}{\Delta \vdash e_1(e_2) : \sigma}$	$\frac{\Delta \vdash e : \{\ell_i : \tau_i \mid i \in 1..n\}}{\Delta \vdash e : \{\ell_i : \tau_i \mid i \in 1..m\}}$
(Val Record)	(Val Select)	(Val Assign)	
$\frac{\Delta \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Delta \vdash [\ell_i : \tau_i = e_i \mid i \in 1..n] : \{\ell_i : \tau_i \mid i \in 1..n\}}$	$\frac{\Delta \vdash e : \{\dots, \ell : \tau, \dots\}}{\Delta \vdash e.\ell : \tau}$	$\frac{\Delta \vdash e_1 : \{\dots, \ell : \tau, \dots\} \quad \Delta \vdash e_2 : \tau}{\Delta \vdash e_1.\ell := e_2 : \tau}$	
(Val Compose)	(Val New)		
$\frac{\Delta \vdash e : \emptyset \Rightarrow K}{\Delta \vdash \text{compose } e : K_{\triangleleft} \Rightarrow K_{\triangleright}}$	$\frac{\Delta \vdash e : \{\ell_i : \tau_i \mid i \in 1..n\} \Rightarrow \sigma \quad \Delta \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Delta \vdash \text{new } e \text{ with } \ell_i := e_i \mid i \in 1..n : \sigma}$		
(Val Reconfig)			
$\frac{\Delta \vdash e_1 : K \Rightarrow K' \quad \Delta \vdash e_2 : I \quad \Delta \vdash e'_i : \sigma_i \quad \forall i \in 1..n \quad \Delta, x : I \oplus K'_{\triangleright} \vdash e_3 : \delta \quad \Delta, x : I \vdash e_4 : \delta}{\Delta \vdash \text{reconfig } x = e_1[e_2] \text{ with } \ell_i := e'_i \mid i \in 1..n \text{ in } e_3 \text{ else } e_4 : \delta}$			

Fig. 8. Typing Rules for Computational Expressions

included in the instance type. Hence, the type system does not distinguish instances of components providing the same ports.

Despite the dependence of reconfiguration on a runtime check, some basic conformance between the configurator type ($K \Rightarrow K'$) and the type of the target object (τ) is tested statically in the (Val Reconfig) rule. We basically use K' to ensure that the continuations of reconfigurations are well-typed, in particular: that no dependencies are left open after the application ($K'_{\circ} = \emptyset$); that the configurator does not override the object's ports ($K'_{\triangleright} \# I$); and that all new requirements must be correctly satisfied by the existing plug assignments, ($K'_{\triangleleft} - K_{\triangleleft} = \{\ell'_i : \sigma_i \mid i \in 1..m\}$).

The rule system of Figure 9 assigns a type of the form $K \Rightarrow K'$ to each composition operation to denote the required (K) and provided (K') resources. Basic composition operations get natural configurator types, which are then elaborated by means of composition. For instance the type of (provides $\ell : \tau$) indicates the providing of an unsatisfied resource ($\ell \circ \tau$), i.e. a resource that must be satisfied before this configurator is used to make a component or reconfigure an instance, and of a new provided port ($\ell \triangleright \tau$). Symmetrically, (requires $\ell : \tau$) adds a new required port ($\ell \triangleleft \tau$) and an available resource ($\ell \bullet \tau$) to a composition context. The typing of $(x[e : \tau])$ indicate that it adds available resources corresponding to an instance of the inner component ($x \bullet \{\ell_j^p : \sigma_j \mid j \in 1..m\}$) and its provided ports ($x.\ell_j^p \bullet \sigma_j \mid j \in 1..m$), and unsatisfied resources that denote the internally required ports ($x.\ell_i^r \circ \tau_i \mid i \in 1..n$). Similar type information is associated with method blocks but using the required set of resources K . Notice the restricted typing environment $|\Delta|$ in the premises of Rule (Comp Method Block), and (Comp Uses). $|\Delta|$ denotes the typing environment retaining the type assignments in Δ that have component or configurator type. This forbids any reference to the heap to be made from well-typed method blocks, and therefore ensures that configurators are closed values.

$$\begin{array}{c}
\text{(Comp Requires)} \qquad \qquad \qquad \text{(Comp Provides)} \\
\Delta \vdash (\text{requires } \ell : \tau) : \emptyset \Longrightarrow \{\ell \bullet \tau, \ell \triangleleft \tau\} \quad \Delta \vdash (\text{provides } \ell : \tau) : \emptyset \Longrightarrow \{\ell \circ \tau, \ell \triangleright \tau\} \\
\\
\text{(Comp Plug)} \\
\Delta \vdash \text{plug } (\pi_1 : \tau) \text{ into } (\pi_2 : \tau) : (\{\pi_2 \circ \tau, \pi_1 \bullet \tau\} \Longrightarrow \{\pi_1 \bullet \tau\}) \\
\\
\text{(Comp Sequencing)} \qquad \qquad \qquad (K' \# K'', K' \# K''') \\
\frac{\Delta \vdash e_1 : K \Longrightarrow K', K_c \quad \Delta \vdash e_2 : K_c, K'' \Longrightarrow K'''}{\Delta \vdash (e_1; e_2) : K, K'' \Longrightarrow K', K'''} \\
\\
\text{(Comp Uses)} \qquad \qquad \qquad (\tau = \{\ell_i : \tau_i\}_{i \in 1..n}, \sigma = \{\ell'_j : \sigma_j\}_{j \in 1..m}) \\
\frac{|\Delta| \vdash e : \tau \Rightarrow \sigma}{\Delta \vdash x[e : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow \{x \bullet \sigma, x.\ell_i \circ \tau_i\}_{i \in 1..n}, x.\ell'_j \bullet \sigma_j\}_{j \in 1..m}} \\
\\
\text{(Comp Method Block)} \\
\frac{|\Delta|, x : \{\ell_i : \tau_i\}_{i \in 1..n}, K_\bullet \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Delta \vdash x_K[\ell_i : \tau_i = e_i]_{i \in 1..n} : K \Longrightarrow K, \{x \bullet \{\ell_i : \tau_i\}_{i \in 1..n}\}}
\end{array}$$

Fig. 9. Typing Rules for Composition Expressions

Rule (Comp Sequencing) combines the effect of two expressions. This rule shows the propagation of resources (K_c) from e_1 to e_2 , meaning that e_2 handles these resources either by keeping them in K''' or by consuming them.

For a sequence of composition operations to be accepted in a compose expression it must denote a complete architecture, in particular the set of unsatisfied resources must be empty ($K_o = \emptyset$). The elimination of these resources from the types is captured by the typing of plug operations: they are typed as having a required resource ($\pi_2 \circ \tau$) that is not propagated to the set of provided resources. This denotes the satisfaction of internal dependencies in a composition. We now state and characterize type safety in λ_χ .

Type safety is a corollary of our main theorem (Theorem 1) that, as a side effect of the traditional progress and type preservation properties, implies the architectural soundness of configurators, components and instances (before and after a reconfiguration action). First, our language is extended with a distinguished value, wrong, to which an expression evaluates whenever a runtime error occurs. A runtime error is defined to occur whenever an operation is undefined, this includes usual cases such as: application of a value which is not an abstraction, assignment to a value which is not a location, selection of a field on a value which is not a record or does not possess the relevant label (notice that this case includes calling a method on a null reference), and so on. Essentially, we include all situations in which the operational semantics is undefined.

In order to prove subject reduction for expression evaluation, we rely on an auxiliary lemma that establishes subject reduction with respect to the application of composition operations, and is used when analysing the cases of the `new` and `reconfig` expressions, where composition and reconfiguration steps take place. The proof of this lemma relies on certain special type annotations, of the form $\llbracket \tau \Rightarrow \sigma \rrbracket$, that keep track, during the process of constructing an object instance from its generating component, of its unsatisfied required ports (τ) on one hand, and of the declared provided ports on the other hand (σ). The final type of the object being built is σ . The type τ is used to verify the

satisfaction of the required ports. We also need to define a notion of conformance between the structure of object instances and resource sets, in order to specify an invariant of the reconfiguration process, and relate such invariant with the result of the runtime matching test performed by the reconfig operation. All these ingredients are presented with full details in [15], and combined to prove Theorem 1. We write $\Gamma \vdash S$ to denote that Γ types heap S .

Theorem 1 (Subject Reduction). *Let $e \in \lambda_\chi \setminus \{\text{nil}\}$ and S be a heap, such that e is a closed expression in S , $\text{nil}(S) = \emptyset$. Let $\Gamma \vdash S$ and $\Gamma \vdash e : \tau$. If $(e; S \downarrow v; S')$ then*

- a) *There is a typing environment Γ' such that $\Gamma \vdash S'$ and $\Gamma' \vdash v : \tau$;*
- b) *The value v is either an abstraction, a component, a configurator, or a location that maps to a record or an object, and*
- c) *$\text{nil}(S') = \emptyset$.*

Notice that the addition of required and provided ports to an instance introduces nil values in the heap (e.g., Rule App Requires). As expected, in well-formed architectures all such ports must become plugged to compatible implementations. Thus, from the theorem's assumption that nil does not occur in the source program, from the invariant $\text{nil}(S) = \emptyset$, and a correct typing of the heap, we conclude that all instances must be structurally well-formed. From the fact that nil is not admissible as a result of an evaluation, we also conclude that “nil dereferencing errors” cannot occur.

5 Concluding Remarks

We have presented a small object-oriented component programming language, by adding to a λ -calculus with imperative records a arguably minimal set of language constructs to express component definition, using method blocks, other components, and connection operations as basic ingredients. Both configuration of components and reconfiguration of objects are uniformly represented at the semantic level by configurators, typed values that represent architectural change and play a role similar to makefiles or project templates in software development support systems. However, configurators and components are first-class values, that can constructed and manipulated dynamically. Therefore, our language is expressive enough to model many sophisticated software management operations, typical of component-based systems, involving dynamic composition, configuration and reconfiguration, even if the well-typed architectural programming fragment is computationally incomplete, for expected reasons. Our main result is a type system enforcing that well-typed programs do not go wrong; in our setting this implies not only absence of “method not implemented” errors, but also architectural consistency of dynamic composition and reconfiguration processes, as made precise by Theorem 1.

Although defined from rather standard language constructs (a λ -calculus with imperative records), our language does not seem straightforwardly encodable, in a type preserving way, in such a canonical language, due to the presence of intensional information at the level of types, to the particular notion of “staging” involved, related with architectural manipulations rather than with source level program manipulation, and to

the particular combination of static and dynamic type checking used. In particular, configurator values carry type information (e.g., as Java class files do) and evaluation of configurator operations (Fig. 6) involve computing with intensional type information in order to ensure soundness of configurator composition and reconfiguration.

It would be interesting to investigate more flexible typing relations, involving subtyping and polymorphism, along the lines of [14], and particularly challenging to identify a natural and useful notion of subtyping for configurator values, given the intensional character of configurator types. At the level of the basic language, it is also conceivable, in principle, to extend the application of reconfiguration not only to objects, but also to component values, we refrained from pursuing that, because that does not seem to increase the expressiveness of our language, and lacks pragmatical motivation.

This work is partially supported by IST-3-016004-IP-09 Sensoria and by Microsoft Research Grant 2002-73. We also acknowledge many useful comments by the reviewers.

References

1. Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Trans. on Progr. Languages and Systems*, 13(2), April 1991.
2. Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in archjava. In *Proc. of the European Conf. on Object-Oriented Programming*. Springer-Verlag, 2002.
3. Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.
4. Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoye. Formalizing dynamic software updating. In *Proc. of the Int. Workshop on Unanticipated Software Evolution*, 2003.
5. Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *Proc. of the European Conf. on Object-Oriented Programming*. Springer-Verlag, 1999.
6. Luca Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages*. ACM Press, 1997.
7. Sophia Drossopoulou, Ferruccio Damiani, Mariangola Dezani-Ciancaglini, and Paola Gianini. Fickle: Dynamic object re-classification. In *Proc. of the European Conf. on Object-Oriented Programming*, 2001.
8. Dominic Duggan. Type-based hot swapping of running modules. In *Proc. of the Int. Conf. on Functional Programming*, 2001.
9. S. Fagorzi and E. Zucca. A calculus for reconfiguration. In *On-line Proc. of the Int. Workshop Developments in Computational Models at ICALP*, 2005.
10. Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1998.
11. Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *Proc. of the Euro. Symp. on Programming*, 2002.
12. Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Call-by-value mixin modules: Reduction semantics, side effects, types. In *Proc. of the Euro. Symp. on Programming*, 2004.
13. João Costa Seco and Luís Caires. A basic model of typed components. In *Proc. of the European Conf. on Object-Oriented Programming*, Cannes, France, 2000. Springer-Verlag.
14. João Costa Seco and Luís Caires. Subtyping First-Class Polymorphic Components. In *Proc. of the Euro. Symp. on Programming*, Edinburgh, 2005. Springer-Verlag.
15. João Costa Seco and Luís Caires. Types for dynamic reconfiguration. Technical Report UNL-DI-1-2006, FCT-UNL, 2006.

16. Peter Sewell. Modules, abstract types, and distributed versioning. In *ACM Symp. on Principles of Programming Languages*, pages 236–247, New York, NY, USA, 2001. ACM Press.
17. Vugranam C. Sreedhar. Mixin'up components. In *Proceedings of the International Conference on Software Engineering*. ACM Press, 2002.
18. Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.
19. Joe Wells and René Vestergaard. Confluent equational reasoning for linking with first-class primitive modules. In *Proc. of the Euro. Symp. on Programming*, 1999.

Size-Change Termination Analysis in k -Bits

Michael Codish^{1,*}, Vitaly Lagoon², Peter Schachte², and Peter J. Stuckey^{2,3}

¹ Department of Computer Science, Ben-Gurion University, Israel

² Department of Computer Science and Software Engineering,
The University of Melbourne, Australia

³ NICTA Victoria Laboratory

`mcodish@cs.bgu.ac.il`

`{lagoon, schachte, pjs}@cs.mu.oz.au`

Abstract. Size-change termination analysis is a simple and powerful technique successfully applied for a variety of programming paradigms. A main advantage is that termination for size-change graphs is decidable and based on simple linear ranking functions. A main disadvantage is that the size-change termination problem is PSPACE-complete. Proving size change termination may have to consider exponentially many size change graphs. This paper is concerned with the representation of large sets of size-change graphs. The approach is constraint based and the novelty is that sets of size-change graphs are represented as disjunctions of size-change constraints. A constraint solver to facilitate size-change termination analysis is obtained by interpreting size-change constraints over a sufficiently large but finite non-negative integer domain. A Boolean k -bit modeling of size change graphs using binary decision diagrams leads to a concise representation. Experimental evaluation indicates that the 2-bit representation facilitates an efficient implementation which is guaranteed complete for our entire benchmark suite.

1 Introduction

Size-change termination analysis [8] is a simple and powerful technique to verify program termination. First, the transition relation of a program is approximated by a set of size-change graphs. Then, termination is guaranteed if all of the idempotent size change graphs in the closure of this set under a composition operation have (possibly different) ranking functions.

A typical example is the analysis of the Prolog program depicted in Figure 1(a) which computes Ackermann's function. The size-change graphs in the figure describe all transitions in computations of this program. Between subsequent function calls, either the first argument decreases in size (Figure 1(b)), or else it does not increase and the second argument decreases in size (Figure 1(c)). As formalised below, these graphs are idempotent, closed under composition and have as ranking functions $f(\bar{u}) = u_1$ and $f'(\bar{u}) = u_2$ respectively.

* Research performed at the University of Melbourne

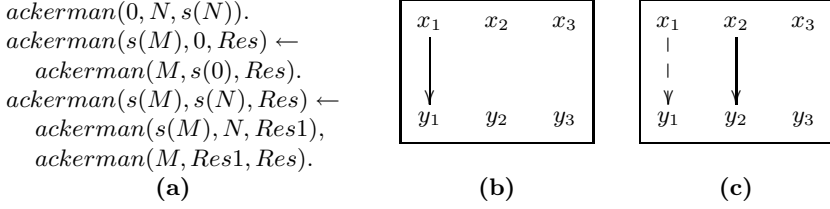


Fig. 1. Ackermann's function with size-change graphs

A major strength of the technique is that for a given set of size-change graphs termination is decidable. An idempotent size-change graph has a ranking function if and only if it has one which indicates that a specific single argument decreases in size. A major weakness is that size change termination is complete for PSPACE. While in practice this rarely occurs, closure under composition may introduce exponentially many additional size-change graphs.

This paper is concerned with the representation of large sets of size-change graphs and supporting operations for closing these representations under composition and testing all graphs in the closure for the existence of ranking functions. The key idea in our approach is to view sets of size-change graphs as constraints. For individual size-change graphs this idea is not new. The TerminWeb [9] analyser maintains sets of size-change graphs, each graph represented as a conjunction of constraints. The novelty in this paper is to illustrate how sets of size-change graphs can be represented accurately through disjunction. For example the two graphs in Figure 1 are captured by the constraint $(x_1 > y_1) \vee (x_1 \geq y_1 \wedge x_2 > y_2)$. Given this view, a set of size change graphs is equivalent to its set of solutions over the domain of non-negative integers, much the same as a Boolean function is equivalent to its set of models. We draw on the motivation that representing large sets of models for Boolean functions is a well studied problem with readily available off-the-shelf tools. The main difficulty is to provide set-based operations for size change termination which operate accurately on these representations.

To support an operation to compose disjunctions of size-change graphs we introduce a non-standard interpretation of the binary size-relations $>$ and \geq . This enables us to model composition of sets of size-change graphs as conjunction. To determine if each of the graphs in a set has a ranking function we apply a previous result [3] to design a suitable test.

Another difficulty is to provide a constraint solver for size-change graphs and their operations. This is achieved by interpreting constraints over a sufficiently large but finite domain (of non-negative integers). Finite domain constraints are then represented as Boolean functions as proposed in [6]. This Boolean representation for finite domain constraints and operations leads to an efficient implementation using binary decision diagrams. Experimental evaluation indicates that the 2-bit representation is guaranteed complete for our entire extensive benchmark suite. Of course the approach we describe does not ameliorate the

PSPACE hardness of the termination problem for size change graphs, the resulting binary decision diagrams can require exponential space and time.

2 Size-Change Termination

This section presents the standard definitions and results for size change graphs. Our definitions are similar to those given in [8] except that they are given in a language of constraints. The constraint representation naturally provides a notion of ordering not present in the original definition [8].

Definition 1 (size-change graphs - I). *A size-change graph is a binary clause of the form $p(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), q(\bar{y})$ where \bar{x} and \bar{y} are the disjoint vectors of arguments and $c(\bar{x}, \bar{y})$ is a conjunction of constraints of the form $x \succ^b y$ with $x \in \bar{x}$, $y \in \bar{y}$ and $b \in \{0, 1\}$. A constraint $x \succ^b y$ corresponds to an edge and is interpreted as $x \geq y + b$: strict ($x > y$) or non-strict ($x \geq y$) when respectively $b = 1$ or $b = 0$.*

Size-Change Graph Notation: Consider a size-change graph $g = p(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), q(\bar{y})$ with $\bar{x} = \langle x_1 \dots, x_n \rangle$ and $\bar{y} = \langle y_1 \dots, y_m \rangle$. We sometimes write g in the form $p_{/n}(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), q_{/m}(\bar{y})$ to make explicit the arities of \bar{x} and \bar{y} . The parameter set of g , is denoted $Par(g) = \{p_{\langle 1 \rangle}, \dots, p_{\langle n \rangle}, q_{\langle 1 \rangle}, \dots, q_{\langle m \rangle}\}$. For a set of size-change graphs G , we denote $Par(G) = \cup \{Par(g) \mid g \in G\}$. A size change graph of the form $p_{/n}(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), p_{/n}(\bar{y})$ is called a *recursive* size change graph. When p and q are clear from the context we refer to $g = c(\bar{x}, \bar{y})$ as the size-change graph. In the examples, edges are depicted by solid and dashed arrows corresponding to strict and non-strict edges. For each pair of nodes $x \in \bar{x}$ and $y \in \bar{y}$ the unique strictest constraint between x and y is depicted.

Example 1. The following size-change graphs are depicted in Figure 2 as $c_1(\bar{x}, \bar{y})$, $c_2(\bar{x}, \bar{y})$ and $c_3(\bar{x}, \bar{y})$ respectively.

$$\begin{aligned} g_1 &= p(x_1, x_2, x_3) \leftarrow x_1 > y_2, x_2 \geq y_2, x_3 > y_3, p(y_1, y_2, y_3). \\ g_2 &= p(x_1, x_2, x_3) \leftarrow x_1 > y_1, x_2 \geq y_1, p(y_1, y_2, y_3). \\ g_3 &= p(x_1, x_2, x_3) \leftarrow x_1 > y_2, x_2 > y_2, p(y_1, y_2, y_3). \end{aligned}$$

Note that by Definition 1 the size-change graph

$$g'_2 = p(x_1, x_2, x_3) \leftarrow x_1 > y_1, x_1 \geq y_1, x_2 \geq y_1, p(y_1, y_2, y_3).$$

is also depicted as $c_2(\bar{x}, \bar{y})$.

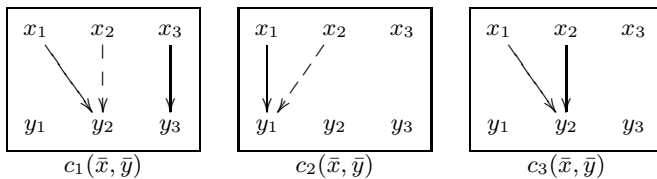


Fig. 2. Size-change graphs

Definition 2 (size-change graph solution). A solution θ for a size-change graph $c(\bar{x}, \bar{y})$ is a valuation on the variables \bar{x} and \bar{y} , $\theta = \{x_1/a_1, \dots, x_n/a_n, y_1/b_1, \dots, y_m/b_m\}$ which is a solution of $c(\bar{x}, \bar{y})$, i.e. $c(\bar{a}, \bar{b})$ is valid.

Solutions can be written as two rows (\bar{a}, \bar{b}) in a matrix, as illustrated in the following example.

Example 2. Consider the following 8 solutions and the size-change graphs $c_1(\bar{x}, \bar{y})$, $c_2(\bar{x}, \bar{y})$ and $c_3(\bar{x}, \bar{y})$ of Figure 2.

$$\begin{array}{llll} s_1 = \begin{bmatrix} 8, 7, 3 \\ 9, 7, 2 \end{bmatrix} & s_2 = \begin{bmatrix} 4, 3, 8 \\ 3, 7, 9 \end{bmatrix} & s_3 = \begin{bmatrix} 8, 7, 2 \\ 9, 6, 3 \end{bmatrix} & s_4 = \begin{bmatrix} 8, 7, 2 \\ 5, 6, 3 \end{bmatrix} \\ s'_1 = \begin{bmatrix} 1, 0, 1 \\ 1, 0, 0 \end{bmatrix} & s'_2 = \begin{bmatrix} 1, 0, 0 \\ 0, 1, 1 \end{bmatrix} & s'_3 = \begin{bmatrix} 1, 1, 0 \\ 1, 0, 1 \end{bmatrix} & s'_4 = \begin{bmatrix} 1, 1, 1 \\ 0, 0, 0 \end{bmatrix} \end{array}$$

s_1 and s'_1 are solutions only for $c_1(\bar{x}, \bar{y})$, s_2 and s'_2 are solutions only for $c_2(\bar{x}, \bar{y})$, s_3 and s'_3 are solutions only for $c_3(\bar{x}, \bar{y})$, s_4 is a solution for $c_2(\bar{x}, \bar{y})$ and $c_3(\bar{x}, \bar{y})$ but not for $c_1(\bar{x}, \bar{y})$ and s'_4 is a solution for all three of the size-change graphs.

Definition 3 (order on size-change graphs). Size-change graphs on the same parameter set are ordered by constraint entailment. A size change graph $c_1(\bar{x}, \bar{y})$ is more general than $c_2(\bar{x}, \bar{y})$ if the solutions of c_1 are a superset of the solutions of c_2 , i.e., $c_2(\bar{x}, \bar{y}) \models c_1(\bar{x}, \bar{y})$. Size-change graphs are equivalent if they have the same sets of solutions, i.e. $c_1(\bar{x}, \bar{y}) \leftrightarrow c_2(\bar{x}, \bar{y})$.

Definition 4 (composition and idempotence of size-change graphs). Let $p(\bar{x}) \leftarrow c_1(\bar{x}, \bar{y}), q(\bar{y})$ and $q(\bar{x}) \leftarrow c_2(\bar{x}, \bar{y}), r(\bar{y})$ be size-change graphs. Their composition is the size-change graph $p(\bar{x}) \leftarrow c_1(\bar{x}, \bar{y}) \circ c_2(\bar{x}, \bar{y}), r(\bar{y})$ given by

$$c_1(\bar{x}, \bar{y}) \circ c_2(\bar{x}, \bar{y}) = \bigwedge \left\{ x \succ^b y \mid \begin{array}{l} x \in \bar{x}, y \in \bar{y}, \\ c_1(\bar{x}, \bar{z}) \wedge c_2(\bar{z}, \bar{y}) \models x \succ^b y \end{array} \right\}.$$

Recursive size-change graph $p(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), p(\bar{y})$ is idempotent if and only if $c(\bar{x}, \bar{y}) \circ c(\bar{x}, \bar{y}) = c(\bar{x}, \bar{y})$. The pairwise composition of sets of size-change graphs G_1 and G_2 , respectively of the form $p(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), q(\bar{y})$ and $q(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), r(\bar{y})$ is: $G_1 \circ G_2 = \{ g_1 \circ g_2 \mid g_1 \in G_1, g_2 \in G_2 \}$.

Definition 5 (closure under composition). Let G be a set of size-change graphs. We denote by G^* the closure of G under composition. This is the smallest superset of G such that if $p(\bar{x}) \leftarrow c_1(\bar{x}, \bar{y}), q(\bar{y}) \in G^*$ and $q(\bar{x}) \leftarrow c_2(\bar{x}, \bar{y}), r(\bar{y}) \in G^*$ then also $p(\bar{x}) \leftarrow c_1(\bar{x}, \bar{y}) \circ c_2(\bar{x}, \bar{y}), r(\bar{y}) \in G^*$.

Example 3. The set of size-change graphs depicted in Figure 1 is closed under composition. Both graphs are idempotent. The graphs in Figure 2 are also idempotent. The graphs in Figure 3 are not idempotent.

Lee *et al.*[8] introduce the property of *size change termination* and prove that a set of size-change graphs G has this property if and only if each idempotent size-change graph $p_{/n}(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), p_{/n}(\bar{y})$ in G^* has a strict “vertical down arrow”

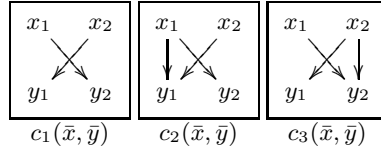


Fig. 3. Non-idempotent size change graphs

of the form $x_i > y_i$. Each individual idempotent graph has a strict vertical down arrow if and only if the following condition holds:

$$\bigvee_{i=1}^n (c(\bar{x}, \bar{y}) \models x_i > y_i). \quad (1)$$

For any (recursive) size change graph $c(\bar{x}, \bar{y})$, a function f mapping tuples of non-negative integers to a well founded domain and such that $c(\bar{x}, \bar{y}) \models f(\bar{x}) > f(\bar{y})$ is called a ranking function for $c(\bar{x}, \bar{y})$. Equation 1 implies that $c(\bar{x}, \bar{y})$ has a ranking function of the form $f(u_1, \dots, u_n) = u_i$.

The result of [8] is generalized in [3] where the authors show that a set of size-change graphs G satisfies size-change termination if and only if the following condition holds: for every recursive (not necessarily idempotent) size-change graph $p_{/n}(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), p_{/n}(\bar{y})$ in G^* , each solution of $c(\bar{x}, \bar{y})$ has a strict “vertical down arrow” of the form $x_i > y_i$. In other words, each individual recursive $c(\bar{x}, \bar{y})$ in G^* must satisfy the condition below:

$$c(\bar{x}, \bar{y}) \models \bigvee_{i=1}^n (x_i > y_i). \quad (2)$$

Equation 2 implies that $c(\bar{x}, \bar{y})$ has a ranking function of the form $f(u_1, \dots, u_n) = \sum a_i u_i$ with all coefficients $a_i \in \{0, 1\}$. The distinction between the tests in Equations 1 and 2 is exemplified by the size change graph $c_1(\bar{x}, \bar{y})$ of Figure 3 which is not idempotent and which has no strict vertical down arrow of the form $x_i > y_i$. However, any solution of $c_1(\bar{x}, \bar{y})$ is also a solution of $c_2(\bar{x}, \bar{y})$ or of $c_3(\bar{x}, \bar{y})$ (in the same figure) which do have vertical down arrows. Note that the function $f(\bar{u}) = u_1 + u_2$ is a ranking function for $c_1(\bar{x}, \bar{y})$. Sidestepping the restriction to idempotent graphs turns out to be important to facilitate the specification of a set-based test for termination given in Section 3. In the following we refer to the implicant in Equation 2 as the *ranking constraint*.

Definition 6 (size-change ranking constraint). Let $p_{/n}(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), p_{/n}(\bar{y})$ be a recursive size-change graph. The corresponding ranking constraint is denoted

$$\mathcal{R}(\bar{x}, \bar{y}) = \bigvee_{i=1}^n x_i \succ^1 y_i.$$

The application of size-change termination to proving termination is based on the observation that if a set of size-change graphs G is a safe approximation of

the transition relation for a program P , and G satisfies size-change termination, then P terminates.

3 Set Based Size-Change Termination

In this section we propose a set based approach to size-change termination. The basic idea is that sets of size-change graphs can be represented as disjunctions of constraints with no loss of information for termination analysis. The contribution is in the design of the set-based operations for size-change termination analysis. The following definition provides the basic representation for a set of size-change graphs as a disjunction of constraints.

Definition 7 (disjunctive representation). *Let $G_{p,q}$ be a set of size-change graphs of the form $p(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), q(\bar{y})$ (p and q are fixed). The disjunctive representation of $G_{p,q}$ is the binary clause denoted $G_{p,q}^\vee = p(\bar{x}) \leftarrow C(\bar{x}, \bar{y}), q(\bar{y})$ where $C(\bar{x}, \bar{y}) = \vee \{ c(\bar{x}, \bar{y}) \mid p(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), q(\bar{y}) \in G_{p,q} \}$. When clear from the context we refer to $C(\bar{x}, \bar{y})$ as the disjunctive representation.*

This definition is easily extended to apply to sets of graphs with different source and target (p and q). In this case the result is a set of disjunctive constraints, (at most) one for each p and q .

Definition 8 (order on disjunctive size-change graphs). *Disjunctive size-change graphs with the same source and target are ordered by entailment. A disjunctive size-change graph G_1 is more general than G_2 if the solutions of G_1 include those of G_2 . Two disjunctive size-change graphs are equivalent if they have the same sets of solutions.*

Example 4. Consider the size-change graphs $p(\bar{x}) \leftarrow c_i(\bar{x}, \bar{y}), p(\bar{y})$ for $i \in \{1, 2, 3\}$ depicted in Figure 3. The sets of graphs $\{c_1(\bar{x}, \bar{y})\}$ and $\{c_2(\bar{x}, \bar{y}), c_3(\bar{x}, \bar{y})\}$ are equivalent. In one direction, graph $c_1(\bar{x}, \bar{y})$ is more general than each of the graphs $c_2(\bar{x}, \bar{y})$ and $c_3(\bar{x}, \bar{y})$ which have fewer solutions (more constraints). In the other direction, observe that $c_1(\bar{x}, \bar{y}) \models x_1 + x_2 > y_1 + y_2 \models x_1 > y_1 \vee x_2 > y_2$ and so any solution of $c_1(\bar{x}, \bar{y})$ is either a solution of $c_2(\bar{x}, \bar{y})$ or of $c_3(\bar{x}, \bar{y})$.

When composing disjunctions of constraints we can no longer consider the original disjuncts as these are not maintained as sets. However we may consider all disjuncts that entail a given constraint. The following definition is intended only as the specification of set-based composition. We do not propose to implement the operation based on this definition. That would be very inefficient.

Definition 9 (composing disjunctive representations). *Let $G_{p,q}$ and $G_{q,r}$ be sets of size-change graphs with disjunctive representations $G_{p,q}^\vee = p(\bar{x}) \leftarrow C_1(\bar{x}, \bar{y}), q(\bar{y})$ and $G_{q,r}^\vee = q(\bar{y}) \leftarrow C_2(\bar{y}, \bar{z}), r(\bar{z})$ respectively. Their disjunctive composition is the size-change graph $G_{p,q}^\vee \circ G_{q,r}^\vee = p(\bar{x}) \leftarrow C(\bar{x}, \bar{y}), r(\bar{z})$ where*

$$C(\bar{x}, \bar{y}) = \bigvee \left\{ c_1(\bar{x}, \bar{y}) \circ c_2(\bar{x}, \bar{y}) \mid \begin{array}{l} c_1(\bar{x}, \bar{y}) \models C_1(\bar{x}, \bar{y}), \\ c_2(\bar{x}, \bar{y}) \models C_2(\bar{x}, \bar{y}) \end{array} \right\}.$$

The following two lemmata justify viewing sets as disjunctions.

Lemma 1 (disjunctive termination). *Consider a set of size-change graphs $G_{p,p}$ of the form $p(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), p(\bar{y})$. Then, all graphs in $G_{p,p}$ satisfy the ranking constraint of Definition 6 if and only if $G_{p,p}^\vee \models \mathcal{R}(\bar{x}, \bar{y})$.*

Proof. Follows from $(a \vee b) \models c$ if and only if $a \models c$ and $b \models c$.

Lemma 2 (disjunctive composition). *Let $G_{p,q}$ and $G_{q,r}$ be sets of size-change graphs. Then, $(G_{p,q} \circ G_{q,r})^\vee \leftrightarrow G_{p,q}^\vee \circ G_{q,r}^\vee$.*

Proof. Let s be a solution of $(G_{p,q} \circ G_{q,r})^\vee$. So there are graphs $g_1 \in G_{p,q}$ and $g_2 \in G_{q,r}$ such that s is a solution of $g_1 \circ g_2$. But g_1 and g_2 respectively entailed $G_{p,q}^\vee$ and $G_{q,r}^\vee$ and hence s is also a solution for $G_{p,q}^\vee \circ G_{q,r}^\vee$. The other direction is similar.

From here on we will not distinguish sets from disjunctions. We will view sets of constraints modulo disjunction. We now proceed to provide a more practical way to implement the composition of sets of size-change graphs. The following example provides the intuition and motivation.

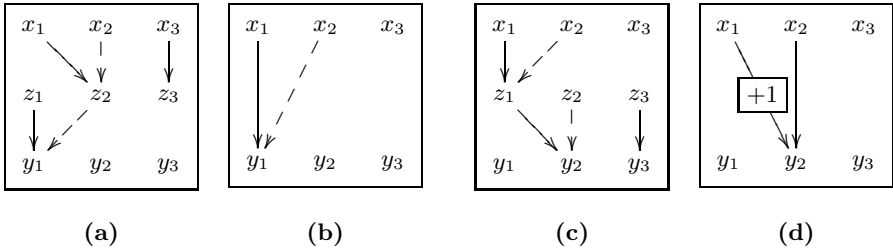


Fig. 4. Constraints of Example 5: **(a-b)** $c_1(\bar{x}, \bar{y}) \circ c_2(\bar{x}, \bar{y}) = c_2(\bar{x}, \bar{y})$; **(c-d)** $c_2(\bar{x}, \bar{y}) \circ c_1(\bar{x}, \bar{y}) \neq \exists \bar{z}. c_2(\bar{x}, \bar{z}) \wedge c_1(\bar{z}, \bar{y})$. (since $\exists z_1. (x_1 > z_1) \wedge (z_1 > y_2) \equiv x_1 + 1 > y_2$)

Example 5. Figure 4(a-b) illustrates the composition of individual size change graphs $c_1(\bar{x}, \bar{y}) \circ c_2(\bar{x}, \bar{y}) = c_2(\bar{x}, \bar{y})$ for the size change graphs of Figure 2. The result of the composition is equivalent to the projected conjunction of the original constraints: $\exists \bar{z}. c_1(\bar{x}, \bar{z}) \wedge c_2(\bar{z}, \bar{y})$. This correspondence follows because the relations $>$ and \geq satisfy $> \circ \geq = >$ and $\geq \circ \geq = \geq$. This gives hope that we might define the composition of size change graphs in terms of renaming, conjunction and projection: $c_1(\bar{x}, \bar{y}) \circ c_2(\bar{x}, \bar{y}) = \exists \bar{z}. c_1(\bar{x}, \bar{z}) \wedge c_2(\bar{z}, \bar{y})$, and lift the resulting operations to sets defined by disjunctions. However the correspondence does not hold when considering the composition of strict relations such as in $c_2(\bar{x}, \bar{y}) \circ c_1(\bar{x}, \bar{y}) = c_3(\bar{x}, \bar{y})$ as depicted in Figure 4(c-d). The corresponding conjunction is $\exists \bar{z}. c_2(\bar{x}, \bar{z}) \wedge c_1(\bar{z}, \bar{y})$ which is equivalent to $x_1 > y_2 + 1 \wedge x_2 > y_2$. The problem is with the constraint $x_1 > y_2 + 1$ which is not of the form $x_i \succ^b y_j$ and the source of the problem is that $> \circ > \neq >$.

We now refine the interpretation of the constraints in a size-change graph so that composition as well as set based composition can be defined in terms of renaming, conjunction and projection. The key is to weaken the greater than relation.

Definition 10 (weak greater-than). *The binary relation \gg over the non-negative integers is given by $\gg = > \cup \{(a, a) \mid a \text{ is even}\}$.*

The intuition behind \gg is the follows: It is stronger than \geq yet weaker than $>$. The projected conjunction $\exists z.(x > z \wedge z > y)$ is not equivalent to $x > y$ because it misses the tuples $(n+1, n)$. But one of $n+1$ or n is even. Hence, using \gg we can assign z to the even value, and we have that $x \gg y \leftrightarrow \exists z.(x \gg z \wedge z \gg y)$.

Definition 11 (size-change graphs - II). *Reconsider Definition 1 of a size-change graph and Definition 6 of the ranking constraint. But this time the relations \succ^1 and \succ^0 are interpreted as \gg and \geq .*

Lemma 3. *Lemma 1 is not influenced when \succ^1 is interpreted as \gg instead of as $>$ in Definitions 8 and 6.*

Proof. (Sketch) For an individual graph $c(\bar{x}, \bar{y}) \models \mathcal{R}(\bar{x}, \bar{y})$ is equivalent to showing there are no solutions of $c(\bar{x}, \bar{y}) \wedge \bigwedge_{i=1}^n (\neg x_i \succ^1 y_i)$. For $\succ^1 \equiv >$, this means finding a loop including a strict arc in the size change graph $c(\bar{x}, \bar{y})$ with \geq arcs added from each y_i to x_i (since $\neg x_i > y_i \leftrightarrow y_i \geq x_i$). For $\succ^1 \equiv \gg$ this amounts to the same thing except the upwards arcs are $\gg \equiv > \cup \{(a, a) \mid a \text{ is odd}\}$. Clearly a loop including a \gg arc and \gg arc is not satisfiable since values taken by variables in a solution must be nonincreasing, and hence identical around the loop, but then the \gg arc requires that the value is even while \gg requires it is odd.

In the remainder of the paper, size-change graphs and termination constraints are to be interpreted in terms of \gg and \geq unless stated otherwise. We are now in position to obtain set-based composition as conjunction.

Lemma 4 (set based composition). *Let $C_1(\bar{x}, \bar{y})$ and $C_2(\bar{x}, \bar{y})$ be the disjunctive representations of sets of size-change graphs G_1 and G_2 respectively of the forms $p(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), q(\bar{y})$ and $q(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), r(\bar{y})$. Then,*

$$C_1(\bar{x}, \bar{y}) \circ C_2(\bar{x}, \bar{y}) = \exists \bar{z}. (C_1(\bar{x}, \bar{z}) \wedge (C_2(\bar{z}, \bar{y})).$$

Proof. After distributing \wedge over \vee it is left to show that the claim holds for individual disjuncts $c_1(\bar{x}, \bar{y})$ and $c_2(\bar{x}, \bar{y})$. This follows because $\gg \circ \gg = \gg$ and $\gg \circ \geq = \geq \circ \gg = \gg$.

Example 6. Consider the composition of the graphs depicted in Figure 4(c). We have $c_1(\bar{x}, \bar{y}) = (x_1 \gg y_1) \wedge (x_2 \geq y_1)$ and $c_2(\bar{x}, \bar{y}) = (x_1 \gg y_2) \wedge (x_2 \geq y_2) \wedge (x_3 \gg y_3)$. Consider the problematic (renamed) pair of relations $(x_1 \gg z_1)$ from c_1 and $(z_1 \gg y_2)$ from c_2 . The projected conjunction $\exists z_1.(x_1 \gg z_1) \wedge (z_1 \gg y_2)$ in the composition $c_1(\bar{x}, \bar{y}) \circ c_2(\bar{x}, \bar{y})$ now results in $x_1 \gg y_2$ as required.

This completes the theoretical specification of all of the components required to perform set-based size change termination analysis. To make this practical we

still have to provide an adequate data structure to represent sets of size-change graphs and support the set-based operations.

4 Finite Domain Size Change Graphs

We proceed to design an analyzer which computes the closure under composition of the given set of size-change graphs and then tests each disjunctive constraint $C(\bar{x}, \bar{y})$ in the closure for the existence of a ranking function using the test $C(\bar{x}, \bar{y}) \models \mathcal{R}(\bar{x}, \bar{y})$ as provided by Lemma 1.

One idea is to apply a general-purpose constraint solver such as CLP(R) [7]. This is the choice taken in TerminWeb [4]. The problem is that CLP(R) does not handle natively the disjunctions found in size-change constraints. TerminWeb represents the disjunctions of size change graphs as sets of binary clauses, and implements set-based operations by considering individual disjuncts.

The alternative approach presented in this paper is based on modeling (disjunctive) size-change graphs by *finite-domain constraints*. All atomic operations of the analyzer take sets as objects.

To obtain a representation based on finite domain constraints we define the *restriction* of a constraint to a finite non-negative integer domain.

Definition 12 (domain restriction). *The restriction of a (size-change) constraint $C(\bar{x}, \bar{y})$ to the first d non-negative integers $[0 \dots d - 1]$ is denoted by $[C(\bar{x}, \bar{y})]_d$ and given by:*

$$[C(\bar{x}, \bar{y})]_d \equiv C(\bar{x}, \bar{y}) \wedge \bigwedge_{i=1 \dots n} x_i, y_i \in [0 \dots d - 1]$$

For all practical purposes there is no loss of information when restricting sets of size change graphs to a sufficiently large domain. The intuition is that for any solution of a set of size change graphs with d nodes, the same ordering between the values can be represented with only d different non-negative integers. However, there are subtleties. The next three lemmata illustrate that the representation and operations are preserved.

Lemma 5. *Let $C_1(\bar{x}, \bar{y})$ and $C_2(\bar{x}, \bar{y})$ be disjunctive size-change constraints with $|\bar{x}| = m$ and $|\bar{y}| = n$. Then $C_1(\bar{x}, \bar{y})$ is equivalent to $C_2(\bar{x}, \bar{y})$ if and only if $[C_1(\bar{x}, \bar{y})]_{m+n}$ is equivalent to $[C_2(\bar{x}, \bar{y})]_{m+n}$.*

Proof. If the constraints are equivalent then clearly the corresponding restrictions are as well. Consider the opposite direction and assume for the purpose of contradiction that $C_1(\bar{x}, \bar{y})$ and $C_2(\bar{x}, \bar{y})$ define the same sets of solutions over the domain of $m + n$ values, but differ in at least one solution over the infinite non-negative integer domain. Assume without loss of generality that $\theta = \{x_1/v_1, \dots, x_m/v_m, y_1/v_{m+1}, \dots, y_n/v_{m+n}\}$ is a solution of $C_1(\bar{x}, \bar{y})$ but not of $C_2(\bar{x}, \bar{y})$. Consider first the case where size change graphs are interpreted in terms of the binary relations $>$ and \geq . Define a solution θ' which maps each

variable of \bar{x} and \bar{y} to its index in the ascending order induced on the corresponding values $\{v_1, \dots, v_{m+n}\}$. We also make sure that two (or more) variables mapped by θ to the same value v are mapped to the same index by θ' . Clearly, all pairwise relations imposed on \bar{x} and \bar{y} by θ are preserved intact by θ' . Thus, θ' is a solution of $C_1(\bar{x}, \bar{y})$ but not of $C_2(\bar{x}, \bar{y})$ in contradiction to the assumption that they define the same set of solutions over the domain of $m + n$ values.

Now consider the case where size change graphs are interpreted over the relations \gg and \geq . We have an additional requirement on θ' from the previous case. If for a pair of variables $x, y \in \bar{x} \cup \bar{y}$ we have $\theta(x) = \theta(y) = v$ and the corresponding $\theta'(x) = \theta'(y) = v'$ then we require that v' is even if and only if v is even. Note that the domain of $m + n$ distinct values is still sufficient for defining θ' .

Lemma 6 (finite-domain termination test). *Let $p_{/n}(\bar{x}) \leftarrow C(\bar{x}, \bar{y}), p_{/n}(\bar{y})$ be a disjunctive (recursive) size change graph. Then,*

$$C(\bar{x}, \bar{y}) \models \mathcal{R}(\bar{x}, \bar{y}) \Leftrightarrow [C(\bar{x}, \bar{y})]_{2n} \models [\mathcal{R}(\bar{x}, \bar{y})]_{2n}$$

Proof (sketch). We need to show that $C(\bar{x}, \bar{y}) \wedge \neg \mathcal{R}(\bar{x}, \bar{y})$ is satisfiable if and only if $[C(\bar{x}, \bar{y}) \wedge \neg \mathcal{R}(\bar{x}, \bar{y})]_{2n}$ is satisfiable. The constraint $C(\bar{x}, \bar{y}) \wedge \neg \mathcal{R}(\bar{x}, \bar{y}) = C(\bar{x}, \bar{y}) \wedge \bigwedge_{i=1}^n \neg(x_i \gg y_i)$ is a constraint based on pairwise order relations between the elements of \bar{x} and \bar{y} . We assume a solution θ of that constraint and show using the same mapping as in the proof of Lemma 5 that the constraint is satisfiable if and only if it is satisfiable over the domain of $2n$ elements.¹ The claim follows by observing (through the straightforward transformation) that $[C(\bar{x}, \bar{y})]_{2n} \models [\mathcal{R}(\bar{x}, \bar{y})]_{2n}$ is equivalent to $[C(\bar{x}, \bar{y})]_{2n} \models \mathcal{R}(\bar{x}, \bar{y})_{2n}$.

Lemma 7 (finite domain composition). *Let $C_1(\bar{x}, \bar{z})$ and $C_2(\bar{z}, \bar{y})$ be disjunctive size-change constraints with $|\bar{x}| = m$ and $|\bar{y}| = n$. Then*

$$[C_1(\bar{x}, \bar{z})]_{m+n} \circ [C_2(\bar{z}, \bar{y})]_{m+n} = [C_1(\bar{x}, \bar{z}) \circ C_2(\bar{z}, \bar{y})]_{m+n}$$

Proof (sketch). The proof technique is similar to that of Lemma 5. We assume a solution ϕ on $\bar{x} \cup \bar{y}$ of $C_1(\bar{x}, \bar{z}) \circ C_2(\bar{z}, \bar{y})$. By definition there exist $c_1(\bar{x}, \bar{z}) \models C_1(\bar{x}, \bar{z})$ and $c_2(\bar{z}, \bar{y}) \models C_2(\bar{z}, \bar{y})$ and solution

$$\theta = \{x_1/v_{x1}, \dots, x_m/v_{xm}, y_1/v_{y1}, \dots, y_n/v_{yn}, z_1/v_{z1}, \dots, z_l/v_{zl}\}$$

of the conjunction $c_1(\bar{x}, \bar{z}) \wedge c_2(\bar{z}, \bar{y})$ extending ϕ (i.e. $\phi(v) = \theta(v), v \in \bar{x} \cup \bar{y}$) and thus, of each of $c_1(\bar{x}, \bar{z})$ and $c_2(\bar{z}, \bar{y})$ individually. We show that this solution can be transformed to another solution θ' with at most $m + n$ distinct values in the range, yet preserving the “ \gg ”-order relations for each pair $(x, z) \in (\bar{x} \times \bar{z})$ and $(z, y) \in (\bar{z} \times \bar{y})$. So θ' is a solution of $[C_1(\bar{x}, \bar{z})]_{m+n}$ and $[C_2(\bar{z}, \bar{y})]_{m+n}$. The transformation starts by ordering the variables in \bar{x} , \bar{y} and \bar{z} with respect to

¹ Note that $C(\bar{x}, \bar{y}) \wedge \neg \mathcal{R}(\bar{x}, \bar{y})$ is not a size-change graph and thus, it is not always satisfiable. However, the proof technique of Lemma 5 applies to any constraints based on pairwise order relations.

their assigned values v_i . Then we “shift” the values for the variables of \bar{x} and \bar{y} until each variable of \bar{z} shares its assigned value with either a variable of \bar{x} or a variable of \bar{y} . The transformation is always possible and thus, it is sufficient to prove the claim only for the solutions with at most $m + n$ distinct values in the range of the substitution. In that case the operation of domain restriction $[\cdot]_{m+n}$ degenerates to an identity, and the two parts of the formula in the claim become the same.

5 Size Change Termination with k Bits

To facilitate efficient size change termination analysis we observe that in practice it is often sufficient to interpret size change constraints over a finite domain with a smaller number of values than nodes in the graphs. As our implementation is based on a Boolean representation of finite domain constraints we will consider values in binary form and typically chose a number of values d of the form $d = 2^k$. Experimental results indicate that all of the size change termination problems in our benchmark suite are guaranteed to be analysed correctly using a 2-bit representation. The following example illustrates the main idea.

Example 7. Consider the disjunctive representation for the graphs in Figure 2:

$$(x_1 \gg y_2 \wedge x_2 \geq y_2 \wedge x_3 \gg y_3) \vee (x_1 \gg y_1 \wedge x_2 \geq y_1) \vee (x_1 \gg y_2 \wedge x_2 \gg y_2).$$

From the results of the previous section we know that for termination analysis we can consider solutions over 6 values. However, note that the constraints in this example are “partitioned” in two blocks of nodes: $I = \{x_1, x_2, y_1, y_2\}$ and $J = \{x_3, y_3\}$. There are no constraints linking the nodes of I and J . As we shall see we can interpret the constraint over the domain of 4 elements i.e., the size of the larger partition.

Definition 13 (partitioning size-change constraints). *We say that a (disjunctive) size-change constraint $C(\bar{x}, \bar{y})$ can be partitioned if the set of arguments $\bar{x} \cup \bar{y}$ can be partitioned into two disjoint non-trivial subsets I and J such that $C(\bar{x}, \bar{y}) \equiv (\exists I.C(\bar{x}, \bar{y})) \wedge (\exists J.C(\bar{x}, \bar{y}))$.*

We proceed to formalize the intuition of Example 7.

Lemma 8. *Let $C_1(\bar{x}, \bar{y})$ and $C_2(\bar{x}, \bar{y})$ be size-change graphs that admit the same partitioning induced by the sets I and J of nodes. Then $C_1(\bar{x}, \bar{y})$ is equivalent to $C_2(\bar{x}, \bar{y})$ if and only if $[C_1(\bar{x}, \bar{y})]_{\max(|I|, |J|)}$ is equivalent to $[C_2(\bar{x}, \bar{y})]_{\max(|I|, |J|)}$.*

Proof. Assume for the purpose of contradiction that $C_1(\bar{x}, \bar{y})$ and $C_2(\bar{x}, \bar{y})$ are not equivalent while $[C_1(\bar{x}, \bar{y})]_{\max(|I|, |J|)}$ and $[C_2(\bar{x}, \bar{y})]_{\max(|I|, |J|)}$ are. That means that either $\exists I.C_1(\bar{x}, \bar{y})$ is not equivalent to $\exists I.C_2(\bar{x}, \bar{y})$ or $\exists J.C_1(\bar{x}, \bar{y})$ is not equivalent to $\exists J.C_2(\bar{x}, \bar{y})$. By Lemma 5 the equivalence of $\exists I.C_1(\bar{x}, \bar{y})$ and $\exists I.C_2(\bar{x}, \bar{y})$ can be tested using a finite domain of at most $|J|$ elements. Similarly, the equivalence of $\exists J.C_1(\bar{x}, \bar{y})$ and $\exists J.C_2(\bar{x}, \bar{y})$ can be tested using a finite domain of at

most $|I|$ elements. Thus, if $C_1(\bar{x}, \bar{y})$ and $C_2(\bar{x}, \bar{y})$ are not equivalent, then there must be a substitution over the domain of (at most) $\max(|I|, |J|)$ elements which distinguishes between the two constraints. Hence, we have a contradiction.

In a similar way we can tighten the bounds of Lemma 6 and Lemma 7 which show distributivity of domain restriction and the operations (composition and testing for termination). Moreover, for correctness of the analysis there is now an additional operation to consider: partitioning.

Lemma 9. *Let size-change constraint $C(\bar{x}, \bar{y})$ admit a partition (I, J) of arguments such that $\max(|I|, |J|) = m > 2$. Then the size-change graph $[C(\bar{x}, \bar{y})]_m$ admits the same partition.*

Lemma 9 does not hold for $m = 2$ because a constraint $x \gg y$ for $x, y \in I$ in the two-value domain implies $y = 0$ and hence, $x' \gg y$ for any x' , including $x' \in J$ (and vice versa). Hence, the restriction to two values introduces new dependencies between the elements of I and J . For $m > 2$ and without loss of generality for $x \in I$ and $y \in J$, we can assign values to x and y so that either $x \gg y$ or $\neg(x \gg y)$ holds.

In a recent paper [1] Ben-Amram and Lee provide the following definitions which we will make use of.

Definition 14 (size relation graph [1]). *Let G be a set of size change graphs. The corresponding size-relation graph, denoted $\text{srg}(G)$, is the annotated digraph with vertex set $\text{Par}(G)$ and an edge from $p_{\langle i \rangle}$ to $q_{\langle j \rangle}$ labelled by $\langle b, g \rangle$ if $g = p_{/n}(\bar{x}) \leftarrow c(\bar{x}, \bar{y}), q_{/m}(\bar{y})$ is a graph in G and $x_i \succ^b y_j$ an edge in g .*

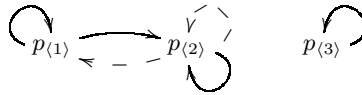


Fig. 5. Size relation graph for the graphs of Figure 2

Definition 15 ($\text{clean}(G)$ [1]). *For a set G of size change graphs, $\text{clean}(G)$ is the set of graphs G minus every arc not belonging to a strongly connected component of $\text{srg}(G)$ that contains a label $b = 1$. If $G = \text{clean}(G)$ we say that G is clean.*

Example 8. Consider the set of graphs G from Figure 2. The corresponding size-relation graph $\text{srg}(G)$ is depicted as Figure 5. Observe that G is clean.

The following lemma enables us to restrict attention to sets of cleaned size-change graphs.

Lemma 10 ([1]). *A set of size change graphs G satisfies size change termination if and only if $\text{clean}(G)$ does.*

To determine the number of bits required to perform size change termination analysis for a set of graphs it is sufficient to check the size of the largest strongest connected component in $srg(clean(G))$.

Definition 16 (diameter). *Let G be a set of size change graphs. The diameter of G is the largest number of parameters with the same predicate symbol in a strongly connected component of $srg(clean(G))$.*

Example 9. The diameter of the set of graphs depicted in Figure 2 is 2.

For a set of size-change graphs G , the strongly connected components of $srg(clean(G))$ indicate a partitioning of the nodes of G . This provides a safe bound on the number of values required to represent G .

6 Implementation and Experimentation

Boolean Encoding: We first illustrate how the binary relations \geq and \gg are modelled for k -bit non-negative integers. Let $\langle v_{k-1}, \dots, v_0 \rangle$ denote the k -bit binary representation of non-negative integer variable v with left most significant binary digit. The k -bit relation $v \geq w$ is standardly modelled inductively by the following Boolean function:

$$\begin{aligned} \langle \rangle \geq \langle \rangle &\equiv 1 \text{ (true)} \\ \langle v_{k-1}, \dots, v_0 \rangle \geq \langle w_{k-1}, \dots, w_0 \rangle &\equiv (v_{k-1} \wedge \neg w_{k-1}) \vee \\ &\quad ((v_{k-1} \leftrightarrow w_{k-1}) \wedge \langle v_{k-2}, \dots, v_0 \rangle \geq \langle w_{k-2}, \dots, w_0 \rangle) \end{aligned}$$

The non-standard relation $v \gg w$ of Definition 10 is modelled as

$$v \gg w \equiv v \geq w \wedge ((v \neq w) \vee even(v))$$

where $(v \neq w) \equiv \neg \bigwedge_{i=1}^n (v_i \leftrightarrow w_i)$ and $even(v) \equiv \neg v_0$ (the least significant bit is 0). Note that the above formula is equivalent to Definition 10.

Example 10. For $k = 2$ the relation $x \geq y$ is modelled by $(x_1 \wedge \neg y_1) \vee ((x_1 \leftrightarrow y_1) \wedge (x_0 \rightarrow y_0))$. Note that the models of this formula correspond to the solutions of the constraint $x \geq y$ on the set of four values $\{0, \dots, 3\}$.

The Boolean encodings of sets of size-change graphs and their set-based operations are obtained from the encoding of the binary relations given above and the set-based definitions of Sections 2 and 3. Size-change graphs are modelled as conjunctions of binary relations. Sets of size-change graphs are modelled as disjunctions of the models of individual size-change graphs. Composition of sets of size-change graphs is modelled through renaming, conjunction and projection. Finally, testing a set of size-change graphs for termination amounts to checking the entailment on two Boolean formula.

A key strength of our approach is that all of the components of the size-change termination analysis can be represented as Boolean formula and standard Boolean operations. This facilitates an implementation based on well-studied data structures and well-engineered tools for representing and manipulating Boolean formulae.

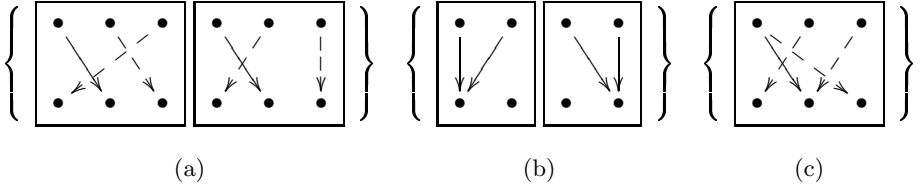


Fig. 6. one-to-one, fan-in and fan-out

Prototype Implementation: To validate our ideas, we have constructed a prototype size-change termination analyser. We use *Reduced Ordered Binary Decision Diagrams* [2] (ROBDDs, often just called BDDs) to represent size-change graphs. ROBDDs are a standard — perhaps *the* standard — representation of Boolean formulae for many applications. They have been applied successfully to representation of sets of constraints over a finite domain [6]. In the context of this work we apply ROBDDs to represent sets of size-change graphs and the respective set-based operations.

Our analyzer comprises about 500 lines of Prolog code and is implemented in SWI-Prolog [10]. It utilizes the freely available CUDD [5] package as a back-end for manipulating BDDs and a previously developed module interfacing CUDD with SWI-Prolog (around 650 lines of C-code.)

Results: The analyzer has been applied to a benchmark suite consisting of 339 size-change termination problems. These problems have been generated from the benchmark suite of TerminWeb [9]. The problems can be obtained from <http://www.cs.bgu.ac.il/~mcodish/TerminWeb/scg.tgz>. All 339 problems have diameter two or less. The total analysis time for the benchmarking suite and a 2-bit representation is 1.2 CPU seconds on a 1GHz machine running GNU/Linux 2.4. The longest running single test takes 70 milliseconds. Preliminary comparison indicates that the performance of our analyzer is far superior to the corresponding components of TerminWeb (orders of magnitude).

We note that our analyzer can also handle hard instances of the underlying PSPACE-complete problem. For instance, the example used in the proof of PSPACE-hardness in Theorem 5 of [8] takes 0.4 second to analyze. Unlike the benchmarks of TerminWeb this example has a diameter of 5 and thus, 3-bit encodings of graph nodes are required for its analysis.

7 Related Work

Ben-Amram and Lee introduce a polynomial algorithm (termed SCP) which covers many instances of size-change termination [1]. SCP is shown to be complete for sets of size-change graphs which are “one-to-one” (the in- and out-degree of all nodes is not more than 1). Their basic SCP algorithm is correct but not complete for sets of graphs which are “fan-in free” (the in-degree of all nodes is not more than 1) and several techniques to handle certain kinds of graphs

with “fan-in” are also proposed. Experimental evaluation indicates that SCP is complete for their benchmark suite (circa 90 SCT problems).

The set of graphs depicted as Figure 6(a) is one-to-one. The SCP algorithm will detect that these graphs satisfy size change termination in polynomial time without computing the expensive closure operation. The graphs in Figure 6(b) have fan-in and SCP does not detect that they are terminating. The graph in Figure 6(c) has both fan-in and fan-out. Its termination also cannot be detected using SCP.

In contrast, our k -bit representation is always complete for any set of size change graphs with diameter 2^{k-1} or less. Experimental evaluation indicates that for our benchmark suite (which extends the one used by Ben-Amram and Lee and consists of 339 SCT problems) all of the examples have diameter 2 or less (after cleanup). Hence the 2-bit size-change termination analysis is guaranteed to be complete. For the examples of Figure 6 our technique requires a 3-bit analysis for (a) and (c) and a 2-bit analysis for (b).

8 Conclusion

This paper proposes a constraint-based approach to size-change termination analysis. We model size-change graphs, sets of size-change graphs and operations for size-change termination using Boolean functions. We draw on experience from Boolean functions where representing large sets of models is well studied. A key step in our design is the non-standard interpretation of size-change relations “ $>$ ” and “ \geq ”. This enables us to encode union and composition of sets of size-change graphs by disjunction and conjunction. The proposed approach has been implemented using BDD-based modeling and BDD operations. The initial performance indicators are highly encouraging.

Acknowledgement

We thank Samir Genaim for generating the size-change termination problems from the benchmark suite of TerminWeb, and the referees for their insightful comments.

References

1. A. M. Ben-Amram and C. S. Lee. Size-change termination in polynomial time. submitted for publication, 2004.
2. R. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
3. M. Codish, V. Lagoon, and P. J. Stuckey. Testing for termination with monotonicity constraints. In G. Gabbrielli, Maurizio; Gupta, editor, *21st International Conference, ICLP*, volume 3668 of *LNCS*. Springer, October 2005.
4. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

5. <http://vlsi.colorado.edu/~fabio/CUDD/>.
6. P. Hawkins, V. Lagoon, and P. J. Stuckey. Solving set constraint satisfaction problems using ROBDD's. *Journal of Artificial Intelligence Research*, 24:106–156, 2005.
7. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP(\mathcal{R}) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.
8. sC. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. *ACM SIGPLAN Notices*, 36(3):81–92, 2001. Proceedings of POPL'01.
9. C. Taboch, S. Genaim, and M. Codish. Terminweb: Semantic based termination analyser for logic programs, 1999-2002.
<http://www.cs.bgu.ac.il/~mcodish/TerminWeb>.
10. J. Wielemaker. An overview of the SWI-Prolog programming environment. In F. Mesnard and A. Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, Dec. 2003. Katholieke Universiteit Leuven. CW 371.

Path Optimization in Programs and Its Application to Debugging^{*}

Akash Lal, Junghee Lim, Marina Polishchuk, and Ben Liblit

Computer Sciences Department,
University of Wisconsin-Madison
{akash, junghee, mpoli, liblit}@cs.wisc.edu

Abstract. We present and solve a path optimization problem on programs. Given a set of program nodes, called critical nodes, we find a shortest path through the program's control flow graph that touches the maximum number of these nodes. Control flow graphs over-approximate real program behavior; by adding dataflow analysis to the control flow graph, we narrow down on the program's actual behavior and discard paths deemed infeasible by the dataflow analysis. We derive an efficient algorithm for path optimization based on weighted pushdown systems. We present an application for path optimization by integrating it with the Co-operative Bug Isolation Project (*CBI*), a dynamic debugging system. CBI mines instrumentation feedback data to find suspect program behaviors, called bug predictors, that are strongly associated with program failure. Instantiating critical nodes as the nodes containing bug predictors, we solve for a shortest program path that touches these predictors. This path can be used by a programmer to debug his software. We present some early experience on using this hybrid static/dynamic system for debugging.

1 Introduction

Static analysis of programs has been used for a variety of purposes including compiler optimizations, verification of safety properties, and improving program understanding. Static analysis has the advantage of considering all possible executions of a program, thus giving strong guarantees on the program's behavior. In this paper, we present a static analysis technique for finding a program execution sequence that is optimal with respect to some criteria. Given a set of program locations, which we call *critical nodes*, we find a trace among all possible program execution traces that touches the maximum number of these critical nodes and has the shortest length among all such traces. Since reachability in programs is undecidable in general, we over-approximate the set of all possible traces through a program by considering all paths in its control flow graph, and solve the optimization problem on this collection of paths. We also consider how to more closely approximate actual program behavior by discarding paths in the control flow graph deemed infeasible by dataflow analysis [1]. We show that the powerful framework of weighted pushdown systems [2] can be used to represent and solve several variations of the path optimization problem.

^{*} Supported in part by a gift from Microsoft Research.


```

int **a;

void main() {
    init(a);
    ...
    process(a);
    ...
}

void clear(int **a) {
    for(...)
        a[i] = NULL;
}

void process(int **a) {
    switch(getchar()) {
        case 'e' :
            clear(a);
            break;
        case 'p' :
            ...
    }
    ...
    a[i][j]++;
}

```

Fig. 1. A buggy program fragment

Why is it important to find paths? Consider the program fragment shown in Figure 1 and suppose that it crashes on some input at line “`a[i][j]++`”. While debugging the program, we find out (using some analysis) that only the statement “`a[i] = NULL`” in `clear()` could have caused a null-pointer dereference at the crash site. However, looking at this line in isolation gives no indication of what the actual bug is. When we construct a path in the program from the entry point of `main()` to the crash site that visits this suspect line in `clear()` we get a path that touches statements shown in bold in Figure 1. It shows that the program can call `clear()` from `process()` and then continue execution onto the crash site. Closer examination of this path may suggest that the `break` statement after `clear()` should have been a `return` statement. Seeing paths allows a richer understanding of program behavior than merely examining isolated statements or procedures.

We have implemented our path optimization algorithm and integrated it with the Cooperative Bug Isolation Project (CBI) [3] to

create the BTRACE debugging support tool. CBI adds lightweight dynamic instrumentation to software to gather information about runtime behavior. Using this data, it identifies suspect program behaviors, called *bug predictors*, that are strongly associated with program failure. Bug predictors expose the causes and circumstances of failure, and have been used successfully to find previously unknown bugs [4]. CBI is primarily a dynamic system based on mining feedback data from observed runs. Our work on BTRACE represents the first major effort to combine CBI’s dynamic approach with static program analysis.

BTRACE enhances CBI output by giving more context for interpreting bug predictors. Using CBI bug predictors as our set of critical nodes, we construct a path from the entry point of the program to the failure site that touches the maximum number of these predictors. CBI associates a numerical score with each bug predictor, with higher scores denoting stronger association with failure. We therefore extend BTRACE to find a shortest path that maximizes the sum of the scores of the predictors it touches. That is, BTRACE finds a path such that the sum of predictor scores of all predictors on the path is maximal, and no shorter path has the same score. We also allow the user to restrict attention to paths that have unfinished calls exactly in the order they appear in a stack trace left behind by the failed program, and to impose constraints on the order in which predictors can be touched. These constraints enhance the utility of BTRACE for

debugging purposes by producing a path that is close enough to the actual failing execution of the program to give the user substantial insight into the root causes of failure. We present experimental results in Section 4 to support this claim.

Under the extra constraints described above, the path optimization problem solved by BTRACE can be stated as follows:

THE BTRACE PROBLEM. *Given the control flow graph (N, E) of a program having nodes N and edges E ; a single node $n_f \in N$ (representing the crash site of a program); a set of critical nodes $B \subseteq N$ (representing the bug predictors); and a function $\mu : B \rightarrow \mathbb{R}$ (representing predictor scores), find a path in the control flow graph that first maximizes $\sum_{n \in S} \mu(n)$ where $S \subseteq B$ is the set of critical nodes that the path touches and then minimizes its length. Furthermore, restrict the search for this optimal path to only those paths that satisfy the following constraints:*

1. **Stack trace.** *Given a stack trace, consider only those paths that reach n_f with unfinished calls exactly in the order they appear in the stack trace.*
2. **Ordering.** *Given a list of node pairs (n_i, m_i) where $n_i, m_i \in B$ and $0 \leq i \leq k$ for some k , consider only those paths that do not touch node m_i before node n_i .*
3. **Dataflow.** *Given a dataflow analysis framework, consider only those paths that are not ruled out as infeasible by the dataflow analysis. The requirements on the dataflow analysis framework are specified in Section 3.4.*

Finding a feasible path through a program when one exists is, in general, undecidable. Therefore, even with powerful dataflow analysis, BTRACE can return a path that will never appear in any real execution of the program. We consider this acceptable as we judge the usefulness of a path by how much it helps a programmer debug her program, rather than its feasibility.

The key contributions of this paper are as follows:

- We present an algorithm that optimizes path selection in a program according to the criteria described above. We use weighted pushdown systems to provide a common setting under which all of the mentioned optimization constraints can be satisfied.
- We describe a hybrid static/dynamic system that combines optimal path selection with CBI bug predictors to support debugging.

The remainder of the paper is organized as follows: Section 2 presents a formal theory for representing paths in a program. Section 3 derives our algorithm for finding an optimal path. Section 4 considers how path optimization can be used in conjunction with CBI for debugging programs and presents experimental results demonstrating that the approach is feasible. Section 5 discusses some of the related work in this area, and Section 6 concludes with some final remarks.

2 Describing Paths in a Program

This section introduces the basic theory behind our approach. In Section 2.1, we formalize the set of paths in a program as a pushdown system. Section 2.2 introduces weighted pushdown systems that have the added ability to associate a value with each path.

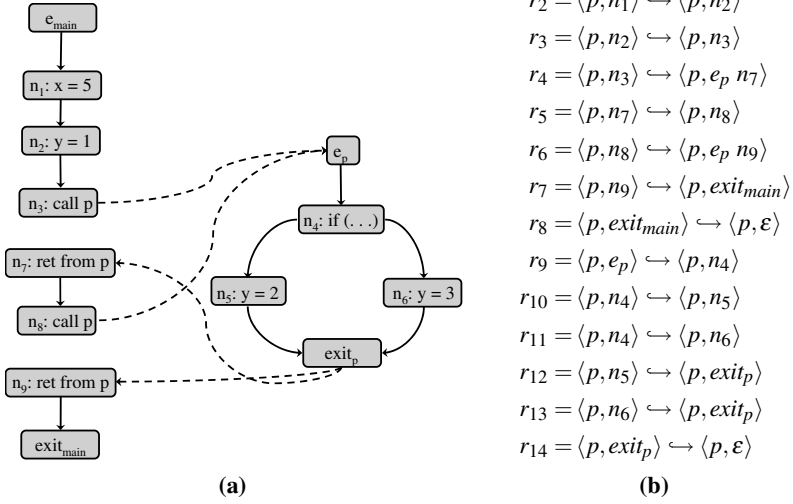


Fig. 2. (a) A control flow graph. The e and $exit$ nodes represent entry and exit points of procedures, respectively. Dashed edges represent interprocedural control flow. (b) A pushdown system that models the control flow graph shown in (a). It uses a single state p and has one rule per CFG edge. Rules r_4 and r_6 correspond to procedure calls and save the return site on the stack. Rules r_8 and r_{14} simply pop-off the top of the stack to reveal the most recent return site.

2.1 Paths in a Program

A control flow graph (CFG) of a program is a graph where nodes are program statements and edges represent possible flow of control between statements. Figure 2a shows the CFG of a program with two procedures. We adopt the convention that each procedure call in the program is represented by two nodes: one is the source of an interprocedural call edge to the callee's entry node and the second is the target of an interprocedural return edge from the callee's exit node back to the caller. In Figure 2a, nodes n_3 and n_7 represent one call from $main$ to p ; nodes n_8 and n_9 represent a second call.

Not all paths (sequences of nodes connected by edges) in the CFG are valid. For example, the path $[e_{main} \ n_1 \ n_2 \ n_3 \ e_p \ n_4 \ n_5 \ exit_p \ n_9]$ is invalid because the call at node n_3 should return to node n_7 , not node n_9 . In general, the valid paths in a CFG are described by a context-free language of matching call/return pairs: for each call, only the matching return edge can be taken at the exit node. For this reason, it is natural to use pushdown systems to describe valid paths in a program [2, 5].

Definition 1. A *pushdown system* (PDS) is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ of finite sets where P is the set of states, Γ is the set of stack symbols and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is the set of pushdown rules. A rule $r = (p, \gamma, q, u) \in \Delta$ is written as $\langle p, \gamma \rangle \hookrightarrow \langle q, u \rangle$.

A PDS is a finite automaton with a stack (Γ^*). It does not take any input, as we are interested in the transition system it describes, not the language it generates.

Definition 2. A **configuration** of a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ is a pair $\langle p, u \rangle$ where $p \in P$ and $u \in \Gamma^*$. The rules of the pushdown system describe a **transition relation** \Rightarrow on configurations as follows: if $r = \langle p, \gamma \rangle \hookrightarrow \langle q, u \rangle$ is some rule in Δ , then $\langle p, \gamma u' \rangle \Rightarrow \langle q, uu' \rangle$ for all $u' \in \Gamma^*$.

The construction of a PDS to represent paths in a CFG is fairly straightforward [2]. An example is shown in Figure 2b. The transition system of the constructed PDS mimics control flow in the program. A sequence of transitions in the transition system ending in a configuration $\langle p, n_1 n_2 \cdots n_k \rangle$, where $n_i \in \Gamma$, is said to have a *stack trace* of $\langle n_1, \dots, n_k \rangle$: it describes a path in the CFG that is currently at n_1 and has unfinished calls corresponding to the return sites n_2, \dots, n_k . In this sense, a configuration stores an abstract run-time stack of the program, and the transition system describes valid changes that the program can make to it.

2.2 Weighted Pushdown Systems

A weighted pushdown system (WPDS) is obtained by associating a *weight* with each pushdown rule. The weights must come from a set that satisfies bounded idempotent semiring properties [2, 6].

Definition 3. A **bounded idempotent semiring** is a quintuple $(D, \oplus, \otimes, \bar{0}, \bar{1})$, where D is a set whose elements are called **weights**, $\bar{0}$ and $\bar{1}$ are elements of D , and \oplus (the combine operation) and \otimes (the extend operation) are binary operators on D such that

1. (D, \oplus) is a commutative monoid with $\bar{0}$ as its neutral element, and where \oplus is idempotent (i.e., for all $a \in D$, $a \oplus a = a$).
2. (D, \otimes) is a monoid with $\bar{1}$ as its neutral element.
3. \otimes distributes over \oplus , i.e., for all $a, b, c \in D$ we have

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c).$$
4. $\bar{0}$ is an annihilator with respect to \otimes , i.e., for all $a \in D$, $a \otimes \bar{0} = \bar{0} = \bar{0} \otimes a$.
5. In the partial order \sqsubseteq defined by: $\forall a, b \in D$, $a \sqsubseteq b$ iff $a \oplus b = a$, there are no infinite descending chains.

Definition 4. A **weighted pushdown system** is a triple $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system, $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ is a bounded idempotent semiring and $f : \Delta \rightarrow D$ is a map that assigns a weight to each pushdown rule.

The \otimes operation is used to compute the weight of concatenating two paths and the \oplus operation is used to compute the weight of merging parallel paths. If σ is a sequence of rules $[r_1, r_2, \dots, r_n] \in \Delta^*$, then define the value of σ as $val(\sigma) = f(r_1) \otimes f(r_2) \otimes \cdots \otimes f(r_n)$. In Definition 3, item 3 is required by WPDSs to efficiently explore all paths, and item 5 is required for termination of WPDS algorithms.

For sets of pushdown configurations S and S' , let $path(S, S')$ be the set of all rule sequences that transform a configuration in S to a configuration in S' . Let $n\Gamma^* \subseteq \Gamma^*$ denote the set of all stacks that start with n . Existing work on WPDSs allows us to solve the following problems [2]:

Definition 5. Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system with $\mathcal{P} = (P, \Gamma, \Delta)$ and let $c \in P \times \Gamma^*$ be a configuration. The **generalized pushdown predecessor (GPP_c) problem** is to find for each regular set of configurations $S \subseteq (P \times \Gamma^*)$:

- $\delta(S) \stackrel{\text{def}}{=} \bigoplus \{ \text{val}(\sigma) \mid \sigma \in \text{path}(S, c) \}$
- a **witness set of paths** $\omega(S) \subseteq \text{path}(S, c)$ such that $\bigoplus_{\sigma \in \omega(S)} \text{val}(\sigma) = \delta(S)$.

The **generalized pushdown successor (GPS_c) problem** is to find for each regular set of configurations $S \subseteq P \times \Gamma^*$:

- $\delta(S) \stackrel{\text{def}}{=} \bigoplus \{ \text{val}(\sigma) \mid \sigma \in \text{path}(c, S) \}$
- a **witness set of paths** $\omega(S) \subseteq \text{path}(c, S)$ such that $\bigoplus_{\sigma \in \omega(S)} \text{val}(\sigma) = \delta(S)$.

For the above definition, we avoid defining a *regular* set of configurations by restricting S to be either a single configuration $\{c'\}$ or $n\Gamma^*$ for some $n \in \Gamma$. The above problems can be considered as backward and forward reachability problems, respectively. Each aims to find the combine of values of all paths between given pairs of configurations ($\delta(S)$). Along with this value, we can also find a witness set of paths $\omega(S)$ that together justify the reported value for $\delta(S)$. This set of paths is always finite because of item 5 in Definition 3. Note that the reachability problems do not require finding the *smallest* witness set, but the WPDS algorithms always find a finite set.

3 Finding an Optimal Path

In this section we solve the specific BTRACE problem defined in Section 1. We begin by developing a solution to the basic path optimization problem without considering dataflow or ordering constraints and then add them back one by one.

3.1 Creating a WPDS

Let (N, E) be a CFG and $\mathcal{P} = (P, \Gamma, \Delta)$ be a pushdown system representing its paths, constructed as described in Section 2.1. Let $B \subseteq N$ be the set of critical nodes. We will use this notation throughout this section. We now construct a WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ that can be solved to find the best path.

For each path, we need to keep track of its length and also the set of critical nodes it touches. Let $V = 2^B \times \mathbb{N}$ be a set whose elements each consist of a subset of B (the critical nodes touched) and a natural number (the length of the path). We want to associate each path with an element of V . This is accomplished by defining a weight, which will summarize a set of paths, as a set of elements from V . The combine operation simply takes a union of the weights, but eliminates an element if there is a better one around, i.e., if there are elements (b, v_1) and (b, v_2) , the one with shorter path length is chosen. This drives the WPDS to only consider paths with shortest length. The extend operation takes a union of the critical nodes and sums up path lengths for each pair of elements from the two weights. This reflects the fact that when a path with length v_1 that touches the critical nodes in b_1 is extended with a path of length v_2 that touches the critical

nodes in b_2 , we get a path of length $v_1 + v_2$ that touches the critical nodes in $b_1 \cup b_2$. The semiring constant $\bar{0}$ denotes an infeasible path, and the constant $\bar{1}$ denotes an empty path that touches no critical nodes and crosses zero graph edges. This is formalized in the following definition.

Definition 6. Let $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ be a bounded idempotent semiring where each component is defined as follows:

- The set of weights D is 2^V , the power set of V .
- For $w_1, w_2 \in D$, define $w_1 \oplus w_2$ as $\text{reduce}(w_1 \cup w_2)$, where
$$\text{reduce}(A) = \{(b, v) \in A \mid \nexists (b, v') \in A \text{ with } v' < v\}$$
- For $w_1, w_2 \in D$, define $w_1 \otimes w_2$ as
$$\text{reduce}(\{(b_1 \cup b_2, v_1 + v_2) \mid (b_1, v_1) \in w_1, (b_2, v_2) \in w_2\})$$
- The semiring constants $\bar{0}, \bar{1} \in D$ are $\bar{0} = \emptyset$ and $\bar{1} = \{(\emptyset, 0)\}$.

To complete the description of the WPDS \mathcal{W} , we need to associate each pushdown rule with a weight. If $r = \langle p, n \rangle \hookrightarrow \langle p, u \rangle \in \Delta$, then associate it with the weight $f(r) = \{(\{n\} \cap B, 1)\}$. Whenever the rule r is used, the length of the path is increased by one and the set of critical nodes grows to include n if n is a critical node. It is easy to see that for a sequence of rules $\sigma \in \Delta^*$ that describes a path in the CFG, $\text{val}(\sigma) = \{(b, v)\}$ where b is the set of critical nodes touched by the path and v is its length.

3.2 Solving the WPDS

An optimal path can be found by solving the generalized pushdown reachability problems on this WPDS. We consider two scenarios here: when we have the crash site but do not have the stack trace of the crash, and when both the crash site and stack trace are available. We start with just the crash site. Let $n_e \in N$ be the entry point of the program, and $n_f \in N$ the crash site.

Theorem 1. In \mathcal{W} , solving $\text{GPS}_{\langle p, n_e \rangle}$ gives us $\delta(n_f \Gamma^*) = \{(b, v) \in V \mid \text{there is a path from } n_e \text{ to } n_f \text{ that touches exactly the critical nodes in } b, \text{ and the shortest such path has length } v\}$. Moreover, $\omega(n_f \Gamma^*)$ is a set of paths from n_e to n_f such that there is at least one path for each $(b, v) \in \delta(n_f \Gamma^*)$ that touches exactly the critical nodes in b and has length v .

The above theorem holds because $\text{paths}(\langle p, n_e \rangle, \langle p, n_f \Gamma^* \rangle)$ is exactly the set of paths from n_e to n_f , which may or may not have unfinished calls. Taking a combine over the values of such paths selects, for some subsets $b \subseteq B$, a shortest path that touches exactly the critical nodes in b , and discards the longer ones. The witness set must record paths that justify the reported value of $\delta(n_f \Gamma^*)$. Since the value of a path is a singleton-set weight, it must have at least one path for each member of $\delta(n_f \Gamma^*)$.

When we have a stack trace available as some $s \in (n_f \Gamma^*)$, with n_f being the top-most element of s , we can use either GPS or GPP .

Theorem 2. In \mathcal{W} , solving $\text{GPS}_{\langle p, n_e \rangle} (\text{GPP}_{\langle p, s \rangle})$ gives us the following values for $W_\delta = \delta(\langle p, s \rangle) (\delta(\langle p, n_e \rangle))$ and $W_\omega = \omega(\langle p, s \rangle) (\omega(\langle p, n_e \rangle))$: $W_\delta = \{(b, v) \in V \mid \text{there is a valid path from } n_e \text{ to } n_f \text{ with stack trace } s \text{ that touches all critical nodes in } b, \text{ and the}$

shortest such path has length v } $\}$. $W_\omega =$ a set of paths from n_e to n_f , each with stack trace s such that there is at least one path for each $(b, v) \in W_\delta$ that touches exactly the critical nodes in b and has length v .

The above theorem allows us to find the required values using either *GPS* or *GPP*. The former uses forward reachability, starting from n_e and going forward in the program, and the latter uses backward reachability, starting from the stack trace s and going backwards. Appendix A presents a detailed discussion on the complexity of solving these problems on our WPDS. The worst-case complexity is exponential in the number of critical nodes and (practically) linear in the size of the program. The exponential complexity in critical nodes is, unfortunately, unavoidable. The reason is that the path optimization problem we are trying to solve is a strict generalization of the traveling salesman problem: our objective is to find a shortest path between two points that touches a given set of nodes. However, we did not find this complexity to be a limitation in our experiments.

Having obtained the above W_δ and W_ω values, we can find an optimal path easily. Let $\mu : B \rightarrow \mathbb{R}$ be a user-defined measure that associates a score with each critical node. We compute a score for each $(b, v) \in W_\delta$ by summing up the scores of all critical nodes in b and then choose the pair with highest score. Extracting a path corresponding to that pair in W_ω gives us an optimal path. Some advantages of having such a user-defined measure are that the user can specify bug predictor scores given by CBI, or make up his own scores. The user can also give a negative score to critical nodes that should be *avoided* by the path. Critical nodes with zero score can be added and used for specifying ordering constraints (Section 3.3). This lets our tool work interactively with the user to find a suitable path. More generally, we can allow the user to give a measure $\hat{\mu} : (2^B \times \mathbb{N}) \rightarrow \mathbb{R}$ that directly associates a score with a path. Using such a measure, the user can decide to choose shorter paths instead of paths that touch more critical nodes.

3.3 Adding Ordering Constraints

We now add ordering constraints to the path optimization problem. Suppose that we have a constraint “node n must be visited before node m ,” which says that we can only consider paths that do not visit m before visiting n . It is relatively easy to add such constraints to the WPDS given above. The extend operation is used to compute the value of a path. We simply change it to yield $\bar{0}$ for paths that do not satisfy the above ordering constraint. For $w_1, w_2 \in D$, redefine $w_1 \otimes w_2$ as *reduce*(A) where

$$A = \{(b_1 \cup b_2, v_1 + v_2) \mid (b_1, v_1) \in w_1, (b_2, v_2) \in w_2, \neg(m \in b_1, n \in b_2)\}$$

If we have more than one ordering constraint, then we simply add more clauses, one for each constraint, to the above definition of extend.

These constraints do not change the worst case asymptotic complexity of solving reachability problems in WPDS. However they do help prune down the paths that need to be explored, because each constraint cuts down on the size of weights produced by the extend operation.

3.4 Adding Dataflow Analysis

So far we have not considered interpreting the semantics of the program other than its control flow. This implies that the WPDS can find infeasible paths: ones that cannot

occur in any execution of the program. An example is a path that assigns $x := 1$ and then follows the true branch of the conditional `if (x == 0)`. In general, it is undecidable to restrict attention to paths that actually occur in some program execution, but if we can rule out many infeasible paths, we increase the chances of presenting a feasible or near-feasible path to the user. This can be done using dataflow analysis.

Dataflow analysis is carried out to approximate, for each program variable, the set of values that the variable can take at each point in the program. When a dataflow analysis satisfies certain conditions, it can be integrated into a WPDS by designing an appropriate weight domain [2, 5]. Examples of such dataflow analyses include linear constant propagation [7] and affine relation analysis [8, 9]. In particular, we can use any bounded idempotent semiring weight domain $\mathcal{S}_d = (D_d, \oplus_d, \otimes_d, \bar{0}_d, \bar{1}_d)$ provided that when given a function $f_d : \Delta \rightarrow D_d$ that associates each PDS rule (CFG edge) with a weight, it satisfies the following property: given any (possibly infinite) set $\Sigma \subseteq \Delta^*$ of paths between the same pair of program nodes, we have $\bigoplus_{\sigma \in \Sigma} val_d(\sigma) = \bar{0}_d$ only if all paths in Σ are infeasible, where $val_d([r_1, \dots, r_k]) = f_d(r_1) \otimes_d \dots \otimes_d f_d(r_k)$. In particular this means that $val_d(\sigma) = \bar{0}_d$ only if σ is an infeasible path. This imposes a soundness guarantee on the dataflow analysis: it can only rule out infeasible paths. Details on how classical dataflow analysis frameworks [1] can be encoded as weight domains can be found in Reps et al. [2]. The basic idea is to encode dataflow transformers that capture the effect of executing a program statement, or a sequence of statements, as weights. The extend operation composes transformers and the combine operation takes their *meet* in the dataflow value lattice.

Such a translation from dataflow transformers to a weight domain allows us to talk about the meet-over-all-paths between configurations of a pushdown system. For example, solving $GPS_{\langle p, n_e \rangle}$ on this weight domain gives us $\delta(\langle p, n_1 n_2 \dots n_k \rangle)$ as the combine (or meet) over the values of all paths from n_e to n_1 that have the stack trace $n_1 n_2 \dots n_k$. This is a unique advantage that we gain over conventional dataflow analysis by using WPDSs.

Given \mathcal{S}_d and f_d as above, we change the weight domain of our WPDS as follows.

Definition 7. Let $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ be a bounded idempotent semiring where each component is defined as follows:

- The set of weights D is $2^{B \times \mathbb{N} \times D_d}$, the power set of the set $2^B \times \mathbb{N} \times D_d$.
- For $w_1, w_2 \in D$, define $w_1 \oplus w_2$ as $reduce_d(w_1 \cup w_2)$ where $reduce_d(A)$ is defined as $\{(b, \min\{v_1, \dots, v_n\}, d_1 \oplus_d \dots \oplus_d d_n) \mid (b, v_i, d_i) \in A, 1 \leq i \leq n\}$
- For $w_1, w_2 \in D$, define $w_1 \otimes w_2$ as $reduce_d(A)$ where A is the set
$$\left\{ (b_1 \cup b_2, v_1 + v_2, d_1 \otimes_d d_2) \mid \begin{array}{l} (b_1, v_1, d_1) \in w_1, (b_2, v_2, d_2) \in w_2, d_1 \otimes_d d_2 \\ \neq \bar{0}_d, (b_1, b_2) \text{ satisfy all ordering constraints} \end{array} \right\}$$
- The semiring constants $\bar{0}, \bar{1} \in D$ are $\bar{0} = \emptyset$ and $\bar{1} = \{(\emptyset, 0, \bar{1}_d)\}$.

Here (b_1, b_2) satisfy all ordering constraints iff for each constraint “visit n before m ,” it is not the case that $m \in b_1$ and $n \in b_2$.

The weight associated with each rule $r = \langle p, n \rangle \hookrightarrow \langle p, u \rangle \in \Delta$ is given by $f(r) = \{(\{n\} \cap B, 1, f_d(r))\}$. Each path is now associated with the set of predictors it touches, its length, and its dataflow value. Infeasible paths are removed during the extend operation as

weights with dataflow value $\bar{0}_d$ are discarded. More formally, for a path $\sigma \in \Delta^*$ in the CFG, $val(\sigma) = \{(b, v, w_d)\}$ if $w_d = val_d(\sigma) \neq \bar{0}_d$ is the dataflow value associated with the path, v is the length of the path, b is the set of critical nodes touched by the path, and the path satisfies all ordering constraints. If σ does not satisfy some ordering constraint or if $val_d(\sigma) = \bar{0}_d$, then $val(\sigma) = \emptyset = \bar{0}$. Analysis using this weight domain is similar to the “property simulation” used in ESP [10], where a distinct dataflow value is maintained for each property-state. We maintain a distinct dataflow weight for each subset of critical nodes.

Instead of repeating Theorems 1 and 2, we just present the case of using *GPS* when the stack trace $s \in (n_f \Gamma^*)$ is available. Results for other cases can be obtained similarly.

Theorem 3. *In the WPDS obtained from the weight domain defined in Definition 7, solving $GPS_{\langle p, n_e \rangle}$ gives us the following values:*

- $\delta(\langle p, s \rangle) = \{(b, v, w_d) \mid \text{there is a path from } n_e \text{ to } n_f \text{ with stack trace } s \text{ that visits exactly the critical nodes in } b, \text{ satisfies all ordering constraints, is not infeasible under the weight domain } \mathcal{S}_d, \text{ and the shortest such path has length } v\}$.
- $\omega(\langle p, s \rangle)$ contains at least one path for each $(b, v, w_d) \in \delta(\langle p, s \rangle)$ that goes from n_e to n_f with stack trace s , visits exactly the predictors in b , satisfies all ordering constraints and has length v . More generally, for each $(b, v, w_d) \in \delta(\langle p, s \rangle)$ it will have paths $\sigma_i, 1 \leq i \leq k$ for some constant k such that $val(\sigma_i) = \{(b, v_i, w_i)\}$, $\min\{v_1, \dots, v_k\} = v$, and $w_1 \oplus_d \dots \oplus_d w_k = w_d$.

The worst case time complexity in the presence of dataflow analysis increases by a factor of $H_d(C_d + E_d)$ where H_d is height of \mathcal{S}_d , C_d is time required for applying \oplus_d , and E_d is the time required for applying \otimes_d .

Theorem 3 completely solves the BTRACE problem mentioned in Section 1. The next section presents the dataflow weight domain that we used for our experiments.

3.5 Example and Extensions for Using Dataflow Analysis

Copy Constant Propagation. We now give an example of a weight domain that can be used for dataflow analysis. We encode copy-constant propagation [11] as a weight domain. A similar encoding is used by Sagiv, Reps, and Horwitz [7]. Copy-constant propagation aims to determine if a variable has a fixed constant value at some point in the program. It interprets constant-to-variable assignments ($x := 1$) and variable-to-variable assignments ($x := y$) and abstracts all other assignments as $x := \perp$, which says that x may not have a constant value. We ignore conditions on branches for now.

Let Var be the set of all global (integer) variables of a given program. Let $\mathbb{Z}_\perp^\top = \mathbb{Z} \cup \{\perp, \top\}$ and $(\mathbb{Z}_\perp^\top, \sqcap)$ be the standard constant propagation meet semilattice obtained from the partial order $\perp \sqsubseteq_{cp} c \sqsubseteq_{cp} \top$ for all $c \in \mathbb{Z}$. Then the set of weights of our weight domain is $D_d = Var \rightarrow (2^{Var} \times \mathbb{Z}_\perp^\top)$. Here, $\tau \in D_d$ represents a dataflow transformer that summarizes the effect of a executing a sequence of program statements as follows: if $env : Var \rightarrow \mathbb{Z}$ is the state of the program before the statements are executed and $\tau(x) = (\{x_1, \dots, x_n\}, c)$ for $c \in \mathbb{Z}_\perp^\top$, then the value of a variable x after the statements are executed is $env(x_1) \sqcap env(x_2) \dots \sqcap env(x_n) \sqcap c$. Let $\tau^v(x)$ be the first component of $\tau(x)$ and $\tau^c(x)$ be the second component. Then we can define the semiring operations

as follows: the combine operation is a concatenation of expressions and the extend operation is substitution. Formally, for $\tau_1, \tau_2 \in D_d$,

$$\tau_1 \oplus_d \tau_2 = \lambda x. (\tau_1^v(x) \cup \tau_2^v(x), \tau_1^c(x) \sqcap \tau_2^c(x))$$

$$\tau_1 \otimes_d \tau_2 = \lambda x. (\bigcup_{y \in \tau_2^v(x)} \tau_1^v(y), \tau_2^c(x) \sqcap (\bigcap_{y \in \tau_2^v(x)} \tau_1^c(y)))$$

The semiring constants are given by $\bar{0}_d = \lambda x. (\emptyset, \top)$ and $\bar{1}_d = \lambda x. (\{x\}, \top)$.

Handling Conditionals. Handling branch conditions is problematic because dataflow analysis in the presence of conditions is usually very hard. For example, finding whether a branch condition can ever evaluate to true, even for copy-constant propagation, is PSPACE-complete [12]. Therefore, we have to resort to approximate dataflow analysis, i.e., we give up on computing meet-over-all-paths. This translates into relaxing the distributivity requirement on the weight domain \mathcal{S}_d . Fortunately, WPDSs can handle non-distributive weight domains [2] by relaxing Definition 3 item 3 as follows. If D is the set of weights, then for all $d_1, d_2, d_3 \in D$,

$$d_1 \otimes (d_2 \oplus d_3) \sqsubseteq (d_1 \otimes d_2) \oplus (d_1 \otimes d_3); \quad (d_1 \oplus d_2) \otimes d_3 \sqsubseteq (d_1 \otimes d_3) \oplus (d_2 \otimes d_3)$$

where \sqsubseteq is the partial order defined by $\oplus : d_1 \sqsubseteq d_2$ iff $d_1 \oplus d_2 = d_1$. Under this weaker property, the generalized reachability problems can only be solved approximately, i.e., instead of obtaining $\delta(c)$ for a configuration c , we only obtain a weight w such that $w \sqsubseteq \delta(c)$. For our path optimization problem, this inaccuracy will be limited to the dataflow analysis. We would only eliminate some of the paths that the dataflow analysis can find infeasible and might find a path σ such that $val_d(\sigma) = \bar{0}_d$. This is acceptable because it is not possible to rule out all infeasible paths anyway. Moreover, it allows us the flexibility of putting in a simple treatment for conditions in most dataflow analyses. The disadvantage is that we lose a strong characterization of the type of paths that will be eliminated.

For copy-constant propagation, we extend the set of weights by $\{\rho_e \mid e \text{ is an arithmetic condition}\}$. We associate weight ρ_e with the rule $\langle p, n \rangle \hookrightarrow \langle p, m \rangle$ ($n, m \in \Gamma$) if the corresponding CFG edge can only be taken when e evaluates to true on the program state at n . For example, we associate the weight $\rho_{x=0}$ with the true branch of the conditional `if (x == 0)` and weight $\rho_{x \neq 0}$ with its false branch. The extend operation is modified such that for $\tau \in D_d$, $\tau \otimes \rho_e$ evaluates the condition e under the information provided by τ and results to $\bar{0}_d$ if e evaluates to false. Otherwise, the extend is simply τ . More details can be found in a companion technical report [13].

Handling Local Variables. A recent extension to WPDSs [5] shows how local variables can be handled by using *merge functions* that allow for local variables to be saved before a call and then merged with the information returned by the callee to compute the effect of the call. This treatment for local variables allows us to restrict each weight to manage the local variables of only one procedure. Details of the construction of these merge functions are given in a companion technical report [13].

4 Integrating BTRACE and CBI

The formalisms of Section 3 may be used for solving a variety of optimization problems concerned with touching key program points along some path. BTRACE represents one application of these ideas: an enhancement to the statistical debugging analysis performed by the Cooperative Bug Isolation Project (CBI).

4.1 A Need for Failure Paths

CBI uses runtime instrumentation and statistical modeling techniques to diagnose bugs in widely deployed software. CBI identifies suspect program behaviors, called *bug predictors*, that are strongly associated with program failure. Candidate behaviors may include branch directions, function call results, values of variables, and other dynamic properties [14]. Each bug predictor is assigned a numerical score in \mathbb{R}^+ that balances two key factors: (1) how much this predictor increases the probability of failure, and (2) how many failed runs this predictor accounts for. Thus, high-value predictors warrant close examination both because they are highly correlated with failure and because they account for a large portion of the overall failure rate seen by end users [4].

A key strength of CBI is that it samples behavior for the entire dynamic lifetime of a run; however, interpreting the resulting predictors, which may be located anywhere in the program prior to the failure point, can be very challenging. Rather than work with isolated bug predictors, the programmer would like to navigate forward and backward along the path that led to failure. BTRACE constructs a path that hits several high-ranked predictors. This can help the programmer draw connections between sections of code that, though seemingly unrelated, act in concert to bring the program down.

4.2 BTRACE Implementation

We have implemented BTRACE using the WPDS++ library [15]. To manage the exponential complexity in the number of bug predictors, we efficiently encode weights using abstract decision diagrams (ADDs) provided by the CUDD library [16]. Additional details on how the semiring operations are implemented on ADDs may be found in a companion technical report [13].

A BTRACE debugging session starts with a list of related bug predictors, believed by CBI to represent a single bug. We designate this list (or some high-ranked prefix thereof) as the critical nodes and insert them at their corresponding locations in the CFG. Branch predictors, however, may be treated as a special case. These predictors associate the direction of a conditional with failure, and therefore can be repositioned on the appropriate branch. This can be seen as one example of exploiting not just the location but also the semantic meaning of a bug predictor; branch predicates make this easy because their semantic meaning directly corresponds to control flow.

For dataflow analysis, we track all integer- and pointer-valued variables and structure fields. We do not track the contents of memory and any write to memory via a pointer is replaced with assignments of \perp to all variables whose address was ever taken. Direct structure assignments are expanded into component-wise assignments to all fields of the structure.

4.3 Case Studies: Siemens Suite

We have applied BTRACE to three buggy programs from the Siemens test suite [17]: TCAS v37, REPLACE v8, and PRINT_TOKENS2 v6. These programs do not crash; they merely produce incorrect output. Thus our analysis is performed without a stack trace, instead treating the exit from `main()` as the “failure” point. We find that BTRACE can be useful even for non-fatal bugs.

TCAS has an array index error in a one-line function that contains no CBI instrumentation and thus might easily be overlooked. Without bug predictors, BTRACE produces the shortest possible path that exits `main()`, revealing nothing about the bug. After adding the top-ranked predictor, BTRACE isolates lines with calls to the buggy function.

REPLACE has an incorrect function return value. BTRACE with the top two predictors yields a path through the faulty statement. Each predictor is located within one of two disjoint chains of function calls invoked from `main()`, and neither falls in the same function as the bug. Thus, while the isolated predictors do not directly reveal the bug, the BTRACE failure path through these predictors does.

PRINT_TOKENS2 has an off-by-one error. Again, two predictors suffice to steer BTRACE to the faulty line. Repositioning of branch predictors is critical here. Even with all nineteen CBI-suggested predictors and dataflow analysis enabled, a correct failure path only results if branch predictors are repositioned to steer the path in the proper direction.

4.4 Case Studies: CCRYPT and BC

We have also run BTRACE on two small open source utilities: CCRYPT v1.2 and BC v1.06. CCRYPT is an encryption/decryption tool and BC is an arbitrary precision calculator. Both are written in C. Fatal bugs in each were first characterized in prior work by Liblit et al. [14]. More detailed discussion of experimental results can be found in a companion technical report [13].

CCRYPT has an input validation bug. Reading end-of-file yields a NULL string (`char *`) that is subsequently dereferenced without being checked first. If given only a stack trace, BTRACE builds an infeasible path that takes several impossible shortcuts through initialization code. These shortcuts also yield NULL values, but in places that are properly checked before use and therefore cannot be the real bug. The path remains the same if we add dataflow analysis (but no bug predictors), or if we add up to fourteen bug predictors (but no dataflow analysis).

However, if BTRACE uses both dataflow analysis and at least eleven bug predictors, the failure path changes to a feasible path that correctly describes the bug: non-NULL values in the well-checked initialization code, and a fatal unchecked NULL value later on. This feasible path also arises from just a stack trace if one manually inserts ordering constraints to require that bug predictors appear after initialization code, e.g. if the initialization code were assumed to be bug-free. The combination of dataflow analysis and bug predictors make such manual, a priori assumptions unnecessary.

BC has a buffer overrun: a bad loop index in `more_arrays()` silently trashes memory. The program keeps running but may eventually crash during a subsequent call

to `bc_malloc()`. The stack trace at the point of failure suggests heap corruption but provides no real clues as to when the corruption occurred or by what piece of code. CBI-identified bug predictors are scattered across several files and their relationship may not be clear on first examination.

Using one bug predictor, BTRACE builds a path that calls `more_arrays()` early in execution. This path is feasible but misleading: `more_arrays()` is always called early in execution, and only a second or subsequent call to `more_arrays()` can cause failure. Using two or more bug predictors forces the path to include a fatal second call to `more_arrays()`, correctly reflecting the true bug. By reading in-progress calls out of the failure trace, we can easily reconstruct the entire stack at the call to `more_arrays()` or any other point of interest and thereby give deeper context to the frontier of the bad code.

CBI actually produces two ranked lists of related bug predictors for BC, suggesting two distinct bugs. BTRACE produces the same path using either list, suggesting that they correspond to a single bug. BTRACE is correct: the two lists do correspond to a single bug. CBI can be confused by sampling noise, statistical approximation, incompleteness of dynamic data, and other factors. BTRACE is a useful second check, letting us unify equivalent bug lists that CBI has incorrectly held apart.

Section 3.2 mentioned that solving the WPDS may require time exponential in the number of bug predictors. We find that the actual slowdown is gradual and that the absolute performance of BTRACE is good. As expected, the *GPS* phase dominates; creating the initial WPDS and extracting a witness path from the solved system take negligible time. The small CCRYPT application has 13,661 CFG nodes, with about 1,300 on a typical failure path. BTRACE requires 0.10 seconds to find a path using zero CCRYPT predictors, increasing gradually to 0.97 seconds with fifteen predictors. Adding more predictors slows the analysis gradually, amplified only when adding a predictor forces BTRACE to build a longer failure path. BC is larger at 45,234 CFG nodes, and a typical failure path produced by BTRACE is about 3,000 nodes long. The complete analysis takes from two to four seconds with up to four predictors.

Adding dataflow analysis slows the analysis by a factor of between four and twelve, depending on configuration details. Analysis with dataflow and realistic numbers of bug predictors takes about thirteen seconds for BC and less than two seconds for CCRYPT.

5 Related Work

The CodeSurfer Path Inspector tool [18, 19] uses weighted pushdown systems for verification: to see if a program can drive an automaton, summarizing a program property, into a bad state. If this is possible, it uses witnesses to produce a faulty program path. It can also use dataflow analyses by encoding them as weights to rule out infeasible paths. We use WPDSs for optimizing a property instead of verifying it, which has not been previously explored.

Liblit and Aiken directly consider the problem of finding likely failure paths through a program [20]. They present a family of analysis techniques that exploit dynamic information such as failure sites, stack traces, and event logs to construct the set of possible paths that a program might have taken. They could not, however, optimize path length

or the number of events touched when all of them might be unreachable in a single path. Our approach is, therefore, more general. BTRACE incorporates these techniques, along with dataflow analysis, within the unifying framework of weighted pushdown systems. Another difference is that instead of using event logs, we use the output of CBI to guide the path-finding analysis. The theory presented in Section 3 can be extended to incorporate event logs by adding ordering constraints to appropriately restrict the order in which events must be visited by a path.

PSE is another tool for finding failing paths [21]. It requires a user-provided description of how the error could have occurred, e.g., “a pointer was assigned the value `NULL`, and then dereferenced.” This description is in the form of a finite state automaton, and the problem of finding a failing run is reduced to finding a backward path that drives this automaton from its *error* state to its initial state. PSE solves this in the presence of pointer-based data structures and aliasing. Our work does not require any user description of the bug that might have caused the crash, but we do not yet handle pointer-based structures. Like PSE, we can use pointer analysis as a preprocessing step to produce more accurate dataflow weights.

In Definitions 6 and 7, we define semirings that are the power set of the values we want to associate with each path. This approach has been presented in a more general setting by Lengauer and Theune [22]. The power set operation is used to add distributivity to the semiring, and a reduction function, such as our *reduce*, ensures that we never form sets of more elements than necessary.

Our lists of bug predictors are derived using the iterative ranking and elimination algorithm of Liblit et al. [4]. Several other statistical debugging algorithms for CBI-style data have been proposed, including ones based on regularized curve fitting [23], sparse disjunction learning [24], probability density function estimation [25], support vector machines [26], and random forests [26]. BTRACE path reconstruction can use predictors arising from any of these techniques; we require only a list of predictors and numerical scores reflecting their importance. Further study may reveal whether certain statistical debugging algorithms yield more useful BTRACE paths than others.

6 Conclusions

We have presented a static analysis technique to build BTRACE, a tool that can find an optimal path in a program under various constraints imposed by a user. Using bug predictors produced by CBI, BTRACE can perform a postmortem analysis of a program and reconstruct a program path that reveals the circumstances and causes of failure. The paths produced by BTRACE might not be feasible, but we intend for them to help programmers understand the bug predictors produced by CBI and locate bugs more quickly. BTRACE provides user options to supply additional constraints in the form of stack traces and ordering constraints, the latter of which allow the user to guide the tool interactively while locating a bug. Our case studies show that the BTRACE path can isolate the chain of events leading to failure, and, given enough predictors, has the ability to lead the programmer directly to the faulty code. More experiments are required to prove the utility of BTRACE in debugging larger software systems, but initial results look promising.

Acknowledgments

We would like to thank Susan Horwitz for her insightful comments on an earlier draft of this paper.

References

1. Aho, A., Sethi, R., Ullman, J.: *Compilers: Principles, Techniques and Tools*. Addison-Wesley (1985)
2. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: *Sci. of Comp. Prog.* Volume 58. (2005) 206–263
3. Liblit, B.: *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley (2004)
4. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: *SIGPLAN Conf. on Prog. Lang. Design and Impl.* (2005)
5. Lal, A., Reps, T., Balakrishnan, G.: Extended weighted pushdown systems. In: *Computer Aided Verification*. (2005) 434–448
6. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: *Symp. on Princ. of Prog. Lang.* (2003) 62–73
7. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci.* **167** (1996) 131–170
8. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: *Symp. on Princ. of Prog. Lang.* (2004)
9. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. In: *European Symp. on Programming*. (2005)
10. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: *SIGPLAN Conf. on Prog. Lang. Design and Impl.* (2002) 57–68
11. Wegman, M., Zadeck, F.: Constant propagation with conditional branches. In: *Symp. on Princ. of Prog. Lang.* (1985) 291–299
12. Müller-Olm, M., Rüthing, O.: On the complexity of constant propagation. In: *European Symp. on Programming*. (2001) 190–205
13. Lal, A., Lim, J., Polishchuk, M., Liblit, B.: BTRACE: Path optimization for debugging. Technical Report 1535, University of Wisconsin-Madison (2005)
14. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: *SIGPLAN Conf. on Prog. Lang. Design and Impl.* (2003)
15. Kidd, N., Reps, T., Melski, D., Lal, A.: WPDS++: A C++ library for weighted pushdown systems (2005) <<http://www.cs.wisc.edu/wpis/wpds++>>.
16. Somenzi, F.: *Colorado University Decision Diagram package*. Technical report, University of Colorado, Boulder (1998)
17. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: *Proc. of the 16th Int. Conf. on Softw. Eng.*, IEEE Computer Society Press (1994) 191–200
18. GrammaTech, Inc.: *CodeSurfer Path Inspector* (2005) <http://www.grammatech.com/products/codesurfer/overview_pi.html>.
19. Balakrishnan, G., Reps, T., Kidd, N., Lal, A., Lim, J., Melski, D., Gruian, R., Yong, S., Chen, C.H., Teitelbaum, T.: Model checking x86 executables with CodeSurfer/x86 and WPDS++. In: *Computer Aided Verification*. (2005)

20. Liblit, B., Aiken, A.: Building a better backtrace: Techniques for postmortem program analysis. Technical Report CSD-02-1203, University of California, Berkeley (2002)
21. Manevich, R., Sridharan, M., Adams, S., Das, M., Yang, Z.: PSE: explaining program failures via postmortem static analysis. In: Found. Softw. Eng. (2004) 63–72
22. Lengauer, T., Theune, D.: Unstructured path problems and the making of semirings (preliminary version). In: WADS. Volume 519 of Lecture Notes in Computer Science., Springer (1991) 189–200
23. Zheng, A.X., Jordan, M.I., Liblit, B., Aiken, A.: Statistical debugging of sampled programs. In Thrun, S., Saul, L., Schölkopf, B., eds.: Advances in Neural Information Processing Systems 16. MIT Press, Cambridge, MA (2004)
24. Zheng, A.X.: Statistical Software Debugging. PhD thesis, Univ. of California, Berkeley (2005)
25. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: SOBER: Statistical mmodel-based bug localization. In: Found. Softw. Eng., New York, NY, USA, ACM Press (2005) 286–295
26. Jiang, L., Su, Z.: Automatic isolation of cause-effect chains with machine learning. Technical Report CSE-2005-32, University of California, Davis (2005)

A Complexity of Solving the WPDS

In this section, we discuss the worst-case running time complexity of solving the WPDS constructed with the weight domain defined in Definition 6. Each of the methods outlined in Theorems 1 and 2 require solving either *GPS* or *GPP* and then reading the value of $\delta(c)$ for some configuration c . We do not consider the time required for reading the witness value as it can be factored into these two steps. Let $|\Delta|$ be the number of push-down rules (or the size of the CFG), $|\text{Proc}|$ the number of procedures in the program, n_e the entry point of the program, $|B|$ the number of critical nodes, and L the length of a shortest path to the most distant CFG node from n_e . The height (length of the longest descending chain) of the semiring we use is $H = 2^{|B|}L$ and the time required to perform each semiring operation is $T = 2^{|B|}$.

To avoid requiring more WPDS terminology, we specialize the complexity results of solving reachability problems on WPDS [2] to our particular use. $GPS_{\langle p, n_e \rangle}$ can be solved in $O(|\Delta| |\text{Proc}| H T)$ time and $GPP_{\langle p, s \rangle}$ requires $O(|s| |\Delta| H T)$ time. Reading the value of $\delta(\langle p, n_e \rangle)$ is constant time and $\delta(\langle p, s \rangle)$ requires $O(|s| T)$ time. We can now put these results together.

When no stack trace is available, the only option is to use Theorem 1. Obtaining an optimal path in this case requires time $O(|\Delta| |\text{Proc}| 2^{2|B|} L)$. When a stack trace is available, Theorem 2 gives us two options. Suppose we have k stack traces available to us (corresponding to multiple failures caused by the same bug). In the first option, we solve $GPS_{\langle p, n_e \rangle}$, and then ask for the value of $\delta(\langle p, s \rangle)$ for each stack trace available. This has worst-case time complexity $O(|\Delta| |\text{Proc}| 2^{2|B|} L + k |s| 2^{|B|})$ where $|s|$ is the average length of the stack traces. The second option requires a stack trace s , solves $GPP_{\langle p, s \rangle}$ and then asks for the value of $\delta(\langle p, n_e \rangle)$. This has worst-case time complexity $O(k |s| |\Delta| 2^{2|B|} L)$. As is evident from these complexities, the second option should be faster, but its complexity grows faster with an increase in k . Note that these are only worst-case complexities, and comparisons based on them need not hold for the average

case. In fact, in WPDS++ [15], the WPDS implementation that we use, solving *GPS* is usually faster than solving *GPP*.¹

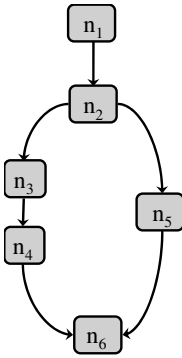


Fig. 3. A simple control flow graph

better to take the shorter right branch at n_2 .) In general, we need to remember a path for each subset of the set of all critical nodes. This is reflected in the design of our weight domain and is what contributes to the exponential complexity with respect to the number of critical nodes.

Let us present some intuition into the complexity results stated above. Consider the CFG shown in Figure 3. If node n_2 is a critical node, then a path from n_1 to n_6 that takes the left branch at n_2 has length 4. The path that takes the right branch has length 3, and touches the same critical nodes as the first path. Therefore, at n_6 , the first path can be discarded and we only need to remember the second path. In this way, branching in the program, which increases the total number of paths through the program, only increases the complexity linearly ($|\Delta|$). Now, if node n_3 is also a critical node, then at n_6 we need to remember both paths: one touches more critical nodes and the other has shorter length. (For a path that comes in at n_1 , and has already touched n_3 , it is

¹ The implementation does not take advantage of the fact that the PDS has been obtained from a CFG. Backward reachability is easier on CFGs as there is at most one known predecessor of a return-site node.

Inference of User-Defined Type Qualifiers and Qualifier Rules

Brian Chin, Shane Markstrum, Todd Millstein, and Jens Palsberg

University of California, Los Angeles
{naerbnc, smarkstr, todd, palsberg}@cs.ucla.edu

Abstract. In previous work, we described a new approach to supporting user-defined *type qualifiers*, which augment existing types to specify and check additional properties of interest. For each qualifier, users define a set of rules that are enforced during static typechecking of programs. Separately, these rules are automatically validated with respect to a user-defined predicate that formalizes the qualifier's intended run-time invariant. We instantiated this approach as a framework for user-defined type qualifiers in C programs, called CLARITY.

In this paper, we extend our earlier approach by resolving two usability issues. First, we show how to perform qualifier inference in the presence of user-defined rules by generating and solving a system of conditional set constraints, thereby relieving users of the burden of explicitly annotating programs. Second, we show how to automatically infer rules that respect a given user-defined invariant, thereby relieving qualifier designers of the burden of manually producing such rules. We have formalized both qualifier and rule inference and proven their correctness. We have also extended CLARITY to support qualifier and rule inference, and we illustrate their utility in practice through experiments with several type qualifiers and open-source C programs.

1 Introduction

Type systems are a natural and powerful discipline for specifying and statically checking properties of programs. However, language designers cannot anticipate all of the properties that programmers will wish to specify, nor can they anticipate all of the practical ways in which such properties can be statically checked. Therefore, it is desirable to allow programmers to refine existing types in order to specify and check additional program properties. A practical form of refinement can be achieved through user-defined *type qualifiers* [6, 7].

In previous work [2], we described a new approach to user-defined type qualifiers that is more expressive and provides stronger guarantees than prior approaches. Users provide a set of *qualifier rules* in a stylized language. These rules declaratively define a qualifier's associated programming discipline and are automatically enforced during static typechecking of programs. Users may also provide a predicate that formalizes a qualifier's intended run-time invariant. This invariant is used to automatically validate the correctness of the provided qualifier rules. We instantiated this approach as a framework for user-defined type qualifiers in C programs, called CLARITY, and illustrated its utility for a variety of qualifiers, including *pos* and *neg* for integers, *nonnull* for pointers, and *tainted* and *untainted* for format strings.

```

qualifier nonzero(int Expr E)
  case E of
    decl int Const C:
      C, where C != 0
  | decl int Expr E1:
      E1, where pos(E1)
  | decl int Expr E1, E2:
      E1 * E2,
      where nonzero(E1) && nonzero(E2)
  restrict
    decl int Expr E1, E2:
      E1 / E2, where nonzero(E2)
  invariant value(E) != 0

```

Fig. 1. A user-defined type qualifiers for nonzero integers.

In this paper, we extend our earlier approach by resolving two usability issues. First, we show how to perform qualifier inference, relieving programmers of the burden of explicitly annotating their programs. We describe an inference algorithm that is parameterized by a set of user-defined qualifier rules. The algorithm generates and solves a system of *conditional set constraints*. We have formalized constraint generation and proven that the constraint system is equivalent to the associated qualifier type system. We have also extended CLARITY to support qualifier inference and have used it to infer qualifiers on several open-source C programs.

Second, we show how to automatically infer rules that respect a given user-defined invariant, relieving qualifier designers of the burden of manually producing such rules. We define a partial order of candidate rules that formalizes the situation when one rule subsumes another. We then describe an algorithm for walking this partial order to generate all valid rules that are not subsumed by any other valid rule. We have implemented rule inference in CLARITY and used it to automatically generate all of our manually produced qualifier rules as well as some valid rules we had not thought of.

The next section reviews our approach to user-defined type qualifiers in the context of CLARITY. Sections 3 and 4 describe qualifier inference and rule inference, respectively. Section 5 discusses our experiments with qualifier and rule inference in CLARITY. Section 6 compares with related work, and section 7 concludes.

2 An Overview of CLARITY

2.1 Qualifier Rules and Qualifier Checking

Figure 1 illustrates the definition of a simple user-defined qualifier for nonzero integers in CLARITY.¹ The first line defines the qualifier `nonzero` to be applicable to all expressions of type `int`. The `case` and `restrict` blocks provide the user-defined typing

¹ This paper focuses on CLARITY’s “value” qualifiers; its “reference” qualifiers are not considered [2].

discipline associated with the new qualifier. Each clause in the case block represents a qualifier rule, consisting of some *metavariable* declarations for use in the rest of the rule, a *pattern*, and a *condition*. The first case clause in Figure 1 indicates that an integer constant can be given the qualifier `nonzero` if the constant is not equal to zero. The second clause indicates that an expression can be given the qualifier `nonzero` if it can be given another user-defined qualifier `pos`, whose definition is not shown. The third clause indicates that the product of two `nonzero` expressions can also be considered `nonzero`.

A `restrict` clause has the same syntax as a case clause, but the semantics is different. Namely, a `restrict` clause indicates that whenever the pattern is matched by some program expression, then the condition should also be satisfied by that expression. Therefore, the `restrict` clause in Figure 1 indicates that all denominator expressions in a division must have the qualifier `nonzero`.

CLARITY includes a *qualifier checker* that statically enforces user-defined qualifier rules on programs. As a simple example, consider the statement

```
nonzero int prod = a*b;
```

where `a` and `b` are each declared to have the type `pos int`. Since `prod` is declared to have the qualifier `nonzero`, the qualifier checker must ensure that `a*b` can be given this qualifier. By the third case clause for `nonzero` in Figure 1, the check succeeds if it can be shown that `a` and `b` each recursively has qualifier `nonzero`. Each of these recursive checks succeeds by the second case clause for `nonzero`, since `a` and `b` are each declared to have qualifier `pos`.

In general, each program expression can be associated with a set of qualifiers. Qualifier checking employs a natural notion of subtyping in the presence of user-defined qualifiers: a type $Q_1\tau$ subtypes another type $Q_2\tau$, where Q_1 and Q_2 are sets of qualifiers and τ is an unqualified type, if $Q_1 \supseteq Q_2$. We have formalized this notion of subtyping and proven that it is sound for all user-defined qualifiers expressible in our rule language [2]. There is no direct notion of subtyping between qualifiers, but this can be encoded in the rules. For example, the second case clause for `nonzero` in Figure 1 has the effect of making `pos int` a subtype of `nonzero int`.

2.2 Qualifier Invariants and Qualifier Validation

In addition to the qualifier rules, CLARITY allows users to provide a predicate that formalizes a qualifier's intended run-time invariant. For example, the invariant clause for `nonzero` in Figure 1 indicates that the value of an expression qualified with `nonzero` should not equal zero, in all run-time execution states. The invariant makes use of a value predicate that our framework provides. Given a qualifier's invariant, CLARITY's *qualifier validator* component ensures that the qualifier's rules are correct, in the sense that they respect this invariant. Qualifier validation happens once, independent of any particular program that uses the qualifier. For each case clause, the qualifier validator generates one proof obligation to be discharged.² Our implementation discharges obligations with the Simplify automatic theorem prover [5].

² We do not validate `restrict` rules, whose correctness depends on a user-specific notion of run-time error.

Each proof obligation requires that a rule's pattern and condition are sufficient to ensure the qualifier's invariant at run time. For example, the qualifier validator generates the following obligation for the first case clause for `nonzero` in figure 1: if an expression E is an integer constant other than zero, then the value of E in an arbitrary execution state is not equal to zero. For the third case clause, the qualifier validator generates the following obligation: if an expression E has the form $E_1 * E_2$ and both E_1 and E_2 satisfy `nonzero`'s invariant in an arbitrary execution state, then E also satisfies `nonzero`'s invariant in that state. These obligations are easily discharged by Simplify.³ On the other hand, if the pattern in the third case clause were erroneously specified as $E_1 + E_2$, the qualifier validator would catch the error, since it is not possible to prove that the sum of two nonzero integers is always nonzero.

CLARITY's qualifier validator is currently limited by the capabilities of Simplify, which includes decision procedures for propositional logic, linear arithmetic, and equality with uninterpreted functions, and additionally includes heuristics for handling first-order quantification. Simplify works well for many kinds of properties, for example arithmetic invariants and simple invariants about pointers such as nonnullness. Simplify is not tailored for reasoning about other useful kinds of invariants, for example shape invariants on data structures. However, our approach to qualifier validation could easily be adapted for use with other decision procedures and theorem provers, including tools requiring some user interaction.

3 Qualifier Inference

The original CLARITY system supports qualifier *checking*: all variables must be explicitly annotated with their qualifiers. In this section, we show how to support qualifier *inference* in the presence of user-defined qualifier rules. We formalize qualifier inference for a simply-typed lambda calculus with references and user-defined qualifiers, as defined by the following grammar:

$$\begin{aligned} e &::= c \mid e_1 + e_2 \mid x \mid \lambda x : \tau. e \mid e_1 \ e_2 \mid \text{ref } e \mid e_1 := e_2 \mid !e \mid \text{assert}(e, q) \\ \tau &::= \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \text{ref } \tau \end{aligned}$$

Let Q be the set $\{q_1, \dots, q_n\}$ of user-defined qualifiers in a program. Sets of qualifiers from Q form a natural lattice, with partial order \supseteq , least-upper-bound function \cap , and greatest-lower-bound function \cup . We denote elements of this lattice by metavariable l ; qualified types are ranged over by metavariable ρ and are defined as follows:

$$\rho ::= l \ \phi \quad \phi ::= \text{int} \mid \rho_1 \rightarrow \rho_2 \mid \text{ref } \rho$$

We present both a type system and a constraint system for qualifier inference and prove their equivalence, and we describe an algorithm for solving the generated constraints. We assume the bound variables in expressions are annotated with unqualified types τ . It is possible to combine qualifier inference with type inference, but separating them simplifies the presentation.

³ Our qualifier validator currently does not properly model overflow.

3.1 Formal Qualifier Rules

We formalize the case rules as defining two kinds of relations. First, some case clauses have the effect of declaring a specificity relation between qualifiers. We formalize these rules as defining axioms for a relation of the form $q_1 \triangleright q_2$. For example, the second case clause in Figure 1 would be represented by the axiom $\text{pos} \triangleright \text{nonzero}$. We use \triangleright^* to denote the reflexive, transitive closure of the user-defined \triangleright relation, and we require \triangleright^* to be a partial order.

The other kind of case clause uses a pattern to match on a constructor (e.g., $+$), and the clause determines the qualifier of the entire expression based on the qualifiers of the immediate subexpressions. We formalize these rules as defining relations of the form R_p^q , where q is a qualifier and p represents one of the constructors in our formal language, ranging over integer constants and the symbols $+$, λ , and ref . The arity of each relation R_p^q is the number of immediate subexpressions of the constructor represented by p , and the domain of each argument to the relation is Q . Each case clause is formalized through axioms for these relations. For example, the third case clause in Figure 1 would be represented by the axiom $R_*^{\text{nonzero}}(\text{nonzero}, \text{nonzero})$ (if our formal language contained multiplication). The first case clause in that figure would be formalized through the (conceptually infinite) set of axioms $R_1^{\text{nonzero}}()$, $R_2^{\text{nonzero}}()$, etc. For simplicity of presentation, we assume that each subexpression is required to satisfy only a single qualifier. In fact, our implementation allows each subexpression to be constrained to satisfy a set of qualifiers, and it would be straightforward to update our formalism to support this ability.

Finally, we formalize the `restrict` rules with an expression of the form $\text{assert}(e, q)$, which requires the type system to ensure that the top-level qualifier on expression e 's type includes qualifier q . For example, the `restrict` rule in Figure 1 is modeled by replacing each denominator expression e in a program with $\text{assert}(e, \text{nonzero})$. The `assert` expression can also be used to model explicit qualifier annotations in programs.

3.2 The Type System

We assume we are given an expression e along with a set A of axioms representing the user-defined qualifier rules, as described above. The qualifier type system is presented in Figure 3, and the axioms in A are implicitly considered to augment this formal system. As usual, metavariable Γ ranges over type environments, which map variables to qualified types. The rule for $\text{assert}(e, q)$ infers a qualified type for e and then checks that q is in the top-level qualifier of this type. The *strip* function used in the rule for lamdas removes all qualifiers from a qualified type ρ , producing an unqualified type τ .

The main novelty in the type system is the consultation of the axioms in A to produce the top-level qualifiers for constructor expressions. For example, consider the first rule in Figure 3, which infers the qualifiers for an integer constant c using a set comprehension notation. The resulting set I includes all qualifiers q' such that the $R_c^{q'}()$ relation holds (according to the axioms in A), as well as all qualifiers q that are “less specific” than such a q' as defined by the \triangleright^* relation. In this way, the rule finds all possible qualifiers that can be proven to hold given the user-defined case clauses. The subsumption

$$\frac{l_1 \supseteq l_2}{l_1 \text{int} \leq l_2 \text{int}} \quad \frac{l_1 \supseteq l_2 \quad \rho \leq \rho' \quad \rho' \leq \rho}{l_1 \text{ref } \rho \leq l_2 \text{ref } \rho'} \quad \frac{l_1 \supseteq l_2 \quad \rho_2 \leq \rho_1 \quad \rho'_1 \leq \rho'_2}{l_1(\rho_1 \rightarrow \rho'_1) \leq l_2(\rho_2 \rightarrow \rho'_2)}$$

Fig. 2. Formal subtyping rules for qualified types

$$\begin{array}{c} \frac{l = \{q \mid R_c^{q'}() \wedge q' \triangleright^* q\}}{\Gamma \vdash c : l \text{int}} \quad \frac{\Gamma \vdash e_1 : l_1 \text{int} \quad \Gamma \vdash e_2 : l_2 \text{int} \quad l = \{q \mid R_+^{q'}(q_1, q_2) \wedge q_1 \in l_1 \wedge q_2 \in l_2 \wedge q' \triangleright^* q\}}{\Gamma \vdash e_1 + e_2 : l \text{int}} \quad \frac{\Gamma(x) = \rho}{\Gamma \vdash x : \rho} \\[10pt] \frac{\text{strip}(\rho_1) = \tau_1 \quad \Gamma, x : \rho_1 \vdash e : \rho_2 \quad \rho_2 = l_2 \phi_2 \quad l = \{q \mid R_\lambda^{q'}(q_2) \wedge q_2 \in l_2 \wedge q' \triangleright^* q\}}{\Gamma \vdash \lambda x : \tau_1. e : l(\rho_1 \rightarrow \rho_2)} \quad \frac{\Gamma \vdash e_1 : l(\rho_2 \rightarrow \rho) \quad \Gamma \vdash e_2 : \rho_2}{\Gamma \vdash e_1 e_2 : \rho} \\[10pt] \frac{\Gamma \vdash e : \rho \quad \rho = l_0 \phi_0 \quad l = \{q \mid R_{\text{ref}}^{q'}(q_0) \wedge q_0 \in l_0 \wedge q' \triangleright^* q\}}{\Gamma \vdash \text{ref } e : l \text{ref } \rho} \quad \frac{\Gamma \vdash e_1 : l \text{ref } \rho \quad \Gamma \vdash e_2 : \rho}{\Gamma \vdash e_1 := e_2 : \rho} \\[10pt] \frac{\Gamma \vdash e : l \text{ref } \rho}{\Gamma \vdash !e : \rho} \quad \frac{\Gamma \vdash e : \rho \quad \rho = l \phi \quad q \in l}{\Gamma \vdash \text{assert}(e, q) : \rho} \quad \frac{\Gamma \vdash e : \rho' \quad \rho' \leq \rho}{\Gamma \vdash e : \rho} \end{array}$$

Fig. 3. Formal qualifier inference rules

rule at the end of the figure can then be used to forget some of these qualifiers, via the subtyping rules in Figure 2. The inference of top-level qualifiers is similar for the other constructors, except that consultation of the R relation makes use of the top-level qualifiers inferred for the immediate subexpressions.

3.3 The Constraint System

In this section we describe a constraint-based algorithm for qualifier inference. The key novelty is the use of a specialized form of *conditional constraints* to represent the effects of user-defined qualifier rules. The metavariable α represents *qualifier variables*, and we generate constraints of the following forms:

$$\alpha \supseteq \alpha \quad q \in \alpha \quad q \in \alpha \Rightarrow \bigvee (\bigwedge q \in \alpha)$$

Given a set C of constraints, let S be a mapping from the qualifier variables in C to sets of qualifiers. We say that S is a *solution* to C if S satisfies all constraints in C . We say that S is the *least solution* to C if for all solutions S' and qualifier variables α in the domain of S and S' , $S(\alpha) \supseteq S'(\alpha)$. It is easy to show that if a set of constraints C in the above form has a solution, then it has a unique least solution.

Constraint Generation. We formalize constraint generation by a judgment of the form $\kappa \vdash e : \delta \mid C$. Here C is a set of constraints in the above form, and the metavariable δ represents qualified types whose qualifiers are all qualifier variables:

$$\delta ::= \alpha \ \phi \quad \phi ::= \text{int} \mid \delta_1 \rightarrow \delta_2 \mid \text{ref } \delta$$

$$\begin{aligned}
\alpha_1 \text{int} \sqsubseteq \alpha_2 \text{int} &\equiv \{\alpha_1 \supseteq \alpha_2\} \\
\alpha_1 \text{ref } \delta_1 \sqsubseteq \alpha_2 \text{ref } \delta_2 &\equiv \{\alpha_1 \supseteq \alpha_2\} \cup \delta_1 \sqsubseteq \delta_2 \cup \delta_2 \sqsubseteq \delta_1 \\
\alpha_1(\delta_1 \rightarrow \delta'_1) \sqsubseteq \alpha_2(\delta_2 \rightarrow \delta'_2) &\equiv \{\alpha_1 \supseteq \alpha_2\} \cup \delta_2 \sqsubseteq \delta_1 \cup \delta'_1 \sqsubseteq \delta'_2
\end{aligned}$$

Fig. 4. Converting type constraints into set constraints

$$\begin{array}{c}
\frac{\alpha' \text{ fresh} \quad \delta' = \alpha' \text{int} \quad \delta = \text{refresh}(\delta')}{\kappa \vdash c : \delta \mid \delta' \sqsubseteq \delta \cup \{C_c^q(\alpha') \mid q \in Q\}} \quad \frac{\kappa \vdash e_1 : \alpha_1 \text{int} \mid C_1 \quad \kappa \vdash e_2 : \alpha_2 \text{int} \mid C_2}{\kappa \vdash e_1 + e_2 : \delta \mid C_1 \cup C_2 \cup \delta' \sqsubseteq \delta \cup \{C_+^q(\alpha_1, \alpha_2, \alpha') \mid q \in Q\}} \\
\frac{\kappa(x) = \delta' \quad \delta = \text{refresh}(\delta')}{\kappa \vdash x : \delta \mid \delta' \sqsubseteq \delta} \quad \frac{\kappa, x : \delta_1 \vdash e : \delta_2 \mid C \quad \delta_1 = \text{embed}(\tau_1) \quad \delta_2 = \alpha_2 \varphi_2}{\kappa \vdash \lambda x : \tau_1. e : \delta \mid C \cup \delta' \sqsubseteq \delta \cup \{C_\lambda^q(\alpha_2, \alpha') \mid q \in Q\}} \\
\frac{\kappa \vdash e_1 : \alpha(\delta_2 \rightarrow \delta') \mid C_1 \quad \kappa \vdash e_2 : \delta'_2 \mid C_2 \quad \delta = \text{refresh}(\delta')}{\kappa \vdash e_1 e_2 : \delta \mid C_1 \cup C_2 \cup \delta'_2 \sqsubseteq \delta_2 \cup \delta' \sqsubseteq \delta} \quad \frac{\kappa \vdash e : \delta_0 \mid C \quad \delta_0 = \alpha_0 \varphi_0}{\kappa \vdash \text{ref } e : \delta \mid C \cup \delta' \sqsubseteq \delta \cup \{C_{\text{ref}}^q(\alpha_0, \alpha') \mid q \in Q\}} \\
\frac{\kappa \vdash e_1 : \alpha \text{ref } \delta' \mid C_1 \quad \kappa \vdash e_2 : \delta'' \mid C_2 \quad \delta = \text{refresh}(\delta')}{\kappa \vdash e_1 := e_2 : \delta \mid C_1 \cup C_2 \cup \delta'' \sqsubseteq \delta' \cup \delta' \sqsubseteq \delta} \quad \frac{\kappa \vdash e : \alpha \text{ref } \delta' \mid C \quad \delta = \text{refresh}(\delta')}{\kappa \vdash !e : \delta \mid C \cup \delta' \sqsubseteq \delta} \quad \frac{\kappa \vdash e : \delta' \mid C \quad \delta' = \alpha \phi \quad \delta = \text{refresh}(\delta')}{\kappa \vdash \text{assert}(e, q) : \delta \mid C \cup \{q \in \alpha\} \cup \delta' \sqsubseteq \delta}
\end{array}$$

Fig. 5. Formal constraint generation rules for qualifier inference

The metavariable κ denotes type environments that map program variables to qualified types of the form δ .

The inference rules defining this judgment are shown in Figure 5. The *embed* function adds fresh qualifier variables to an unqualified type τ in order to turn it into a qualified type δ , and *refresh*(δ) is defined as *embed*(*strip*(δ)). To keep the constraint generation purely syntax-directed, subsumption is “built in” to each rule: the *refresh* function is used to create a fresh qualified type δ , which is constrained by a subtype constraint of the form $\delta' \sqsubseteq \delta$. Subtype constraints are also generated for applications and assignments, as usual. We treat a subtype constraint as a shorthand for a set of qualifier-variable constraints, as shown in Figure 4.

Each rule for an expression with top-level constructor p produces one conditional constraint per qualifier q in Q , denoted C_p^q . Informally, the constraint C_p^q *inverts* the user-defined qualifier rules, indicating all the possible ways to prove that an expression with constructor p can be given qualifier q according to the axioms in A . For example, both the second and third case clauses in Figure 1 can be used to prove that a product $a*b$ has the qualifier *nonzero*, so our implementation of constraint generation in CLARITY produces the following conditional constraint:

$$\text{nonzero} \in \alpha_{a*b} \Rightarrow ((\text{nonzero} \in \alpha_a \wedge \text{nonzero} \in \alpha_b) \vee (\text{pos} \in \alpha_{a*b}))$$

More formally, let $\text{zip}(R_p^q(q_1, \dots, q_m), \alpha_1, \dots, \alpha_m)$ denote the constraint $q_1 \in \alpha_1 \wedge \dots \wedge q_m \in \alpha_m$. Let $\{a_1, \dots, a_u\}$ be all the axioms in A for the relation R_p^q , and let $\{q_1, \dots, q_v\} = \{q' \in Q \mid q' \triangleright q\}$. Then $C_p^q(\alpha_1, \dots, \alpha_m, \alpha')$ is the following conditional constraint:

$$q \in \alpha' \Rightarrow \left(\bigvee_{1 \leq i \leq u} \text{zip}(a_i, \alpha_1, \dots, \alpha_m) \vee \bigvee_{1 \leq i \leq v} q_i \in \alpha' \right)$$

We have proven the equivalence of our constraint system with the type system presented in the previous subsection; details are in our companion technical report [3].

Theorem: $\emptyset \vdash e : \rho$ if and only if $\emptyset \vdash e : \delta \mid C$ and there exists a solution S to C such that $S(\delta) = \rho$.

Constraint Solving. We solve the constraints by a graph-based propagation algorithm, which either determines that the constraints are unsatisfiable or produces the unique least solution. Figure 6 shows a portion of the constraint graph generated for the statement `int prod = a*b`. On the left side, the graph includes one node for each qualifier variable, which is labeled with the corresponding program expression. Each node contains a bit string of length $|Q|$ (not shown in the figure), representing the qualifiers that may be given to the associated expression. All bits are initialized to true, indicating that all expressions may be given all qualifiers. If bit i for node α ever becomes false during constraint solving, this indicates that α cannot include the i th qualifier in any solution.

Because our algorithm propagates the *inability* for an expression to have a qualifier, the direction of flow is opposite what one might expect. For each generated constraint of the form $\alpha_1 \supseteq \alpha_2$, the graph includes an edge from α_1 to α_2 . For each conditional constraint, the graph contains a representation of its *contrapositive*. For example, the right side of Figure 6 shows an *and-or* tree that represents the following constraint:

$$((\text{nonzero} \notin \alpha_a \vee \text{nonzero} \notin \alpha_b) \wedge (\text{pos} \notin \alpha_{a*b})) \Rightarrow \text{nonzero} \notin \alpha_{a*b}$$

The tree's root has an outgoing edge to the `nonzero` bit of the node `a*b`, and the leaves similarly have incoming `nonzero`-bit edges. In the figure, edges to and from individual

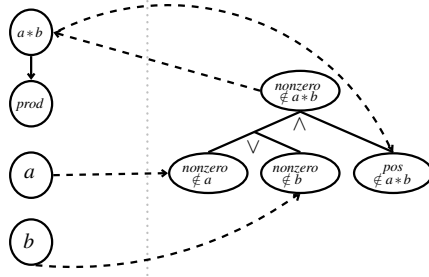


Fig. 6. An example constraint graph

bits are dotted. The root of each and-or tree maintains a counter of the number of subtrees it is waiting for before it can “fire.” Our example tree has a counter value of 2.

To solve the constraints, we visit the root of each and-or tree once. If its counter is greater than 0, we do nothing. Otherwise, the outgoing edge from its root is traversed, which falsifies the associated bit and propagates this falsehood to its successors recursively until quiescence. For example, if the and-or tree in Figure 6 ever fires, that will falsify the nonzero bit of $a*b$, which in turn will falsify the nonzero bit of $prod$.

After the propagation phase is complete, we employ the constraints of the form $q \in \alpha$ to check for satisfiability. For each such constraint, if the bit corresponding to qualifier q in node α is false, then we have a contradiction and the constraints are unsatisfiable. Otherwise, the least solution is formed by mapping each qualifier variable α to the set of all qualifiers whose associated bit in node α is true.

Complexity Analysis. Let n be the size of a program, m be the size of the axioms in A , and q be the number of user-defined qualifiers. There are $O(n)$ qualifier variables, $O(n^2)$ constraints of the form $\alpha \supseteq \alpha$, $O(qn)$ constraints of the form $q \in \alpha$, and $O(qn)$ conditional constraints generated, each with size $O(m)$. Therefore, the constraint graph has $O(n^2)$ edges between qualifier-variable nodes, each of which can be propagated across q times. There are $O(qnm)$ edges in total for the and-or trees, and there are $O(qnm)$ edges between the qualifier-variable nodes and the and-or trees, each of which can be propagated across once. Therefore, the total number of propagations, and hence the total time complexity, is $O(qn(n+m))$.

4 Rule Inference

Writing qualifier rules can be tedious and error prone. The qualifier validator that is part of our framework reduces errors by checking that each user-defined rule respects its associated qualifier’s invariant. However, other errors are possible. For example, a user-defined rule may be correct but be overly specific, and there may be useful rules that are completely omitted. To reduce the burden on qualifier designers and to reduce these kinds of errors, we have created a technique for automatically *inferring* correct case rules from a qualifier’s invariant.

A naive approach to rule inference is to generate each candidate rule and use the qualifier validator to remove all candidates that do not respect the intended invariant. However, since qualifier validation is relatively expensive, requiring usage of decision procedures, and since there are an exponential number of candidates in the number of qualifiers, it is desirable to minimize the number of candidates that need to be explicitly considered.⁴ To efficiently search the space of candidate rules, we define a partial order \preceq that formalizes the situation when one candidate subsumes another.

The most precise partial ordering on case clauses is logical implication. For example, the third case clause in Figure 1 corresponds to the following formula, obtained by replacing qualifiers with their invariants:

⁴ Conceptually, there are an infinite number of candidates, due to constants. We handle constants through a simple heuristic that works well in practice. For each qualifier, we only consider a single candidate rule (possibly) containing constants, which is derived from the qualifier’s invariant by replacing all references to $value(E)$ with a metavariable ranging over constants.

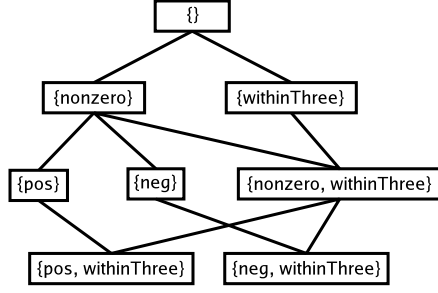


Fig. 7. An example of \preceq_S for four qualifiers

$$\text{value}(E1) \neq 0 \wedge \text{value}(E2) \neq 0 \Rightarrow \text{value}(E1 * E2) \neq 0$$

The above clause subsumes a clause that requires both $E1$ and $E2$ to be `pos` instead of `nonzero`, since the above formula logically implies the formula associated with the new clause. Unfortunately, precisely computing this partial order requires an exponential number of calls to decision procedures to reason about logical implication, which is exactly what we are trying to avoid.

Instead, our approach is to use logical implication to define a partial ordering on individual qualifiers, but to then lift this partial ordering to case clauses in a purely syntactic way. Therefore, we need only make a quadratic number of calls to the decision procedures in order to compute the partial order. This approximation of the “true” partial ordering is still guaranteed to completely exhaust the space of candidates, but it is now possible to produce qualifier rules that are redundant. As we show in Section 5, however, our approach works well in practice. The rest of this section formalizes our partial order and describes the rule inference algorithm; more details are in our companion technical report [3].

The Partial Order. We assume that every qualifier $q \in Q$ has an invariant, which is a unary predicate that we also denote q . We also assume that no two qualifiers have logically equivalent invariants. Then we define a partial order \preceq_Q on qualifiers as follows:

$$q_1 \preceq_Q q_2 \triangleq \forall x. q_1(x) \Rightarrow q_2(x)$$

This partial order is computed by $|Q|^2$ queries to Simplify. We similarly use $|Q|^2$ Simplify queries to compute mutual exclusivity of pairs of qualifiers:

$$q_1 \perp_Q q_2 \triangleq \forall x. \neg(q_1(x) \wedge q_2(x))$$

Let $S = \mathcal{P}(Q)$. We lift \preceq_Q to sets of qualifiers (or *qualsets*) s_1 and s_2 in S as follows:

$$s_1 \preceq_S s_2 \triangleq \forall q_2 \in s_2. \exists q_1 \in s_1. q_1 \preceq_Q q_2$$

When considering qualsets for use in a candidate case clause, we restrict our attention to qualsets that are *canonical*. We define a set $s \in S$ as canonical if the following condition holds:

$$\forall q_1, q_2 \in s. \neg(q_1 \perp_Q q_2) \wedge (q_1 \preceq_Q q_2 \Rightarrow q_1 = q_2)$$

It is easy to prove that \preceq_S is a partial order on canonical qualsets. An example of the \preceq_S partial order over canonical qualsets that may include any of the four qualifiers nonzero, pos, neg, and withinThree (whose invariant requires the value to be ≥ -3 and ≤ 3) is shown in Figure 7. We lift \preceq_S to tuples of canonical qualsets in the obvious way:

$$(s_1, \dots, s_k) \preceq_T (t_1, \dots, t_k) \triangleq \forall i \in \{1, \dots, k\}. s_i \preceq_S t_i$$

Finally, we can describe the partial order on candidate case clauses. A candidate c can be considered to be a triple containing the constructor p used as the pattern; a tuple of qualsets (s_1, \dots, s_k) , one per subexpression of p , representing the clause's condition; and the qualifier q that the clause is defined for. We define the partial ordering on case clauses c_1 and c_2 as follows:

$$(p_1, (s_1, \dots, s_k), q_1) \preceq (p_2, (t_1, \dots, t_j), q_2) \triangleq \\ p_1 = p_2 \wedge k = j \wedge (t_1, \dots, t_k) \preceq_T (s_1, \dots, s_k) \wedge q_1 \preceq_Q q_2$$

We have proven that if $c_1 \preceq c_2$ then in fact c_1 logically implies c_2 [3].

The Algorithm. Consider generating all valid case rules for a single qualifier q . Further, fix a particular constructor p to use in the rule's pattern, and assume that this constructor has exactly one subexpression. Let W be a worklist of pairs of the form (s, l) where s is a qualset and l is a list of qualifiers. Initialize the set W to $\{(\emptyset, [q_1, q_2, q_3, \dots])\}$, where $[q_1, q_2, q_3, \dots]$ is an ordered list of all the qualifiers in reverse topological order according to \preceq_Q . Using reverse topological order ensures we will generate qualsets for use in a case rule from most-general to most-specific, which is necessary given the contravariance in the definition of \preceq . We similarly maintain W in sorted order according to a reverse topological sort of the first component of each pair. We also maintain a set T of valid case rules, initialized to \emptyset .

1. If W is empty, we are done and T contains all the valid non-redundant rules. Otherwise, remove the first pair (s, l) in W .
2. If there is some candidate $(p', s', q') \in T$ such that $(p', s', q') \preceq (p, s, q)$ then s is redundant, so we drop the pair (s, l) and return to the previous step. Otherwise, we continue to the next step.
3. We run our framework's qualifier validator on (p, s, q) . If it passes, we add (p, s, q) to T . If not, then we need to check less-specific candidates. For each $q \in l$, we add the pair $(s \cup \{q\}, l')$ to W , where l' is the suffix of l after q . These pairs are placed appropriately in W to maintain its sortedness, as described earlier.

In the case when the constructor p has $k > 1$ subexpressions, we need to enumerate k -ary multisets. To do so, the worklist W now contains k -tuples of pairs of the form (s, l) . When adding new elements to W , we apply the procedure described in Step 3 above to each component of the k -tuple individually, keeping all other components unchanged. The only subtlety is that we want to avoid generating redundant tuples. For example, if $q_1 \preceq_Q q_2$, then the tuple $(\{q_2\}, \{q_2\})$ could be a successor of both $(\{q_1\}, \{q_2\})$ and

Table 1. Qualifier inference results

qualifier sets			nonnull			nonnull/pos/neg/nz		
program	kloc	vars	cons	gen (s)	solv (s)	cons	gen (s)	solv (s)
identd-1.0	0.19	624	1381	0.09	0.01	2757	0.15	0.01
mingetty-0.9.4	0.21	488	646	0.04	0.01	1204	0.06	0.01
bftpd-1.0.11	2.15	1773	3768	0.39	0.05	6426	0.58	0.08
bc-1.04	4.75	4769	14913	1.21	0.13	27837	5.78	0.18
grep-2.5	10.43	4914	15719	0.75	0.55	28343	7.84	0.71
snort-2.06	52.11	29013	99957	36.39	46.81	176852	290.24	58.07

$(\{q_2\}, \{q_1\})$. To avoid this duplication, we only augment a component of a k -tuple when generating new candidates for W in Step 3 if it is either the component that was last augmented along this path of the search, or it is to the right of that component. This rule ensures that once a component is augmented, the search cannot “go back” and modify components to its left. In our example, $(\{q_2\}, \{q_2\})$ would not be generated from $(\{q_1\}, \{q_2\})$ in Step 3, because the last component to have been augmented must have been the second one (since all components begin with the empty set).

Finally we describe the full algorithm for candidate generation. We enumerate each qualifier q in topological order according to \preceq_Q . For each such qualifier, we enumerate each constructor p in any order and use the procedure described above to generate all the valid non-redundant rules of the form $(p, (s_1, \dots, s_k), q)$. The set T is initialized to \emptyset at the beginning of this algorithm and is augmented throughout the entire process. In this way, candidates shown to be valid for some qualifier q can be used to find a later candidate for a target q' to be redundant. For example, a rule allowing the sum of two `pos` expressions to be considered `pos` will be found to subsume a rule allowing the sum of two `pos` expressions to be considered `nonzero`. When this algorithm completes, the set T will contain all valid rules such that none is subsumed by any other valid rule according to \preceq . Finally, we augment T with rules that reflect the specificity relation among qualifiers, such as the second case rule in Figure 1. These rules are derived directly from the computed \preceq_Q relation.

5 Experiments

5.1 Qualifier Inference

We implemented qualifier inference in CLARITY and ran it on six open-source C programs, ranging from a few hundred to over 50,000 lines of code, as shown in Table 1. Each test case was run through the inferencer twice. The first time, the inferencer was given a definition only for a version of `nonnull`, with a `case` clause indicating that an expression of the form `&E` can be considered `nonnull` and a `restrict` clause requiring dereferences to be to `nonnull` expressions. The second time, the inferencer was additionally given versions of the qualifiers `pos`, `neg`, and `nonzero` for integers, each with 5 case rules similar to those in Figure 1. For each run, the table records the number

of constraints produced as well as the time in seconds for constraint generation and constraint solving.

Several pointer dereferences fail to satisfy the `restrict` clause for `nonnull`, causing qualifier inference to signal inconsistencies. We analyzed each of the signaled errors for `bc` and inserted casts to `nonnull` where appropriate to allow inference to succeed. In total, we found no real errors and inserted 107 casts. Of these, 98 were necessary due to a lack of flow-sensitivity in our type system. We plan to explore the incorporation of targeted forms of flow-sensitivity to handle commonly arising situations. Despite this limitation, the qualifier rules were often powerful enough to deduce interesting invariants. For example, on `bc`, 37% (163/446) of the integer lvalues were able to be given the `nonzero` qualifier and 5% (24/446) the `pos` qualifier. For `snort`, 8% (561/7103) of its integer lvalues were able to be given the `nonzero` qualifier, and 7% (317/4571) of its pointer lvalues were able to be given the `nonnull` qualifier (without casts).

5.2 Rule Inference

We implemented rule inference in the context of CLARITY and performed two experiments. First, we inferred rules for `pos`, `neg`, and `nonzero`, given only their invariants. In the second experiment, we additionally inferred rules for `withinThree`. For the first experiment, our rule inference algorithm automatically generated all of the case rules we had originally hand-written for the three qualifiers. In addition, rule inference generated several valid rules that we had not written. For example, one new rule allows the negation of a `nonzero` expression to also be `nonzero`. The second experiment produced no new rules for `nonzero`, `pos`, and `neg`, indicating their orthogonality to the `withinThree` qualifier. However, it did generate several nontrivial rules for `withinThree` that we had not foreseen. For example, one rule allows a sum to be considered `withinThree` if one operand is `withinThree` and `pos` while the other operand is `withinThree` and `neg`. In both experiments, no redundant rules were generated.

The first experiment required 18 queries to the decision procedures in order to compute the \preceq_Q and \perp_Q relations, for use in the overall \preceq partial order, and 142 queries to validate candidate rules. In contrast, the naive generate-and-test algorithm would require 600 queries. The second experiment required 32 queries to compute \preceq_Q and \perp_Q as well as 715 queries for candidate validation, while the naive algorithm would require 3136 queries. The first experiment completed in under six minutes, and the second experiment in under 26 minutes. The running times are quite reasonable, considering that rule inference need only be performed once for a given set of qualifiers, independent of the number of programs that employ these qualifiers.

6 Related Work

Our framework is most closely related to the CQUAL system, which also allows users to define new type qualifiers for C programs [6]. The main novelty in our approach is the incorporation of user-defined qualifier rules, which are not supported in CQUAL. Our qualifier inference algorithm extends the technique used for inference in CQUAL [6] to handle such user-defined rules via a form of conditional constraints. Our notion of

rule inference has no analogue in CQUAL. CQUAL includes a form of qualifier polymorphism, and follow-on work extended CQUAL's type system to be flow sensitive [7], while CLARITY currently lacks both of these features.

Work on *refinement types* [8] allows programmers to create subtypes of ML datatype definitions. Intersection types allow a function to have multiple type signatures with varying refinements, playing a role analogous to our case rules. A refinement inference algorithm is provided for a functional subset of ML. Later work [4] considered the interaction of intersection types with computational effects, and recent work extends these ideas to a flow-sensitive setting [10]. These two systems are more powerful than our type qualifiers, but they do not support full type inference.

HM(X) [11] is a Hindley-Milner-style type inference system that is parameterized by the form of constraints. Our situation is dual to that one: while HM(X) has a fixed type system that is parameterized by a constraint system, qualifier inference in our framework uses a fixed form of constraints but is parameterized by the qualifier rules.

Rule inference is related to work on *predicate abstraction* [9, 1] and on finding the *best transformer* [12, 13]. These algorithms use decision procedures to precisely abstract a program with respect to a set of predicates. Rule inference is similar, as it produces an abstraction automatically from the user-defined invariants. However, this abstraction is produced *once*, independent of any particular program.

7 Conclusions

We have described two forms of inference that reduce the burden on users of our approach to user-defined type qualifiers. *Qualifier inference* employs user-defined rules to infer qualifiers on programs, obviating the need for manual program annotations. We described an algorithm for qualifier inference based on generating and solving a system of conditional set constraints. *Rule inference* employs decision procedures to automatically produce qualifier rules that respect a qualifier's user-defined invariant, reducing the burden on qualifier designers. We described a partial order on candidate qualifier rules that allows us to search the space of candidates efficiently without losing completeness. We have implemented both qualifier and rule inference in the CLARITY system for C, and our experimental results illustrate their utility in practice.

Acknowledgments

This research was supported in part by NSF ITR award #0427202 and by a generous gift from Microsoft Research. Thanks to Craig Chambers, Vass Litvinov, Scott Smith, and Frank Tip for discussions that led to a simplification of the presentation of the constraint system. Thanks to Rupak Majumdar for useful feedback on the paper.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213. ACM Press, 2001.

2. B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–95, New York, NY, USA, 2005. ACM Press.
3. B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. Technical Report CSD-TR-050041, UCLA Computer Science Department, October 2005.
4. R. Davies and F. Pfenning. Intersection types and computational effects. In *ICFP '00: Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 198–208. ACM Press, 2000.
5. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.
6. J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.
7. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12. ACM Press, 2002.
8. T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 268–277, New York, NY, USA, 1991. ACM Press.
9. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
10. Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 213–225. ACM Press, 2003.
11. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.
12. T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 252–266, 2004.
13. G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 530–545, 2004.

Assertion Checking over Combined Abstraction of Linear Arithmetic and Uninterpreted Functions

Sumit Gulwani¹ and Ashish Tiwari²

¹ Microsoft Research, Redmond, WA 98052
sumitg@microsoft.com

² SRI International, Menlo Park, CA 94025
tiwari@csl.sri.com

Abstract. This paper presents results on the problem of checking equality assertions in programs whose expressions have been abstracted using combination of linear arithmetic and uninterpreted functions, and whose conditionals are treated as non-deterministic.

We first show that the problem of assertion checking for this combined abstraction is coNP-hard, even for loop-free programs. This result is quite surprising since assertion checking for the individual abstractions of linear arithmetic and uninterpreted functions can be performed efficiently in polynomial time.

Next, we give an assertion checking algorithm for this combined abstraction, thereby proving decidability of this problem despite the underlying lattice having infinite height. Our algorithm is based on an important connection between unification theory and program analysis. Specifically, we show that weakest preconditions can be strengthened by replacing equalities by their unifiers, without losing any precision, during backward analysis of programs.

1 Introduction

We use the term *equality assertion* or simply *assertion* to refer to an equality between two program expressions. By *assertion checking*, we mean checking whether a given assertion is an invariant at a given program point.

Reasoning about assertions in programs is an undecidable problem. Hence, assertion checking is typically performed over some (sound) abstraction of the program. This may give rise to false positives, i.e., some assertions that are true in the original program may not be true in the abstract version. There is an efficiency-precision trade-off in the choice of the abstraction. A more precise abstraction leads to fewer false positives but is also harder to reason about.

Linear arithmetic and uninterpreted functions¹ are two most commonly used expression languages for creating program abstractions. There are several

¹ An uninterpreted function F of arity n satisfies only one axiom: If $e_i = e'_i$ for $1 \leq i \leq n$, then $F(e_1, \dots, e_n) = F(e'_1, \dots, e'_n)$. Uninterpreted functions are commonly used to abstract programming language operators that are otherwise hard to reason about. They are also used to abstract procedure calls.

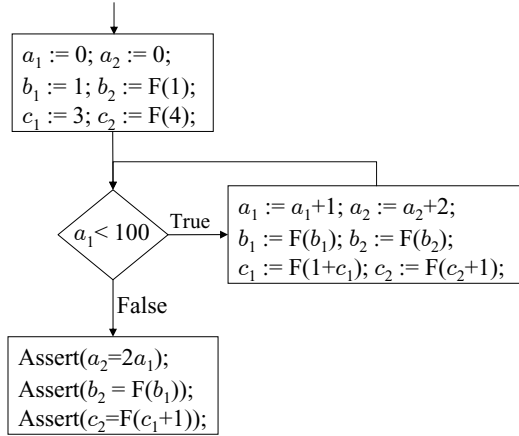


Fig. 1. This program illustrates the difference between precision of performing analysis over the abstractions of linear arithmetic (which can verify only the first assertion), uninterpreted functions (which can verify only the second assertion), and their combination (which can verify all assertions). F denotes some function without any side-effects and can be modeled as an uninterpreted function for purpose of proving the assertions.

papers that describe how to do assertion checking for each of these abstractions. (Section 6 on related work describes some of this work.) The combined expression language of linear arithmetic and uninterpreted functions yields a more precise abstraction than the ones obtained from either of these two expression languages. For example, consider the program shown in Figure 1. Note that all assertions at the end of the program are true. If this program is analyzed over the abstraction of linear arithmetic (using, for example, the abstract interpreter described in [14] or [6]), then only the first assertion can be validated. This is because discovering the relationship between b_1 and b_2 , and between c_1 and c_2 , involves reasoning about uninterpreted functions. Similarly, if this program is analyzed over the abstraction of uninterpreted functions (using, for example, the abstract interpreter described in [10]), then only the second assertion can be validated. However, an analysis over the combined abstraction of linear arithmetic and uninterpreted functions can verify all assertions.

Even though there has been a lot of work for reasoning about the abstractions of linear arithmetic and that of uninterpreted functions, the problem of assertion checking over the combined abstraction of linear arithmetic and uninterpreted functions has not been considered before. In this paper, we consider the problem of checking equality assertions in programs whose expressions have been abstracted using linear arithmetic and uninterpreted functions. We also abstract all program conditionals as non-deterministic because otherwise the problem is easily shown to be undecidable even for the individual abstractions of linear arithmetic [17] and uninterpreted functions [16]. (An analysis that performs an imprecise reasoning over the combined abstraction of linear arithmetic and uninterpreted functions but takes conditional guards into account would

also be useful in practice, and can be used, for example, for array bounds checking. The related work section mentions our recent work on combining abstract interpreters, which can be used to construct such an analysis.) The abstracted program model is formally described in Section 2.

In Section 3, we show that the problem of assertion checking in the combined abstraction of linear arithmetic and uninterpreted functions is coNP-hard. This is true even for loop-free programs, in which case it is coNP-complete. This result is quite surprising because assertion checking in the individual abstractions of linear arithmetic and uninterpreted functions entails polynomial-time algorithms (even for programs with loops). Karr's algorithm [14, 17] can be used to perform assertion checking when program expressions have been abstracted using linear arithmetic operators. Gulwani and Necula's algorithm [9, 10] performs assertion checking in programs whose expressions have been abstracted using uninterpreted functions. Both these algorithms run in polynomial time. However, our coNP-hardness result shows that there is no way to combine these algorithms to do assertion checking for the combined abstraction in polynomial time (unless $P=coNP$). A similar combination problem has been studied extensively in the context of decision procedures. Nelson and Oppen have given a famous combination result for combining decision procedures for disjoint, convex and quantifier-free theories with only polynomial-time overhead [20]. The theories of linear arithmetic and uninterpreted functions are disjoint, convex, and quantifier-free and have polynomial time decision procedures. Hence, the Nelson-Oppen combination methodology can be used to construct a polynomial-time decision procedure for the combination of these theories. In this paper, we show that, unfortunately, there is no polynomial-time combination scheme for assertion checking in the combined abstraction of linear arithmetic and uninterpreted functions (unless $P=coNP$).

In Section 4, we give an assertion checking algorithm for the combined abstraction (of linear arithmetic and uninterpreted functions) thereby showing that this problem is decidable. This result is again surprising because the underlying abstract lattice has infinite height, which implies that a standard abstract interpretation [6] based algorithm cannot terminate in a finite number of steps. However, our algorithm leverages the fact that our goal is not to discover all equality invariants, but to check whether a given assertion is an invariant. A central component of our algorithm is a *general* result that allows replacement of equalities generated in weakest precondition computation by their unifiers (Lemma 2). For theories that admit a singleton or finite complete set of unifiers, respectively called unitary and finitary theories, this replacement can be effectively done. The significance of this connection between assertion checking and unification is discussed further in Section 5. We make the paper self-contained by presenting (in Section 4.1) a novel unification algorithm for the combined theory of linear arithmetic and uninterpreted functions, which is used in our assertion checking algorithm.

2 Program Model

We assume that each procedure in a program is abstracted using the flowchart nodes shown in Figure 2. In the assignment node, x refers to a program variable while e denotes some expression in the underlying abstraction. We refer to the language of such expressions as *expression language of the program*. The expression languages for the abstractions of linear arithmetic, uninterpreted functions and their combination are as follows:

- Linear arithmetic:

$$e ::= y \mid c \mid e_1 \pm e_2 \mid c \times e$$

Here y denotes some variable while c denotes some arithmetic constant.

- Uninterpreted functions:

$$e ::= y \mid F^n(e_1, \dots, e_n)$$

Here F^n denotes some uninterpreted function of arity n . We allow n to be zero (for representing nullary uninterpreted functions).

- Combination of linear arithmetic and uninterpreted functions:

$$e ::= y \mid c \mid e_1 \pm e_2 \mid c \times e \mid F^n(e_1, \dots, e_n)$$

A non-deterministic assignment $x := ?$ denotes that the variable x can be assigned any value. Such non-deterministic assignments are used as a safe abstraction of statements (in the original source program) that our abstraction cannot handle precisely.

Non-deterministic conditionals, represented by $*$, denote that the control can flow to either branch irrespective of the program state before the conditional. They are used as a safe abstraction of guarded conditionals, which our abstraction cannot handle precisely. We abstract away the guards in conditionals because otherwise the problem of assertion checking (when the expression language of the program involves combination of linear arithmetic and uninterpreted functions) can be easily shown undecidable from either of the following two results. Müller-Olm and Seidl have shown that the problem of assertion

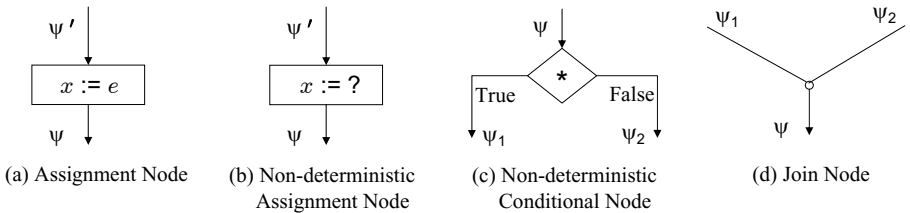


Fig. 2. Flowchart nodes in our abstracted program model

checking in programs that use guarded conditionals and linear arithmetic expressions is undecidable [17]. Müller-Olm, Rüthing, and Seidl have also proved a similar undecidability result when the expression language involves uninterpreted functions [16].

A join node has two incoming edges. Note that a join node with more than two incoming edges can be reduced to multiple join nodes each with two incoming edges.

3 coNP-Hardness of Assertion Checking

In this section, we show that the problem of assertion checking when the expression language of the program involves combination of linear arithmetic and uninterpreted functions (and the flowchart representation of the program consists of nodes shown in Figure 2) is coNP-hard.

The key observation in proving this result is that a disjunctive assertion of the form $g = a \vee g = b$ can be encoded as the non-disjunctive assertion $F(a) + F(b) = F(g) + F(a + b - g)$. The procedure **Check**(g,m) generalizes this encoding for the disjunctive assertion $g = 0 \vee \dots \vee g = m - 1$ (which has $m - 1$ disjuncts), as stated in Lemma 1. Once such a disjunction can be encoded, we can reduce the unsatisfiability problem to the problem of assertion checking as follows.

Consider the program shown in Figure 3. We will show that the assert statement in the program is true iff the input boolean formula ψ is unsatisfiable. Note that, for a fixed ψ , the procedures **IsUnsatisfiable** and **Check** can be reduced to one procedure whose flowchart representation consists of only the nodes shown in Figure 2. (These procedures use procedure calls and loops with guarded conditionals only for expository purposes.) This can be done by unrolling the loops and inlining procedure **Check** inside procedure **IsUnsatisfiable**. The size of the resulting procedure is polynomial in the size of the input boolean formula ψ .

The procedure **IsUnsatisfiable** contains k non-deterministic conditionals, which together choose a truth value assignment for the k boolean variables in the input boolean formula ψ , and accordingly set its clauses to true (1) or false (0). The boolean formula ψ is unsatisfiable iff at least one of its clauses remains unsatisfied in every truth value assignment to its variables, or equivalently, $g \in \{0, \dots, m-1\}$ in all executions of the procedure **IsUnsatisfiable**. The procedure **Check**(g,m) performs the desired check as stated in the following lemma.

Lemma 1. *The assert statement in **Check**(g,m) is true iff $g \in \{0, \dots, m - 1\}$.*

Proof. The following properties hold for all $0 \leq i \leq m - 1$.

- E1. If $0 \leq j \leq i$, then $h_{i,j} = h_{i,0}$.
- E2. If $g \in \{0, \dots, m - 1\}$, then $h_i = h_{i,g}$.
- E3. If $g \notin \{0, \dots, m - 1\}$, then h_i cannot be expressed as a linear combination of $h_{i,0}, \dots, h_{i,m-1}$.

```

IsUnsatisfiable( $\psi$ )
  % Suppose formula  $\psi$  has  $k$  variables  $x_1, \dots, x_k$ 
  %           and  $m$  clauses numbered 1 to  $m$ .
  % Let variable  $x_i$  occur in positive form in clauses #  $A_i[0], \dots, A_i[c_i]$ 
  %           and in negative form in clauses #  $B_i[0], \dots, B_i[d_i]$ .
  for  $i = 1$  to  $m$  do
     $e_i := 0$ ; %  $e_i$  represents whether clause  $i$  is satisfiable or not.
  for  $i = 1$  to  $k$  do
    if (*) then % set  $x_i$  to true
      for  $j = 0$  to  $c_i$  do
         $e_{A_i[j]} := 1$ ;
      else % set  $x_i$  to false
        for  $j = 0$  to  $d_i$  do
           $e_{B_i[j]} := 1$ ;
   $g := e_1 + e_2 + \dots + e_m$ ; % Count how many clauses have been satisfied.
  Check( $g, m$ );

Check( $g, m$ )
  % This procedure checks whether  $g \in \{0, \dots, m-1\}$ .
   $h_0 := F(g)$ ;
  for  $j = 0$  to  $m-1$  do
     $h_{0,j} := F(j)$ ;
  for  $i = 1$  to  $m-1$  do
     $s_{i-1} := h_{i-1,0} + h_{i-1,i}$ ;
     $h_i := F(h_{i-1}) + F(s_{i-1} - h_{i-1})$ ;
    for  $j = 0$  to  $m-1$  do
       $h_{i,j} := F(h_{i-1,j}) + F(s_{i-1} - h_{i-1,j})$ ;
  Assert( $h_{m-1} = h_{m-1,0}$ );

```

Fig. 3. A program that illustrates the coNP-hardness of assertion checking when the expression language uses combination of linear arithmetic and uninterpreted functions.

The above properties can be proved easily by induction on i . If $g \in \{0, \dots, m-1\}$, then the assert statement is true because:

$$\begin{aligned}
 h_{m-1} &= h_{m-1,g} \text{ (follows from property E2)} \\
 &= h_{m-1,0} \text{ (follows from property E1)}
 \end{aligned}$$

If $g \notin \{0, \dots, m-1\}$, then it follows from property E3 that the assert statement is falsified. ■

Lemma 1 implies that the assert statement in procedure `IsUnsatisfiable(ψ)` is true iff the input boolean formula ψ is unsatisfiable. Hence, the following theorem holds.

Theorem 1. *Assertion checking for programs with non-deterministic conditionals and whose expression language is a combination of linear arithmetic and uninterpreted functions is coNP-hard.*

Since `IsUnsatisfiable` can be represented as a loop-free program, Theorem 1 holds even for loop-free programs.

4 Algorithm for Assertion Checking

In this section, we give an assertion checking algorithm for our abstracted program model when the expression language of the program involves combination of linear arithmetic and uninterpreted functions. We prove that this algorithm terminates, which establishes the decidability of assertion checking for the combined abstraction. It remains an open problem to establish an upper complexity bound for this algorithm.

For purpose of describing and proving correctness of our algorithm, we first establish some results on unification in the combined theory of linear arithmetic and uninterpreted functions in the next sub-section.

4.1 Unification in the Combined Theory

A *substitution* σ is a mapping that maps variables to expressions such that for every variable x , the expression $\sigma(x)$ contains variables only from the set $\{y \mid \sigma(y) = y\}$. A substitution mapping σ can be (homomorphically) lifted to expressions such that for every expression e , we define $\sigma(e)$ to be the expression obtained from e by replacing every variable x by its mapping $\sigma(x)$. Often, we denote the application of a substitution σ to an expression e using postfix notation as $e\sigma$. We sometimes treat a substitution mapping σ as the following formula, which is a conjunction of non-trivial equalities between variables and their mappings:

$$\bigwedge_{x: x \neq x\sigma} x = x\sigma$$

A substitution σ is a *unifier* for an equality $e_1 = e_2$ (in theory T) if $e_1\sigma = e_2\sigma$ (in theory T). A substitution σ is a unifier for a set of equalities E if σ is a unifier for each equality in E . A substitution σ_1 is *more-general* than a substitution σ_2 if there exists a substitution σ such that $x\sigma_2 = (x\sigma_1)\sigma$ for all variables x .² A set C of unifiers for E is *complete* when for any unifier σ for E , there exists a unifier $\sigma' \in C$ that is more-general than σ . Theories can be classified based on whether all equalities in that theory have a complete set of unifiers whose cardinality is at most 1 (unitary theory), or finite (finitary theory), or whether some equality does not have any finite complete set of unifiers (infinitary theory).

In the remaining part of this section, we show that the combined theory of linear arithmetic and uninterpreted functions is finitary. For this purpose, we describe an algorithm that computes a complete set of unifiers for an equality in the combined theory. We describe this algorithm using a set of inference rules (listed in table 1) in the style of [4].

² The more-general relation is reflexive, i.e., a substitution is more-general than itself.

Table 1. Inference rules for unification in the combination theory

Unif0:	$\frac{(E \cup \{e = e\}, \sigma)}{(E, \sigma)}$
Unif1:	$\frac{(E \cup \{x = e\}, \sigma)}{(E\sigma', \sigma\sigma')}$
if x does not occur in e . Here $\sigma' = \{x \mapsto e\}$ and $E\sigma'$ denotes $\{e_1\sigma' = e_2\sigma' \mid (e_1 = e_2) \in E\}$.	
Unif2:	$\frac{(E \cup \{F(e_1, \dots, e_n) = F(e'_1, \dots, e'_n) + e\}, \sigma)}{(E \cup \{e_1 = e'_1, \dots, e_n = e'_n, e = 0\}, \sigma)}$

Table 1 describes some inference rules that operate on states. A state (E, σ) is a pair consisting of a set E of equalities (between expressions involving combination of linear arithmetic and uninterpreted functions) and a substitution σ . The Unif0 rule removes trivial equalities from E . The Unif1 rule can be applied after selecting some equality from E that can be rewritten in the form $x = e$ such that variable x does not occur in expression e . The Unif2 rule is applied after selecting some equality that can be rewritten in the form $F(e_1, \dots, e_n) = F(e'_1, \dots, e'_n) + e$ for some uninterpreted function F and expressions e_i, e'_i and e .

The notation $\{x_1 \mapsto e_1, \dots, x_k \mapsto e_k\}$ denotes the substitution mapping that maps variable x_i to e_i (for $1 \leq i \leq k$) and all other variables to themselves. We use the notation $(E, \sigma) \vdash (E', \sigma')$ to denote that the state (E', σ') is obtained from (E, σ) by applying some inference rule. Similarly, $(E, \sigma) \vdash^* (E', \sigma')$ denotes that the state (E', σ') can be obtained from the state (E, σ) by applying some sequence of inference rules.

To generate a complete set of unifiers C for an equality $e_1 = e_2$, we start with the state $(\{e_1 = e_2\}, I)$, where I is the identity mapping, and apply the inference rules repeatedly until no more inference rules can be applied. For all derivations that end with some state of the form (\emptyset, σ) , we put σ in C . Theorem 2 stated below implies that the set C thus obtained is indeed a set of unifiers for the equality $e_1 = e_2$. Theorem 3 implies that this set C of unifiers is complete. The proofs of these theorems are by induction on the length of the derivation and are given in the full version of this paper [13].

Theorem 2 (Soundness). *If $(E, I) \vdash^* (\emptyset, \sigma)$, then σ is a unifier for E .*

Theorem 3 (Completeness). *Suppose σ is a unifier for E . Then there is a derivation $(E, I) \vdash^* (\emptyset, \sigma_0)$ such that σ_0 is a more-general unifier for E than σ .*

The following theorem implies that the set C is finite.

Theorem 4 (Finite Complete Set of Unifiers). *Every derivation $(E, I) \vdash^* (E', \sigma')$ takes a finite number of steps. Consequently, E has a finite complete set of unifiers.*

The proof of Theorem 4 is given in the full version of this paper [13]. The key proof idea is to show that every derivation $(E, I) \vdash^* (E', \sigma')$ takes a finite

number of steps and then use Konig's lemma to bound the total number of derivations.

We next illustrate the application of the inference system.

Example 1. Consider the following derivation of a unifier for the equality $Fx + Fy = Fa + Fb$.

$$\begin{array}{ll}
 (\{Fx + Fy = Fa + Fb\}, I) & \\
 (\{Fx = Fb, y = a\}, I) & \text{Unif2} \\
 (\{x = b, y = a\}, I) & \text{Unif2} \\
 (x = b, \{y \mapsto a\}) & \text{Unif1} \\
 (\emptyset, \{x \mapsto b, y \mapsto a\}) & \text{Unif1}
 \end{array}$$

Thus $\{x \mapsto b, y \mapsto a\}$ is a unifier for $Fx + Fy = Fa + Fb$. Note that the alternate choice for the first Unif2 application yields another unifier $\{x \mapsto a, y \mapsto b\}$ for the given equality. No other unifier can be generated by applying the inference rules. Hence, these two unifiers constitute a complete set of unifiers for the given equality.

Example 2. As another example, consider generating a complete set of unifiers for the equality $x + Fx + Fy = a + Fa + F(a + 1)$. Since each variable occurs below an uninterpreted symbol, only the Unif2 rule is applicable. There are four choices, either $x = a$, or $x = a + 1$, or $y = a$, or $y = a + 1$. We show a derivation for the second choice below.

$$\begin{array}{ll}
 (\{x + Fx + Fy = a + Fa + F(a + 1)\}, I) & \\
 (\{x + Fy = a + Fa, x = a + 1\}, I) & \text{Unif2} \\
 (\{a + Fa - Fy = a + 1\}, \{x \mapsto a + Fa - Fy\}) & \text{Unif1} \\
 (\{a = a + 1, a = y\}, \{x \mapsto a + Fa - Fy\}) & \text{Unif2} \\
 (\{0 = 1\}, \{x \mapsto a + Fa - Fy, y \mapsto a\}) & \text{Unif1}
 \end{array}$$

The above derivation is now stuck with no inference rule being applicable. Note that only the first choice $x = a$ and the fourth choice $y = a + 1$ successfully generate a unifier, which in both cases is $\{x \mapsto a, y \mapsto a + 1\}$. This unifier yields a singleton complete set of unifiers for the given equality.

4.2 Algorithm

Our algorithm for assertion checking over the combined abstraction is based on weakest precondition computation. It represents invariants at each program point by a formula that is a disjunction of substitution mappings. We show that any program invariant in our abstracted program model can be represented using such formulas (Lemma 2).

Suppose the goal is to check whether an assertion $e_1 = e_2$ is an invariant at program point π . The algorithm performs a backward analysis of the program computing a formula ψ (which is a disjunction of substitution mappings) at each program point such that ψ must hold for the assertion $e_1 = e_2$ to be true at program point π . This formula is computed at each program point from the

formulas at the successor program points in an iterative manner. The algorithm uses the transfer functions described below to compute these formulas across the flowchart nodes shown in Figure 2. The algorithm declares $e_1 = e_2$ to be an invariant at π if the formula computed at the beginning of the program after fixed-point computation is a tautology in the combined theory of linear arithmetic and uninterpreted functions.

In the following transfer functions, we use the notation $\mathbf{Unif}(E)$, where E is some conjunction of equalities E , to denote the formula that is a disjunction of all unifiers in some complete set of unifiers for E . (If E is unsatisfiable, then E does not have any unifier and $\mathbf{Unif}(E)$ is simply *false*.) The formula $\mathbf{Unif}(E)$ can be computed by using the algorithm described in Section 4.1.

Initialization: The formula at all program points except π is initialized to be the trivial formula *true*. The formula at program point π is initialized to be $\mathbf{Unif}(e_1 = e_2)$.

Assignment Node: See Figure 2 (a). The formula ψ' before an assignment node $x := e$ is obtained from the formula ψ after the assignment node by substituting x by e in ψ , and invoking \mathbf{Unif} on each resulting disjunct.

$$\psi' = \bigvee_i \mathbf{Unif}(\psi^i[e/x]), \text{ where } \psi = \bigvee_i \psi^i$$

Non-deterministic Assignment Node: See Figure 2 (b). The formula ψ' before a non-deterministic assignment node $x := ?$ is obtained from the formula ψ after the non-deterministic assignment node by substituting program variable x by some fresh constant (i.e., a fresh nullary uninterpreted function symbol) α , and invoking \mathbf{Unif} on each resulting disjunct.

$$\psi' = \bigvee_i \mathbf{Unif}(\psi^i[\alpha/x]), \text{ where } \psi = \bigvee_i \psi^i$$

Non-deterministic Conditional Node: See Figure 2 (c). The formula ψ before a non-deterministic conditional node is obtained by taking the conjunction of the formulas ψ_1 and ψ_2 on the two branches of the conditional, and invoking \mathbf{Unif} on each resulting disjunct.

$$\psi = \bigvee_{i,j} \mathbf{Unif}(\psi_1^i \wedge \psi_2^j), \text{ where } \psi_1 = \bigvee_i \psi_1^i \text{ and } \psi_2 = \bigvee_j \psi_2^j$$

Join Node: See Figure 2 (d). The formulas ψ_1 and ψ_2 on the two predecessors of a join node are same as the formula ψ after the join node.

$$\psi_1 = \psi \text{ and } \psi_2 = \psi$$

Fixed-Point Computation: In presence of loops in procedures, the algorithm goes around each loop until the formulas computed at each program point in two successive iterations of a loop are equivalent, or if any formula becomes *false*.

Correctness. We now prove that the above algorithm is correct, i.e., an assertion $e_1 = e_2$ holds at program point π iff the algorithm claims so. For this purpose, we first state a useful lemma (Lemma 2) that states an interesting connection between program analysis and unification theory. This lemma is true in general: it is independent of the logical theory and also holds for programs with guarded conditionals. The proof of this lemma is given in the full version of this paper [13].

Lemma 2. *An equality $e_1 = e_2$ holds at a program point π iff $\text{Unif}(e_1 = e_2)$ holds at π . In fact, a formula ϕ containing $e_1 = e_2$ holds at a program point π iff $\phi[\text{Unif}(e_1 = e_2)/(e_1 = e_2)]$ holds at π .*

Lemma 2 implies that the formula computed by our algorithm before the flowchart is the (real) weakest precondition of the formula after those nodes. Also, note that the algorithm starts with a formula which is an invariant at π iff the given assertion is an invariant at π (follows from Lemma 2). The correctness of the algorithm now follows from the fact that the algorithm starts with the correct assertion at π and iteratively computes the correct weakest precondition at each program point in a backward analysis.

Termination. We now prove that the above algorithm terminates in a finite number of steps. It suffices to show that the weakest precondition computation across a loop terminates in a finite number of iterations. This follows from the following lemma.

Lemma 3. *Let C be a chain ψ_1, ψ_2, \dots of formulas that are disjunctions of substitutions. Let $\psi_i = \bigvee_{\ell=1}^{m_i} \psi_i^\ell$ for some integer m_i and substitutions ψ_i^ℓ . Suppose*

- (a) $\psi_{i+1} = \bigvee_{\ell=1}^{m_i} \bigvee_{j=1}^{n_i} \text{Unif}(\psi_i^\ell \wedge \alpha_i^j)$, for some substitutions α_i^j .
- (b) $\psi_i \not\Rightarrow \psi_{i+1}$.

Then, C is finite.

The proof of Lemma 3 is by establishing a well founded ordering on ψ_i' s, and is given in the full version of this paper [13]. Lemma 3 implies termination of our assertion checking algorithm. (Note that the weakest preconditions ψ_1, ψ_2, \dots generated by our algorithm at any given program point inside a loop in successive iterations satisfy condition (a), and hence $\psi_{i+1} \Rightarrow \psi_i$ for all i . Lemma 3 implies that there exists j such that $\psi_j \Rightarrow \psi_{j+1}$ and hence $\psi_j \equiv \psi_{j+1}$, at which point the fixed-point computation across that loop terminates.) Hence, the following theorem holds.

Theorem 5. *Assertion checking for programs with non-deterministic conditionals and whose expression language is a combination of linear arithmetic and uninterpreted functions is decidable.*

The decidability of assertion checking for the combined abstraction is rather surprising given that the abstract lattice over sets of equalities between expressions

in the combined theory has an infinite height. This suggests that an abstract interpretation based forward analysis algorithm that operates over this lattice may not terminate across loops (unless widening techniques are employed, which may lead to imprecise analysis). For example, consider the following program.

```
InfiniteHeightExample()
  x := 0;
  while (*) do { x := x + 1 };
  Assert( $x = 0 \vee \dots \vee x = m$ );
```

The disjunctive assertion at the end of the program can be encoded using an equality assertion. The procedure `Check(x,m)` (on page 284) does exactly this. Clearly, the assertion at the end of the program is not true. To invalidate this assertion, the abstract interpreter will have to go around the loop m times. Hence, it will not terminate across loops (because if it did terminate in say t steps, then it will not be able to invalidate the assertion $x = 0 \vee \dots \vee x = t$). Our algorithm terminates because it performs a backward analysis (which is good enough for assertion checking) instead of performing a forward analysis (which is required for discovering all valid equalities).

5 Assertion Checking and Unification

The results in this paper point out an interesting connection between assertion checking in programs over a given abstraction and the unification problem for the theory defining that abstraction. Lemma 2 implies that we can replace an assertion by a formula representing a complete set of unifiers for that assertion. This result is quite general and holds for programs with even guarded conditionals and any expression language. This allows for strengthening of weakest preconditions computed using standard transfer functions, by applying `Unif()` to the result *without* losing any precision. This observation is the basis for the close connection between assertion checking and unification.

The theories of linear arithmetic and uninterpreted functions are unitary. However, equalities in the combined theory of linear arithmetic and uninterpreted functions may not have a complete set of unifiers with a cardinality of at most 1. This disparity appears to be responsible for the coNP-hardness of assertion checking for the combined abstraction of linear arithmetic and uninterpreted functions (as opposed to the fact that the abstractions of linear arithmetic and uninterpreted functions have polynomial-time assertion checking algorithms [14, 10]). The presence of multiple unifiers in a minimal complete set allows for encoding of disjunctions in the combined abstraction. For example, the assertion $F(x) + F(3 - x) = F(1) + F(2)$ has two unifiers $x = 1$ and $x = 2$ in its minimal complete set of unifiers. This assertion will be true at any program point iff $x = 1$ or $x = 2$ on all paths leading to this assertion.

The decidability of assertion checking for the combined abstraction (of linear arithmetic and uninterpreted functions) can be attributed to fact that the combined theory is finitary. Observe that the weakest precondition computation of

an assertion, as described in Section 4.2, terminates across a loop because there are only finitely many ways that the assertion can be true.

6 Related Work

We are not aware of any work related to assertion checking for the combined abstraction of linear arithmetic and uninterpreted functions. However, there has been a lot of work on assertion checking and invariant generation over individual abstractions of linear arithmetic and uninterpreted functions.

Program Analysis over Abstraction of Linear Arithmetic. Karr described an algorithm to reason about programs using the abstraction of linear equalities. This algorithm performs a forward analysis of the program and computes a set of linear equalities at each program point [14, 17] in an iterative manner. Gulwani and Necula gave a randomized algorithm that performs an equally precise reasoning but more efficiently [8]. Cousot gave a more precise algorithm that reasons about programs using the abstraction of linear inequalities wherein the facts computed at each program point are linear inequality relationships between program variables [7]. Müller-Olm and Seidl have described a modular linear arithmetic analysis to reason about finite-bit machine arithmetic [19]. There has also been some work on extending some of these analyses to an interprocedural setting [18, 11].

Program Analysis over Abstraction of Uninterpreted Functions. Kildall's algorithm [15] performs abstract interpretation over the lattice of sets of Herbrand equivalences (i.e., equivalences between expressions involving uninterpreted functions) but it runs in exponential time. Alpern, Wegman, and Zadeck (AWZ) gave a polynomial-time algorithm that reasons about programs treating all operators as uninterpreted functions [1]. The AWZ algorithm is less precise than Kildall's algorithm, but is quite popularly used for global value numbering in compilers. Rüthing, Knoop and Steffen's (RKS) polynomial-time algorithm also reasons about programs using the abstraction of uninterpreted functions. The RKS algorithm is more precise than the AWZ algorithm but remains less precise than Kildall's algorithm. Recently, Gulwani and Necula gave a polynomial-time algorithm that is as precise as Kildall's algorithm with respect to assertion checking in programs using the abstraction of uninterpreted functions [9, 10].

Combination of Abstract Interpreters. We have recently described a general methodology to combine abstract interpreters for two abstractions to construct an abstract interpreter for the combination of those abstractions [12]. This methodology can be used to construct an efficient polynomial-time algorithm that performs analysis over the combined abstraction of linear arithmetic and uninterpreted functions and also takes conditional guards into account. However, this algorithm does not perform the most precise reasoning over the combined abstraction of linear arithmetic and uninterpreted functions. Note that the algorithm that we have described in this paper performs the most precise reasoning over the combined abstraction of linear arithmetic and uninterpreted functions, but it does not take conditional guards into account.

Unification for Combination of Theories. The unification problem for the combined theory of linear arithmetic and uninterpreted functions is a simple variant of the unification problem for abelian groups with additional uninterpreted functions. This latter problem is usually referred to as the *general unification problem* for abelian groups [3]. The first algorithm for generating unifiers for the general unification problem for abelian groups was obtained as a corollary of the general result for *combining* unification algorithms [21] and was later refined [2]. The generic combination unification algorithm involves solving the so-called “unification with constants” and “constant elimination” problems [21], or “unification with linear constant restriction” [2] problem for the individual theories. In this paper, we have presented a different unification algorithm for the combined theory of linear arithmetic and uninterpreted functions. Our presentation of this unification algorithm is using inference rules, which are simple to understand and implement.

Decision Procedures for Combination of Theories. Nelson and Oppen gave a general methodology for combining decision procedures for disjoint, convex and quantifier-free theories with only polynomial-time overhead [20]. Shostak gave an efficient variant of this algorithm for the specific case of solvable theories. Clark, Dill and Levitt have described a decision procedure, based on Shostak’s method, for combination of linear arithmetic and uninterpreted functions in presence of boolean connectives [5]. It must be mentioned that the problem of assertion checking in programs over a certain abstraction (and in particular for combination of two abstractions) is harder than developing a decision procedure for that abstraction. This is because even though a decision procedure can be used to verify an assertion along a particular program path, a program can potentially have an infinite number of paths. However, if a program is annotated with appropriate invariants at all join points, then a decision procedure can be easily used to verify those invariants as well as assertions across straight-line program fragments.

7 Conclusion

In this paper, we show that assertion checking in programs whose expressions have been abstracted using linear arithmetic and uninterpreted functions is coNP-hard (even for loop-free programs). We also give an algorithm for assertion checking for this abstraction, thereby proving decidability of this problem. These results are obtained by closely analyzing the expressiveness of a theory and its effect on the assertion checking problem. First, the ability to encode disjunctions is identified to be an important factor in making assertion checking hard. Second, the classification of a theory as unitary, finitary, or infinitary—based on whether it admits a singleton, finite, or infinite complete set of unifiers has bearing on the hardness and tractability of the assertion checking problem. We show that assertions can be replaced by their unifiers for purpose of checking if they are invariant. We believe that these observations will be significant when other similar or more general abstractions are considered for program analysis.

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th Annual ACM Symposium on POPL*, pages 1–11, 1988.
2. F. Baader and K. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In *11th International Conference on Automated Deduction*, volume 607 of *LNAI*, pages 50–65, 1992.
3. F. Baader and W. Snyder. Unification theory. In *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
4. L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *J. of Automated Reasoning*, 31(2):129–168, 2003.
5. C. W. Barrett, D. L. Dill, and J. R. Levitt. Validity checking for combinations of theories with equality. In *First International Conference on Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201, 1996.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on POPL*, pages 234–252, 1977.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on POPL*, pages 84–96, 1978.
8. S. Gulwani and G. C. Necula. Discovering affine equalities using random interpretation. In *30th Annual ACM Symposium on POPL*, Jan. 2003.
9. S. Gulwani and G. C. Necula. Global value numbering using random interpretation. In *31st Annual ACM Symposium on POPL*, Jan. 2004.
10. S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *Static Analysis Symposium*, volume 3148 of *LNCS*, pages 212–227, 2004.
11. S. Gulwani and G. C. Necula. Precise interprocedural analysis using random interpretation. In *32nd Annual ACM Symposium on POPL*, Jan. 2005.
12. S. Gulwani and A. Tiwari. Combining abstract interpreters. *Submitted for publication*, Nov. 2005.
13. S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. Technical Report MSR-TR-2006-01, Microsoft Research, Jan. 2006.
14. M. Karr. Affine relationships among variables of a program. In *Acta Informatica*, pages 133–151. Springer, 1976.
15. G. A. Kildall. A unified approach to global program optimization. In *1st ACM Symposium on POPL*, pages 194–206, Oct. 1973.
16. M. Müller-Olm, O. Rüthing, and H. Seidl. Checking herbrand equalities and beyond. In *VMCAI*, volume 3385 of *LNCS*, pages 79–96. Springer, Jan. 2005.
17. M. Müller-Olm and H. Seidl. A note on Karr’s algorithm. In *31st International Colloquium on Automata, Languages and Programming*, pages 1016–1028, 2004.
18. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *31st ACM Symposium on POPL*, pages 330–341, Jan. 2004.
19. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *European Symposium on Programming*, pages 46–60, 2005.
20. G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.
21. M. Schmidt-Schauss. Unification in a combination of arbitrary disjoint equational theories. *J. Symbolic Computation*, 8(1-2):51–99, 1989.

Embedding Dynamic Dataflow in a Call-by-Value Language*

Gregory H. Cooper and Shriram Krishnamurthi

Brown University, Providence, RI 02912, USA
{greg, sk}@cs.brown.edu

Abstract. This paper describes FrTime, an extension of Scheme designed for writing interactive applications. Inspired by functional reactive programming, the language embeds dynamic dataflow within a call-by-value functional language. The essence of the embedding is to make program expressions evaluate to nodes in a dataflow graph. This strategy eases importation of legacy code and permits incremental program construction. We have integrated FrTime with the DrScheme programming environment and have used it to develop several novel applications. We describe FrTime’s design and implementation in detail and present a formal semantics of its evaluation model.

1 Introduction

This paper describes FrTime (pronounced “father time”), a programming language built atop the DrScheme environment [9]. FrTime is an exploration of an important point in the design space of dynamic dataflow, or functional reactive [7, 16, 19], programming.

To make FrTime as familiar as possible to current programmers, the language reuses much of the infrastructure, including the syntax, of an existing call-by-value language. In this case the host language is a purely functional subset of Scheme, although the strategy we describe could be applied to other call-by-value languages as well. The embedding strategy reuses the host language’s evaluator to make program execution construct a graph of dataflow dependencies; a dataflow *engine* subsequently reacts to events and propagates changes through this graph. FrTime conservatively extends basic language constructs to trigger graph creation when used in the context of time-varying values. Pure Scheme programs are also FrTime programs with the same meaning they have in Scheme and may be incorporated into FrTime programs without modification.

The design of FrTime reflects a desire to satisfy three main goals.

1. Programs should be able to respond to and process events from external sources. For example, one application of FrTime is as a scripting language for a debugger [15]. A debugger script must respond to events from the program under investigation, which arrive at an unspecified frequency that cannot be known *a priori*. This suggests that the language should embrace a push-driven implementation strategy, where the arrival of an event triggers a computation that propagates up a tree of dependencies.

* This work is partially supported by NSF grant CCR-0305949.

2. FrTime programs should be able to make maximal use of a programming environment for incremental development. This especially means that programmers must be able to write expressions in the read-eval-print loop (REPL), observe and name their values, use them to build larger expressions, and so on. With such support, the REPL can serve as one of the primary interfaces for many programs, thereby saving programmers from having to construct a separate, explicit interface. For example, the paper about the scriptable debugger [15] discusses the debugging of a shortest-paths algorithm by interactively adding script fragments that provide increasingly better clues. FrTime's support for dynamic dataflow graph construction saves us the need to *build* an interaction mode for the debugger. Instead we reuse the DrScheme REPL, inheriting its many features and also its careful handling of many subtle issues [11].
3. As a practical matter, FrTime needs to reuse as much of an existing evaluator as possible. In particular, the underlying evaluator in DrScheme is quite complex and supports a large legacy codebase. Ideally, therefore, FrTime needs an evaluation strategy that can reuse this evaluator and seamlessly integrate as much legacy code as possible to inherit a large library of useful functionality. (For example, by inheriting DrScheme's graphics library, the scriptable debugger supports graphical display of the target program's state.) A corollary is that legacy programs should be incrementally convertible into FrTime. That is, it should be possible to begin with an existing Scheme application and run it under FrTime, then gradually change fragments of it to use dataflow features. This must not require a significant source transformation (such as conversion into continuation-passing or monadic style).

In this paper we present the semantics and implementation of FrTime. In particular, we describe the language's embedding strategy and how it satisfies the goals stated above. We also provide an operational semantics that specifies the language's evaluation model. FrTime has been distributed with the DrScheme programming environment since 2003 and has been used to develop several non-trivial applications, including a scriptable debugger [15], a spreadsheet, and a version of the Slideshow [10] presentation system enhanced with interactive animations.

2 The FrTime Language

FrTime extends the language of DrScheme [9] with support for dynamic dataflow through a notion of *signals*, or time-varying values. The language is inspired and informed by work on functional reactive programming (FRP) [7, 16, 19], which extends Haskell [14] with similar features.

The most basic signals are those that represent time itself. For example, there is a signal called *seconds*, which counts the number of seconds elapsed since a specific point in the past. *Seconds* is an example of a *behavior*—a signal that is defined at every point in time, or *continuous*. If we apply a primitive function f to a behavior, the result is a new behavior, whose value is computed by applying f to the argument at (conceptually) every point in time.¹ In other words, FrTime *lifts* primitive functions to the domain of

¹ Operationally, the language only applies f to the argument initially and each time it changes.

behaviors. For example, if we evaluate *(even? seconds)*, the result is a new behavior that indicates, at every moment, whether the current value of *seconds* is even.

In addition to behaviors, there are signals called event streams that carry sequences of discrete values. For example, we have built an interface to the DrScheme window toolkit that provides an event stream called *key-strokes*, which carries key events. Unlike with behaviors, primitive procedures cannot be applied to event sources. FrTime instead provides a collection of event-processing combinators that are analogous to common list-processing routines. For example, the raw *key-strokes* stream contains events for key presses and releases. Applications that don't care about the releases can elide them with *(filter-e char? key-strokes)*. This produces a new event stream that only carries the events whose values are characters.

There is similarly an analog of *map* called *map-e*, which we could use to convert all of the alphabetic characters to upper case. Another combinator, called *collect-e*, resembles Haskell's *scanl*; it consumes an event stream, an initial accumulator, and a transformer. For each event occurrence, *collect-e* applies the transformer to the new event and the accumulator, yielding a new accumulator which is emitted on the resulting event stream. By passing **empty** and **cons** as the second and third arguments, we can build a list of all the occurrences of a given event.

FrTime provides primitives for converting between behaviors and event streams. One is *hold*, which consumes an event stream and an initial value and returns a behavior that starts with the initial value and changes to the last event value each time an event occurs. Conversely, *changes* consumes a behavior and returns an event stream that emits the value of the behavior each time it changes.

On the surface, signals bear some similarity to constructs found in other languages. Behaviors change over time, like mutable data structures or the return values of impure procedures, and event streams resemble the infinite lazy lists (also called streams) common to Haskell and other functional languages. The key difference is that FrTime tracks dataflow relationships between signals and automatically recomputes them to maintain programmer-specified invariants.

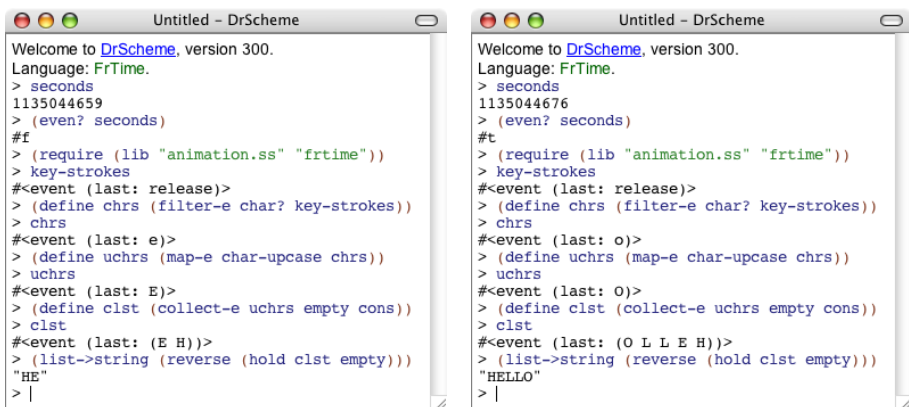


Fig. 1. Screenshots of a single interactive FrTime session, taken 17 seconds apart

FrTime runs in the DrScheme programming environment. Figure 1 presents two screenshots from the same interactive session in DrScheme, taken about seventeen seconds apart. In this session we first evaluate *seconds* and (*even? seconds*). Then we load the FrTime animation library, which creates a new, empty window (not shown). As we type into this new window, key press and release events arrive on the *key-strokes* event stream. We create *chrs* by filtering out the release events, and *uchrs* by converting these to upper-case with *map-e*. We make *collect-e* accumulate a list of characters, then apply *hold* to produce a behavior, which we reverse and convert to a string.

Evaluating a signal at the DrScheme prompt registers a dependency between that signal and the graphical object that represents it in the interactions window. Thus, when the signal's value changes, FrTime automatically triggers an update of the display. This explains why the two screenshots in Fig. 1 show different values for many expressions, even though they are taken from the same session. This is an important example of integrating the language with the environment in order to respect the language's unconventional abstractions. Conversely, the language supports the environment's notion of interactive, incremental program construction. For example, as we build up the string of *key-strokes*, we can name and observe each intermediate result, checking that it behaves as we expect before adding the next piece.

3 Evaluation Strategy

In this section we describe FrTime's evaluation strategy, which satisfies the goals set forth in the Introduction. Firstly, it employs a push-driven update mechanism: events initiate computation, and changes cause dependent parts of the program to recompute. Secondly, the language supports incremental program construction; the programmer can interleave program construction, evaluation, and observation. Finally, it reuses the Scheme evaluator and permits reuse of existing Scheme library code, which supports incremental conversion of Scheme programs to use FrTime's dataflow features.

FrTime is a collection of syntactic abstractions and value definitions implemented in Scheme. Executing a FrTime program means running the Scheme evaluator in an environment containing the FrTime definitions. These definitions *make executing the program build a graph of its dataflow dependencies*. The nodes of this graph correspond to program expressions, and the arcs indicate flow of values from one expression to another. An expression that does not utilize any dataflow elements evaluates as a standard, pure Scheme expression, yielding the same value it would have in Scheme.

Because evaluation is push-driven, a program's reactivity originates through dependence on primitive event sources, for example a timer, a keyboard, a mouse, or a network data stream. The FrTime *engine* listens to events from these sources and routes them to the interested parts of the program's dataflow graph. Values change at the corresponding nodes of the dataflow graph and propagate along the dependency arcs.

In the remainder of this section, we explain how evaluating a FrTime expression constructs a graph of dataflow dependencies, and how the language implements reactivity through subsequent traversal of this graph. We discuss some of the difficulties that arise from a push-driven update model and how we solve them.

3.1 Dataflow Graph Construction and Manipulation

Suppose the programmer enters the expression $(+ 3 4)$ at the FrTime REPL. Its evaluation proceeds in the traditional call-by-value fashion, first reducing subexpressions to values, then applying the specified operation to them.

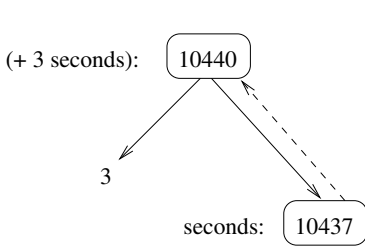
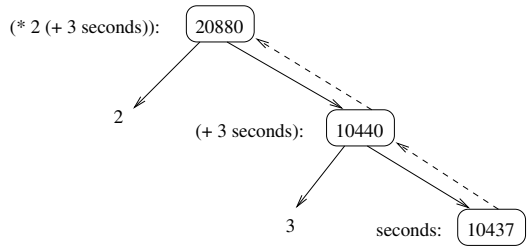
FrTime is meant to extend pure Scheme with a notion of signals, so if we start with a pure Scheme expression and replace some constant values with signals, the result should be a legal FrTime program. For example, we should be able to refer to “the time 3 seconds from now” by writing $(+ 3 \text{ seconds})$. However, evaluating such an expression in a standard Scheme evaluator does not yield the desired dataflow semantics. Scheme primitives like $+$ only know how to process ordinary, constant Scheme values (in this case numbers). At best, the result might be to add 3 to the *current value* of *seconds*. This would produce a constant value reflecting the state of the system at the moment of evaluating the expression, but it would fail to update with the passage of time. In reality, the situation is worse; FrTime’s signals are implemented as data structures, so passing *seconds* to $+$ is a type mismatch and causes a runtime exception.

Clearly, *ordinary* Scheme evaluation does not work for FrTime. This means that we must either write a new evaluator for FrTime, or extend Scheme evaluation to accommodate FrTime’s novel features. Since we want to reuse as much of Scheme as possible, we take the latter approach by interposing a mechanism that prevents the direct application of Scheme primitives to signals. Specifically, we define the FrTime evaluation environment so that the names of Scheme primitives refer to lifted versions of the same. Lifting wraps a primitive with code that checks for signal arguments and, if there are any, constructs and returns a new signal. For example, the FrTime expression $(+ 3 \text{ seconds})$ reduces to the following Scheme code:

```
(if (or (signal? 3) (signal? seconds))
    (make-signal (λ () (+ (current-value 3) (current-value seconds))) 3 seconds)
    (+ 3 seconds))
```

This first tests for signals among the argument subexpressions. Since *seconds* is a signal, the conditional selects the first branch. The procedure **make-signal** consumes a thunk (nullary procedure), boxed above, and any number of *producer* values to which the thunk refers. It returns a new signal whose value is defined, at any point in time, by the result of calling the thunk. In this case, it applies the addition primitive to the current values of the constant 3 and the signal *seconds*. The procedure *current-value* acts like the identity function on constant values, so $(\text{current-value } 3)$ reduces to 3. On signals, *current-value* projects the signal’s current value, an ordinary Scheme constant. Thus the addition primitive inside the thunk sees only constants, so there are no errors. The fact that signals like *seconds* change over time underscores the necessity of the thunk: the language needs to re-evaluate the procedure to update the signal when any of the producers change.

The additional arguments to **make-signal** (here 3 and *seconds*) are the producers on which the new signal depends. They may include both constants and signals; **make-signal** ignores the constants and registers a dependency with each of the signals. Registration gives the producers explicit references to the new signal (instead of the other way around, as a reader might initially assume). These *reverse* references are essential to

**Fig. 2.** Dataflow graph for $(+ 3 \text{ seconds})$ **Fig. 3.** Dataflow graph for $(* 2 (+ 3 \text{ seconds}))$

implementing push-driven evaluation: when a signal changes, the runtime system follows them to determine which signals need recomputation. Figure 2 shows the resulting signal graph. Rounded boxes depict signals, solid arrows are normal data references, and dashed arrows represent the reverse references needed for push-driven updates.

The addition of these reverse references would ordinarily expand reachability into a symmetric relation, to the detriment of effective memory management. To solve this problem, we make the reverse references *weak*, which tells the memory manager to ignore them when computing reachability. If a signal is no longer reachable from the application, it will be reclaimed. Since there may be a significant delay between objects' becoming unreachable and their reclamation by the garbage-collector, it is possible that the engine will continue recomputing such *dead* signals for some time. This strategy is unacceptable in general, so we need a mechanism for deleting signals when they cease to belong in the system. We describe such a mechanism in Sect. 3.3.

The FrTime evaluation model applies to all expressions, even those that do not use signals. For example, the FrTime expression $(+ 3 4)$ reduces to the following Scheme expression:

```
(if (or (signal? 3) (signal? 4))
    (make-signal (λ () (+ (current-value 3) (current-value 4))) 3 4)
    (+ 3 4))
```

Because the constants 3 and 4 are not signals, the entire expression is clearly equivalent to its raw Scheme counterpart, and yields the constant 7. This illustrates one of our design goals: that pure Scheme expressions should evaluate in FrTime as they would in standard Scheme. This means that programmers can easily mix pure Scheme and FrTime code, which creates a smooth migration path for porting Scheme code.

Because user-defined signals like $(+ 3 \text{ seconds})$ are indistinguishable from primitive signals like *seconds*, evaluation of FrTime expressions works even when operations on signals nest. For example, if a programmer writes $(* 2 (+ 3 \text{ seconds}))$, the inner $(+ \dots)$ subexpression evaluates first, yielding a signal like the one described above. The evaluation of the $(* \dots)$ application proceeds in an analogous manner, constructing a new signal that depends upon the value of $(+ 3 \text{ seconds})$. We show the resulting graph in Fig. 3.

Expressions can arbitrarily nest and mix computations involving constants and signals. For example, if we write $(+ (+ 1 2) \text{ seconds})$, the $(+ 1 2)$ evaluates as in Scheme, reducing to the constant 3, after which evaluation proceeds exactly as above for

(+ 3 seconds). Only one new signal is created, and the resulting dataflow graph is identical to the one shown in Fig. 2.

The dataflow graph construction that occurs when a FrTime program runs is just the first step in its evaluation. The interesting part—the program’s reactivity—begins once the graph is constructed and continues as long as the system runs and events arrive. This involves primitive signals changing in response to external events and propagating through the dataflow graph. For example, once every second, a timer triggers a change in *seconds*, which in turn triggers recomputation of every signal that *depends* on *seconds*, such as (+ 3 seconds) in our example above. Changes then propagate to transitive dependents, such as (* 2 (+ 3 seconds)).

When the engine recomputes a signal, it compares the new value with the previous one. If they are the same (according to Scheme’s **eq?** procedure), the engine does not schedule the signal’s dependents. For example, a signal defined by an expression like (*quotient seconds* 10) depends on *seconds* but only changes after *seconds* increases by ten. Consumers of this signal, like (> (*quotient seconds* 10) 100), only recompute every ten seconds, not every second.

3.2 Glitch Prevention

Scheduling recomputation is an important semantic issue. For example, consider the expression (< *seconds* (+ 1 seconds)). This evaluates to a signal that should always have the value **true**, since *n* is always less than *n* + 1.

However, life is not so simple in a push-driven update model. Each change in *seconds* triggers recomputation of the overall expression and the inner (+ 1 seconds) signal, and the order in which FrTime recomputes these signals affects the answer. If it updates the (+ 1 seconds) signal first, then the top-level < compares up-to-date versions of *seconds* and (+ 1 seconds), yielding **true**. On the other hand, if it updates the top-level signal first, it then compares the up-to-date *seconds* with the stale (+ 1 seconds)—which is equal to the new value of *seconds*—yielding **false**.

This situation, where a signal is recomputed before all of its subordinate signals are up-to-date, is called a *glitch* [5]. Such behavior is unacceptable as it results in redundant computation and, much worse, causes signals to violate invariants.

We need a traversal strategy that prevents glitches. The crucial property is that no signal should update until everything on which it depends is also up-to-date. Unfortunately, the obvious candidates of depth-first and breadth-first search are susceptible to glitches, as the preceding example shows. However, a slightly modified breadth-first search achieves the goal. Specifically, we approximate the graph’s structure by assigning each signal a *height*, which exceeds that of all its producers. To make a valid depth assignment possible, the dataflow graph must be acyclic. This restriction has the benefit of guaranteeing that update propagation terminates, but it also seems to impose a severe limit on the language’s expressive power. We explain in Sect. 3.5 how FrTime supports programs with cyclic dependencies.

Computing signal heights is relatively simple. Since **make-signal** receives all of the new signal’s producers, it only needs to compute their maximum and add 1 to it. Instead of a standard first-in-first-out queue, the engine uses a priority queue to process nodes in order of increasing height. Since each signal is higher than everything on which it depends, this strategy guarantees the absence of glitches and redundant computation.

3.3 Dynamic Reconfiguration

The height-guided recomputation strategy works under the assumption that the dataflow graph does not change in the middle of an update cycle. Unfortunately, this is an unreasonable assumption: the need to reconfigure the graph dynamically arises naturally from combining behaviors with basic Scheme features.

We illustrate some of the intricacies of dynamic reconfiguration through a simple example involving the use of a time-varying condition in an **if** expression:

```
(let* ([len (modulo seconds 4)]
      [lst (build-list len add1)])
  (if (zero? len)
      0
      (list-ref lst (sub1 len))))
```

In this program, *len* cycles through the values 0, 1, 2, 3, and *lst* is the list (1 . . . *len*). When *len* is 0, the value of the whole expression is 0, and otherwise it is the last element of *lst*, which is also equal to *len*.

Evaluating this program proves to be somewhat tricky. Since the **if**'s condition is time-varying, the result of the whole expression needs to switch dynamically between the branches (either of which may also be time-varying), forwarding the value of whichever branch the condition currently selects.

In general, evaluating a branch is only legal when the condition selects it. For example, when the first branch is selected above, evaluating the second branch would raise an exception by attempting to extract the element of *lst* at position -1 . The threat of such problems means that, when the condition changes, a new branch must be constructed and the old one disabled, or deleted, before evaluation proceeds. Thus the structure of the dataflow graph must change in the middle of an update cycle. In this example, the need arises from the use of behaviors in conditionals; an analogous situation arises when the function position of an application is time-varying.

Changing the structure of the dataflow graph in the middle of an update cycle creates a number of hazards that must be handled carefully. Constructing new dataflow graph fragments is precarious because the existing graph may be in an inconsistent state, with some signals updated but others stale. In this example, *lst* has a large height because *build-list* is a recursive procedure that constructs a complex fragment of dataflow graph. However, since *zero?* is a primitive, (*zero? len*) has height 2, and when it becomes **false** (triggering construction of the second branch), *lst* still has the stale value **empty**. Evaluating the new branch (which tries to extract an element from *lst*) would raise an exception. To avoid this problem, FrTime constructs the new branch *without computing the initial node values*. Instead it enqueues the new nodes for update, and the recomputation algorithm initializes them after reaching the proper height.

Another problem is that the new fragment's height may exceed that of the old one. To prevent glitches, the engine needs to adjust height assignments to reflect the new graph topology before performing any more updates. It must also notify the priority queue of any changes in heights of signals that are already enqueued for update.

Deleting a fragment of the dataflow graph is also subtle. To prevent any unwanted evaluation, FrTime must delete all of the signals in the dead branch, including those

already enqueued for recomputation. This means that the height of all of these signals must strictly exceed that of the condition. Deleting a signal involves removing all edges incident on it, which makes it unreachable and ensures that it will not be scheduled for recomputation again. However, since a change may have already scheduled the deleted signal for recomputation, deletion replaces the signal's update procedure with a no-op, preventing ill effects from any final update attempt. (This technique is more efficient than the alternative of removing the deleted signals from the priority queue.)

Determining which signals to delete can be tricky, too. It is not simply the set of all signals reachable from the root of the deleted fragment; this would include many signals merely referenced within the branch (in the example, signals like *len* and *lst*). However, taking only the signals directly created by evaluation of the branch yields an underapproximation. This is because the branch may contain other dynamic expressions, which in turn constructed signals after the creation of the enclosing branch. FrTime needs to track construction within this extended notion of the expression's dynamic extent. The semantics presented in Sect. 4 provides an abstract model of this mechanism, but the details involved in implementing it efficiently are beyond the scope of this paper.

3.4 Incremental Construction

FrTime's evaluation model differs from the approaches taken in the Haskell FRP systems [7, 16]. In those, a program specifies the structure of a dynamic dataflow computation, but the actual reactivity is implemented in an interpreter called *reactimate*. This interpreter runs in an infinite loop, blocking interaction through the REPL until the computation is finished. In many applications, we need to support REPL-style interaction in the middle of the reactive program's execution.

FrTime supports REPL interaction by implementing reactivity in a separate thread. The user is assigned one thread, typically corresponding to the DrScheme REPL, while the FrTime dataflow engine, which constructs and manipulates the program's dataflow graph, runs in a separate thread. These threads communicate through a message queue; at the beginning of each update cycle, the engine empties the queue and processes the messages. Each message corresponds either to an event occurrence or to a request for construction of a new dataflow graph fragment. When the user enters an expression at the REPL prompt, the REPL sends a message to the dataflow engine, which evaluates it and responds with the root of the resulting graph. Control returns to the REPL, which issues a new prompt for the user, while in the background the engine continues processing events and updating signals.

On the surface, it may appear that the Haskell systems could achieve similar behavior simply by spawning a new thread to evaluate the call to *reactimate*. Control flow would return to the REPL, apparently allowing the user to extend or modify the program. However, this background process would still not return a value or offer an interface for probing or extending the running dataflow computation. The values of signals running inside a *reactimate* session, like the dataflow program itself, reside in the procedure's scope and hence cannot escape or be affected from the outside. In contrast, FrTime's message queue allows users to submit new program fragments dynamically, and *evaluating an expression returns a live signal* which, because of the engine's background execution, reflects part of a running computation.

$$\begin{aligned}
x \in \langle \text{var} \rangle &::= (\text{variable names}) & p \in \langle \text{prim} \rangle &::= + \mid - \mid * \mid / \mid < \mid > \mid \dots \\
\sigma \in \langle \text{loc} \rangle &::= (\text{store locations}) & t, n \in \langle \text{num} \rangle &::= 0 \mid 1 \mid 2 \mid \dots \\
u, v \in \langle \text{v} \rangle &::= \perp \mid \text{true} \mid \text{false} \mid \langle \text{num} \rangle \mid \langle \text{prim} \rangle \mid (\lambda (\langle \text{var} \rangle^*) (\langle \text{e} \rangle)) \mid \langle \text{loc} \rangle \\
e \in \langle \text{e} \rangle &::= \langle \text{v} \rangle \mid \langle \text{var} \rangle \mid (\langle \text{e} \rangle \langle \text{e} \rangle^*) \mid (\text{delay } \langle \text{e} \rangle \langle \text{num} \rangle) \mid (\text{if } \langle \text{e} \rangle \langle \text{e} \rangle \langle \text{e} \rangle) \\
E \in \langle \text{E} \rangle &::= [] \mid (\langle \text{v} \rangle^* \langle \text{E} \rangle \langle \text{e} \rangle^*) \mid (\text{delay } \langle \text{E} \rangle \langle \text{num} \rangle) \mid (\text{if } \langle \text{E} \rangle \langle \text{e} \rangle \langle \text{e} \rangle) \\
s \in \langle \text{sig-type} \rangle &::= (\text{lift } \langle \text{prim} \rangle \langle \text{v} \rangle^*) \mid (\text{delay } \langle \text{loc} \rangle \langle \text{num} \rangle \langle \text{loc} \rangle) \mid \text{input} \\
&\quad \mid (\text{dyn } (\lambda (\langle \text{var} \rangle) (\langle \text{e} \rangle) \langle \text{loc} \rangle \langle \text{loc} \rangle)) \mid (\text{fwd } \langle \text{loc} \rangle) \mid \text{const}
\end{aligned}$$

Fig. 4. Grammars for FrTime values, expressions, evaluation contexts, and signal types

3.5 Cycles

We explain in Sect. 3.2 how our height assignment strategy restricts the dataflow graph to be acyclic. However, programs with cyclic signal networks arise naturally in many applications. For example, in user interfaces, we often want two sets of widgets that display and control the same underlying model, such as RGB and HSV views in a color-selection window. Since either set of widgets must be able to influence the other, they are mutually dependent. Forbidding cycles altogether would disallow expression of such patterns, making the language unacceptably weak.

In the current implementation, we make a compromise consistent with that made by other dataflow languages [4, 5, 16, 18, 19]. We provide a **delay** operator that reflects the value that its argument had at a specific interval in the past. If a cycle includes a signal created by **delay**, then that cycle cannot cause the system to enter a tight loop, since the delay halts update propagation until the future. We therefore assign a height of 0 to **delay**-ed signals. As long as each cycle passes through a **delay**, a consistent height assignment is possible, and evaluation is safe.

4 Semantics

We have developed a formal semantics of FrTime’s evaluation model, which highlights the push-driven update strategy and the embedding in a call-by-value functional host language. Figure 4 shows the grammars for values, expressions, evaluation contexts, and signal types. Values include the undefined value (\perp), booleans, numbers, primitive procedures, λ -abstractions, and store locations (which identify signals). Expressions include values, procedure applications, delays, and conditionals. Evaluation contexts [8] enforce a left-to-right, call-by-value order on subexpression evaluation. Signal types, which we explain in detail below, describe the different signal variants.

Figure 5 presents semantic domains and operations over them. δ , a parameter to the system, defines reduction for primitives. Σ denotes a set of signal locations and X means a set of *external events*, each of which contains a location, a value, and an occurrence time (when it enters the system). I refers to a set of *internal events*, which contain only target locations and (optionally) values. A store S maps signal locations to triples containing a *current value*, a *signal type*, and a *set of dependents*. For notational

$\delta : \langle \text{prim} \rangle \times \langle v \rangle \times \dots \rightarrow \langle v \rangle$	(primitive evaluation)
$\Sigma \subset \langle \text{loc} \rangle$	(store location set)
$I \subset \langle \text{loc} \rangle \cup (\langle \text{loc} \rangle \times \langle v \rangle)$	(internal event set)
$X \subset \langle \text{loc} \rangle \times \langle v \rangle \times \langle \text{num} \rangle$	(external event set)
$S : \langle v \rangle \rightarrow \langle v \rangle \times \langle \text{sig-type} \rangle \times 2^{\langle \text{loc} \rangle}$	(signal in store)
$\mathcal{V}_S(v) = v', \text{ where } S(v) = (v', -, -)$	(current value projection)
$A(\Sigma, v_0, v) = \begin{cases} \Sigma & \text{if } v \neq v_0 \\ \emptyset & \text{otherwise} \end{cases}$	(signals affected by change)
$\text{reg}(\sigma, \Sigma, S) = S[\sigma' \mapsto (v, s, \Sigma' \cup \{\sigma\})]_{\forall \sigma' \in \Sigma [S(\sigma') = (v, s, \Sigma')]}$	(dependency registration)
$\mathcal{D}_S(\Sigma) = \bigcup_{\sigma \in \Sigma} \Sigma', \text{ where } S(\sigma) = (-, -, \Sigma')$	(dependency lookup)
$\text{dfrd}_S(I) = \mathcal{D}_S^+(\{\sigma \mid \sigma \in I \vee (\sigma, -) \in I\})$	(deferred recomputations)
$\text{del}(S, \Sigma) = \left(\begin{array}{l} S[\sigma \mapsto (v, s, \Sigma' \setminus \Sigma)]_{\forall \sigma [S(\sigma) = (v, s, \Sigma')],} \\ \bigcup_{\sigma \in \Sigma} \begin{cases} \Sigma' & \text{if } S(\sigma) = (-, (\text{dyn } - -), \Sigma') \\ \emptyset & \text{otherwise} \end{cases} \end{array} \right)$	(dependency removal)

Fig. 5. Semantic domains and operations

convenience when dealing with behaviors and constants, the store permits lookup of constants, which `const` signals. This simplifies the definition of \mathcal{V}_S , which projects the current value of any signal or constant. Other important operations include *reg*, which registers one signal's dependence on a set of other signals, and \mathcal{D}_S , which computes the set of signals dependent upon any of a set of signals. *dfrd* computes the set of stale signals that are deferred, or not ready for immediate update (the $^+$ indicates transitive, irreflexive closure). Finally, *del* eliminates references to deleted signals from a given store. As explained in Sect. 3.3, FrTime needs to delete signals recursively from nested dynamic branches. To facilitate this, *del* not only returns the modified store but also finds all the nested `dyn` signals, whose children must be deleted.

FrTime's evaluation model divides naturally into two layers. One is the context-sensitive rewriting system that captures the call-by-value functional core and the extension that constructs the dataflow graph. Figure 6 shows the transformation rules that comprise this layer. These *construction* rules reduce expressions in the context of a store and a set of internal events. The δ , β_v , and *IF* reductions are standard for languages derived from the λ -calculus; they neither read nor change any of the additional elements in the tuple. The *LIFTed* versions of these rules describe how the system extends the dataflow graph when behaviors are used with primitive procedures, user-defined procedures, and conditionals.

The *LIFTed* rules explain only the construction of the dataflow graph. The reactivity is described by the layer of *update* rules, which are presented in Fig. 7. These specify how the system evolves when each variety of signal updates:

lift Application of a primitive to one or more behaviors results in the lifting of the application (rule δ -LIFT). This yields a new `lift` signal that records the primitive and its arguments. The new signal is enqueued for update, which invokes rule *U-LIFT* after all the arguments are up-to-date. The rule computes the signal's value by applying the primitive to the current values of the arguments. If the new value differs from the old one, the signal's dependents are enqueued for update.

$$\begin{array}{c}
 \frac{\{v_1, \dots, v_n\} \wedge \langle \text{loc} \rangle = \emptyset}{\langle S, I, E[(p \ v_1 \dots v_n)] \rangle \rightarrow \langle S, I, E[\delta(p, v_1, \dots, v_n)] \rangle} \quad (\delta) \\
 \\
 \frac{\begin{array}{l} \{v_1, \dots, v_n\} \wedge \langle \text{loc} \rangle = \{\sigma_1, \dots, \sigma_k\} \neq \emptyset \quad \forall i \in [1..k]. S(\sigma_i) = (v_i, s_i, \Sigma_i) \\ S' = \text{reg}(\sigma, \{\sigma_1, \dots, \sigma_k\}, S[\sigma \mapsto (\perp, (\text{lift } p \ v_1 \dots v_n), \emptyset)]) \end{array}}{\langle S, I, E[(p \ v_1 \dots v_n)] \rangle \rightarrow \langle S', I \cup \{\sigma\}, E[\sigma] \rangle} \quad (\delta\text{-LIFT}) \\
 \\
 \langle S, I, E[(\lambda (x_1 \dots x_n) e) \ v_1 \dots v_n] \rangle \rightarrow \langle S, I, E[e[v_1/x_1] \dots [v_n/x_n]] \rangle \quad (\beta_v) \\
 \\
 \frac{S' = S[\sigma_1 \mapsto (\perp, (\text{dyn } (\lambda (x) (x \ v_1 \dots v_1)) \ \sigma \ \sigma_2), \emptyset)][\sigma_2 \mapsto (\perp, (\text{fwd } \perp), \emptyset)]}{\langle S, I, E[(\sigma \ v_1 \dots v_n)] \rangle \rightarrow \langle \text{reg}(\sigma_1, \{\sigma\}, S'), I \cup \{\sigma_1\}, E[\sigma_2] \rangle} \quad (\beta_v\text{-LIFT}) \\
 \\
 \begin{array}{l} \langle S, I, E[(\text{if true } e_1 \ e_2)] \rangle \rightarrow \langle S, I, E[e_1] \rangle \\ \langle S, I, E[(\text{if false } e_1 \ e_2)] \rangle \rightarrow \langle S, I, E[e_2] \rangle \end{array} \quad (\text{IF}) \\
 \\
 \frac{S' = S[\sigma_1 \mapsto (\perp, (\text{dyn } (\lambda (x) (\text{if } x \ e_1 \ e_2)) \ \sigma \ \sigma_2), \emptyset)][\sigma_2 \mapsto (\perp, (\text{fwd } \perp), \emptyset)]}{\langle S, I, E[(\sigma \ v_1 \dots v_n)] \rangle \rightarrow \langle \text{reg}(\sigma_1, \{\sigma\}, S'), I \cup \{\sigma_1\}, E[\sigma_2] \rangle} \quad (\text{IF-LIFT}) \\
 \\
 \frac{S' = \text{reg}(\sigma_2, \{\sigma\}, S[\sigma_1 \mapsto (\perp, \text{input}, \emptyset)][\sigma_2 \mapsto (\perp, (\text{delay } \sigma \ n \ \sigma_1), \emptyset)])}{\langle S, I, E[(\text{delay } \sigma \ n)] \rangle \rightarrow \langle S', I \cup \{\sigma_2\}, E[\sigma_1] \rangle} \quad (\text{DELAY})
 \end{array}$$

Fig. 6. Construction rules

delay, input Delaying a signal requires two new signals: a *consumer* (of type `delay`) observes changes in the argument and directs events to a *producer* that arrive after the given interval (rule U-DELAY). The producer has type `INPUT` and simply forwards the delayed value carried by the latest event (rule U-INPUT). Because communication passes through the external event mechanism, there is no direct dependence; this is why **delay** breaks cycles. In general, input signals can channel values into the system from the external event queue. They are thus useful not only for **delay** but can also model events from all manner of input sources, such as a mouse or a network port.

dyn, fwd Signals of type `dyn` modify the structure of the dataflow graph in response to changes in a given *trigger* signal. These signals are used to implement both conditionals (**if** expressions) and applications with a signal in the function position. For conditionals, the trigger is the condition, while for applications the trigger is the function. Each `dyn` signal contains an update procedure (the *u* field in rule U-DYN), which FrTime applies to the current value of the trigger (σ_1) to yield a new branch of dataflow graph (rooted at σ_3). The branch is connected to the rest of the graph by a permanent `fwd` signal, which forwards the value of the current branch. The `dyn` signal's Σ field, normally used to track dependents, tracks all the signals created by the most recent invocation of the update procedure. These are the signals that must be deleted when a change in the trigger invalidates the existing branch. Each application of *del* removes references to these signals in the store and accumulates the set of signals created by nested `dyn` signals. These also must be deleted and may in turn have children requiring deletion. The language thus applies *del* repeatedly until no deletions remain.

The rules described above leave the precise scheduling of updates non-deterministic. However, they enforce a topological order, which guarantees the absence of glitches and

$$\begin{array}{c}
\frac{I \ni \sigma \notin \text{dfrd}_S(I) \quad S(\sigma) = (v_0, (\text{lift } p \ v_1 \dots), \Sigma) \quad \delta(p, \mathcal{V}_S(v_1), \dots) = v}{\langle X, S, I, t \rangle \hookrightarrow \langle X, S[\sigma \mapsto (v, (\text{lift } p \ v_1 \dots), \Sigma)], I \setminus \{\sigma\} \cup A(\Sigma, v_0, v), t \rangle} \text{ (U-LIFT)} \\
\\
\frac{\sigma \in I \quad S(\sigma) = (\perp, (\text{delay } \sigma \ n \ \sigma_1), \Sigma)}{\langle X, S, I, t \rangle \hookrightarrow \langle X \cup \{(\sigma_1, \mathcal{V}_S(\sigma), t + n)\}, S, I \setminus \{\sigma\}, t \rangle} \text{ (U-DELAY)} \\
\\
\frac{(\sigma, v) \in I \quad S(\sigma) = (v_0, \text{input}, \Sigma)}{\langle X, S, I, t \rangle \hookrightarrow \langle X, S[\sigma \mapsto (v, \text{input}, \Sigma)], I \setminus \{\sigma\} \cup A(\Sigma, v_0, v), t \rangle} \text{ (U-INPUT)} \\
\\
\frac{\begin{array}{l} \sigma \in I \quad S(\sigma) = (\perp, (\text{dyn } u \ \sigma_1 \ \sigma_2), \Sigma) \\ S(\sigma_2) = (v, (\text{fwd } _), \Sigma_2) \quad (S^*, \emptyset) = \text{del}^*(S, \Sigma) \\ \langle S^*, I, (u \ \mathcal{V}_S(\sigma_1)) \rangle \rightarrow^* \langle S', I', \sigma_3 \rangle \quad \Sigma' = \text{dom}(S') \setminus \text{dom}(S) \\ S_1 = \text{reg}(\sigma_2, \{\sigma_3\}, S'[\sigma \mapsto (\perp, (\text{dyn } u \ \sigma_1 \ \sigma_2), \Sigma')][\sigma_2 \mapsto (v, (\text{fwd } \sigma_3), \Sigma_2)]) \end{array}}{\langle X, S, I, t \rangle \hookrightarrow \langle X, S_1, (I' \setminus \Sigma) \setminus \{\sigma\}, t \rangle} \text{ (U-DYN)} \\
\\
\frac{\sigma \in I \quad S(\sigma) = (v_0, (\text{fwd } \sigma'), \Sigma) \quad S(\sigma') = (v, _, _)}{\langle X, S, I, t \rangle \hookrightarrow \langle X, S[\sigma \mapsto (v, (\text{fwd } \sigma'), \Sigma)], I \setminus \{\sigma\} \cup A(\Sigma, v_0, v), t \rangle} \text{ (U-FWD)} \\
\\
\langle X, S, \emptyset, t \rangle \hookrightarrow \langle X, S, \{(\sigma, v) \mid (\sigma, v, t + 1) \in X\}, t + 1 \rangle \text{ (U-SHIFT)}
\end{array}$$

Fig. 7. Update rules

makes the state at the end of each update cycle well-defined. When there are no more internal update events to process, the system is stable and awaits the arrival of new events. Time advances to the next step, and any external events scheduled for the new time shift into the set of internal events (rule U-SHIFT).

5 Related Work

There is a large body of research on dataflow programming. An early language was Lucid [18], a pure, first-order dataflow language based on synchronous streams. Lustre [4] offers a similar programming model to that of Lucid, but with restrictions that support compilation to finite automata and real-time performance guarantees. Lustre also adds a notion of user-defined clocks, allowing streams to compute at different rates. Lucid Synchrone [12] extends Lustre with ML-style type inference, pattern-matching, and first-class functions. Signal [2] is similar to Lustre but is based on relations rather than functions, so the evaluation model is non-deterministic. There are other synchronous languages, such as Esterel [3], whose programming models are imperative.

Functional reactive programming (FRP) [7, 16, 17, 19] merges the model of synchronous dataflow programming with the expressive power of Haskell, a statically-typed, higher-order functional language. In addition, it adds support for *switching* (dynamically reconfiguring a program's dataflow structure) and introduces a conceptual separation of signals into (continuous) *behaviors* and (discrete) *events*.

There has been significant work on implementation models for FRP. Real-time FRP [20] FRP is close in spirit to the synchronous dataflow languages, where the focus is on bounding resource consumption. Parallel FRP [17] adds a notion of non-determinism and explores compilation of FRP programs to parallel code. Elliott discusses several functional implementation strategies for FRP systems [6], which suffer from various

practical problems such as time- and space-leaks. A newer version, Yampa [16], fixes these problems at the expense of some expressive power: while Fran [7] extended Haskell with first-class signals, the Yampa programmer builds a network of *signal functions* in a custom syntax, through a set of *arrow* combinators [13]. FrTime’s linguistic goals are more in line with those of Fran—integrating signals with the Scheme language in as seamless a manner as possible. Importantly, because Scheme is eager, the implementation has precise control over when signals begin evaluating, which helps to prevent time-leaks. In addition, the use of state in the implementation allows more control over memory usage, which helps to avoid space-leaks. The evaluation model leads to several other differences, as described in Section 3.

Frappé [5] is a Java library for building FRP-style dynamic dataflow graphs. Its evaluation model is similar to FrTime’s, in the sense that computation is driven by external events, not by a central clock. However, the propagation strategy is based on a “hybrid push-pull” algorithm, whereas FrTime’s is entirely push-driven, which makes conditional evaluation more challenging. A more important difference from FrTime is that Frappé is a library, not a language. It is intended less for end-user programming than as runtime support for an FRP compiler that targets Java.

Adaptive functional programming (AFP) [1] supports incremental recomputation of function results when their inputs change. As in FrTime, execution occurs in two stages. First the program runs, constructing a graph of its data dependencies. The user then changes input values and tells the system to recompute their dependents. The key difference from FrTime is that AFP requires transforming the program into *destination-passing style*. This prevents the easy import of legacy code and complicates the task of porting existing libraries. The structure of AFP also leads to a more linear recomputation process, where the program re-executes from the first point affected by the changes.

6 Conclusions and Future Work

We have presented FrTime, an implementation of functional reactive programming for a call-by-value language. We have described its novel evaluation model, which accomplishes the goals set forth in the Introduction. We have also provided a formal semantic model for reasoning about FrTime evaluation more abstractly. The language is integrated and distributed with the DrScheme programming environment. We have developed interfaces for various libraries and built several non-trivial applications with it.

Our primary focus for future research is to improve performance of the update strategy. Currently, there is significant overhead involved when moving from Scheme’s top-down, stack-based execution model to FrTime’s push-driven, queue-based update algorithm. In particular, we have noticed severe degradations in performance when running code from existing Scheme libraries under FrTime. FrTime permits fine control (not described in this paper) over the boundary between the two execution strategies, and we are interested in developing mechanical techniques for optimizing the decision.

Acknowledgements. We are grateful to Antony Courtney, Paul Hudak, Guillaume Marceau, and John Peterson for valuable discussions about this work. We also thank the anonymous reviewers for their suggestions.

References

1. U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 247–259, 2002.
2. A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.
3. G. Berry. *The Foundations of Esterel*. MIT Press, 1998.
4. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 178–188, 1987.
5. A. Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages*, pages 29–44, 2001.
6. C. Elliott. Functional implementations of continuous modeled animation. In *Programming Languages: Implementations, Logics, and Programs*, pages 284–299, 1998.
7. C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–277, 1997.
8. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102(2):235–271, 1992.
9. R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.
10. R. B. Findler and M. Flatt. Slideshow: Functional presentations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 224–235, 2004.
11. M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming languages as operating systems (or, Revenge of the Son of the Lisp Machine). In *ACM SIGPLAN International Conference on Functional Programming*, pages 138–147, 1999.
12. G. Hamon and M. Pouzet. Modular Resetting of Synchronous Data-flow Programs. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 289–300, 2000.
13. J. Hughes. Generalizing monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.
14. S. P. Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98*. 1999.
15. G. Marceau, G. H. Cooper, S. Krishnamurthi, and S. P. Reiss. A dataflow language for scriptable debugging. In *IEEE International Symposium on Automated Software Engineering*, pages 218–227, 2004.
16. H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *ACM SIGPLAN Workshop on Haskell*, pages 51–64, 2002.
17. J. Peterson, V. Trifonov, and A. Serjantov. Parallel functional reactive programming. In *Practical Aspects of Declarative Languages*, pages 16–31, 2000.
18. W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press U.K., 1985.
19. Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM Conference on Programming Language Design and Implementation*, pages 242–252, 2000.
20. Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *ACM SIGPLAN International Conference on Functional Programming*, pages 146–156, 2001.

Polymorphic Type Inference for the JNI^{*}

Michael Furr and Jeffrey S. Foster

University of Maryland, College Park
{furr, jfoster}@cs.umd.edu

Abstract. We present a multi-lingual type inference system for checking type safety of programs that use the Java Native Interface (JNI). The JNI uses specially-formatted strings to represent class and field names as well as method signatures, and so our type system tracks the flow of string constants through the program. Our system embeds string variables in types, and as those variables are resolved to string constants during inference they are replaced with the structured types the constants represent. This restricted form of dependent types allows us to directly assign type signatures to each of the more than 200 functions in the JNI. Moreover, it allows us to infer types for user-defined functions that are parameterized by Java type strings, which we have found to be common practice. Our inference system allows such functions to be treated polymorphically by using instantiation constraints, solved with semi-unification, at function calls. Finally, we have implemented our system and applied it to a small set of benchmarks. Although semi-unification is undecidable, we found our system to be scalable and effective in practice. We discovered 155 errors and 36 cases of suspicious programming practices in our benchmarks.

1 Introduction

Foreign function interfaces (FFIs) allow programs to call functions written in other languages. FFIs are important because they let new languages access system libraries and legacy code, but using FFIs correctly is difficult, as there is usually little or no compile-time consistency checking between native and foreign code. As a result, programs that use FFIs may produce run-time typing errors or even violate type safety, thereby causing program crashes or data corruption.

In this paper we develop a multi-lingual type inference system that checks for type safe usage of Java’s FFI to C, called the Java Native Interface (JNI).¹ In the JNI, most of the work is done in C “glue code,” which translates data between the two languages and then in turn invokes other routines, often in system or user libraries. Java primitives can be accessed directly by C glue code because they have the same representations as C primitives, e.g., a Java integer can be given the C type `int`. However all Java objects, no matter what class they are from, are assigned a single opaque type `jobject` by the JNI. Since `jobject`

^{*} This research was supported in part by NSF CCF-0346982 and CCF-0430118.

¹ The JNI also contains support for C++, but our system only analyzes C code.

contains no further Java type information, it is easy for a programmer to use a Java object at a wrong type without any compile-time warnings.

We present a new type inference system that embeds Java type information in C types. When programmers manipulate Java objects, they use JNI functions that take as arguments specially-formatted strings representing class names, field names, and method signatures. Our inference system tracks the values of C string constants through the code and translates them into Java type annotations on the C `object` type, which is a simple form of dependent types.

We have found that JNI functions are often called with parameters that are not constants—in particular, programmers often write “wrapper” functions that group together common sequences of JNI operations, and the Java types used by these functions depend on the strings passed in by callers. Thus a novel feature of our system is its ability to represent partially-specified Java classes in which class, field, and method names and type information may depend on string variables in the program. During type inference these variables are resolved to constants and replaced with the structured types they represent. This allows us to handle wrapper functions and to directly assign type signatures to the more than 200 functions in the JNI.

Our system also supports polymorphism for functions that are parameterized by string variables and Java types. Our type inference algorithm generates instantiation constraints [1] at calls to polymorphic functions, and we present an algorithm based on semi-unification [2, 3] for solving the constraints.

We have implemented our system and applied it to a small set of benchmarks. Although semi-unification is undecidable, we found our algorithm to be both scalable and effective in practice. In our experiments, we found 155 errors and 36 suspicious but non-fatal programming mistakes in our benchmarks, suggesting that programmers do misuse the JNI and that multi-lingual type inference can reveal many mistakes.

In summary, the main contributions of this paper are as follows:

- We develop a multi-lingual type inference system that embeds Java types inside of C. Our system uses a simple form of dependent types so that Java object types may depend on the values of C string constants and variables.
- We present a constraint-based type inference algorithm for inferring multi-lingual types for C glue code. During inference, as string variables are resolved to constant strings they are replaced with the structured Java types they represent. Our system uses instantiation constraints to model polymorphism for JNI functions and user-defined wrapper functions.
- We describe an implementation of our type inference system. We have applied our implementation to a small set of benchmarks, and as a result, we found a number of bugs in our benchmark suite.

This work complements and extends our previous work on the OCaml-to-C FFI [4]. Our previous system was monomorphic and worked by tracking integers and memory offsets into the OCaml heap. Our previous system also did not model objects, which clearly limits its applicability to the JNI. Our new system

can model accesses to Java objects using string constants and variables, and performs parametric polymorphic type inference.

2 Background

In this section we motivate our system by briefly describing the JNI [5]. The JNI is typically used to access low-level C libraries which are impractical to recode in Java. Libraries may have a significant amount of development time invested and interfacing it with Java via the JNI avoids duplicated programming work. Also, low-level operating system functions are typically only provided by means of a C library (`libc`) and so the JNI must be used to access them. To call a C function from Java, the programmer first declares a Java method with the `native` keyword and no body. When this native method is invoked, the Java runtime finds and invokes the correspondingly-named C function. Since Java and C share the same representation for primitive types such as integers and floating point numbers, C glue code requires no special support to manipulate them. In contrast, Java objects, such as instances of `Object`, `Class`, or `int[]`, are all represented with a single opaque C type `jobject` (often an alias of `void *`), and glue code invokes functions in the JNI to manipulate `jobjects`. For example, to get the object `Point.class`, which represents the class `Point`, a programmer might write the following C code²:

```
jobject pointClass = FindClass("java/awt/Point");
```

Here the `FindClass` function looks up a class by name. The resulting object `pointClass` is used to access fields and methods, as well as create new instances of class `Point`. For example, to access a field, the programmer next writes

```
jfieldID fid = GetFieldID(pointClass, "x", "I");
```

After this call, `fid` contains a representation of the location of the field `x` with type `I` (a Java `int`) in class `Point`. This last parameter is a terse encoding of Java types called a *field descriptor*. Other examples are `F` for float, `[I` for array of integers, and `Ljava/lang/String;` for class `String`. Notice this is a slightly different encoding of class names than used by `FindClass`. Our implementation enforces this difference, but we omit it from the formal system for simplicity.

Finally, to read this field from a `Point` object `p`, the programmer writes

```
jobject p = ...;
int y = GetIntField(p, fid);
```

The function `GetIntField` returns an `int`, and there is one such function for each primitive type and one function `GetObjectField` for objects.

Thus we can see that a simple field access that would be written `int y = p.x` in Java requires three JNI calls, each corresponding to one internal step of the

² The JNI functions discussed in this section are actually invoked slightly differently and take an additional parameter, as discussed in Section 4.

```

int my_getIntField(jobject obj, char *field) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, field, "I");
    return GetIntField(obj, fid);
}

```

Fig. 1. JNI Wrapper Function Example

JVM: getting the type of the object, finding the offset of the field, and retrieving its contents. And while a Java compiler only accepts the code `y = p.x` if it is type correct, errors in C glue code, such as typos in the string `java/awt/Point`, `x`, or `I`, will produce a run-time error. There are also several other places where mistakes could hide. For example, the programmer must be careful to maintain the dependence between the type of `x` and the call to `GetIntField`. If the type of `x` were changed to `float`, then the call must also be changed, to `GetFloatField`, something that is easy to overlook. Moreover, since `pointClass` and `p` both have type `jobject`, either could be passed where the other is expected with no C compiler warning, which we have seen happen in our benchmarks. Invoking a Java method is similar to extracting a field. We omit the details due to lack of space.

One common pattern we have seen in JNI code is wrapper functions that specialize JNI routines to particular classes, fields, or methods. For example, Fig. 1 contains a function `my_getIntField` that extracts an integer field from an object. This routine invokes the JNI function `GetObjectClass`, which returns an object representing the class of its argument (as opposed to `FindClass`, which looks up a class by name). Calling `my_getIntField` is safe if the first parameter has an integer field whose name is given by the second parameter. Thus this function is parameterized by the object and by the name of the field, but not its type. Since this wrapper function might be called multiple times with different objects and different field names, we need to perform polymorphic type inference, not only in the types of objects but also in the values of string parameters.

3 Type System

In this section, we describe our multi-lingual type inference system. The input to our system is a collection of Java classes and a C program. Our type inference system analyzes the C program and generates *constraints* that describe how it uses Java objects. We also add constraints based on the type signatures of native methods and the actual class definitions from Java. We then solve the constraints to determine whether the C code uses Java objects consistently with the way they are declared, and we report any discrepancies as type errors.

Extracting type information from Java class files is relatively straightforward. Thus most of the work in our system occurs in the C analysis and constraint resolution phases, so these are the focus of this section.

3.1 Multi-lingual Types

To check that C glue code uses Java types correctly, we must reconstruct the Java types for C variables that are declared with types like `jobject`, `jfieldID`, and `jmethodID`. Before giving our type language formally, consider the function `my_getIntField` in Fig. 1. Our system will give this function the following type, which corresponds to the informal description given at the end of Section 2:

$$\text{my_getIntField} : \{\nu; \langle \nu_{\text{field}} : \text{JInt} \rangle \circ \phi; \mu\} \text{jobject} \times \text{str}\{\nu_{\text{field}}\} \rightarrow \text{JInt} \text{jobject}$$

This is the type of a function that takes two parameters and returns a Java integer (which has the same representation as a C `int`). Note we use the constructor `jobject` to denote any Java type, not just objects. The second parameter is a C string whose contents are represented by the variable ν_{field} . The first parameter is an object that contains a field with name ν_{field} and Java type `int`. Here we have embedded Java type information into the bare C type in order to perform checking. The type of `my_getIntField` places no constraints on the class name ν of the first parameter or its other fields ϕ and methods μ . In order to infer this type, we need to track intermediate information about `cls` and `fid` as well. A detailed explanation of how this type is inferred is given in Section 3.4.

Our formal type grammar is given in Fig. 2. C types `ct` include void, integers, string types `str{s}` (corresponding to C types `char *`), and function types. In type `str{s}`, the string s may be either a known constant string “*Str*” or a type variable ν that is later resolved to a constant string. For example, in Fig. 1, `field` is given type `str{ ν_{field} }`, and ν_{field} is then used as a field name in the type of `my_getIntField`.

Our type language embeds a Java type jt in the C type `jobject`. In order to model the various ways the JNI can be used to access objects, we need a richer representation of Java types than just simple class names. For example, the wrapper function in Fig. 1 may be safely called with mutually incompatible classes as long as they all have the appropriate integer field. Thus our Java types include type variables α , the primitives `JVoid` and `JInt`, and *object descriptions* o of the form $\{s; F; M\}$, which represents an object whose class is named s with *field set* F and *method set* M . Since our inference system may discover the fields

$$\begin{aligned} \text{ct} &::= \text{void} \mid \text{int} \mid \text{str}\{s\} \mid (\text{ct} \times \cdots \times \text{ct}) \rightarrow \text{ct} \\ &\quad \mid jt \text{jobject} \mid (f, o) \text{jfieldID} \mid (m, o) \text{jmethodID} \\ jt &::= \alpha \mid \text{JVoid} \mid \text{JInt} \mid o \mid jt \text{JClass} \mid \text{JTStr}\{s\} \\ o &::= \{s; F; M\} \\ s &::= \nu \mid \text{“Str”} \\ f &::= s : jt \\ F &::= \phi \mid \emptyset \mid \langle f; \cdots; f \rangle \circ F \\ m &::= s : (jt \times \cdots \times jt) \rightarrow jt \\ M &::= \mu \mid \emptyset \mid \langle m; \cdots; m \rangle \circ M \end{aligned}$$

Fig. 2. Type Language

$$\begin{aligned}
FindClass &: (\mathbf{str}\{\nu\}) \rightarrow \mathbf{JTStr}\{\nu\} \text{ JClass jobject} \\
GetObjectClass &: (\{\nu; \phi; \mu\} \text{ jobject}) \rightarrow \{\nu; \phi; \mu\} \text{ JClass jobject} \\
GetFieldID &: (o \text{ JClass jobject} \times \mathbf{str}\{\nu_2\} \times \mathbf{str}\{\nu_3\}) \rightarrow (f, o) \text{ jfieldID jobject} \\
&\quad \text{where } o = \{\nu_1; \langle f \rangle \circ \phi; \mu\} \text{ and } f = \nu_2 : \mathbf{JTStr}\{\nu_3\} \\
GetIntField &: (o \text{ jobject} \times (f, o) \text{ jfieldID jobject}) \rightarrow \mathbf{JInt} \text{ jobject} \\
&\quad \text{where } o = \{\nu_1; \langle f \rangle \circ \phi; \mu\} \text{ and } f = \nu_2 : \mathbf{JInt}
\end{aligned}$$

Fig. 3. Sample JNI Type Signatures. All unconstrained variables are quantified

and methods of an object incrementally, we allow these sets to grow with the composition operator \circ . A set is *closed* if it is composed with \emptyset , and it is *open* if it is composed with a variable ϕ or μ . Since we never know just from C code whether we have accessed all the fields and methods of a class, field and method sets become closed only when unified with known Java types.

Instances of Java’s `Class` class are essential for using the JNI, and therefore we distinguish them from other objects with the type *jt* `JClass`. For example, in Fig. 1 the variable `cls` is given type $\{\nu; \langle \nu_{field} : \mathbf{JInt} \rangle \circ \phi; \mu\} \text{ JClass jobject}$, meaning it is the class of `obj`. This separation is required so that an instance object is not passed to a function like `GetFieldID` which requires a class object.

Lastly, Java objects whose types are described by string s have type $\mathbf{JTStr}\{s\}$. During inference, when the value of s is determined, $\mathbf{JTStr}\{s\}$ will be replaced by the appropriate type. For example, initially the result type of `my_getIntField` is determined to be $\mathbf{JTStr}\{\text{“I”}\}$, which is immediately replaced by \mathbf{JInt} .

The types $(f, o) \text{ jfieldID}$ and $(m, o) \text{ jmethodID}$ represent intermediate JNI values for extracting field f or method m . The name of a field or method is a string s . For example, `fid` in Fig. 1 has type $(\nu_{field} : \mathbf{JInt}, o)$ where o is our representation of `obj`. We include o so that we can check that this field identifier is used with an object of the correct type.

Given this type grammar, we can precisely describe the types of the JNI functions. Fig. 3 gives the types for the functions we have seen so far in this paper. For instance, the function `GetIntField` takes an object with a field f and a `jfieldID` describing f , and returns an integer. Notice that the object type o is open, because it may have other fields in addition to f . As we discussed in Section 2, it is important that these functions be polymorphic. In the type signatures in Fig. 3, any unconstrained variables are implicitly quantified.

3.2 Constraint Generation

The core of our system is a type inference algorithm that traverses C source code and infers the types in Fig. 2. Due to lack of space, we omit explicit typing rules for the source language, since they are mostly standard. Each C term of type `jobject` is initially assigned type $\alpha \text{ jobject}$ for fresh α , and similarly for

`jfieldID` and `jmethodID`. As we traverse the program, we generate *unification constraints* of the form $a = b$ (where a and b range over `ct`, `jt`, etc) whenever two terms must have the same type. For example, for the code in Fig. 1, we unify the type of `cls` with the return type of `GetObjectClass`. Since the only way to manipulate `jobject` types is through the JNI, in general our analysis only generates constraints at assignment, function definition, and function application. Because we use unification for `jt` types rather than subtyping, our inference algorithm could fail to unify two Java objects where one object is a subtype of the other. However, we did not find this to be a problem in practice (see Section 4).

To support polymorphism, we use *instantiation constraints* of the form $a \preceq_i b$ to model substitutions for quantified variables for each instantiation site i [2, 3]. Formally, we use the following two rules for generalization and instantiation:

$$\begin{array}{c}
 \text{(LET)} \\
 \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma[f \mapsto (t_1, fv(\Gamma))] \vdash e_2 : t_2}{\Gamma \vdash \text{let } f = e_1 \text{ in } e_2 : t_2}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{(INST)} \\
 \frac{\Gamma(f) = (t, \vec{\alpha}) \quad (t, \vec{\alpha}) \preceq_i (\beta, \vec{\alpha}) \quad \beta \text{ fresh}}{\Gamma \vdash f_i : \beta}
 \end{array}$$

In (LET), we represent a polymorphic type as a pair $(t, \vec{\alpha})$, where t is the base type and $\vec{\alpha}$ is the set of variables that may *not* be quantified. Then in (INST), we generate an instantiation constraint $(t, \vec{\alpha}) \preceq_i (\beta, \vec{\alpha})$ where i is unique to this instantiation site. This constraint requires that there exist a substitution S_i such that $S_i(t) = \beta$. Our type rule also demands that $\vec{\alpha} \preceq_i \vec{\alpha}$, i.e., S_i does not instantiate the free variables of the environment. The main advantage to this notation for polymorphism is that it allows us to traverse the source code in any order. In particular, we can generate instantiation constraints for a call to function f before we have seen a definition of f . A full discussion of these rules is beyond the scope of this paper and can be found elsewhere [3].

We have formalized a checking version of our type system in terms of lambda-calculus extended with strings, let-bound polymorphism, and primitives representing Java objects. We also include as primitive a set of JNI functions for operating on Java objects, and the small-step semantics for the language includes a reduction rule for each of these functions. We believe it is straightforward to prove that the reduction rules preserve solutions.

Theorem 1 (Soundness). *If $\Gamma \vdash e : ct$, then there exists a value v such that $e \rightarrow^* v$ and $\Gamma \vdash v : ct$.*

Proof. The proof is available in our companion technical report [6].

In our implementation, we do not keep track of the set $fv(\Gamma)$ for functions. Since C does not have nested functions, we simply issue warnings at any uses of global variables of type `jobject`. In general we have found that programmers use few globals when interacting with the JNI. We do not issue warnings for global `char *` types since these are common in C programs. While this makes our implementation unsound, doing so would generate a very high rate of false positives. Our current implementation also does not check for global variables of type `jfieldID`, `jmethodID`, or any Java types embedded in C data structures.

3.3 Constraint Solving

Our type inference algorithm generates a set of constraints that we need to solve in order to decide if the program is well-typed. To solve the constraints, we use a variant of Fähndrich et al’s worklist algorithm for semi-unification [2].

To simplify our constraint resolution rules below, whenever we refer to field and method sets we always assume they have been flattened so that they are composed with either \emptyset or a variable. During the process of unification, unknown strings ν will be replaced by known constant strings $\mathbf{str}\{“Str”\}$. As this happens, we need to ensure that our object types still make sense. In particular, Java classes may not contain two fields with the same name but different types.³ Thus we also impose a well-formedness constraint on all field sets: any two fields with the same name in a field set must have the same type. Formally, for a field $f = s : jt$ we define $fname(f) = s$ and $fjtype(f) = jt$. Then a field set $\langle f_1; \dots; f_n \rangle$ is well-formed if $fname(f_i) = fname(f_j) \Rightarrow fjtype(f_i) = fjtype(f_j)$ for all i, j . Methods however, unlike fields, may be overloaded in Java, and so we do not apply the above well-formedness condition to them.

During constraint solving, our system may unify a string variable ν with a constant string “Str”. When this occurs, our system uses the *Eval* function shown below to convert a type $\mathbf{JTStr}\{s\}$ into the Java type it represents:

$$\begin{aligned} Eval(\mathbf{JTStr}\{“V”\}) &\Rightarrow \mathbf{JVoid} \\ Eval(\mathbf{JTStr}\{“I”\}) &\Rightarrow \mathbf{JInt} \\ Eval(\mathbf{JTStr}\{“Ljava/lang/String;”\}) &\Rightarrow \{“java/lang/String”; \dots; \dots\} \\ Eval(\mathbf{JTStr}\{“Ljava/awt/Point;”\}) &\Rightarrow \{“java/awt/Point”; \dots; \dots\} \\ &\vdots \end{aligned}$$

We use a similar function to convert the string representation of a method signature into a list of Java types.

We express constraint solving in terms of rewrite rules, shown in Fig. 4. Given a set of constraints C , we apply these rules exhaustively, replacing the left-hand side with the right-hand side until we reach a fixpoint. Technically because we use semi-unification this algorithm may not terminate, but we have not found a case of this in practice. The complete list of rewrite rules is long and mostly standard, and so Fig. 4 contains only the interesting cases. The exhaustive set of rules may be found in our companion technical report [6].

In Fig. 4, the (Closure) rule unifies two terms b and c when they are both instantiations of the same variable a at the same instantiation site. Intuitively, this rule enforces the property that substitution S_i must replace variable a consistently [3]. The rule (\mathbf{JTStr} Ineq) applies the usual semi-unification rule for constructed types. Since the substitution S_i renames the left-hand side to yield the right-hand side in a constraint \preceq_i , the right-hand side must have the same shape. Thus in (\mathbf{JTStr} Ineq), we unify jt with $\mathbf{JTStr}\{\nu\}$ where ν is fresh and then propagate the semi-unification constraint to s and ν .

³ Although an overloaded field via inheritance is possible, their manipulation in C is not supported by our system and was not observed in our benchmarks.

In (FieldSet InEq), we match up the fields of two non-empty field sets. We directly propagate instantiation constraints to fields f_k and f'_j with the same field name or variable. We then make an instantiation constraint to F' from the remaining fields of the left-hand side for which there does not exist a field with the same name on the right-hand side. Recall that we assume the sets have been flattened, so that F' is either a variable or \emptyset . We then generate the analogous constraint for F and the right-hand side. Notice that this process may result in a field set with multiple fields with variables ν for names. Our implicit well-formedness requirement from Section 3 will handle the case where these variables are later unified with known strings.

We omit the rules for method sets due to lack of space. These rules are similar to the field set rules except for one important detail. Suppose we have an open method set $\langle x : \alpha \rightarrow \alpha \rangle \circ \mu$ which is unified with a closed method set $\langle x : \text{JInt} \rightarrow \text{JInt}; x : \text{JVoid} \rightarrow \text{JVoid} \rangle \circ \emptyset$. Since the method x is overloaded, we do not know if α should unify with JInt or JVoid . Therefore, if this occurs during unification, our tool emits a warning and removes the constraint. Also, we could unify return types of methods with otherwise equal signatures, but do not do so in our current implementation.

The next three rules handle strings. (Str Sub) replaces one string variable by another. (Str Resolve) uses *Eval* to replace occurrences of $\text{JTStr}\{\nu\}$ with their appropriate representations. (Str Eq) and (Str Neq) test string constants for equality.

Because $\text{JTStr}\{\}$ encodes its type as a string, we use a slightly different rewrite rule for this case. In rule (JTStr Sub), if a $\text{JTStr}\{\}$ type is unified with a type variable, then the variable is replaced as normal. However, if a $\text{JTStr}\{s\}$ type is unified with a void type as in (JTStr Void), then we add the constraint that the $s = \text{"V"}$, since *Eval*(s) must produce a void type. We use a similar rule for integers. Similarly, the rule (JTStr Obj) adds the constraint that s must have the same name as the object it unifies with.

3.4 Example

In this section, we demonstrate our inference system on the `my_getIntField` function from Fig. 1. Initially our analysis assigns each parameter and local variable of this function a fresh type (we omit the variable `fid` for brevity as it only provides redundant constraints on `my_getIntField`):

$$\text{obj} : \alpha_{obj} \text{ jobject} \quad \text{cls} : \alpha_{cls} \text{ jobject} \quad \text{field} : \text{str}\{\nu_{field}\}$$

The first line of the function calls the `GetObjectClass` function (call this instantiation site 1). After looking up its type in the environment (shown in Fig. 3 with quantified type variables ν , ϕ , and μ), we add the following constraints:

$$\begin{aligned} \{\nu; \phi; \mu\} \text{ jobject} &\preceq_1 \alpha_{obj} \text{ jobject} \\ \{\nu; \phi; \mu\} \text{ JClass jobject} &\preceq_1 \alpha_{cls} \text{ jobject} \end{aligned}$$

(Closure)	$C \cup \{a \preceq_i b\} \cup \{a \preceq_i c\} \Rightarrow C \cup \{a \preceq_i b\} \cup \{b = c\}$	
(JTStr Ineq)	$C \cup \{ \text{JTStr}\{s\} \preceq_i jt \} \Rightarrow C \cup \{jt = \text{JTStr}\{\nu\}\} \cup \{s \preceq_i \nu\}$ $\nu \text{ fresh}$	
(FieldSet InEq)	$C \cup \{\langle f_1; \dots; f_n \rangle \circ F \preceq_i \langle f'_1; \dots; f'_m \rangle \circ F'\} \Rightarrow$ $C \cup \{ \text{ftype}(f_k) \preceq_i \text{ftype}(f'_j) \mid \text{fname}(f_k) = \text{fname}(f'_j) \}$ $\cup \{ \langle f_k \mid \exists j. \text{fname}(f_k) = \text{fname}(f'_j) \rangle \preceq_i F' \}$ $\cup \{ F \preceq_i \langle f'_j \mid \exists k. \text{fname}(f_k) = \text{fname}(f'_j) \rangle \}$	
(Str Sub)	$C \cup \{\nu_1 = \nu_2\} \Rightarrow C[\nu_1 \mapsto \nu_2]$	
(Str Resolve)	$C \cup \{\nu = \text{"Str"}\} \Rightarrow C[\nu \mapsto \text{"Str"}][\text{JTStr}\{\text{"Str"}\} \mapsto \text{Eval}(\text{"Str"})]$	
(Str Eq)	$C \cup \{\text{"Str}_1" = \text{"Str}_2"\} \Rightarrow C$	$\text{Str}_1 = \text{Str}_2$
(Str Neq)	$C \cup \{\text{"Str}_1" = \text{"Str}_2"\} \Rightarrow \text{error}$	$\text{Str}_1 \neq \text{Str}_2$
(JTStr Sub)	$C \cup \{\alpha = \text{JTStr}\{s\}\} \Rightarrow C[\alpha \mapsto \text{JTStr}\{s\}]$	
(JTStr Void)	$C \cup \{\text{JVoid} = \text{JTStr}\{s\}\} \Rightarrow C \cup \{s = \text{"V"}\}$	
(JTStr Obj)	$C \cup \{\{\nu; F; M\} = \text{JTStr}\{s\}\} \Rightarrow C \cup \{s = \nu\}$	

Fig. 4. Selected Constraint Rewrite Rules

Applying our constraint rewrite rules yields the following new constraints:

$$\begin{array}{ll}
 \alpha_{obj} = \{\nu_2; \phi_2; \mu_2\} & \alpha_{cls} = \{\nu_3; \phi_3; \mu_3\} \text{ JClass} \\
 \nu \preceq_1 \nu_2 & \nu \preceq_1 \nu_3 \\
 \phi \preceq_1 \phi_2 & \phi \preceq_1 \phi_3 \\
 \mu \preceq_1 \mu_2 & \mu \preceq_1 \mu_3 \\
 \nu_2, \phi_2, \mu_2 \text{ fresh} & \nu_3, \phi_3, \mu_3 \text{ fresh}
 \end{array}$$

Then rule (Closure) in Fig. 4 generates the constraints $\nu_2 = \nu_3$, $\phi_2 = \phi_3$, and $\mu_2 = \mu_3$ to require that the substitution corresponding to this call is consistent. Next, `my_getIntField` calls `GetFieldID`, and after applying our constraint rules, we discover (among other things): $\phi_2 = \langle \nu_{field} : \text{JTStr}\{\text{"I"}\} \rangle \circ \phi_4$ with ϕ_4 fresh. Now since we have a `JTStr` with a known string, our resolution rules call `Eval` to replace it, yielding $\phi_2 = \langle \nu_{field} : \text{JInt} \rangle \circ \phi_4$. The last call to `GetIntField` generates several new constraints, but they do not affect the types. Thus after the function has been analyzed, it is given the type

`my_getIntField` : $\{\nu_1; \langle \nu_{field} : \text{JInt} \rangle \circ \phi_4; \mu\} \text{ jobject} \times \text{str}\{\nu_{field}\} \rightarrow \text{JInt jobject}$

In other words, this function accepts any object as the first parameter as long as it has an integer field whose name is given by the second parameter, exactly as intended.

4 Implementation and Experiments

We have implemented the inference system described in Section 3 in the form of two tools used in sequence during the build process. The first tool is a light-weight Java compiler wrapper. The wrapper intercepts calls to `javac` and records

the class path so that the second tool can retrieve class files automatically. The wrapper itself does not perform any code analysis. The second tool applies our type inference algorithm to C code and issues warnings whenever it finds a type error. Our tool uses CIL [7] to parse C source code and the OCaml JavaLib [8] to extract Java type information from compiled class files.

Our implementation contains some additional features not discussed in the formal system. In addition to the type `jobject`, the JNI contains a number of typedefs (aliases) for other object types, such as `jstring` for Java `Strings`. These are all aliases of `jobject`, and so their use is not required by the JNI, and they do not result in any more checking by the C compiler. Our system does not require their use either, but since they are a form of documentation we enforce their intended meaning, e.g., values of type `jstring` are assigned a type corresponding to `String`. We found 3 examples in our benchmarks where programmers used the wrong alias. The JNI also defines types `jvoid` and `jint`, representing Java voids and integers, as aliases of the C types `void` and `int`, respectively. Our system does not distinguish between the C name and its j-prefixed counterpart.

Rather than being called directly, JNI functions are actually stored in a table that is passed as an extra argument (usually named `env`) to every C function called from Java, and this table is in turn passed to every JNI function. For example, the `FindClass` function is actually called with `(*env)->FindClass(env, ...)`. Our system extracts FFI function names via syntactic pattern matching, and we assume that the table is the same everywhere. Function pointers that are not part of the JNI are not handled by our system, and we do not generate any constraints when they are used in a program.

The JNI functions for invoking Java methods must take a variable number of arguments, since they may be used to invoke methods with any number of parameters. Our system handles the commonly-used interface, which is JNI functions declared to be `varargs` using the `...` convention in C. However, the JNI provides two other calling mechanisms that we do not model: passing arguments as an array, and passing arguments using the special `va_list` structure. We issue warnings if either is used.

Although our type system is flow-insensitive, we treat the types of local variables flow-sensitively. Each assignment updates the type of a variable in the environment, and we add a unification constraint to variables of the same name at join points in the control flow graph. See [4] for details.

Lastly, our implementation models strings in a very simple way to match how they are used in practice in C glue code. We currently ignore string operations like `strcat` or destructive updates via array operations. We also assume that strings are always initialized before they are used, since most compilers produce a warning when this is not the case.

We ran our analysis tool on a suite of 11 benchmarks that use the JNI. Fig. 5 shows our results. All benchmarks except `pgpjava` are glue code libraries that connect Java to an external C library. The first 7 programs are taken from the Java-Gnome project [9], and the remaining programs are unrelated. For each

Program	C LOC	Java LOC	Time (s)	Errs	Warnings	False Pos	Impr
libgconf-java-2.10.1	1119	670	2.4	0	0	10	0
libglade-java-2.10.1	149	1022	6.9	0	0	0	1
libgnome-java-2.10.1	5606	5135	17.4	45	0	0	1
libgtk-java-2.6.2	27095	32395	36.3	74	8	34	18
libgtkhtml-java-2.6.0	455	729	2.9	27	0	0	0
libgtkmozembed-java-1.7.0	166	498	3.3	0	0	0	0
libvte-java-0.11.11	437	184	2.5	0	26	0	0
jnetfilter	1113	1599	17.3	9	0	0	0
libreadline-java-0.8.0	1459	324	2.2	0	0	0	1
pgpjava	10136	123	2.7	0	1	0	1
posix1.0	978	293	1.8	0	1	0	0
Total				155	36	44	22

Fig. 5. Experimental Results

program, Fig. 5 lists the number of lines of C and Java code, the analysis time in seconds (average of 3 runs), and the number of messages reported by our tool, divided manually into four categories as described below. The running time includes the C code analysis (including extracting Java types from class files) but not the parsing of C code. The measurements were performed on a 733 MHz Pentium III machine with 1GB of RAM.

Our tool reported 155 errors, which are programming mistakes that may cause a program to crash or to emit an unexpected exception. Surprisingly, the most common error was declaring a C function with the wrong arity, which accounted for 68 errors (30 in libgtk and 38 in libgnome). All C functions called from Java must start with one parameter for the JNI environment and a second parameter for the invoking object or class. In many cases the second parameter was omitted in the call, and hence any subsequent arguments would be retrieved from the wrong stack location, which could easily cause a program crash.

56 of the errors were due to mistakes made during a software rewrite. Programs that use the JNI typically use one of two techniques to pass C pointers (e.g., GUI window objects) through Java: they either pass the pointer directly as an integer, or they embed the pointer as an integer field inside a Java object. Several of the libraries in the Java-Gnome project appear to be switching from the integer technique to the object technique, which requires changing Java declarations in parallel with C declarations, an error-prone process. Our tool detected many cases when a Java native method specified an `Object` parameter but the corresponding C function specified an integer parameter, or vice-versa. This accounted for 4 errors in libgnome, 25 in libgtk, and 27 in libgtkhtml.

Type mismatches accounted for 17 of the remaining errors. 6 errors occurred because a native Java method was declared with a `String` argument, but the C code took a byte array argument. In general Java strings must be translated to C strings using special JNI functions, and hence this is a type error. Another type error occurred because one C function passed a (non-array) Java object to

another C function expecting a Java array. Since both of these are represented with the type `jobject` in C, the C compiler did not catch this error.

Finally, 14 errors were due to incorrect namings. 11 of these errors (9 in `jnetfilter` and 2 in `libgtk`) were caused by calls to `FindClass` with an incorrect string. Ironically, all 9 `jnetfilter` errors occurred in code that was supposed to construct a Java exception to throw—but since the string did not properly identify the exception class, the JVM would throw a `ClassNotFoundException` exception instead. The remaining 3 errors were due to giving incorrect names to C functions corresponding to Java native methods; such functions must be given long names following a particular naming scheme, and it is easy to get this wrong.

We also ran our tool on a development Java 1.6 compiler, code-named Mustang. Our tool produced 480 messages, one of which was an actual error in which the JNI glue code did not properly distinguish between the types `int` and `long`. If used on a big-endian 64-bit machine, the C function would access only the higher 32 bits of the value, creating a runtime error [10]. The remaining messages were all false positives or imprecision messages. This program is not present in Figure 5 because we had to play special tricks to use our tool on the source code due to its intricate bootstrapping build process and were therefore unable to calculate an accurate running time.

Most of the errors we found are easy to trigger with a small amount of code. In cases such as incorrectly-named function, errors would likely be immediately apparent as soon as the native method is called. Thus clearly many of the errors are in code that has not been tested very much, most likely the parts of libraries that have not yet been used by Java programmers.

Our tool also produced 36 warnings, which are suspicious programming practices that do not actually cause run-time errors. One warning arose when a programmer called the function `FindClass` with a field descriptor of the form `Ljava/lang/String`; rather than a fully qualified class name `java/lang/String`. Technically this is an error [5], but the Sun JVM we tested allows both versions, so we only consider this a warning. Another example that accounted for 2 warnings was due to incorrectly declaring a function to return the wrong type, but then returning the correct type in the function body.

Finally, 33 warnings were due to the declaration of C functions that appear to implement a specific native method (because they have mangled names), but do not correspond to any native Java method. In many cases there was a native method in the Java code, but it had been commented out or moved without deleting the C code. This will not cause any run-time errors, but it seems helpful to notify the programmer about this dead code.

Our tool also produced 44 false positives, which are warnings about correct code, and 22 imprecision warnings, which occurred when the analysis had insufficient information about the value of a string. All of the false positives were caused by the use of subtyping inside of C code, which our analysis does not model precisely because it uses unification. 16 of the warnings were due to unification failures with partially specified methods that could not be resolved.

The other 6 warnings occurred when the programmer called a Java method by passing arguments via an array, which our analysis does not model.

5 Related Work

In prior work, we presented a system for inferring types for programs that use the OCaml-to-C FFI [4]. Our new work on the JNI differs in several ways. To infer types for the JNI, we need to track string values through the source code, whereas for the OCaml FFI we mainly needed to track integers and pointer offsets. Our new system can directly assign polymorphic types to JNI functions and to user-written wrapper functions, in contrast to our previous system which did not track sufficient interprocedural information to allow this and was not polymorphic. Our new system also includes support for indexing into records using strings for field names and method signatures, and those strings may not be known until constraint resolution. Our previous system did not support objects.

Recently several researchers have developed sophisticated techniques to track strings in programs [11, 12, 13, 14]. One goal of these systems is to check that dynamically-generated SQL queries are well-formed by creating a language which describes all possible strings for a given expression. For purposes of checking clients of the JNI, we found that simple tracking of strings is sufficient.

Nishimura [15] presents an object calculus which can statically infer kinded-types for first-class method names. Their system has similar restrictions to ours like not supporting inheritance or overloaded methods. Our work differs in that we are typing C code and must analyze the value of C strings instead of working with a pure object calculus.

There are many different foreign function interfaces with various design trade-offs [16, 17, 18, 19, 20, 5]. We believe that the techniques we develop in this paper and in our previous work [4] can be adapted to many FFIs.

An alternative to using FFIs directly is to use automatic interface generators to produce glue code. SWIG [21] generates glue code based on an interface specification file. Exu [22] provides programmers with a light-weight system for automatically generating JNI-to-C++ glue code for the common cases. Mockingbird [23] is a system for automatically finding matchings between two types written in different languages and generating the appropriate glue code. Our benchmark suite contained custom glue code that was generated by hand.

In addition to the JNI, there has been significant work on other approaches to object-oriented language interoperation, such as the commercial solutions COM [24], SOM [25] and CORBA [26]. Barret [27] proposes the PolySPIN system as an alternative to CORBA. All of these systems check for errors mainly at run-time (though in some cases interface generators can be used to provide some compile-time checking). The Microsoft common-language runtime (CLR) [28, 29] provides interoperation by being the target of compilers for multiple different languages, and the CLR includes a strong static type system. However, the type system only checks CLR code, and not unmanaged code that it may invoke.

Grechanik et al [30] present a framework called ROOF that allows many different languages to interact using a common syntax. This system includes both run-time and static type checking for code that uses ROOF. It is unclear whether ROOF supports polymorphism and whether it can infer types for glue code in isolation.

6 Conclusion

We have presented a multi-lingual type inference system for checking that programs use the JNI safely. Our system tracks the values of C strings to determine what names and type descriptors are passed to JNI functions. Thus we are able to infer what type assumptions C glue code makes about Java objects and check whether they are consistent with the actual Java class definitions. Our system treats wrapper functions and JNI functions polymorphically, allowing them to be parametric even in string arguments. Using an implementation of our system, we found many errors and suspicious practices in our suite of benchmarks. Our results suggest that our static checking system can be an important part of ensuring that the JNI is used correctly, and we believe that the same ideas can be applied to other object-oriented FFIs.

References

1. Rehof, J., Fähndrich, M.: Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In: *Proceedings of the 28th Annual ACM Symposium on Principles of Programming Languages*, London, United Kingdom (2001)
2. Fähndrich, M., Rehof, J., Das, M.: Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. In: *Proceedings of the 2000 ACM Conference on Programming Language Design and Implementation*, Vancouver B.C., Canada (2000)
3. Henglein, F.: Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems* **15** (1993) 253–289
4. Furr, M., Foster, J.S.: Checking Type Safety of Foreign Function Calls. In: *Proceedings of the 2005 ACM Conference on Programming Language Design and Implementation*, Chicago, Illinois (2005) 62–72
5. Liang, S.: *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley (1999)
6. Furr, M., Foster, J.S.: Polymorphic Type Inference for the JNI. Technical Report CS-TR-4759, University of Maryland, Computer Science Department (2005)
7. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: *Compiler Construction, 11th International Conference*, Grenoble, France (2002)
8. Cannasse, N.: (Ocaml javalib) <http://team.motion-twin.com/ncannasse/javaLib/>.
9. Java-Gnome Developers: (Java bindings for the gnome and gtk libraries) <http://java-gnome.sourceforge.net>.
10. Furr, M., Foster, J.S.: Java SE 6 "Mustang" Bug 6362203 (2005) http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6362203.

11. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Static Analysis, 10th International Symposium, San Diego, CA, USA (2003)
12. DeLine, R., Fähndrich, M.: The Fugue Protocol Checker: Is your software Baroque? Technical Report MSR-TR-2004-07, Microsoft Research (2004)
13. Gould, C., Su, Z., Devanbu, P.: Static Checking of Dynamically Generated Queries in Database Applications. In: Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, UK (2004) 645–654
14. Thiemann, P.: Grammar-Based Analysis of String Expressions. In: Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Language Design and Implementation, Long Beach, CA, USA (2005)
15. Ernst, S.N.: Static Typing for Dynamic Messages. In: Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages, San Diego, California (1998)
16. Blume, M.: No-Longer-Foreign: Teaching an ML compiler to speak C “natively”. In: BABEL’01: First International Workshop on Multi-Language Infrastructure and Interoperability, Firenze, Italy (2001)
17. Finne, S., Leijen, D., Meijer, E., Jones, S.P.: Calling hell from heaven and heaven from hell. In: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, Paris, France (1999) 114–125
18. Fisher, K., Pucella, R., Reppy, J.: A framework for interoperability. In: BABEL’01: First International Workshop on Multi-Language Infrastructure and Interoperability, Firenze, Italy (2001)
19. Huelsbergen, L.: A Portable C Interface for Standard ML of New Jersey. <http://www.smlnj.org/doc/SMLNJ-C/smlnj-c.ps> (1996)
20. Leroy, X.: The Objective Caml system (2004) Release 3.08, <http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf>.
21. Beazley, D.M.: SWIG: An easy to use tool for integrating scripting languages with C and C++. In: USENIX Fourth Annual Tcl/Tk Workshop. (1996)
22. Bubba, J.F., Kaplan, A., Wileden, J.C.: The Exu Approach to Safe, Transparent and Lightweight Interoperability. In: 25th International Computer Software and Applications Conference (COMPSAC 2001), Chicago, IL, USA (2001)
23. Auerbach, J., Barton, C., Chu-Carroll, M., Raghavachari, M.: Mockingbird: Flexible stub compilation from paris of declarations. In: Proceedings of the 19th International Conference on Distributed Computing Systems, Austin, TX, USA (1999)
24. Gray, D.N., Hotchkiss, J., LaForge, S., Shalit, A., Weinberg, T.: Modern Languages and Microsoft’s Component Object Model. *cacm* **41** (1998) 55–65
25. Hamilton, J.: Interlanguage Object Sharing with SOM. In: Proceedings of the Usenix 1996 Annual Technical Conference, San Diego, California (1996)
26. Object Management Group: Common Object Request Broker Architecture: Core Specification. Version 3.0.3 edn. (2004)
27. Barrett, D.J.: Polylingual Systems: An Approach to Seamless Interoperability. PhD thesis, University of Massachusetts Amherst (1998)
28. Hamilton, J.: Language Integration in the Common Language Runtime. *ACM SIGPLAN Notices* **38** (2003) 19–28
29. Meijer, E., Perry, N., van Yzendoorn, A.: Scripting .NET using Mondrian. In: ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary (2001)
30. Grechanik, M., Batory, D., Perry, D.E.: Design of large-scale polylingual systems. In: Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, UK (2004) 357–366

Type Safety of Generics for the .NET Common Language Runtime

Nicu G. Fruja

Computer Science Department, ETH Zürich, Switzerland
fruja@inf.ethz.ch

Abstract. The Microsoft .NET Common Language Runtime (CLR) offers support for generic types and methods. We develop a mathematical specification for the generics design through a type system and a model for the semantics of a subset of bytecode instructions with generics. We formalize the type-consistency checks performed for the subset by the CLR bytecode verifier. We then prove that adding support for generics maintains the type safety of the CLR.

1 Introduction

We have proved in [5] the soundness of the CLR bytecode verifier. The soundness proof takes advantage of the models for the CLR semantics [6, 7]. Kennedy and Syme proposed adding support for generics in the .NET CLR [11]. Their proposal was at the basis of the generics implementation in the .NET Framework v2.0 [1]. The official specification for the CLR generics support is given in prose form in the ECMA Standard [4].

Several versions of Eiffel turned out to be unsafe also due to the variance on generic parameters [3, 8]. Type holes [10] have been identified also in Generic Java [9, 12]. Consequently, the following question arises: *Is type safety preserved after adding generics as specified in the ECMA Standard [4]?*

So far, this question has only been addressed by Yu, Kennedy and Syme in [13]. They focus on aspects of the generics implementation, e.g., specialization of generic code up to data representation, efficient support for runtime types. As their goal was not the type safety, their formalization does not include: *variance on generic parameters*, *constraint types* and *boxed types* (critical for the specification of constraint types). These are exactly the generics features due to which the type safety might be violated.

In the context of generic parameter variance, virtual method calls are problematic in ensuring the type safety. Let us assume, for example, that the bytecode contains a virtual *call* of the method $C::M$. Let $D::M$ be the method that will be *invoked* at runtime. The following aspects are critical for the type safety: (1) due to the variance, the signature of $D::M$ does not necessarily match the signature of $C::M$; (2) if $C::M$ is a generic method, $D::M$ shall also be generic but its constraint types do not necessarily match the corresponding constraint types of $C::M$.

This paper also considers generic parameter variance, constraint types, boxed types, addresses the above aspects and answers positively through Theorem 1 to the above question. A by-product of our work is the identification of a few bugs and gaps in the ECMA Standard [4].

Notational Conventions. Beside the list operations *pop*, *top*, *length*, \cdot (the operation *append* for lists), we use two other operations: for a list L , $drop(L, n)$ returns the list resulting from dropping the last n elements from L and $split(L, n)$ splits off the last n elements of L , i.e., $split(L, n)$ is the pair (L', L'') where $L' \cdot L'' = L$ and $length(L'') = n$.

The rest of the paper is organized as follows. Section 2 gives a formalization of the polymorphic CLR type system. Section 3 provides a formal specification for the statical and operational semantics of a subset of bytecode instructions relevant for generics. Section 4 develops mathematical specifications for the type-consistency checks performed by the verifier and for the statically well-typed methods accepted by the verifier. Section 5 proves that the runtime execution of well-typed methods (with generics) does not corrupt the memory. Section 6 concludes.

2 Type System

This section defines a mathematical framework for the polymorphic type system of CLR. A *type* is a value type, a reference type or a generic parameter. A *value type* is either a value class (whose objects are composite values, i.e., values composed from other values) or a primitive type. The *reference types* are the object classes (whose objects are reference objects), the interfaces, the pointer types¹ and the boxed types. For every value type T , there exists a reference type $boxed(T)$ called *boxed type*. The value of a type $boxed(T)$ is a location where a value of type T can be stored. *Only* the verifier has knowledge of the boxed types. In the bytecode, a boxed type can only be referred to as **object** or as interfaces implemented by the associated value type.

$$\begin{aligned} Type &= Value Type \cup RefType \cup GenericParam \\ Value Type &= ValueClass \cup PrimitiveType \\ RefType &= ObjClass \cup Interface \cup PointerType \cup BoxedType \end{aligned}$$

Additionally, **void** can be used only as a method return type and **Null** (the type of **null**) is used only in the bytecode verification.

The methods are identified through *method references*, i.e., elements of the universe $MRef$. The references include the *signatures* consisting of the argument types and return type. We consider only instance (including virtual) methods.

A class or an interface whose declaration is parameterized by one or more types is called *generic type*. We denote by $GenericType$ the universe of generic

¹ As the pointer types have been studied in [5], their use is very limited in this paper: a pointer (evaluated to an address) can be loaded on the stack by an *Unbox* instruction and can be passed as the **this** pointer to a call through the instructions *CallVirt* and *Constrained.CallVirt*.

Table 1. Selector functions for generic types/methods

$genParamNo : Map(GenericType \cup MRef, \mathbb{N})$	number of generic parameters of a type/method
$genArg : Map(GenericType \cup MRef, List(Type \setminus PointerType))$	generic arguments of a type/method
$constr_i^T : Type \setminus (PointerType \cup GenericParam)$	i -th constraint of generic type T
$constr_i^{mref} : Type \setminus PointerType$	i -th constraint of generic method $mref$
$(var_i^{I'n})_{i=0}^{n-1} : List(\{+, -, \emptyset\})$	variances array of generic interface $I'n$

types: $GenericType \subseteq ValueClass \cup ObjClass \cup Interface$. Typically, $C'n$ gives the name of a generic type C with n type parameters. A method declared within a type, whose signature includes one or more generic parameters (not present in the type declaration itself) is called *generic method*. The class generic parameters are written in the bytecode as $!i$ whereas the method generic parameters are addressed as $!!i$. Thus, $!i$ denotes the i -th class generic parameter (numbered from left-to-right in the relevant class declaration). Similarly, $!!i$ designates the i -th method generic parameter (numbered from left-to-right in the relevant method declaration). $GenericParam$ denotes the universe of generic parameters.

Every generic parameter can have an optional constraint consisting of a type. This differs slightly from [4] which allows a constraint to have more than one type. The restriction does not reduce the complexity but simplifies the exposition. The generic types and methods can be *instantiated*² by replacing every generic parameter with a *generic argument*. Every generic argument shall be a subtype (*when boxed*) of the type given in the corresponding constraint (see Definition 1 for the subtype relation).

The generic interfaces can be covariant or contravariant in one or more of its generic parameters. A *covariant generic parameter* is marked with “+” in the interface declaration whereas “-” is used to denote a *contravariant generic parameter*³. For the sake of notation, we mark with “ \emptyset ” the non-variant parameters.

Table 1 gathers the selector functions which we define to deal with generics.

Definition 1 introduces the subtype relation. The relation is defined also for *open generic types*, i.e., generic types involving generic parameters. We use “ \circ ” to denote a *substitution*. Thus, $T \circ [U_i/X_i]_{i=0}^{n-1}$ is the type T where each generic parameter X_i is substituted by the type U_i .

Definition 1 (Subtype Relation). *The subtype relation \sqsubseteq is the least reflexive and transitive relation such that*

- if T_1 is a non-generic object class which extends / implements the class / interface T_2 , or
- if T_1 is $boxed(T)$ where T is a non-generic value class which extends / implements the class / interface T_2 , or

² *GenericType* has *only* instantiated generic types and *not* generic types’ raw names.

³ Examples of .NET languages that support contravariance are Java 5.0 (through “wildcards” with lower bounds), Sather, Scala, OCaml.

- if T_1 is $\text{boxed}(X)$ where X is a generic parameter constrained by T_2 , or
- if T_1 is **Null** and $T_2 \in \text{RefType}$, or
- if $T_1 \in \text{RefType}$ and $T_2 = \text{object}$, or
- if T_1 is $C\langle U_0, \dots, U_{n-1} \rangle$ where $C'n$ is a generic object class with the generic parameters $(X_i)_{i=0}^{n-1}$ which extends / implements the class / interface T and T_2 is given by $T \circ [U_i/X_i]_{i=0}^{n-1}$, or
- if T_1 is $\text{boxed}(C\langle U_0, \dots, U_{n-1} \rangle)$ where $C'n$ is a generic value class with the generic parameters $(X_i)_{i=0}^{n-1}$ which extends / implements the class / interface T and T_2 is given by $T \circ [U_i/X_i]_{i=0}^{n-1}$, or
- if T_1 is $T\langle U_0, \dots, U_{n-1} \rangle$ and T_2 is $T\langle V_0, \dots, V_{n-1} \rangle$ and for every $i = 0, n-1$ the following conditions hold:
 - if $\text{var}_i^{T'n} = \emptyset$ or $V_i \in \text{ValueType}$ or $V_i \in \text{GenericParam}$, then $U_i = V_i$;
 - if $\text{var}_i^{T'n} = +$, then $U_i \sqsubseteq V_i$;
 - if $\text{var}_i^{T'n} = -$, then $V_i \sqsubseteq U_i$;

then $T_1 \sqsubseteq T_2$.

To state Definition 2 and Definition 3, we need to define the *negation* $-(\text{var}_i)$ of a variances array (var_i) : $-\text{var}_i$ is $+$, if $\text{var}_i = -$, $-\text{var}_i$ is $-$, if $\text{var}_i = +$, and $-\text{var}_i$ is \emptyset , if $\text{var}_i = \emptyset$.

To enforce type safety, the ECMA Standard [4] imposes, unlike [8], several requirements on the instance methods declared by a generic interface which is co-/contra-variant in at least one generic parameter. These methods shall be valid according to Definition 3. However, that definition requires the notion of *valid type with respect to a variances array* which we specify in Definition 2.

Definition 2 (Valid Type). *The predicate **validType** checks the validity of a type T with respect to an array (var_i) of variances.*

$\text{validType}(T, (\text{var}_i)) : \Longleftrightarrow$

$T \notin \text{GenericType} \cap \text{GenericParam} \vee$

T is a generic parameter $!!j$ of the enclosing method \vee

T is a generic parameter $!j$ of the enclosing type $\wedge \text{var}_j \in \{+, \emptyset\} \vee$

$T = C\langle U_0, \dots, U_{n-1} \rangle \in \text{GenericType} \wedge$

$\forall k = 0, n-1$

$(\text{var}_k^{C'n} = + \implies \text{validType}(U_k, (\text{var}_i))) \wedge$

$(\text{var}_k^{C'n} = - \implies \text{validType}(U_k, -(\text{var}_i))) \wedge$

$(\text{var}_k^{C'n} = \emptyset \implies \text{validType}(U_k, (\text{var}_i)) \wedge \text{validType}(U_k, -(\text{var}_i)))$

Closed Generic Types Not Valid? The ECMA Standard [4, Partition II, §9.7] states that $T = C\langle U_0, \dots, U_{n-1} \rangle$ in the above definition shall refer to a “closed” generic type. This does not make a lot of sense, since in this case the definition has nothing to do with the array of variances. This remark and the experiments we have run with CLR indicate that T shall not necessarily be a closed type.

Definition 3 specifies when a method is valid with respect to a variances array. A method is valid if its return type “behaves covariantly” whereas its

argument types and possibly constraint types “behave contravariantly”. The functions *retType*, *argTypes* and *argNo* (introduced in Table 3) are used for a method to retrieve the return type, the list of argument types and its length, respectively. Note that the argument indexed with 0 gives the **this** pointer.

Definition 3 (Valid Method). *The predicate **validMeth** checks the validity of the method *mref* declaration with respect to the array (var_i) of variances.*

$$\begin{aligned} \text{validMeth}(mref, (var_i)) : \iff & \\ & \text{validType}(\text{retType}(mref), (var_i)) \wedge \\ & \forall j = 1, \text{argNo}(mref) - 1 \quad \text{validType}(\text{argTypes}(j), -(var_i)) \wedge \\ & \forall j = 0, \text{genParamNo}(mref) - 1 \quad \text{validType}(\text{constr}_j^{mref}, -(var_i)) \end{aligned}$$

The declaration of a generic interface is *valid* if all the instance methods declared by the interface and all the implemented interface types are valid with respect to the variances array of the given interface⁴.

Definition 4 (Valid Interface Declaration). *The predicate **validDecl** checks the validity of the generic interface $I'n$ declaration.*

$$\begin{aligned} \text{validDecl}(I'n) : \iff & \forall I'n::M \quad \text{validMeth}(I'n::M, (var_i^{I'n})) \wedge \\ & \forall J \text{ implemented by } I'n \quad \text{validType}(J, (var_i^{I'n})) \end{aligned}$$

The type safety proof in Section 5 takes advantage of the following lemmas. Lemma 1 shows that the subtype relationship of a given type varies *directly* with the relationship of the covariant generic parameters and *inversely* with the relationship of the contravariant generic parameters.

Lemma 1. *If the types T , $(U_i)_{i=0}^{n-1}$, $(V_i)_{i=0}^{n-1}$ and the array $(var_i)_{i=0}^{n-1}$ of variances are such that $\text{validType}(T, (var_i)_{i=0}^{n-1})$ and*

$$\forall i = 0, n-1 ((var_i = + \implies V_i \sqsubseteq U_i) \wedge (var_i = - \implies U_i \sqsubseteq V_i) \wedge (var_i = \emptyset \implies V_i = U_i))$$

$$\text{then } T \circ [V_i/!i]_{i=0}^{n-1} \sqsubseteq T \circ [U_i/!i]_{i=0}^{n-1}.$$

Proof. By induction on the structure of the (possibly generic) type T . Definition 2 is applied. \square

Lemma 2 proves that, if $T_1 \sqsubseteq T_2$, then the instantiation of the generic parameters (corresponding to an enclosing generic class and/or method) occurring in T_1 and T_2 with generic arguments satisfying the corresponding constraints preserves the subtype relation between T_1 and T_2 .

⁴ [4, Partition II, §9.7] is unclear. It requires that “every inherited interface declaration” shall be valid with respect to the variances array. Firstly, the interfaces are not “inherited” but implemented. Secondly, it is not about the interface “declaration” but about the interface type present in the **extends** clause of the given interface.

Lemma 2. Let T_1 and T_2 be two types such that $T_1 \sqsubseteq T_2$. Assume T_1 and T_2 occur in the declaration of $C'n$ possibly in the declaration of a generic method $C'n::M$. Let $(U_i)_{i=0}^{n-1}$ and $(V_j)_{j=0}^{m-1}$ be generic arguments for $C'n$ and $C'n::M$, respectively, assumed to satisfy the constraints:

$$\begin{aligned} \text{boxed}(U_i) &\sqsubseteq \text{constr}_i^{C'n} \circ [U_i/!i]_{i=0}^{n-1}, & \text{for every } i = 0, n-1 \\ \text{boxed}(V_j) &\sqsubseteq \text{constr}_j^{C'n::M} \circ ([U_i/!i]_{i=0}^{n-1} \cdot [V_j/!!j]_{j=0}^{m-1}), & \text{for every } j = 0, m-1 \end{aligned}$$

It then holds $T_1 \circ [U_i/!i]_{i=0}^{n-1} \cdot [V_j/!!j]_{j=0}^{m-1} \sqsubseteq T_2 \circ [U_i/!i]_{i=0}^{n-1} \cdot [V_j/!!j]_{j=0}^{m-1}$.

Proof. By induction on the structure of the (possibly generic) type T_1 . Definition 1 is applied. \square

Since it is possible for different methods to have identical signatures when the declaring types are instantiated, the method references have the signatures uninstantiated. Unlike the signatures, the type constraints are instantiated in our approach unless explicitly stated otherwise. To get the instantiated return type and argument types of a method reference, we define *inst* as the substitution that shall be applied to the reference:

$$\begin{aligned} \text{inst}(C::M) &:= \\ &[\text{genArg}(C)(i)/!i]_{i=0}^{\text{genParamNo}(C)-1} \cdot [\text{genArg}(C::M)(i)/!!i]_{i=0}^{\text{genParamNo}(C::M)-1} \end{aligned}$$

3 Bytecode Semantics

In this section we formally define the semantics of the instructions from Table 2 by means of a small-step operational semantics modeled in ASM⁵ syntax in Table 5. The static semantics is given in terms of the functions defined in Table 3 while the dynamic state of the considered bytecode language is described by the functions introduced in Table 4.

The reasons for considering only the instructions from Table 2 are the following. Adding generic types increases the complexity of \sqsubseteq on which *CastClass* and *IsInstance* strongly depend. To give a flavor of the boxed types (possibly involving generic parameters) critical for handling generic arguments, we consider also the instructions *Box*, *Unbox* and *Unbox.Any*. To accommodate method calls on generic parameter values, we analyze also *Constrained.CallVirt*. The most critical feature for type safety is the generic parameter variance. As this aspect is reflected in virtual method calls, *CallVirt* and *Return* are also considered. The other CLR instructions are left out since they do not pose any problems in ensuring type safety of the generics features.

We briefly describe the instruction semantics defined in Table 5. Every instruction is executed under the assumption that the current method *meth* is instantiated, i.e., every generic parameter is replaced by the corresponding generic argument. Consequently, *inst(meth)* is applied to every instruction. Every time

⁵ The definition of ASMs is skipped here, because ASMs can be correctly understood as pseudo-code operating over abstract (domains of) data. See their definition in [2].

Table 2. CLR instructions considered

$Instr =$ *CastClass*(*ObjClass* \cup *Interface* \cup *ValueClass*)
 | *IsInstance*(*ObjClass* \cup *Interface* \cup *ValueClass*)
 | *Box*(*Type*)
 | *Unbox*(*ValueType*)
 | *Unbox.Any*(*Type*)
 | *CallVirt*(*Type*, *MRef*)
 | *Constrained*(*GenericParam*).*CallVirt*(*Type*, *MRef*)
 | *Return*

Table 3. Static functions

<i>code</i>	: <i>Map</i> (<i>MRef</i> , <i>List</i> (<i>Instr</i>))	list of instructions of a method body
<i>argTypes</i>	: <i>Map</i> (<i>MRef</i> , <i>List</i> (<i>Type</i>))	argument types of a method
<i>argNo</i>	: <i>Map</i> (<i>MRef</i> , \mathbb{N})	number of arguments of a method
<i>retType</i>	: <i>Map</i> (<i>MRef</i> , <i>Type</i>)	return type of a method
<i>lookup</i>	: <i>Map</i> (<i>Type</i> \times <i>MRef</i> , <i>MRef</i>)	<i>dynamic binding</i> function

Table 4. Dynamic functions

<i>memVal</i>	: <i>Map</i> (<i>Address</i> \times <i>Type</i> , <i>Value</i>)	memory function
<i>meth</i>	: <i>MRef</i>	current method
<i>pc</i>	: <i>Pc</i>	current program counter of <i>meth</i>
<i>argVal</i>	: <i>Map</i> (\mathbb{N} , <i>Value</i>)	argument values of <i>meth</i>
<i>evalStack</i>	: <i>List</i> (<i>Value</i>)	current evaluation stack of <i>meth</i>
<i>actualTypeOf</i>	: <i>Map</i> (<i>ObjRef</i> , <i>Type</i>)	runtime type of an object reference
<i>addressOf</i>	: <i>Map</i> (<i>ObjRef</i> , <i>Address</i>)	address of value type in a boxed object

an exception occurs, control is passed to the exception handling mechanism defined in [7] which preserves type safety as proved in [5]. Since we do not consider exceptions here, we do not model this control switching either.

The instruction *CastClass*(*C*) checks if the topmost value of the *evalStack* is of type *C*. If not, an exception is thrown. The instruction *IsInstance*(*C*) pops from the *evalStack* a reference to an (possibly boxed) object. If the object is not an instance of *C*, **null** is pushed on the *evalStack*. The *Box*(*T*) instruction (where *T* can also be a generic parameter) turns a boxable value into its boxed form. Applied to a value type, the instruction loads a boxed object created through the macro *NewBox* defined below on the *evalStack*.

let $r = \text{NewBox}(val, T)$ **in** $P \equiv$ **let** $r = \text{new}(\text{ObjRef})$ **and** $adr = \text{new}(\text{Address}, T)$ **in**
 $\text{WRITEMEM}(adr, T, val)$
 $\text{addressOf}(r) := adr$
 $\text{actualTypeOf}(r) := T$
seq P

Table 5. Execution of the bytecode instructions

match $code(meth)(pc)$	
$CastClass(C) \circ inst(meth) \rightarrow$	
let $r = top(evalStack)$ in	
if $r = \text{null} \vee actualTypeOf(r) \sqsubseteq C \circ inst(meth)$ then	
$pc := pc + 1$	
$IsInstance(C) \circ inst(meth) \rightarrow$	
let $(evalStack', [r]) = split(evalStack, 1)$ in	
$pc := pc + 1$	
if $actualTypeOf(r) \not\sqsubseteq C \circ inst(meth)$ then	
$evalStack := evalStack' \cdot [\text{null}]$	
$Box(T) \circ inst(meth) \rightarrow$	
$pc := pc + 1$	
if $T \circ inst(meth) \in ValueType$ then	
let $(evalStack', [val]) = split(evalStack, 1)$ in	
let $r = NewBox(val, T \circ inst(meth))$ in	
$evalStack := evalStack' \cdot [r]$	
$Unbox(T) \circ inst(meth) \rightarrow$	
let $(evalStack', [r]) = split(evalStack, 1)$ in	
if $r \neq \text{null} \wedge actualTypeOf(r) = T \circ inst(meth)$ then	
$evalStack := evalStack' \cdot [addressOf(r)]$	
$pc := pc + 1$	
$Unbox.Any(T) \circ inst(meth) \rightarrow$	
let $(evalStack', [r]) = split(evalStack, 1)$ in	
if $T \circ inst(meth) \in ValueType$ then	
$evalStack := evalStack' \cdot [memVal(addressOf(r), T \circ inst(meth))]$	
$pc := pc + 1$	
elseif $r = \text{null} \vee actualTypeOf(r) \sqsubseteq T \circ inst(meth)$ then $pc := pc + 1$	
$CallVirt(_, C::M) \circ inst(meth) \rightarrow$	
let $(evalStack', [r] \cdot vals) = split(evalStack, argNo(C::M))$ in	
$evalStack := evalStack'$	
$VIRTCALL(r, C::M \circ inst(meth), vals)$	
$Constrained(T).$	
$CallVirt(_, C::M) \circ inst(meth) \rightarrow$	
let $(evalStack', [adr] \cdot vals) = split(evalStack, argNo(C::M))$ in	
$evalStack := evalStack'$	
if $T \circ inst(meth) \in RefType$ then	
let $r = memVal(adr, T \circ inst(meth))$ in	
$VIRTCALL(r, C::M \circ inst(meth), vals)$	
elseif $T \circ inst(meth) \in ValueClass \wedge$	
$T::M \circ inst(meth)$ implements $C::M \circ inst(meth)$ then	
$INVOKE(T::M \circ inst(meth), [adr] \cdot vals)$	
else let $r = NewBox(memVal(adr, T \circ inst(meth)), T \circ inst(meth))$ in	
$VIRTCALL(r, C::M \circ inst(meth), vals)$	
$Return \circ inst(meth) \rightarrow$	
if $retType(meth) = \text{void}$ then $RESULT([])$	
else $RESULT([top(evalStack)])$	

$NewBox(val, T)$ creates a fresh object reference and allocates an address where val of type T is stored through `WRITEMEM`. The definition of `WRITEMEM` (which can be found in [6]) is beyond the scope of this paper.

The *Unbox* instruction takes a reference to a boxed object from the *evalStack* and loads the address of the value embedded into the boxed object. An exception is thrown if the object is not a boxed object or the value type of the value in the box does not match the instantiation of the type operand of the instruction. Unlike *Unbox*, for value types, the *Unbox.Any* instruction leaves the value, not the address of the value, on the *evalStack*. Moreover, the type embedded in *Unbox* can only represent value types and instantiations of generic value types. The function *memVal* (whose definition is given in [6]) is used in *Unbox.Any* to compute the value of a given type stored at a given address. For reference types, *Unbox.Any* has the same effect as *CastClass*.

To specify the virtual method calls implied by the instructions *CallVirt* and *Constrained.CallVirt*, we need to define the *lookup* function.

Definition 5 (Lookup). *Given a type and a (possibly generic) method reference, the function $lookup : Map(Type \times MRef, MRef)$ determines the method to be invoked at runtime when the given method is called on an object whose runtime type is the given type.*

$$\begin{aligned}
 &lookup(C, D::M\langle T_0, \dots, T_{n-1} \rangle) := \\
 &\quad \text{if } C::M \in MRef \wedge \\
 &\quad \quad (D \in ObjClass \implies C::M \text{ overrides } D::M) \wedge \\
 &\quad \quad (D \in Interface \setminus GenericType \implies C::M \text{ implements } D::M) \wedge \\
 &\quad \quad (D = I\langle U_0, \dots, U_{m-1} \rangle \in Interface \cap GenericType \implies \\
 &\quad \quad \quad C::M \text{ implements } I\langle V_0, \dots, V_{m-1} \rangle::M \wedge \\
 &\quad \quad \quad \forall j = 0, m-1 ((var_j^{I'} m = + \implies V_j \sqsubseteq U_j) \wedge \\
 &\quad \quad \quad \quad (var_j^{I'} m = - \implies U_j \sqsubseteq V_j) \wedge \\
 &\quad \quad \quad \quad (var_j^{I'} m = \emptyset \implies U_j = V_j))) \\
 &\quad \text{then } C::M\langle T_0, \dots, T_{n-1} \rangle \\
 &\quad \text{elseif } C = \text{object} \text{ then undef} \\
 &\quad \text{else } lookup(C', D::M\langle T_0, \dots, T_{n-1} \rangle) \text{ where } C' \text{ is the direct base class of } C
 \end{aligned}$$

The ECMA Standard [4] does not specify what is the effect of adding generic parameter variance on the dynamical method lookup. As one can see in Definition 5, the definition of *lookup* becomes more complex: $lookup(-, D::M)$ shall not necessarily be a method which overrides or implements $D::M$.

The instruction $CallVirt(T, C::M)$ calls the virtual method $C::M$ whose (possibly open generic) return type is T . It pops the necessary number of arguments from the *evalStack*. Based on the type of the **this** pointer, it looks up the method to be invoked (through the `INVOKE` macro defined in Table 6) with the popped arguments dynamically by means of the *lookup* function. *lookup* is applied to $C::M \circ inst(meth)$, i.e., the method $C::M$ where *only* the generic parameters present in C or possibly in the generic argument list of M are replaced by the generic arguments indicated in *inst(meth)*.

$$\begin{aligned}
 &VirtCall(r, C::M, vals) \equiv \text{let } D::M = lookup(actualTypeOf(r), C::M) \text{ in} \\
 &\quad \text{if } r \neq \text{null} \text{ then } INVOKE(D::M, [r] \cdot vals)
 \end{aligned}$$

Table 6. Invoking a method and returning from a method

INVOKE($C::M, args$) \equiv	RESULT($vals$) \equiv
PUSHFRAME	POPFRAME
seq	seq
$pc := 0$	$pc := pc + 1$
$evalStack := []$	$evalStack := evalStack \cdot vals$
$meth := C::M$	
SETARG($C::M, args$)	

The instruction $Constrained(T).CallVirt(S, C::M)$ calls the virtual method $C::M$ (whose return type is S) on a value of a generic parameter T . It pops the necessary number of arguments from the $evalStack$. The first value is expected to be a pointer (evaluated to an address) adr . If T is a reference type, then adr is dereferenced and passed as the **this** pointer to $VIRTCALL$. If T is a value type and T implements $C::M$, then adr is passed as the **this** pointer to the method implemented by T which is then called with INVOKE. If T is a value type which does not implement $C::M$, then adr is dereferenced, boxed, and passed as the **this** pointer to a virtual call of $C::M$. Normally, the above outlined transformation of the **this** pointer is performed at compile time, depending on the type of adr and the method being called. However, such a transformation would not be possible when the type of the **this** pointer is a generic type (unknown at compile time). Thus, the prefix *Constrained* allows .NET compilers to make a call to a virtual function in an *uniform way* independent of whether the **this** pointer is a value type or reference type.

The *Return* instruction returns from the current method $meth$ by means of the RESULT macro which we define in Table 6. If the return type of $meth$ is not **void**, $evalStack$ shall contain a value to be returned through RESULT.

The macros PUSHFRAME and POPFRAME in Table 6 are used to push a new frame and to pop the current frame, respectively. The macro SET($C::M, args$) sets the arguments of $C::M$, i.e., the $argVal$ function, to the values $args$.

As stated at the end of Section 2, the method references have the signatures uninstantiated. Therefore, for example, the return type (specified in the signature) of a method which overrides/implements another method shall not necessarily match the return type (specified in the signature) of the overridden/implemented method. The conditions that shall actually be satisfied when a generic method $C::M$ overrides/implements $D::M$ are listed below:

(at) for every $i = 1, argNo(C::M) - 1$,

$$argTypes(C::M)(i) \circ inst(C::M) = argTypes(D::M)(i) \circ inst(D::M)$$

(rt) $retType(C::M) \circ inst(C::M) = retType(D::M) \circ inst(D::M)$

(ct) for every $j = 0, genParamNo(D::M) - 1$,

$$constr_j^{C::M} \text{ is not defined or } constr_j^{D::M} \sqsubseteq constr_j^{C::M}$$

No More Restrictive? Concerning (**ct**), [4, Partition II,§9.9] states that any constraint type specified by the overriding method shall be “no more restrictive” than the corresponding constraint type specified in the overridden method. However, this does not match the Microsoft implementation [1]. It seems that the Microsoft verifier checks whether one of the following conditions is satisfied: either the constraint type in the overriding method is not defined, i.e., as it would have been **object**, or the constraint types (assumed to be instantiated) coincide.

4 Bytecode Verification

The bytecode verification is performed on a *per-method* basis. The verifier simulates the bytecode execution. It attempts to associate a *stack state* $evalStackT$ with every instruction. The stack state is a list of types which specifies the number of values on the *evalStack* at that point in the code and for each slot of the *evalStack* a required type that shall be present in that slot. Before simulating the execution of an instruction, the verifier performs several type-consistency checks specified by means of the predicate *check* defined in Table 7. Its definition follows the specification of the ECMA Standard [4, Partition III]. The stack state of an instruction is constrained by referring to the stack states of the next instruction. Table 8 defines the function *succ* which, given an instruction and a stack state, computes the stack states of the next instruction.

To deal with stack states, we introduce the relations \sqsubseteq_{suf} and \sqsubseteq_{len} . If L' and L'' are two lists of types of lengths m and n , respectively, then

$$L' \sqsubseteq_{suf} L'' :\iff m \geq n \text{ and } L'(m - n + i) \sqsubseteq L''(i) \text{ for every } i = 0, n - 1.$$

$$L' \sqsubseteq_{len} L'' :\iff m = n \text{ and } L'(i) \sqsubseteq L''(i) \text{ for every } i = 0, m - 1.$$

To shorten the specification of *check* and *succ*, we use the following notation:

$$void(T) := \text{if } T = \text{void then } [] \text{ else } [T]$$

Table 7. Type-consistency checks performed by the verifier

$check(meth, pos, evalStackT) :\iff$	
match $code(meth)(pos)$	
$CastClass(_)$	$\rightarrow evalStackT \sqsubseteq_{suf} [\text{object}]$
$IsInstance(_)$	$\rightarrow evalStackT \sqsubseteq_{suf} [\text{object}]$
$Box(T)$	$\rightarrow evalStackT \sqsubseteq_{suf} [T]$
$Unbox(_)$	$\rightarrow evalStackT \sqsubseteq_{suf} [\text{object}]$
$Unbox.Any(_)$	$\rightarrow evalStackT \sqsubseteq_{suf} [\text{object}]$
$CallVirt(_, C::M)$	$\rightarrow evalStackT \sqsubseteq_{suf} argTypes(C::M) \circ inst(C::M)$
$Constrained(T)$	
$CallVirt(_, C::M)$	$\rightarrow boxed(T) \sqsubseteq C \wedge evalStackT \sqsubseteq_{suf}$ $[T\&] \cdot [argTypes(C::M)(i)]_{i=1}^{argNo(C::M)-1} \circ inst(C::M)$
$Return$	$\rightarrow evalStackT \sqsubseteq_{len} void(retType(meth))$

Table 8. Determining successor stack states
$$\begin{array}{ll}
succ(meth, pos, evalStackT) := & \\
\text{match } code(meth)(pos) & \\
\text{CastClass}(C) & \rightarrow \{pop(evalStackT) \cdot [C]\} \\
\text{IsInstance}(C) & \rightarrow \{pop(evalStackT) \cdot [C], pop(evalStackT) \cdot [Null]\} \\
\text{Box}(T) & \rightarrow \text{if } T \in \text{RefType} \text{ then } \{pop(evalStackT) \cdot [T]\} \\
& \quad \text{else } \{pop(evalStackT) \cdot [boxed(T)]\} \\
\text{Unbox}(T) & \rightarrow \{pop(evalStackT) \cdot [T\&]\} \\
\text{Unbox.Any}(T) & \rightarrow \{pop(evalStackT) \cdot [T]\} \\
\text{CallVirt}(T, C::M) & \rightarrow \{drop(evalStackT, argNo(C::M)) \cdot void(T \circ inst(C::M))\} \\
\text{Constrained}(_). & \\
\text{CallVirt}(T, C::M) & \rightarrow \{drop(evalStackT, argNo(C::M)) \cdot void(T \circ inst(C::M))\} \\
\text{Return} & \rightarrow \emptyset
\end{array}$$

Since we have no definition of a particular bytecode verifier, we need a characterization of the type properties of the bytecode that is accepted by the verifier. This leads us to Definition 6. A method is *well-typed* if the verifier succeeds to compute a valid stack state for every instruction of the method. Definition 6 makes this precise: a method is well-typed if there exists a stack state family that satisfies an initial condition, the type-consistency checks and the relations dictated by a top-down pass through the bytecode and by the rules for merging stack states specified in [4, Partition III, §1.8.1.3].

Definition 6 (Well-typed Method). *A method $mref$ is well-typed if there exists a family $(evalStackT_i)_i$ of stack states satisfying the following conditions:*

- (wt1) $evalStackT_0 = []$.
- (wt2) $check(mref, pos, evalStackT_{pos})$ holds for every position pos in $mref$.
- (wt3) If $evalStackT' \in succ(mref, pos, evalStackT_{pos})$, then $evalStackT' \sqsubseteq_{len} evalStackT_{pos+1}$.

5 Type Safety

We prove that the bytecode with generics is type safe. As the bytecode verifier statically checks the type safety of the bytecode, we need to show that the verifier is sound. That means that if the verifier succeeds to compute a valid stack state for every instruction of a method, i.e., the method is well-typed according to Definition 6, then several type safety properties are ensured to hold at runtime. For example, the *evalStack* shall have at runtime values of the types assigned in the stack state and the same length as the stack state. Furthermore, the generic arguments of instantiated generic types and methods shall satisfy the corresponding constraints.

For this section, we assume the following: (1) every method is well-typed; (2) every generic interface declaration is valid; (3) the generic arguments of all

the generic types and method references satisfy at verification time the corresponding constraints. In particular, this means that every generic argument used in the *inst* functions satisfies the corresponding constraints.

The following two lemmas establish relations between the argument types, respectively the return type of the method called at compile time, i.e., the method embedded in a *CallVirt*, and the argument types, respectively the return type of the method that is determined through a lookup and is invoked at runtime.

Lemma 3. *If $D::M = \text{lookup}(T, C::M)$, then $T \sqsubseteq \text{argTypes}(D::M)(0)$ and for every $i = 1, \text{argNo}(C::M) - 1$*

$$\text{argTypes}(C::M)(i) \circ \text{inst}(C::M) \sqsubseteq \text{argTypes}(D::M)(i) \circ \text{inst}(D::M)$$

Proof. By Definition 3, Definition 4, Definition 5, Lemma 1, Lemma 2, (**at**). \square

Lemma 4. *If $D::M = \text{lookup}(T, C::M)$ holds for some type T , then*

$$\text{retType}(D::M) \circ \text{inst}(D::M) \sqsubseteq \text{retType}(C::M) \circ \text{inst}(C::M)$$

Proof. By Definition 3, Definition 4, Definition 5, Lemma 1, Lemma 2, (**rt**). \square

Lemma 5 shows that a generic method determined through a lookup has the same generic arguments as the original method and the generic arguments satisfy the corresponding constraints.

Lemma 5. *If $D::M$ and $C::M$ are generic methods and T is a type such that $D::M = \text{lookup}(T, C::M)$ and $\text{boxed}(\text{genArg}(C::M)(i)) \sqsubseteq \text{constr}_i^{C::M}$ for every $i = 0, \text{genParamNo}(C::M) - 1$, then $\text{genParamNo}(D::M) = \text{genParamNo}(C::M)$ and for every $i = 0, \text{genParamNo}(C::M) - 1$*

$$\text{boxed}(\text{genArg}(C::M)(i)) = \text{boxed}(\text{genArg}(D::M)(i)) \sqsubseteq \text{constr}_i^{D::M}$$

Proof. By Definition 3, Definition 4, Definition 5, Lemma 1, Lemma 2, (**ct**). \square

We assume that the generic arguments of a (instantiated) generic type or generic method satisfy at verification time the corresponding constraints. As the generic arguments might be open generic types, the following question appears: *Do they satisfy the constraints also after the runtime instantiation?* The following proposition answers positively to this question.

Proposition 1 (Preserving Constraints). *The instantiated (not necessarily closed) type $C\langle T_0, \dots, T_{n-1} \rangle$ occurs in the declaration of a generic type $D'm$ possibly in the declaration of a generic method $D'm::M$. Assume that $(T_i)_{i=0}^{n-1}$ satisfy the constraints of $C'n$. If $D'm$ and $D'm::M$ are instantiated at runtime with the generic arguments $(U_j)_{j=0}^{m-1}$ and $(V_k)_{k=0}^{p-1}$ (which are assumed to satisfy the constraints of $D'm$ and $D'm::M$, respectively), then $(T_i \circ \rho)_{i=0}^{n-1}$ satisfy the constraints of $C'n$ where ρ is the substitution $\rho = [U_j / !j]_{j=0}^{m-1} \cdot [V_k / !!k]_{k=0}^{p-1}$ defined in the context of $D'm$ and $D'm::M$ declarations. A similar result holds also for a referenced generic method, i.e., $C::M\langle T_0, \dots, T_{n-1} \rangle$ instead of $C\langle T_0, \dots, T_{n-1} \rangle$.*

Proof. By Lemma 2. \square

The typing judgment $\vdash val : T$ is defined as follows. Thus, $\vdash val : T$ holds if at least one of the following holds: (1) $T \in RefType \setminus (PointerType \cup BoxedType)$ and either val is **null** or $actualTypeOf(val) \sqsubseteq T$; (2) $T = S\&$ and $\exists r \in ObjRef$ such that $addressOf(r) = val$ and $actualTypeOf(r) = S$; (3) $boxed(S) \sqsubseteq T$ where $S \in ValueType$ and $actualTypeOf(val) = S$. (4) the CLR type associated (see [4, Partition III, §1.1]) to val is a value type which is a subtype of T .

We now extend the soundness proof done in [5] for the CLR without generics to the CLR with generics. The theorem proved in [5] guarantees that several type-safety invariants hold at runtime for well-typed methods without generics. We consider here only the invariants which are affected upon adding generics. Additionally, an invariant (**constr**) for generic methods is considered. The invariant (**stack1**) guarantees that $evalStack$ has the same length as the assigned stack state. The invariant (**stack2**) ensures that the values on the $evalStack$ are of the types assigned in the stack state. By (**arg**), we have that the arguments contain values of the declared types. The invariant (**constr**) ensures that the generic arguments of a generic method satisfy the declared constraints.

Theorem 1 (Type Safety). *The following invariants are satisfied at runtime for the current method meth:*

- (**stack1**) $length(evalStack) = length(evalStackT_{pc})$.
- (**stack2**) $\vdash evalStack(i) : evalStackT_{pc}(i) \circ inst(meth)$,
for every $i = 0, length(evalStack) - 1$.
- (**arg**) $\vdash argVal(0) : argTypes(meth)(0)$. If $meth$ takes at least two arguments,
 $\vdash argVal(i) : argTypes(meth)(i) \circ inst(meth)$, for every $i = 1, argNo(meth) - 1$.
- (**constr**) If $meth$ is generic, $boxed(genArg(meth)(i)) \sqsubseteq constr_i^{meth}$, for every
 $i = 0, genParamNo(meth) - 1$.

Proof. The proof is by induction on the run of the model for the operational semantics. The invariants obviously hold in the initial state of the virtual machine, i.e., for the **entrypoint**. Due to the lack of space, we consider here only two critical cases for virtual method calls and method returns.

Case 1. $code(meth)(pc) = CallVirt(T, C::M) \circ inst(meth)$: Since $meth$ is well-typed, we get $check(meth, pc, evalStackT_{pc})$ from Definition 6 (**wt2**). According to the definition of $check$ in Table 7, $evalStackT_{pc} \sqsubseteq_{suf} argTypes(C::M) \circ inst(C::M)$. By (**constr**) and Lemma 2, we have⁶

$$evalStackT_{pc} \circ inst(meth) \sqsubseteq_{suf} (argTypes(C::M) \circ inst(C::M)) \circ inst(meth) \quad (1)$$

By (1) and by the induction hypothesis – that is, by the invariants (**stack1**) and (**stack2**) – there exists a list of values L , two lists of types L' and L'' and the values $(val_i)_{i=0}^{argNo(C::M)-1}$ such that: $evalStack = L \cdot [val_i]_{i=0}^{argNo(C::M)-1}$,

$$evalStackT_{pc} = L' \cdot L'', length(L'') = argNo(C::M), \text{ for every } i = 0, length(evalStack) - argNo(C::M) - 1 \quad \vdash L(i) : L'(i) \circ inst(meth) \quad (2)$$

⁶ Note that $evalStackT_{pc}$ might involve generic parameters.

$$\text{for every } i = 0, \argNo(C::M) - 1 \quad \vdash \text{val}_i : L''(i) \circ \text{inst}(\text{meth}) \quad (3)$$

By (1), (2) and (3), we have for every $i = 0, \argNo(C::M) - 1$

$$\vdash \text{val}_i : (\argTypes(C::M)(i) \circ \text{inst}(C::M)) \circ \text{inst}(\text{meth}) \quad (4)$$

When the instruction $\text{CallVirt}(T, C::M) \circ \text{inst}(\text{meth})$ is executed, the macro $\text{VIRTCALL}(\text{val}_0, C::M \circ \text{inst}(\text{meth}), [\text{val}_i]_{i=1}^{\argNo(C::M)-1})$ is invoked. Assuming that val_0 is not **null**, **INVOKE** does the following for the next current method, i.e., $D::M = \text{lookup}(\text{actualTypeOf}(\text{val}_0), C::M \circ \text{inst}(\text{meth}))$:

- sets the pc to 0,
- sets the evalStack to $[]$ (this, the above initialization of the pc and Definition 6 (**wt1**) ensure that the invariants (**stack1**) and (**stack2**) hold also upon entering $D::M$)
- sets through **SETARG**, the arguments $(\argVal(j))_{j=0}^{\argNo(D::M)-1}$ (corresponding to $D::M$) to $(\text{val}_i)_{i=0}^{\argNo(C::M)-1}$. Note that we have $\argNo(C::M) = \argNo(D::M)$.

It remains to prove that the invariants (**arg**) and (**constr**) hold also for $D::M$. By Lemma 3, we have

$$\text{actualTypeOf}(\text{val}_0) \sqsubseteq \argTypes(D::M)(0) \quad (5)$$

and the following relations for every $i = 1, \argNo(C::M) - 1$:

$$\begin{aligned} &(\argTypes(C::M)(i) \circ \text{inst}(C::M)) \circ \text{inst}(\text{meth}) = \\ &\argTypes(C::M \circ \text{inst}(\text{meth}))(i) \circ \text{inst}(C::M \circ \text{inst}(\text{meth})) \sqsubseteq \\ &\argTypes(D::M)(i) \circ \text{inst}(D::M) \end{aligned} \quad (6)$$

The invariant (**arg**) for the first argument is ensured by (5) and the relation $\argTypes(D::M)(0) \circ \text{inst}(D::M) = \argTypes(D::M)(0)$ (the type of the **this** pointer is instantiated in contrast with the other argument types). For the other arguments, (**arg**) follows from (4) and (6).

To prove (**constr**), we assume that $D::M$ is a generic method. This implies that also the method $C::M$ is generic. By Lemma 5, we get $\text{genParamNo}(D::M) = \text{genParamNo}(C::M)$ and for every $i = 0, \text{genParamNo}(C::M) - 1$:

$$\text{boxed}(\text{genArg}(C::M \circ \text{inst}(\text{meth}))(i)) = \text{boxed}(\text{genArg}(D::M)(i)) \sqsubseteq \text{constr}_i^{D::M}$$

Proposition 1 applied to $C::M$ and the above relations imply (**constr**).

Case 2. $\text{code}(\text{meth})(pc) = \text{Return} \circ \text{inst}(\text{meth})$: Since the current method meth is well-typed, Definition 6 (**wt2**) and the definition of check in Table 7 imply $\text{evalStackT}_{pc} \sqsubseteq_{\text{len}} \text{void}(\text{retType}(\text{meth}))$.

Let $T' = \text{retType}(\text{meth})$. If T' is **void**, then evalStackT_{pc} shall be $[]$. If T' is not **void**, then $\text{evalStackT}_{pc} = [T' \circ \text{inst}(\text{meth})]$. Note that T' might be an open generic type. The induction hypothesis – that is, the invariants (**stack1**)

and **(stack2)** applied to *meth* – implies that *evalStack* is [] if the return type of *meth* is **void** and *evalStack* = [val] where $\vdash val : T' \circ inst(meth)$ if the return type of *meth* is not **void**. Let $D::M$ be the reference of *meth*. Accordingly,

$$\vdash val : T' \circ inst(D::M) \quad (7)$$

If $D::M$ is the **entrypoint**, there is nothing to prove. Otherwise, let $E::M'$ be the method which invoked $D::M$ through an instruction *CallVirt*($T'', C::M$) (the case when the call is through a *Constrained.CallVirt* instruction is similar). We need to show that the invariants **(stack1)** and **(stack2)** hold for $E::M'$ after $D::M$ returns.

When the instruction *Return* $\circ inst(D::M)$ is executed, the frame of $D::M$ is popped off by **RESULT**, $E::M'$ becomes the current method *meth*, the *pc* of $E::M'$ is incremented by 1 and *val* is pushed on the *evalStack* of $E::M'$.

The invariants held before **VIRTCALL** selected $D::M$ through a lookup applied to $C::M$. Similarly as in Case 1, there exists the lists of types L' and L'' that satisfy the relations (2). By Definition 6 (**wt3**) and definition of *succ* for *CallVirt*, we have: $drop(evalStackT_{pc}, argNo(C::M)) \cdot void(T'' \circ inst(C::M)) = L' \cdot void(T'' \circ inst(C::M)) \sqsubseteq_{len} evalStackT_{pc+1}$. By this, the invariant **(constr)** for *meth* and Lemma 2, we get

$$(L' \cdot void(T'' \circ inst(C::M))) \circ inst(meth) \sqsubseteq_{len} evalStackT_{pc+1} \circ inst(meth) \quad (8)$$

If T'' is **void**, Lemma 4 implies that also T' is **void**. **(stack1)** and **(stack2)** follow then from (8) and (2). If T'' is not **void**, by Lemma 4 we get $T' \circ inst(D::M) \sqsubseteq T'' \circ inst(C::M)$. This relation, **(constr)** and Lemma 2 imply $(T' \circ inst(D::M)) \circ inst(meth) \sqsubseteq (T'' \circ inst(C::M)) \circ inst(meth)$. This, together with (8), (2) and (7) guarantees the invariants **(stack1)** and **(stack2)**. \square

6 Conclusion

We have provided a mathematical specification for the CLR generics design via a type system and a model for the semantics of a subset of bytecode instructions with generics. We have formalized the type-consistency tests checked for the subset by the CLR bytecode verifier. Finally, we have proved that adding generics maintains the type safety of the CLR.

Acknowledgment. The author gratefully acknowledges the four reviewers, Viktor Schuppan and Horatiu Julia for their valuable comments and suggestions.

References

1. Microsoft .NET Framework 2.0. <http://msdn.microsoft.com/netframework/>.
2. Egon Börger and Robert F. Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
3. W. R. Cook. A Proposal for Making Eiffel Type-Safe. *Comput. J.*, 32(4):305–311, 1989.

4. Common Language Infrastructure (CLI) – Standard ECMA-335, 2002.
5. Nicu G. Fruja. The Soundness of the .NET CLR Bytecode Verifier. submitted.
6. Nicu G. Fruja. A Modular Design for the .NET CLR Architecture. In A. Slissenko D. Beauquier and E. Börger, editors, *12th International Workshop on Abstract State Machines, ASM 2005, Paris, France*, pages 175–199, March 2005.
7. Nicu G. Fruja and Egon Börger. Analysis of the .NET CLR Exception Handling. In Vaclav Skala and Piotr Nienaltowski, editors, *3rd International Conference on .NET Technologies, .NET 2005, Pilsen, Czech Republic*, pages 65–75, June 2005.
8. Mark Howard, Éric Bezault, Bertrand Meyer, Dominique Colnet, Emmanuel Stapf, Karine Arnout, and Markus Keller. Type-safe covariance: Competent compilers can catch all catcalls. Draft at <http://se.ethz.ch/~meyer/>, April 2003.
9. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
10. A. Jeffrey. Generic Java type inference is unsound. The Types Forum, 2001.
11. Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation, PLDI 2001, Snowbird, Utah, United States*, pages 1–12, May 2001.
12. Mirko Viroli and Antonio Natali. Parametric Polymorphism in Java: an Approach to Translation based on Reflective Features. *SIGPLAN Not.*, 35(10):146–165, 2000.
13. Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of Generics for the .NET Common Language Runtime. In *31st ACM Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy*, pages 39 – 51, January 2004.

The Weird World of Bi-directional Programming

Benjamin C. Pierce

University of Pennsylvania

Abstract. Programs generally work in just one direction, from input to answer. But sometimes we need a program to work in two directions: after calculating an answer, we want to *update* the answer and then somehow calculate backwards to find a correspondingly updated input. Of course, in general, a given update to the answer may not correspond to a unique update on the input, or to any at all; we need an “update translation policy” that tells us which updates can be translated and how to choose among translations when there are many possibilities. The question of how to determine such a policy has been called the *view update problem* in the database literature.

Many approaches to this problem have been devised over the years; most have taken existing database query languages (such as SQL) as their starting points and then proposed ways of describing or inferring update policies. More recently, several groups have begun working to design entirely new languages in which programs are inherently bi-directional – i.e., in which every program can be read from left to right as a map from inputs to answers *and* from right to left as (roughly) a map from updated answers to updated inputs. Moreover, bi-directionality in these languages is treated compositionally: each primitive works in both directions, and the two directions of compound programs can be derived from the two directions of their subcomponents.

This talk charts some interesting regions of the world of bidirectional programming and bi-directional language design, using as a touchstone our experiences at the University of Pennsylvania in the context of the Harmony project, where bi-directional languages – one for transforming trees and another for relational data – play a crucial role in the architecture of a universal data synchronizer.

Author Index

- Ahmed, Amal 7, 69
- Broberg, Niklas 180
- Caires, Luís 214
- Carpinetti, Samuele 197
- Chin, Brian 264
- Clarke, Dave 1
- Codish, Michael 230
- Cooper, Gregory H. 294
- Costa Seco, João 214
- Drossopoulou, Sophia 1
- Fluet, Matthew 7
- Foster, Jeffrey S. 309
- Fruja, Nicu G. 325
- Furr, Michael 309
- Gulwani, Sumit 279
- Hofmann, Martin 22
- Islam, Nayeem 162
- Jay, Barry 100
- Jia, Limin 131
- Jost, Steffen 22
- Kesner, Delia 100
- Koutavas, Vasileios 146
- Krishnamurthi, Shriram 294
- Lagoon, Vitaly 230
- Lal, Akash 246
- Laneve, Cosimo 197
- Leino, K. Rustan M. 115
- Leroy, Xavier 54
- Liblit, Ben 246
- Lim, Junghee 246
- Markstrum, Shane 264
- Millstein, Todd 264
- Morrisett, Greg 7
- Müller, Peter 115
- Mycroft, Alan 38
- Noble, James 1
- Palsberg, Jens 264
- Peyton Jones, Simon 38
- Pierce, Benjamin C. 342
- Polishchuk, Marina 246
- Rudiak-Gould, Ben 38
- Sands, David 180
- Schachte, Peter 230
- Stuckey, Peter J. 230
- Summers, Alexander J. 84
- Tiwari, Ashish 279
- van Bakel, Steffen 84
- Walker, David 131
- Wand, Mitchell 146
- Yu, Dachuan 162