Lecture Notes in Computer Science 1766
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

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

Mehdi Jazayeri   Rüdiger G.K. Loos
David R. Musser   (Eds.)

Generic Programming

International Seminar on Generic Programming
Dagstuhl Castle, Germany, April 27-May 1, 1998
Selected Papers

Springer

Series Editors

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

Volume Editors

Mehdi Jazayeri
Technical University of Vienna
Information Systems Institute, Distributed Systems Group
Argentinierstr. 8/184-1, 1040 Vienna, Austria
E-mail: jazayeri@alum.mit.edu

Rüdiger G.K. Loos
Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Sand 13, 72076 Tübingen, Germany
E-mail: loos@informatik.uni-tuebingen.de

David R. Musser
Rensellaer Polytechnic Institute, Computer Science Department
Troy, NY 12180-3590, USA
E-mail: musser@cs.rpi.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Generic programming : selected papers / International Seminar on
Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998.
Mehdi Jazayeri . . . (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ;
Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; 1766)
ISBN 3-540-41090-2

CR Subject Classification (1998): F.3, D.3, D.2, D.1, E.1, F.1, I.3.5, I.1

ISSN 0302-9743
ISBN 3-540-41090-2 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Steingräber Satztechnik GmbH, Heidelberg
Printed on acid-free paper SPIN: 10719708 06/3142 5 4 3 2 1 0

Preface

This volume contains contributed papers from participants in the Generic Programming Seminar held 27 April – 1 May 1998 at the Dagstuhl Conference Center in Wadern, Germany.

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Among the most obvious achievements of this relatively new programming paradigm are new libraries of software components, both in areas of fundamental and broadly used algorithms and data structures – the Standard Template Library – and in more specialized areas such as computer algebra, graph theory, and computational geometry. As useful as such components may be, however, they are probably less important than the overall generic programming methodology being developed. The papers collected here are reports from the field on the major problems and emerging solutions of generic programming methodology.

June 2000

Mehdi Jazayeri
Rüdiger Loos
David Musser

Organization

The Generic Programming Seminar was held 27 April – 1 May 1998 at the Conference Center of the Schloß Dagstuhl, located in Wadern, Germany (near Saarbrücken). There were forty nine attendees from ten countries. The formal program of the meeting included thirty seven lectures, a panel session on software library standardization, and a discussion of open problems and projects. Many informal discussions also took place, one of the many benefits of the Dagstuhl setting. The traditional Dagstuhl social event was at this meeting a Baroque concert held in the chapel of the Schloß.

Organizers

Mehdi Jazayeri	Technical University of Vienna
Rüdiger Loos	Tübingen University
David Musser	Rensselaer Polytechnic Institute
Alexander Stepanov	SGI

Attendees

David Abrahams	Mark of the Unicorn, Inc.
Giuseppe Attardi	University of Pisa
Matt Austern	SGI
Ulrich Breymann	Hochschule Bremen
Stephen Cerniglia	Rensselaer Polytechnic Institute
George Collins	University of Delaware
James Crotinger	Los Alamos National Lab
Krzysztof Czarnecki	Daimler-Benz
James Dehnert	SGI
Angel Diaz	IBM Watson Research Center
Axel Dold	Ulm University
Matthew Dwyer	Kansas State University
Ulrich Eisenecker	Fachhochschule Heidelberg
Holger Gast	Tübingen University
Robert Glück	Copenhagen University
Friedrich von Henke	University of Ulm
Hoon Hong	North Carolina State University
Mehdi Jazayeri	Technical University of Vienna
Johann Jeuring	Utrecht University
Nicolai Josuttis	Bredex
Erich Kaltofen	North Carolina State
Ullrich Köthe	Rostock University

Uwe Kreppel	Tübingen University
Wolfgang Küchlin	Tübingen University
Dietmar Kühl	Konstanz University
Gary Leavens	Iowa State University
Karl Lieberherr	Northeastern University
Rüdiger Loos	Tübingen University
Kurt Mehlhorn	MPI, Saarbrücken University
David Musser	Rensselaer Polytechnic Institute
Stefan Näher	Halle University
Oscar Nierstrasz	Bern University
Martin Odersky	University of South Australia
William Ogden	Ohio State University
Arturo Sanchez-Ruiz	University of Central Venezuela
Stefan Schirra	Saarbrücken University
Wolfgang Schreiner	RISC-Linz
Sibylle Schupp	Rensselaer Polytechnic Institute
Christoph Schwarzweller	Tübingen University
Murali Sitaraman	W. Virginia University
Georg Trausmuth	Technical University of Vienna
David Vandevoorde	Hewlett-Packard
Todd Veldhuizen	University of Waterloo
Bruce Weide	Ohio State University
Karsten Weihe	Konstanz University
Roland Weiss	Tübingen University
Reinhard Wilhelm	Saarbrücken University
Alexandre Zamulin	Russian Academy of Sciences, Novosibirsk
Wolf Zimmermann	Karlsruhe University

Table of Contents

Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
{dehnert,stepanov}@sgi.com

Abstract. Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types; e.g., copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.

Keywords: Generic programming, operator semantics, concept, regular type.

Introduction

For over three decades, Computer Science has been pursuing the goal of software reuse. There have been a variety of approaches, none of them as successful as similar attempts in other engineering disciplines. Generic programming [5] offers an opportunity to achieve what these other approaches have not. It is based on the principle that software can be decomposed into components which make only minimal assumptions about other components, allowing maximum flexibility in composition.

Reuse has been successful in the area of libraries. Examples include system interface libraries such as Unix [3], numeric libraries such as Lapack [1], and window management libraries such as X [6]. However, these libraries have the characteristic that they use fully specified interfaces that support a pre-determined set of types, and make little or no attempt to operate on arbitrary user types. These fully specified interfaces have been instrumental in encouraging use, by allowing users to comfortably use them with full knowledge of how they will behave. Paradoxically, however, this strength turns into a weakness: for example, while people can use the C library routine sqrt on any machine with predictable results, they cannot use it when a new type like quad-precision floating point is added. In order to make progress, we must overcome this limitation.

Generic programming recognizes that dramatic productivity improvements must come from reuse without modification, as with the successful libraries.

Breadth of use, however, must come from the separation of underlying data types, data structures, and algorithms, allowing users to combine components of each sort from either the library or their own code. Accomplishing this requires more than just simple, abstract interfaces—it requires that a wide variety of components share the same interface so that they can be substituted for one another. It is vital that we go beyond the old library model of reusing identical interfaces with pre-determined types, to one which identifies the minimal requirements on interfaces and allows reuse by similar interfaces which meet those requirements but may differ quite widely otherwise. Sharing similar interfaces across a wide variety of components requires careful identification and abstraction of the patterns of use in many programs, as well as development of techniques for effectively mapping one interface to another.

We call the set of axioms satisfied by a data type and a set of operations on it a *concept*. Examples of concepts might be an integer data type with an addition operation satisfying the usual axioms; or a list of data objects with a first element, an iterator for traversing the list, and a test for identifying the end of the list. The critical insight which produced generic programming is that highly reusable components must be programmed assuming a minimal collection of such concepts, and that the concepts used must match as wide a variety of concrete program structures as possible. Thus, successful production of a generic component is not simply a matter of identifying the minimal requirements of an arbitrary type or algorithm—it requires identifying the common requirements of a broad collection of similar components. The final requirement is that we accomplish this without sacrificing performance relative to programming with concrete structures. A good generic library becomes a repository of highly efficient data structures and algorithms, based on a small number of broadly useful concepts, such that a library user can combine them and his own components in a wide variety of ways.

The C++ Standard Template Library (STL) [7] is the first extensive instance of this paradigm in wide use. It provides a variety of data containers and algorithms which can be applied to either built-in or user types, and successfully allows their composition. It achieves the performance objectives by using the C++ template mechanism to tailor concept references to the underlying concrete structures at compile time instead of resolving generality at runtime. However, it must be extended far beyond its current domain in order to achieve full industrialization of software development. This requires identifying the principles which have made STL successful.

In our search for these principles, we start by placing generic programming in an historic progression. The first step was a generalized machine architecture, exemplified by the IBM 360, based on a uniform view of the machine memory as a sequence of bytes, referenced by uniform addresses (pointers) independent of the type of data being referenced. The next step was the C programming language [4], which was effectively a generalized machine language for such architectures, providing composite data types to model objects in memory, and pointers as identifiers for such memory objects with operations (dereferencing and increment/decrement) that were uniform across types.

The C++ programming language [8] was the next step. It allows us to generalize the use of C syntax, applying the built-in operators to user types as well, using class definitions, operator overloading, and templates. The final step in this progression is generic programming, which generalizes the semantics of C++ in addition to its syntax. If we hope to reuse code containing references to the standard C++ operators, and apply it to both built-in and user types, we must extend the semantics as well as the syntax of the standard operators to user types. That is, the standard operators must be understood to implement well-defined concepts with uniform axioms rather than arbitrary functions. A key aspect of this is generalizing C's pointer model of memory to allow uniform and efficient traversal of more general data structures than simple arrays, accomplished in the STL by its iterator concepts.

This extension of C built-in operator semantics is the key to at least part of the STL's success in finding widely applicable concepts. The development of built-in types and operators on them in programming languages over the years has led to relatively consistent definitions which match both programmer intuition and our underlying mathematical understanding. Therefore, concepts which match the semantics of built-in types and operators provide an excellent foundation for generic programming.

In this paper, we will investigate some of the implications of extending built-in operator semantics to user-defined types. We will introduce the idea of a regular type as a type behaving like the built-in types, and will investigate how several of the built-in operators should behave when applied to such user-defined types.

Regular Types

The C++ programming language allows the use of built-in type operator syntax for user-defined types. This allows us, as programmers, to make our user-defined types *look* like built-in types. Since we wish to extend semantics as well as syntax from built-in types to user types, we introduce the idea of a *regular type*, which matches the built-in type semantics, thereby making our user-defined types *behave* like built-in types as well.

The built-in types in C++ vary substantially in the number and semantics of the built-in operators they support, so this is far from a rigorous definition. However, we observe that there is a core set of built-in operators which are defined for all built-in types. These are the constructors, destructors, assignment and equality operators, and ordering operators. Figure 1 gives their syntax in C++. We use C++ as our basis, although our principles are not language-specific.

The first four fundamental operations in Figure 1 have default definitions for all built-in types and structures in C/C++. The remaining ones have default definitions only for built-in types, but could be defined reasonably for structures as well. Equality and inequality could be defined component-wise, and the order-

Default constructor	T a;
Copy constructor	T a = b;
Destructor	˜T(a);
Assignment	a = b;
Equality	a == b
Inequality	a != b
Ordering, e.g.	a < b

Fig. 1. Fundamental Operations on Type T

ing operators could be defined with a lexicographic order, using the component orderings recursively. (The ordering case is interesting. C++ does not define total ordering operations on pointer types, and it is not possible to define efficient operations which would produce the same results for all implementations. Even without such a portability guarantee, however, there are applications for which universal availability of efficient operators is useful. But this subject is beyond the scope of the current paper.)

As we shall see below, the default definitions of these operations are not always adequate, but their semantics when applied to the built-in types provide the model we want for our more general requirements on regular types. In the remainder of this paper, we shall attempt to identify the essential semantics of these operations, which we call the *fundamental operations* on a type T. By doing so, we will fill in some of the details of a precise definition of regular types.

Copy, Assignment, and Equality

First, we consider the interactions among the copy constructor, assignment, and equality operators. These operations are central to our understanding of a programming language. What can we say about them?

1. `T a = b; assert(a==b);`
   Our first axiom simply says that after constructing a new object `a` of type `T`, with an initial value copied from object `b`, we expect objects `a` and `b` to be equal. Furthermore, we expect this construction to be equivalent to constructing a with a default constructor and then assigning the value of `b` to it:
2. `T a; a = b;` $\iff$ `T a = b;`
   So far, our axioms would be satisfied equally well by a language like C++ which copies values on assignment, or by a language like Lisp which simply copies addresses leaving both names pointing to the same copy of the value. Our next axiom says that we intend the C++ copy semantics:
3. `T a = c;  T b = c;  a = d;  assert(b==c);`
   Here, after assigning the same value `c` to both `a` and `b`, we expect to be able to modify `a` without changing the value of `b`. In fact, we want an even

stronger condition. If `zap` is an operation which always changes the value of its operand, we expect the following to hold:

4. `T a = c;  T b = c;  zap(a);  assert(b==c && a!=b);`
   That is, `b` and `c` do not continue to be equal simply because their values were changed along with `a`'s, but rather because changing `a`'s value did not change theirs.

As an example of the power of these axioms, let us considered a small example. Figure 2a contains a simple template function for swapping two values, and figure 2b contains a code fragment which specifies the expected semantics of that swap function.

a. ***swap*** function	**b.** ***Swap*** specification
`template <class T>`	`T a, b;`
`void swap (T& x, T& y) {`	`...`
`    T tmp = x;`	`T old_a = a;`
`    x = y;`	`T old_b = b;`
`    y = tmp;`	`Swap (a, b);`
`}`	`Assert (a == old_b);`
	`Assert (b == old_a);`

Fig. 2. Generic swap function

Now, if we substitute the body of the swap function for its call in the specification, and apply the axioms above along with the usual axioms of equality (in particular transitivity), we get the expected result, as shown in figure 3.

a. ***swap***	**b. Assertions**
`T a,b;`	`//`
`T old_a = a;`	`// a == old_a`
`T old_b = b;`	`// b == old_b`
`swap (T& a, T& b) {`	`//`
`  T tmp = a;`	`// tmp==a && tmp==old_a`
`  a = b;`	`// a==b && a==old_b`
`  b = tmp;`	`// b==tmp && b==old_a`
`}`	`// b==old_a && a==old_b`

Fig. 3. Validation of swap function

The axioms above provide us with a reasonable characterization of the semantics of copy constructors and assignment operators in terms of the equality of their operands after they are applied. A copy constructor creates a new object equal to the object from which it is copied; and assignment copies its right-hand side operand to its left-hand side object, leaving their values equal.

However, we do not yet have a satisfactory definition of the equality of two objects of a regular type. We shall investigate this question in the next section.

Equality of Regular Types

Some writers have defined equality as a relation that is reflexive, symmetric, and transitive. While these are certainly attributes of equality, they do not constitute a definition. To see this, simply consider a hypothetical equality function which always returns true. It has these three attributes, but certainly does not satisfy our expectations for an equality operator. We must look further.

Logicians might define equality via the following equivalence:

$$x == y \iff \forall \text{ predicate } P, P(x) == P(y)$$

That is, two values are equal if and only if no matter what predicate one applies to them, one gets the same result. This appeals to our intuition, but it turns out to have significant practical problems. One direction of the equivalence:

$$x == y \implies \forall \text{ predicate } P, P(x) == P(y)$$

is useful, provided that we understand the predicates P for which it holds. We shall return to this question later. The other direction, however:

$$\forall \text{ predicate } P, P(x) == P(y) \implies x == y$$

is useless, even if P is restricted to well behaved predicates, for the simple reason that there are far too many predicates P to make this a useful basis for deciding equality. Again, we must look further.

Fortunately, our computer hardware generally defines an equality relation on the built-in types which it implements efficiently. This equality relation is normally bitwise equality (although there are sometimes minor deviations like distinct positive and negative zero representations). Starting from this basis, there is a natural default equality for types composed of simpler types; i.e., equality of corresponding parts of the composite objects. (Although this definition is natural, neither C nor C++ provides a default equality operator for composite types.) While this definition is appealing for arbitrary data structures, we must resolve several questions.

In order to apply this definition to build an equality operator for a composite type from the equality operators on the types of its parts, we must identify its parts. Intuitively, they are the (data) members of a struct, but this is still not sufficient.

First, a C/C++ struct cannot represent all objects of interest. Specifically, it cannot represent an object of variable size. (This design decision was made because allowing variable size types would not allow the creation of arrays of those types with efficient access.) As a result, objects which are naturally variable sized must be constructed in C++ out of multiple simple structs, connected by pointers. In such cases, we say that the object has *remote* parts. For such objects, the equality operator must compare the remote parts of two objects rather than the pointers to them, since we would not like objects with equal remote parts to compare unequal simply because the remote parts were in different memory and their addresses were unequal.

The next subtle problem in identifying the parts of composite objects is that such objects sometimes contain components which are not essential to our concept of value. A good example of this situation is a struct which contains a count of the number of pointers which reference it, perhaps for memory management purposes. We do not view such a reference count component to be part of the value of the object, and would not want otherwise equal objects to compare unequal simply because of unequal reference counts. Our second caveat then is that an equality operator should ignore inessential components.

The final problem relates to the question of where one object ends and another begins. In the physical world, we would think of the legs, seat, back, and arms of a chair as being parts of the chair—the chair would be very different without them. However, we would not think of a person or dog sitting in the chair as a part of the chair, even though it is closely associated with the chair, at least for a time. Similarly if we were writing a program to produce a graphics display of a scene containing a chair, we might represent the chair as a struct containing pointers to remote parts (i.e., other structs) for its legs, seat, etc. However, even if we were to keep track of who or what was sitting in the chair by keeping a pointer to that other object in the chair object, we should probably not consider the other object to be part of the chair. That is, some components of composite objects reflect relationships between objects, and should not be considered as parts for equality testing purposes.

These observations leave us with a definition of equality which is workable in practice, although it still leaves room for judgment:

Definition 1. *Two objects are equal if their corresponding parts are equal (applied recursively), including remote parts (but not comparing their addresses), excluding inessential components, and excluding components which identify related objects.*

Once we have identified the parts of an object which must be tested for equality, we know from the earlier discussion that at least those parts must be copied by copy constructors or assignment operators. In particular, these operators must make copies of remote parts, rather than simply copying the pointers to them.

Now let us return to part of the logicians' definition of equality. Recall that we would like the following statement to be true:

$$x == y \implies \forall \text{ “reasonable” function } \texttt{foo},\ \texttt{foo}(x) == \texttt{foo}(y)$$

It is necessary to limit our expectations to some subset of possible functions `foo`. For instance, this statement will not hold for the "address-of" function applied to distinct objects with equal values, nor will it hold for any other function which distinguishes between individual objects rather than between their values. Considering an example will demonstrate some of the challenges facing a designer of generic components.

Most of us would intuitively assume that a visible accessor function—that is, a public function which returns the value of some component of a composite type—would be a reasonable function which should satisfy the above condition. However, that assumption constrains the combination of the equality operator definition and the choice of the visible parts of an object. To see how, suppose that we define a rational number object as a pair `(p, q)` representing its numerator and denominator. Given such an object type, we cannot define equality mathematically and still allow `p` and `q` to be visible parts, since doing so would yield the following incorrect deduction:

$$\begin{aligned} \texttt{r1 == r2} &\implies \texttt{r1.p == r2.p} \\ \texttt{(1,2) == (2,4)} &\implies \texttt{1 == 2} \end{aligned}$$

Faced with this situation, several reasonable design decisions are possible which preserve our intuition. First, we could avoid defining an equality operator (perhaps defining an `equiv` function with the mathematical definition instead). Second, we could avoid making `p` and `q` visible parts of our rational number type. Finally, we could require that any rational number represented by this type is always in reduced form; i.e., its numerator and denominator have no common divisors.

Optimization

It is fair to ask why all of these details are important. After all, we can always take an arbitrary type definition with a sufficiently extensive set of operations defined on it, and write programs which use it effectively by following its own usage expectations. Most software development has operated this way to date, hand crafting each new component type to use the features exported by the types on which it depends, and exporting features designed to simplify its own implementation or that of known clients.

Generic programming, however, changes the rules substantially. If we are to succeed in producing widely reusable components, idiosyncratic interfaces are no longer usable. A component programmer must be able to make some fundamental assumptions about the interfaces she uses, without ever seeing their implementations or even imagining their applications. Similarly, her eventual users must provide the types implementing those interfaces, and if the same types are to interface with a variety of generic components, the interfaces must be consistent with one another.

The operations we have discussed here, equality and copy, are central because they are used by virtually all programs. They are also critically important because they are the basis of many optimizations which inherently depend upon knowing that copies create equal values, while not affecting the values of objects not involved in the copy. Such optimizations include, for example, common subexpression elimination, constant and copy propagation, and loop-invariant code hoisting and sinking. These are routinely applied today by optimizing compilers to operations on values of built-in types. Compilers do not generally apply them to operations on user types because language specifications do not place the restrictions we have described on the operations of those types.

However, users do apply such optimizations by hand. They often do so without thinking because they intuitively expect the conditions to apply. If they are to produce efficient generic components without seeing the underlying type definitions, they must be able to make the assumptions which allow such optimizations. Our axioms, then, are necessary to allow users to reliably make the optimizations commonly made both by optimizing compilers and by optimizing programmers.

Ultimately, we would like compilers to be able to perform such optimizations at a high semantic level as well as they do at the built-in type level. This will require more formal adherence to the axioms we have described for the fundamental operations. But we would like to go further. Specifically, let us return to the equality principle mentioned above:

$$x == y \implies \forall \text{ ``reasonable'' function } \texttt{foo},\ \texttt{foo}(x) == \texttt{foo}(y)$$

Again, what is a reasonable function? For optimization purposes, there are several classes of functions we would like to capture. First are the standard operators on built-in types that do not have side effects, for example `a+b`, `c-d`, or `p%q`. Second are the visible member accesses; e.g., `s.first` or `c->imaginary`. A third class is the well-known pure functions; e.g., `abs(x)`, `sqrt(y)`, and `cos(z)`. Knowledge of most of these could be built into compilers if we made the appropriate restrictions on the user definitions of them. However, there are many more possibilities among arbitrary functions defined by users. Compilers cannot identify them all without assistance, both because the compiler cannot always see the function definitions, and because the compiler cannot make the necessary distinctions between essential and inessential parts or between pointers to remote parts or to related objects. The ultimate solution, then, must be to identify the important attributes, and allow programmers to specify them explicitly. This is an important language design issue, but is beyond the scope of this paper.

Complexity

It is often claimed that complexity is only an attribute of an implementation, and not properly part of component specification. This is wrong, and becomes more so with generic components. Users (and algorithms) make basic assumptions about operator complexity, and make decisions about the data structures or algorithms they will use based on those assumptions. Consider several examples:

- We expect the push and pop operations on a stack to require amortized constant time. If this expectation were not met, we would often use a different data structure, or perhaps implement a stack explicitly based on another data structure known to behave that way (such as an STL vector). This means that a stack implementation which does a re-allocation and copy whenever the stack grows is not just a poor implementation, it is an unacceptable implementation.
- We expect an implementation of character strings to have a copy constructor which is linear in the length of the string being copied. A well-known C++ standard library implementation contains a string copy constructor which is quadratic, requiring hours to copy a million-character string on a large server. Obviously, such an implementation is unusable for large strings.
- In the C++ STL, it would be possible for bidirectional iterators to support the random access iterator interface; i.e., providing operations to step through the data by more than one element at a time. However, it is important to keep them distinct—the best algorithms for some functions (e.g., rotate or random shuffle) differ dramatically for bidirectional and random access iterators.

For the fundamental operations, users also have intuitive expectations of complexity. For regular types, we therefore require that constructors, destructors, and assignment operators be linear (average-case) in the *area* (i.e., the total size of all parts) of the object involved. Similarly, we require that the equality operator have linear worst-case complexity. (The average-case complexity of equality is typically nearly constant, since unequal objects tend to test unequal in an early part.)

Summary

In this paper, we have investigated several of the fundamental operations on built-in types in C++, and identified characteristics they should have when applied to user-defined types. This process is central to defining broadly applicable concepts which can enable generic programming to produce components which can be reused with a wide variety of built-in and user-defined types. We believe, based on the success of the C++ STL, that this scientific approach of observing widespread commonality in existing programs, and then axiomatizing its properties consistent with existing programming and mathematical practice, holds promise that we will ultimately achieve the elusive goal of widespread software reuse.

References

1. J. Demmel, *LAPACK: A portable linear algebra library for supercomputers*, Proc. of the 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design, December 1989.
2. Aaron Kershenbaum, David R. Musser, and Alexander A. Stepanov, *Higher-Order Imperative Programming*, Computer Science Technical Report 88-10, Rensselaer Polytechnic Institute, April 1988.
3. Brian W. Kernighan and John R. Mashey, *The Unix Programming Environment*, Computer 14(4), 1981, pp. 12-24.
4. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
5. David R. Musser and Alexander A. Stepanov, *Generic Programming*, in P. Gianni, ed., *Symbolic and Algebraic Computation: International Symposium ISSAC 1988*, Lecture Notes in Computer Science v. 38, Springer-Verlag, Berlin, 1989, pp. 13-25.
6. Robert W. Scheifler and James Gettys, *X Window System*, 3rd Ed., Digital Press, 1992.
7. Alexander Stepanov and Meng Lee, *The Standard Template Library*, Tech. Report HPL-95-11, HP Laboratories, November 1995.
8. Bjarne Stroustrup, *The C++ Programming Language*, 3rd Ed., Addison-Wesley, Reading, MA, 1997.

Requirement Oriented Programming
Concepts, Implications, and Algorithms

David Musser[1], Sibylle Schupp[1], and Rüdiger Loos[2]

[1] Department of Computer Science
Rensselaer Polytechnic Institute
{musser,schupp}@cs.rpi.edu
[2] Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
loosr@acm.org

Abstract. Generic programming is requirement oriented programming. We study the concept of genericity and generic concepts. Support of generic programming is exemplified by TECTON formalizations of algebraic and STL concepts. We call for a common effort to establish and maintain generic libraries.

Keywords: generic programming, requirement oriented programming, concept description language, specification language, formal verification, standard template library, algebraic concept definitions, TECTON

1 Introduction

We define *generic* programming as *requirement oriented* programming. First of all this means that generic programs are minimal with respect to requirements. Secondly, we understand "requirement" in a sufficiently broad sense that, indeed, every important aspect of programs can be formally stated as a requirement.

Traditionally requirements have been limited to *specification* requirements, including correctness and invariant assertions. Generic specifications describe the minimal assumptions under which the programs are correct, and thus they describe abstract algorithms that may be instantiated in a broad range of applications. Type checking and verification of generic programs pose new challenges but are only one aspect of genericity.

More important and novel are *algorithmic* requirements. This means that generic algorithms guarantee performance assertions. We address algorithmic requirements in a companion paper [19] on SUCHTHAT [18].

Other important aspects of generic programs are related to portability, to allocation of system resources like storage, and to configuration and maintainability. Even documentation can become easier to read if it is done in a generic way, focused systematically on common aspects. We broadly call these requirements *software engineering* requirements.

Algorithmic and software engineering requirements have to be based on experiments whereas specification requirements are closer to a foundational framework of genericity.

1.1 Organization of the Paper

In the second section we study the concept of genericity in algebra, in algebraic specifications, and in programming languages.

In the third section we give an introduction to TECTON, a language for expressing generic concepts. We give some concept formalizations from algebra and the Standard Template Library, STL [1,12,20].

We have implemented a tool for processing TECTON, which checks concept descriptions for adherence to the languages syntax, sort-correctness, and identifier *availability* rules, the latter being a significant generalization of scope rules of previous specification (or programming) languages. In the fourth section we report measurements of the current implementation.

The last section puts forward a proposal to establish and maintain generic libraries.

2 The Concept of Genericity

2.1 Related Concepts in Computer Science

The concept of an algorithm as defined by Knuth [11] is characterized by a set of five properties: definiteness, finiteness, input, output, and effectiveness. As observed by Collins [4], relaxing the requirement of definiteness—that each step of an algorithm must be precisely defined—leads to the concept of an *abstract algorithm* or an *algorithm schema*, earlier terms for what is today more frequently referred to as a generic algorithm. Thus an algorithm in the traditional sense can be regarded as a generic algorithm for which all sources of indefiniteness—genericity—are made definite.

2.2 Related Concepts in Algebra

In the nineteen twenties, led by Emmy Noether, Grete Herrmann, Emil Artin, and others, algebra turned from working in concrete domains to abstract algebra and abstract algebraic domains. The intention was to base the proofs of the theorems on minimal requirements and to peel away from their inherent generality all additional, non-essential features available in concrete domains. Their student van der Waerden published this fresh view of algebra under the title *Modern Algebra*, meaning abstract algebra. This book was so successful, with numerous editions appearing over many years, that van der Waerden eventually decided to rename it simply *Algebra*.

We see a similar turn from the concrete to the abstract in programming. We expect that following the same principles "generic programming" will become so popular that one day it will be called simply "programming."

We also are convinced that "modern" algebra is *the* adequate theoretical foundation of genericity.[1]

[1] We maintain the distinction between the algebra-based notion of genericity and the set theoretical notion of type polymorphism.

2.3 Related Concepts: Algebraic Specification and Polymorphism

Abstract data types and in their context *algebraic specifications* [21] were a first important step toward abstraction and genericity. They were originally developed for specification and verification purposes, and it took a long time before they entered programming languages as modules or classes.

Following this trail one recognizes that *algebra* stresses the view that it is not sets of *elements* but the *properties of their operations* that define structures. Axioms in algebra therefore come closest to the notion of requirements at the heart of genericity.[2]

Another related notion is polymorphism. *Ad hoc polymorphism*[3] is the name for overloading the same operator with conceptually unrelated functions. Moreover, from "On Understanding Types, Data Abstraction, and Polymorphism" by Cardelli and Wegner [3] we learn: "Universally polymorphic functions will normally work on an infinite number of types (all the types having a given common structure) .... In terms of implementation, a universally polymorphic function will execute the *same* code for arguments of any admissible type, whereas an ad hoc polymorphic function may execute *different* code for each type of argument."

Generic functions are not universally polymorphic in the sense just given. Although they can be applied to an infinite number of parameter types, e.g., different container types or different iterators, the executed code varies considerably, depending on the parameter types. Furthermore, requirements—the core of semantics of generic algorithms—cannot be adequately expressed in the theoretical framework of the cited paper or in similar frameworks.

2.4 Related Concepts: The STL

The STL is the result of a long-term research project on generic programming and Tecton [9,10,7,15], including the development of preceding generic libraries in Scheme, Ada [13], and C++ [14]. "The STL is not a set of programs but a set of requirements" [12]. Accordingly, the STL pushes abstractions via core requirements to *families of abstractions*, far behind the usual abstract data types notion of abstraction (hiding of implementation details). It provides fine-grained reusable components which are highly adaptable but still come with efficiency guarantees. It strictly separates objects and methods—called containers and algorithms—and avoids the otherwise combinatorial explosion of classes and the dominance of objects over algorithms. Containers and algorithms are related by iterators, which generalize the notion of pointers as used for indexing through storage. Storages resources are factored out into allocators, which are defaulted in such a way that many programs do not contain any explicit storage allocation or deallocation.

The STL incorporates genericity by requirements in all three respects mentioned above, and has become a highly visible achievement in software method-

[2] This applies in a strong sense only to specification requirements, but indirectly also to algorithmic and software engineering requirements.

[3] The term is due to Strachey, according to [3].

ology. Its use within other C++-libraries and its transfer to other programming languages calls for a formal description. Progress toward such a description in TECTON is reported in [16]; it is noteworthy that in the process of working it out, several significant gaps in existing informal specifications of the STL were discovered.

Clearly, any implementation of the STL can guarantee requirements only to an extent to which the underlying implementation language supports such guarantees. In practice, therefore, they remain in many respects the responsibility of the implementor and user—as *meta*-concepts, the validity of which in a given program is not guaranteed by C++ or STL in any sense. This is not a criticism since every library can only be used within its specifications. But the goals of the STL raise the question whether the support of requirements can be extended by the means of a language.

3 Tecton, a Language for Generic Concepts

3.1 Requirements and Concept Instances

The central keyword of the TECTON[4] concept description language [8,7,15] is `requires`. Concept descriptions, which are a named sets of requirements, are the building blocks of TECTON.

Semantic requirements are introduced by `generates` or `requires` clauses. `Requires` is followed by quantified formulas, usually expressed as equations, over function descriptors and sorts. The latter are introduced by an `introduces` clause or are imported from other concepts.

Inheritance of concepts happens by `refines` or `uses` clauses. They not only make previously defined sorts and function descriptors available but import at the same time their associated requirements. The main goal of a TECTON translator is the generation of the *proof obligations* governing concept inheritance. Some of them can be discharged by a check in the concept dictionary, while others have to be established offline.

The most powerful and novel capability of TECTON is *concept instantiation* with replacements of available sorts, function descriptors, and concepts. It is this flexibility which avoids a proliferation of new names and which contributes most to a clear structure of related concepts.[5] Replacement of concepts resembles template class parameters and their instantiation. However, TECTON does not require an *a priori* declaration of parameters; instead, every constituent of a concept may be replaced at some later time—except requirements: they are the invariants of TECTON's genericity. Experience shows that the number of possible replacements grows very rapidly and that it would be impractical to

[4] Greek *τεκτων*: builder, craftsman. The name is an invitation to edify programs with conceptual clarity.

[5] Readers familiar with AXIOM, a computer algebra system that supports user-defined algebraic concepts, might consider how a systematic renaming capability similar to that of TECTON could simplify AXIOM programming.

anticipate them all with explicit parameter lists. At the same time, replacements cannot be done arbitrarily—Tecton requires both the original concept and its replacement to be previously defined and the rules of replacement ensure that semantic requirements are observed (proof obligations are generated where necessary).

3.2 Tecton by Examples

We begin with several Tecton *sentences* that define several familiar concepts of general binary relations.

```
Definition: Domain
  uses Boolean;
  introduces domain.
```

This *definition* sentence has two clauses, one that states that the `Domain` concept uses the `Boolean` concept (definable in Tecton but not shown here) and the other that states `Domain` introduces a sort named `domain`. Thus in any subsequent use or refinement of the `Domain` concept all of the sorts and function descriptors of `Boolean` are available with unmodified semantics, and additionally one may refer to a `domain` sort, which was not available in `Boolean`. However, nothing is known of the semantics of `domain` other than it denotes a set; i.e., it could be mapped to any set whatever.

```
Definition: Binary-relation
  refines Domain;
  introduces R: domain x domain -> bool.
```

The `refines` clause of this sentence states that in the `Binary-relation` concept all of the vocabulary of `Domain` is available (including that of `Boolean`). The semantics of any parts of `Domain` that were used, rather than introduced or refined, remain unchanged; hence, since `Boolean` was used in `Domain` it cannot be modified by `Binary-relation`. On the other hand the semantics of `domain` can be modified, and it is modified by the second clause, which states that there is a function `R` defined on `domain` $\times$ `domain` with range `bool`, the sort defined in `Boolean`. However, nothing more is required of this function, so it could any binary relation.

```
Definition: Reflexive
  refines Binary-relation;
  requires (for x: domain) x R x.
```

This sentence further refines `Binary-relation` into a concept that requires a certain semantic property, reflexivity, to be true of `R`. But there are still many possible mappings from `R` into specific relations. The following sentences refine the possible relations `R` can map to with other requirements.

```
Definition: Irreflexive refines Binary-relation;
  requires (for x: domain) not x R x.
```

```
Definition: Symmetric refines Binary-relation;
  requires (for x, y: domain) x R y implies y R x.

Definition: Antisymmetric refines Binary-relation;
  requires (for x, y: domain) x R y and y R x implies x = y.

Definition: Transitive refines Binary-relation;
  requires (for x, y, z: domain) x R y and y R z implies x R z.
```

With these concepts at hand, we can easily combine them to describe another fundamental concept.

```
Definition: Equivalence-relation refines Reflexive,
  Symmetric, Transitive.
```

We see in these examples that a TECTON concept description defines a set of structures (many-sorted algebras) and not just a single one; there are, for example many structures having the properties of an equivalence relation.

Next, we present definitions of a few order relation concepts.

```
Definition: Partial-order refines Reflexive [with ≤ as R],
  Antisymmetric [with ≤ as R], Transitive [with ≤ as R].
```

This example shows the use of *replacements* to modify the vocabulary associated with a previously defined concept: within `Partial-order` the binary relation is referred to as $\leq$, while `R` is banished from the available vocabulary (though it could be reintroduced with a separate meaning in later concept refinements).

```
Definition: Total-order refines Partial-order;
  requires (for x, y: domain) x ≤ y or y ≤ x.

Definition: Strict-partial-order
  refines Irreflexive [with < as R],
          Transitive [with < as R].

Lemma: Partial-order implies Strict-partial-order.
```

A *lemma* is a TECTON sentence that states a semantic property about the structures in the concepts involved, either stating additional properties or stating subset relationships; in either case the claim should logically follow from the concepts' defining properties. In the example, the lemma claims that every structure that belongs to `Partial-order` also belongs to `Strict-partial-order`; equivalently, the requirements of `Partial-order` imply those of `Strict-partial-order`.

```
Extension: Partial-order
  introduces
    <: domain x domain -> bool,
    >: domain x domain -> bool,
    ≥ : domain x domain -> bool;
  requires (for x, y: domain)
    (x < y)  = (x ≤ y and x != y),
    (x > y)  = (not x ≤ y),
    (x ≥ y) = (x > y or x = y).
```

In this last example of sentences about ordering concepts, several function descriptors are added to the vocabulary of `Partial-order` and are defined in terms of it. An *extension* differs from a definition with a `refines` clause in that no modifications are allowed to the semantics of the concept being extended; it differs from a definition with a `uses` clause in that no new concept name is introduced, which helps in avoiding proliferation of concept names.

The next example extends the concept `Real`, whose original definition is not shown here. The first two defining equations use a `where` clause[6] to specify the unique integer `n` satisfying the stated condition. The last equation defines a function in terms of the previous ones. The sorts `reals` and `integers` as well as the function descriptors for `<`, ≤, ≥, `+`, and `1` are not introduced here but are available from previous definitions.

```
Extension: Real
  introduces
    ceiling: reals -> integers,
    floor: reals -> integers,
    entier: reals -> integers;
  requires (for r: reals; n: integers)
    ceiling(r) = n where n - 1 < r and r ≤ n,
    floor(r) = n where n  ≤ r and r < n + 1,
    entier(r) = if r ≥ 0 then floor(r) else ceiling(r).
```

Let us use this example to elaborate on the semantics of sorts and function descriptors. There is a map from sorts to sets such that a sort s names the set S_s, here the set of real or integral numbers. More precisely, S_{reals} and $S_{integers}$ are the carrier sets of the structures $\mathbb{R}$ and $\mathbb{Z}$, respectively. Similarly, function descriptors

```
    ceiling: reals -> integers,
      floor: reals -> integers,
     entier: reals -> integers;
```

are mapped to mathematical functions

$$\lceil\cdot\rceil : \text{carrier}(\mathbb{R}) \longrightarrow \text{carrier}(\mathbb{Z}),$$

$$\lfloor\cdot\rfloor : \text{carrier}(\mathbb{R}) \longrightarrow \text{carrier}(\mathbb{Z}),$$

$$[\cdot] : \text{carrier}(\mathbb{R}) \longrightarrow \text{carrier}(\mathbb{Z}).$$

[6] The `where` clause in TECTON is similar to Hilbert's ι-quantor.

A structure consists of a pair: a family of carriers indexed by sorts and a family of functions indexed by function descriptors. A concept denotes a family of structures that all satisfy a common set of requirements. In the case of the structure $\mathbb{R}$ and structure $\mathbb{Z}$ we are used to thinking of single, concrete structures, a notion that is the exception rather than the rule in TECTON.

Another important point about TECTON semantics is that a structure denoted by a concept is not restricted to the carriers and functions introduced and available to the concept but has in general additional sorts and function descriptors. The earlier examples of binary relations and order relations provide illustrations of this point.

The central property of the reals is expressed by the concept `Continuous -order`. We trace the concept `Real` in the concept graph and drop everything not related to order relations.

```
Definition: Real
  uses Continuous-order [with reals as domain],
  ...
```

The `uses` clause imports the concept instance `Continuous-order` in which the sort `reals` is substituted for sort `domain`. The concept instance thus becomes a *part* of concept `Real`. In its own definition it imports sorts, function descriptors and their defining properties from the concept `Total-order` by a `refines` clause.

```
Definition: Continuous-order
  refines Total-order; uses Set;
  introduces Dedekind-cut: sets x domain x sets -> bool;
  requires (for S, T: sets; z: domain) Dedekind-cut(S, z, T) =
    (for x, y: domain) x in S and y in T implies  x < z and z < y,
    ((for z: domain) (for some S, T: sets) Dedekind-cut(S, z, T)).
```

The function descriptor introduced here formalizes the notion of a Dedekind cut. The first requirement defines a three place predicate for element `z`, the *cut*, separating the two sets `S` and `T`. The `for` keyword denotes the universal, and the `for some` keyword the existential quantifier.

The second formula requires all elements of a continuous order to be definable as cuts. As one sees, the definition is not constructive and uses second order variables `S` and `T` for sets of elements of the underlying domain.

3.3 STL Formalizations in Tecton

The examples of the preceding section are all of familiar mathematical concepts. In this section we how briefly how TECTON can be used to formalize computer science concepts. To this end we show a few examples drawn from a TECTON formal specification of the C++ STL container and iterator concepts [16]. In writing these specifications our starting point was the set of semi-formal concept descriptions given in [5], which is already organized as a concept hierarchy and is more complete in its statement of semantics than the actual C++ standard [6]. For expressing state changes our structures include functions with state domains and ranges, leaving the direct expression of imperative constructs to an algorithmic level in SuchThat [18]. For an alternative approach to specifying formally the semantics of STL, see [22]; in that approach, using Object Oriented Abstract State Machines, an operational style is used at the most basic semantic level.

The first example is a definition the `Back-insertion-sequence` concept as a refinement of the `Sequence` concept (not shown here).

```
Definition: Back-insertion-sequence
refines Sequence;
introduces
  back(nonempty-sequences) -> objects,
  push-back(sequences, objects) -> nonempty-sequences,
  pop-back(nonempty-sequences) -> sequences;
requires (for s: sequences; s1: nonempty-sequences;
              x: objects; i: iterators)
  back(s1) = s1*i where s1++i = end(s1),
  access(back(push-back(s, x))) = access(x),
  pop-back(push-back(s, x)) access-equivalent-to s.
```

Sorts `sequences` and `nonempty-sequences` have already been introduced in the `Sequence` concept, and `objects` and `iterators` in an ancestor of `Sequence`. For some of the functions introduced with those concepts, we are forced for technical reasons to depart from the exact syntax used in C++; here, the dereferencing operator `*` and the increment operator `++` are explicitly binary operations on a sequence and an iterator, whereas in C++ the sequence storage argument is implicit.

The final example is extensive and we lack space to explain it in detail, but the reader will note how a number of semantic issues that are only implicitly present in previous decriptions of STL are here made explicit and therefore subject to formal reasoning, and in turn, correct implementation.

```
Definition: Vector
  refines
    Random-access-container [with vectors as containers,
      nonempty-vectors as nonempty-containers],
    Back-insertion-sequence [with vectors as sequences,
      nonempty-vectors as nonempty-sequences];
  introduces
    capacity(vectors) -> naturals,
    reserve(vectors, naturals) -> vectors,
    usable(vectors, iterators) -> bool;
  generates
    vectors freely using construct, push-back;
  requires
    (for v, v1: vectors; w: nonempty-vectors; n: naturals;
         i, i1, j, k: iterators; e: elements)
```

```
capacity(v) >= size(c),
capacity(reserve(v, n)) >= n,
v1 = reserve(v, n)
  implies size(v1) = size(v) and v1 access-equivalent-to v,
n <= capacity(v) implies reserve(v, n) = v,
usable(v, begin(v)),
usable(v, end(v)),
valid(range(v, i, j))
    and usable(v, i) and usable(v, j) and k in range(v, i, j)
  implies usable(v, k),
i in range(w, begin(w), end(w))
  implies not(usable(erase(w, i), end(w))),
i in range(w, begin(w), end(w))
    and j in range(w, begin(w), end(w))
    and position(w, i) < position(w, j)
    and n = size(range(w, i, j))
    and k in range(w, end(w) - n, end(w))
  implies not(usable(erase(w, i, j), k)).
```

4 Availability Checking

Most programming languages allow the declaration of identifiers to state their intended use. In compiler technology the checking of whether declaration and use of identifiers agree is called *static semantics* (despite the fact that declaration and use can be expressed by a context sensitive grammar which reveals the problem as an advanced syntactic one). Since TECTON extends the specification of identifiers considerably its static semantics becomes much more elaborate. In addition, TECTON's aim at a natural style of specification leaves lots of technical information implicit when it can be understood and retrieved from the context. Let us assume that an identifier i is used in concept C_k which imports concept C_{k-1}, which in turn imports recursively all the way up to concept C_0 where the identifier i was introduced. Let us compare with respect to declarations TECTON concepts with JAVA packages. In TECTON identifier i stands alone for itself leaving the concept inheritance implicit whereas in JAVA one has to write

$$C_0.\cdots.C_{k-1}.C_k.i.$$

Since TECTON concepts are more fine grained than JAVA packages the problem is compounded: after 125 TECTON concept definitions we observed an average k of 12 for the length of the inheritance chain of concepts. The problem of regaining the suppressed information is called the *availability* problem in TECTON. Implementing availability checking was a challenging problem and required the development of new algorithms whose details will be reported elsewhere. Here we only report a few empirical measurements.

The following table summarizes measurements of two files of TECTON concepts. The first, *stltec*, formalizes all STL container and iterator concepts [16].

The second, *algtec* [17], formalizes a fragment of commutative algebra; see [2] for a motivation.

	STL	Algebra
number of concepts	76	125
number of superiors	117	185
number of parts	48	54
number of sorts	56	125
number of function descriptors	141	167
number of available sorts	181	179
number of available functions	141	155
number of sort replacements	47	87
number of function replacements	18	33
number of concept replacements	14	26
lines of input	823	993
used heap size (MB)	3.68	2.81
total heap size (MB)	4	4
total time (s)	3.04	0.99

5 Towards a Standard Concept Library

Experience with support of genericity by concept specifications leads us to propose developing an organized way to establish concept libraries. Although it is not meaningful to standardize generic programming in general, it is important to formalize the most commonly used notions, much as was done with mathematical definitions, e.g., `Strict-partial-order` or `Ring`, which are commonly accepted definitions and conventions everybody adheres to.

We believe that the wide acceptance of the STL is not only due to the technical quality of its implementations in an advanced language but also to a great extent due to its conceptual simplicity and rigor. The iterator and container abstractions have found applications and generalizations outside of the STL. It would be a great achievement if this could be based on precisely defined concepts and formal specifications.

Other important aspects of generic programming could also gain from generally agreed upon concept libraries. If a textbook of a scientific discipline can be formalized by an elaborated set of concepts with related lemmas, one has a non-trivial basis for the verification of programs in that application area and does not have to wait until automatic theorem provers reach that level of sophistication. Verifications of existing programs and conceptions of new generic ones can benefit from concept libraries.

Acknowledgments

We thank our students Quincy Stokes, Holger Gast, Peter Hrenka, Wolfgang Maier, Oriel Maute, Tobias Döhler, Christoph Schwarzweller and Uwe Kreppel for contributing ideas and programs to the project, and our colleagues Deepak Kapur and Alexander Stepanov for long lasting cooperation on TECTON.

References

1. M. H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library.* Addison-Wesley, 1998.
2. Bruno Buchberger, George E. Collins, and Rüdiger Loos. *Computer Algebra, Symbolic and Algebraic Computation.* Springer-Verlag, 1982.
3. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, Dec 1985.
4. George E. Collins. Algebraic algorithms. CS801 Lecture Notes, University of Wisconsin, 1971.
5. Silicon Graphics. Standard template library programming guide. `http://www.sgi.com/Technology/STL/`, 1998.
6. International Organization for Standardization (ISO). *ISO/IEC Final Draft International Standard 14882: Programming Language C++*, 1998.
7. Deepak Kapur and David Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report 92-20, RPI, Troy, 1992.
8. Deepak Kapur, David Musser, and Alexander Stepanov. Operators and algebraic structures. In *Proc. of Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*, Oct 1981.
9. Deepak Kapur, David R. Musser, and Alexander A. Stepanov. Operators and algebraic structures. In *Proc. of Conference on Functional Programming Languages and Computer Architecture.* ACM, 1981.
10. Deepak Kapur, David R. Musser, and Alexander A. Stepanov. Tecton: A language for manipulating generic objects. In J. Staunstrup, editor, *Proc. of Program Specification Workshop*, volume 134 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1982.
11. Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 3 edition, 1998.
12. David Musser and Atul Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.
13. David Musser and Alexander Stepanov. A library of generic algorithms in Ada. In *Proc. ACM SIGAda Conference, Boston*, pages 9–11, Dec 1987.
14. David Musser and Alexander Stepanov. Algorithm-oriented generic libraries. *Software-practice and experience*, 27(7):623–642, Jul 1994.
15. David R. Musser. The Tecton Concept Description Language. `http://www.cs.rpi.edu/~musser/gp/tecton/tecton1.ps.gz`, July 1998.
16. David R. Musser. Tecton description of STL container and iterator concepts. `http://www.cs.rpi.edu/~musser/gp/tecton/container.ps.gz`, August 1998.
17. David R. Musser, Sibylle Schupp, Christoph Schwarzweller, and Rüdiger Loos. Tecton Concept Library. Technical Report WSI-99-2, Fakultät für Informatik, Universität Tübingen, January 1999.
18. Sibylle Schupp. *Generic Programming*—SUCHTHAT *One Can Build an Algebraic Library.* PhD thesis, Universität Tübingen, 1996.
19. Sibylle Schupp and Rüdiger Loos. SUCHTHAT—Generic programming works. In this volume.

20. Alexander Stepanov and Meng Lee. The Standard Template Library. Technical report, HPL-94-34, April 1994. revised July 7, 1995.
21. Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 13, pages 677–788. Elsevier Science Publishers, 1990.
22. A. V. Zamulin. Language independent container specification. In this volume.

Generative Programming and Active Libraries
Extended Abstract

Krzysztof Czarnecki[1], Ulrich Eisenecker[2], Robert Glück[3],
David Vandevoorde[4], and Todd Veldhuizen[5]

[1] DaimlerChrysler AG (Research and Technology, Ulm) czarnecki@acm.org
[2] Fachhochschule Heidelberg Ulrich.Eisenecker@t-online.de
[3] University of Copenhagen (Dept. of Computer Science) glueck@diku.dk
[4] Edison Design Group daveed@vandevoorde.com
[5] Indiana University (Computer Science Dept.) tveldhui@acm.org

Abstract. We describe *generative programming*, an approach to generating customized programming components or systems, and *active libraries*, which are based on this approach. In contrast to conventional libraries, active libraries may contain metaprograms that implement domain-specific code generation, optimizations, debugging, profiling and testing. Several working examples (Blitz++, GMCL, Xroma) are presented to illustrate the potential of active libraries. We discuss relevant implementation technologies.

1 Introduction

The main goal of generic programming is to improve reusability by providing parameterized components which can be *instantiated* for different choices of parameters. The C++ Standard Template Library [24] is remarkable because generic algorithms can be orthogonally combined with generic container representations. The aims of *generative programming* resemble those of generic programming but differ in several important aspects that can enhance the power of generic programming.

Generative Programming (Sect. 2) is about modeling families of software systems by software entities such that, given a particular requirements specification, a highly customized and optimized instance of that family can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge (see [6,9]). Generative Programming involves all phases of the software development process, from the specification of the abstract software entities, to the design and implementation of generative software systems, and the mapping of these onto hardware within constraints (such as space and speed).

We refer to libraries which apply concepts of generative programming as *Active Libraries* (Sect. 3). Active libraries require implementation technologies beyond what is provided by traditional compilers (Sect. 3.2). To illustrate these ideas, we present working examples: Blitz++ and the Generative Matrix Computation Library (GMCL) (Sect. 4), which are C++ libraries that generate customized components; the Tau package (Sect. 5), an extensible profiling tool; and the Xroma system (Sect. 6) for extensible compilation.

2 Generative Programming: Goals and Principles

The goal of modern programming is often to design *domain-specific abstractions* and implement these abstractions using some programming language. In providing these abstractions, we have to deal with two major problems: (1) domain-specific knowledge may get lost in the implementation because there exists a semantic gap between domain-specific abstractions and features offered by a programming language; (2) flexibility and generality in the design may incur considerable performance penalties at runtime.

Generative programming (GP) is an approach to generating customized components and systems. The goals are to (a) decrease the conceptual gap between domain concepts and program code (known as achieving high *intentionality*), (b) achieve high reusability and adaptability, (c) simplify managing many variants of intermediate and end-products, and (d) increase efficiency (both in space and execution time). To meet these goals, GP deploys several principles:

Parameterization of differences: As in generic programming, parameterization allows us to represent—in a compact way—families of components (i.e. components with many commonalities).

Analysis and modeling of dependencies and interactions: Not all parameter value combinations are valid, and the values of some parameters may imply the values of some other parameters. These dependencies are referred to as *horizontal configuration knowledge*, since they occur between parameters at one level of abstraction.

Overhead elimination and domain-specific optimizations: By generating components statically (at compile time), much of the overhead due to unused code, run-time checks and unnecessary levels of indirection may be eliminated. Complicated domain-specific optimizations may also be performed (for example, loop transformations for scientific codes).

Separating problem and solution spaces: The problem space consists of the application-oriented concepts and features in terms of which application designers would like to express their needs, whereas the solution space contains elementary, reusable implementation components (e.g., generic components) which may be combined, analyzed and transformed by metaprograms in order to produce the desired software entity. Configuration knowledge is used to map from the problem space to the solution space. Configuration knowledge may capture specific information about default settings and parameter dependencies, illegal feature combinations, specific construction and optimization rules. The distinction between problem space, solution space and configuration knowledge allows the specification of systems by abstract features (while generic programming expects parameters that instantiate a concrete component).

Separation of concerns: This term, coined by Dijkstra, refers to the importance of dealing with one important issue at a time. To avoid program code which deals with many issues simultaneously, generative programming tries to separate each issue into a distinct set of code. These pieces of code are then combined to generate a needed component. This idea is borrowed from Aspect-Oriented Programming [21].

2.1 Towards Generative Programming

There are three other programming paradigms which have goals similar to those of Generative Programming: Generic Programming, Domain-Specific Languages (DSLs), and Aspect-Oriented Programming (AOP). Generative Programming is broader in scope than these, but borrows important ideas from each:

Generic Programming may be summarized as “reuse through parameterization.” Generic programming allows components which are extensively customizable, yet retain the efficiency of statically configured code [24]. This technique can eliminate dependencies between types and algorithms that are not necessary. For example, iterators allow generic algorithms which work efficiently on both dense and sparse matrices [28].

However, generic programming limits code generation to substituting concrete types for generic type parameters, and welding together pre-existing fragments of code in a fixed pattern. It does not allow generation of completely new code. *Generative* programming is more general because it provides automatic configuration of generic components from abstract specifications, and a more powerful form of parameterization. For example, automatic configuration allows abstract parameters such as an optimization flag (e.g., *speed*, *space*, or *accuracy*). Such parameters map to configurations of implementation components rather than just single parameter components.

Domain-Specific Languages (DSLs) provide specialized language features that increase the abstraction level for a particular problem domain; they allow users to work closely with domain concepts, at the cost of language generality. Domain-specific languages range from widely-used languages for numerical and symbolic computation (e.g., Mathematica) to less well-known languages for telephone switches and financial calculations (to name just a few application domains). DSLs are able to perform domain-specific optimizations and error checking. On the other hand, traditional DSLs typically lack support for generic programming. Shortcomings of traditional DSLs may be overcome by embedding DSLs in existing languages, as advocated in [18].

Aspect-Oriented Programming. Most current programming methods and notations concentrate on finding and composing functional units, which are usually expressed as objects, modules and procedures. However, several properties such as error handling and synchronization cannot be expressed cleanly and locally using current (e.g., object-oriented) notations and languages. Instead, they are expressed by small code fragments scattered throughout several functional components.

Aspect-Oriented Programming (AOP) [21] decomposes problems into functional units and *aspects* (such as error handling and synchronization). In an AOP system, components and aspects are *woven* together to obtain a system implementation that contains an intertwined mixture of aspects and components (called *tangled* code). Weaving can be performed at compile time (e.g., using a compiler or a preprocessor) or at runtime (e.g., using dynamic reflection). In any case, weaving requires some form of metaprogramming (see [4]). Generative programming has a wider scope that includes automatic configuration and generic programming techniques, and provides new ways of interacting with the compiler and development environment.

Putting It Together: Generative Programming. The concept of generative programming encompasses properties of the previous three paradigms, as well as some additional techniques to achieve the goals listed in Sect. 2:

- DSL techniques are used to improve intentionality of program code, and to enable domain-specific optimizations and error checking.
- AOP techniques are used to achieve *separation of concerns* by isolating aspects from functional components.
- Generic Programming techniques are used to parameterize over types, and iterators are used to separate out data storage and traversal aspects.
- Configuration knowledge is used to map between the problem space and solution space. Different parts of the configuration knowledge can be used at different times in different contexts (e.g., compile time or runtime or both).

3 Active Libraries

As noted in the previous section, Generative Programming requires metaprogramming for weaving and automatic configuration. Supporting domain-specific notations may require syntactic extensions and non-textual and interactive representations. Libraries based on Generative Programming ideas thus need both implementation code, and *metacode* which can implement syntax extensions and rendering of textual and non-textual program representations, perform code generation, and apply domain-specific optimizations.[1]

Active libraries are not passive collections of routines or objects, as are traditional libraries, but take an active role in generating code. Active libraries provide abstractions and can optimize those abstractions themselves. They may generate components, specialize algorithms, optimize code, automatically configure and tune themselves for a target machine, check source code for correctness, and report compile-time, domain-specific errors and warnings. They may also describe

[1] We are stretching the traditional meaning of *library*; we do not imply a specific packaging technology.

themselves to tools such as profilers and debuggers in an intelligible way or provide domain-specific debugging, profiling, and testing code themselves. Finally, they may contain code for rendering domain-specific textual and non-textual program representations and for interacting with such representations.

This perspective forces us to redefine the conventional interaction between compilers, libraries, and applications. Active Libraries may be viewed as knowledgeable agents, which interact with each other to produce concrete components. Such agents need infrastructure supporting communication between agents, code generation and transformation, and interaction with the programmer.

3.1 Types of Active Libraries

Active libraries possess different levels of sophistication, characterized by the type of metaprogramming used: (i) active libraries that extend a compiler, (ii) active libraries that extend the environment to provide domain-specific tool support, and (iii) active libraries that contain *metacode* to analyze and transform the domain-specific concepts at different times and in different contexts.

Extending a Compiler. Active libraries may extend a compiler by providing domain-specific abstractions with automatic means of producing optimized program code. They may compose and specialize algorithms, automatically tune code for a target machine and instrument code. Both Blitz++ and the Generative Matrix Computation Library are examples of libraries that extend a compiler (Sect. 4).

Domain-Specific Tool Support. Active libraries may extend the programming environment to provide domain-specific debugging support, domain-specific profiling and code analysis capabilities, and so on. An example is the interaction between Tau and Blitz++ (Sect. 5). An example of a programming platform which supports these ideas as well as Extended Metaprogramming described below is the Intentional Programming (IP) system [29].

Extended Metaprogramming. Active Libraries may contain *metacode* which can be executed to compile, optimize, adapt, debug, analyze, visualize, and edit the domain-specific abstractions. Active libraries may generate different code depending on the deployment context; for example, they may query the hardware and operating system about their architecture. Furthermore, the same metacode may be used at different times and in different contexts; for example, based on the compile-time knowledge of some context properties which remain stable during runtime, some metacode may be used to perform optimizations at compile time and other metacode may be injected into the application to allow for optimization and reconfiguration at runtime.

3.2 Implementation Technologies

Active Libraries require languages and techniques which open up the programming environment. Issues that need to be addressed when constructing active libraries are language support, transformation and analysis, and tool interfacing. We survey these issues in this section. We list those technologies which we believe are most relevant at the time of this writing and which have been put to good use in existing generators.

Language Support. A key issue in the construction of active libraries is support for generative tasks by the implementation language. Generative programming is a novel concept and existing languages are not well-equipped for it.

C++ Templates The C++ language includes some compile-time processing abilities quite by accident, as a byproduct of template instantiation. Nested templates allow compile-time data structures to be created and manipulated, encoded as types; this is the basis of the expression templates technique [33]. The template metaprogram technique [34] exploits the template mechanism to perform arbitrary computations at compile time; these “metaprograms” can perform code generation by selectively inlining code as the “metaprogram” is executed. This technique has proven a powerful way to write code generators for C++.

Extensible compilation and Metalevel processing In metalevel processing systems, library writers are given the ability to directly manipulate language constructs. They can analyze and transform syntax trees, and generate new source code at compile time. The MPC++ metalevel architecture system [19] provides this capability for the C++ language. MPC++ even allows library developers to extend the syntax of the language in certain ways (for example, adding new keywords). Other examples of metalevel processing systems are Open C++ [2], Magik [10], and Xroma (Sect. 6). An important differentiating factor is whether the metalevel processing system is implemented as a pre-processor, an open compiler, or an extensible programming environment (e.g., IP [29]). A potential disadvantage of metalevel processing systems is the complexity of code which one must write: modern languages have complicated syntax trees, and so code which manipulates these trees tends to be complex as well. The situation can be improved by replacing the direct manipulation of syntax trees with a declarative metalanguage for manipulating code. For example, the FOG system [37] provides such a metalanguage for C++.

Meta-languages Program generation is a specific metacomputation task that stretches the capabilities of usual programming languages. Few programming languages support code manipulation and transformation. Design and implementation of such generative programming languages requires novel concepts (e.g., type systems [31]). An important concept that stems from the area of partial evaluation is that of two-level (or more generally, multi-level) programming languages. Two-level languages contain static code (which is evaluated at compile-time) and dynamic code (which is compiled, and later executed at run-time). Multi-level languages [15] can provide a simpler approach to writing program generators (see for example, the Catacomb system [30]).

Program Transformation. Another key issue in the construction of active libraries is the use of program analysis and transformation to optimize and weave the implementation code. Exactly *when* these activities take place depends on the binding-times of the library components and the abstract specifications. Three fundamental operations are the essence of a wide spectrum of automatic transformation methods [16]: program specialization, program composition, and program inversion. Here we shortly discuss program specialization, program composition, and the transformation of programs at run-time.

Program Specialization provides means to tailor generic and highly parameterized components to specific needs and applications. One of the best developed specialization techniques is *partial evaluation* [1,8,20]. An extensive theory and literature on specialization and code generation was developed in this field.

An important discovery was that the concept of *generating extensions* [12] unifies a wide class of apparently different program generators. Examples include parsing, translation, theorem proving, and pattern matching [14]. Through partial evaluation, components which handle variability at run-time can be transformed into *component generators* (or *generating extensions* in the terminology of the field) which handle variability at compile-time [23]. The *Futamura projections* [13] provide the theoretical cornerstone for this technique.

In some cases, such transformations avoid the need for library developers to work with complex meta-level processing systems. Automatic tools for turning a general component into a component generator (or *generating extension*) now exist for various programming languages such as Prolog, Scheme, and C (see [20]).

Program Composition The construction of software by sharing and combining existing implementation components is a main activity when producing customized software. Unfortunately, program hierarchies and modularity do not come for free: they add intermediate data structures, redundant computations, interface code and error checking. Program composition techniques (e.g., [32,36]) can remove such redundancies, and allow fusing components without paying an unacceptably high price.

Runtime Code Generation systems allow libraries to generate customized code at run-time. This makes it possible to perform optimizations which depend on information not available until run-time, for example, the structure of a sparse matrix or the number of processors in a parallel application. Examples of such systems which generate native code are `C (Tick-C) [11,26], Fabius [22], Tempo-C [3], and DyC [17]. Code generation speeds as high as 6 cycles per generated instruction have been achieved. Runtime code modification can also be achieved using dynamic reflection facilities available in languages such as Smalltalk and CLOS.

Interfacing. The concept of active libraries requires us to redefine the interaction of traditional programming tools. We mentioned self-instrumentation of active libraries, which allows new possibilities for profiling, debugging, and program analysis.

Extensible Tools To provide domain-specific support in programming environments, we need tools that provide hooks for libraries to define customized debugging support, profiling, etc. Examples of such tools are Tau (Sect. 5) and the Intentional Programming system [29].

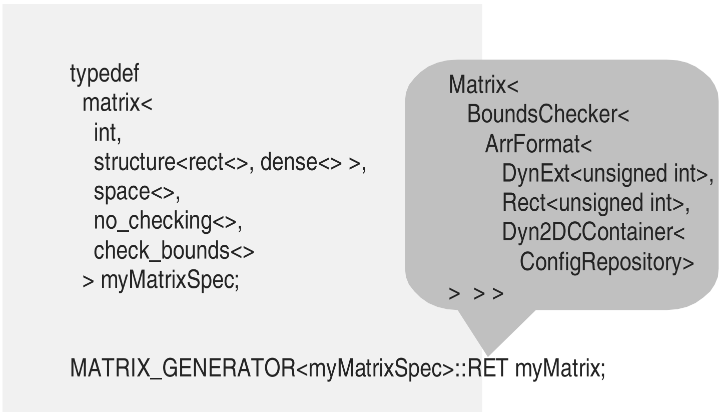
4 Examples: Active Library Extending a Compiler

Recently there have been several projects in scientific computing which fit the description of active libraries. This trend is a result of two main factors: a desire for high-level abstractions which closely model the problem domain, and the inability of traditional compilers to optimize such abstractions. We describe two such libraries here.

Blitz++ The Blitz++ library [35] provides array objects for C++ similar to those in Fortran 90. The largest performance problem for arrays in C++ has been temporaries which result from overloaded operators. Blitz++ solves this problem using the *expression templates* technique [33], which allows it to generate custom evaluation kernels for array expressions. The library performs many loop transformations (tiling, reordering, collapsing, unit stride optimizations, etc.) which have previously been the responsibility of optimizing compilers. Blitz++ generates different code depending on the target architecture. The *template metaprogram* technique [34] is used to generate specialized algorithms for operations on small vectors and matrices.

The Generative Matrix Computation Library (GMCL) is able to generate matrix components with a selected combination of features such as element types (real numbers), density (dense and sparse), storage formats (row- and column-wise, several sparse formats), memory allocation (dynamic and static), error checking (bounds, compatibility, memory allocation), and operations (addition, subtraction, multiplication). Only some combinations of these features are valid; these are specified by a configuration DSL. A configuration DSL has the form of a parameter space, where the component to be configured has a number of parameters and the parameter values can be parameterized themselves. A particular configuration expression is translated into a concrete configuration of generic components. Currently there exist two implementations of the GMCL, one in the Intentional Programming system and one in C++

(see [25,4] for details). The C++ GMCL makes widespread use of expression templates [33], generative programming idioms in C++ [9,4], and many template metaprogramming facilities, e.g., control structures for static metaprogramming [5]. Figure 1 demonstrates the translation of a matrix configuration expression into a configuration of generic components in C++ GMCL. The translation is performed by a template metaprogram. GMCL contains another generator for generating efficient implementations of matrix expressions (e.g., “(A+B)*(C+D)”). This generator reads out the properties of the operands from their configuration repositories (Figure 1).



```
typedef
  matrix<
    int,
    structure<rect<>, dense<> >,
    space<>,
    no_checking<>,
    check_bounds<>
  > myMatrixSpec;
```

```
Matrix<
  BoundsChecker<
    ArrFormat<
      DynExt<unsigned int>,
      Rect<unsigned int>,
      Dyn2DCContainer<
        ConfigRepository>
> > >
```

```
MATRIX_GENERATOR<myMatrixSpec>::RET myMatrix;
```

Fig. 1. Sample C++ code specifying a rectangular dense matrix with bounds checking. The generated matrix type is shown in the gray area

The C++ implementation of the matrix component [25] comprises 7500 lines of C++ code (6000 lines for the configuration generator and the matrix components and 1500 lines for the operations). The matrix configuration DSL covers more than 1840 different kinds of matrices. Despite the large number of supported matrix variants, the performance of the generated code is comparable with the performance of manually coded variants. This is achieved through exclusive use of static binding, which is often combined with inlining.

5 Example: Active Library for Domain-Specific Tool Support

Tau [27] is a package for tuning and analysis of parallel programs. Unlike most profilers, Tau allows libraries to instrument themselves. Self-instrumentation solves some important problems associated with tracing and profiling large libraries. Libraries often have two (or more) layers: a user-level layer, and layers of internal implementation routines. Automatic instrumentation mixes these layers, which may result in a swamp of hard-to-interpret information. The problem is compounded when template libraries (with their long symbol names and many template instances) are used. With self-instrumentation such as Tau offers, active libraries can present themselves to profilers and tracers in an understandable way: time spent in internal routines can be properly attributed to the responsible interface routines, and complex symbol names (especially template routines) can be rewritten to conform to a user's view of the library. For example, Blitz++ uses pretty-printing to describe expression templates kernels to Tau. When users profile applications, they see the time spent in array expressions such as "A=B+C+D", rather than incomprehensible template types and runtime library routines.

6 Xroma: Extensible Translation

Xroma (pronounced Chroma) is a system being developed by one of us (Vandevoorde) specifically to support the concept of active libraries. We describe it here to illustrate in detail an extensible compilation system and metalevel processing.

The Xroma system provides an API for manipulating syntax trees, and a framework which allows library developers to extend the compilation process (Figure 2). Syntax trees which can be manipulated are called *Xromazene*. The parser of the Xroma language produces well-specified Xromazene that is fed to the Xroma translation framework. The grammar of the Xroma language is static: it is not possible to add new syntax structures. Instead, the translation is extensible by defining new Xromazene transformations. The Xroma language describes both the transformations and the code upon which they act.

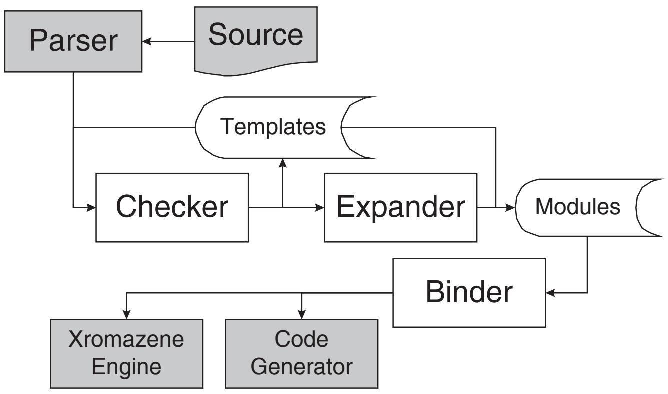


Fig. 2. General organization of Xroma

Libraries in Xroma are composed of *modules*, which are collections of types, procedures and templates. Modules may contain *active components* which are able to generate or transform code.

Xroma Components. The Xroma system exposes several components: a *checker*, an *expander*, a module manager, a template manager and a *binder*. Active components can attach themselves to these Xroma components. The output of the binder is a Xromazene file that can be interpreted on a Xromazene engine or fed to a traditional optimizing code generator. The binder can activate modules e.g., to implement domain-specific global optimizations. The checker's role is to verify semantic consistency of types, procedures and templates. If any of these elements has an active component attached, that component can take over control of checking. For example, it could relax the type-matching requirements between a call site and the available procedure declarations (overload set), or ensure early enforcement of template instantiation assumptions. The expander is a second pass that must reduce application-specific nodes in the syntax tree to the generic set of nodes understood by the binder. Often, an active component does not want to take control of the translation of a whole procedure, type or template. Instead, it is usually only desirable to intercept specific constructs. The Xroma system allows these events to be captured by active libraries at checker or expander time: type, procedure and template definitions; call-sites (before and after overload resolution); object definitions (constructor calls), and template instantiation.

Xroma Example. The following example demonstrates a miniature active library that enforces a constraint on template type-arguments: the type for which the template is instantiated should not be a reference type. Two modules are presented: the first one makes use of the active library, and the second implements the library. The first module defines a template type `Value` whose parameter may only be a non-reference type; this is enforced by the `NotRef` component (defined later):

```
module["program"] AnnotationDemo {       // 1
   synonym module XD = XromasomeDemo;    // 2

   // Declare the Value type             // 3
   export template[type T]               // 4
   type[XD::NotRef(0)] Value {           // 5 Annotation-brackets
      export proc init(ref rdonly T);    // 6
      var T v_;                          // 7
   }                                     // 8

   // Example uses of Value:             // 9
   proc init() {                         // 10 Module initialization
      var Value[int[32]] a;              // 11 Okay
      ref Value[int[32]] b;              // 12 Okay
      var Value[ref int[32]] c;         // 13 Error!
   }                                    // 14
}                                       // 15 End program module
```

The keyword `type` (line 5) is similar to the C++ keyword `class`. Square brackets are used to denote template parameters (line 4), template arguments (lines 11-13) and annotations. Annotations (line 5) allow the Xroma programmer to attach active components to types and procedures. In the example above, this component is `NotRef(0)` where “0” indicates that the first template argument should be checked. The Xroma translation system attaches this annotation to the template and executes the associated active component every time an attempt is made to instantiate the template (lines 11-13). The special procedure `init()` (lines 6,10) corresponds to object constructors and module initializers.

Here is the definition of the `NotRef` module:

```
module XromasomeDemo {                                 // 1
   synonym module XT = Xroma::Translator;              // 2

   export type NotRef extends XT::Xromasome {          // 3
      export proc init(int[32] a) a_(a) {}             // 4
      virtual phase()->(XT::Phase request)             // 5
         { request.init(XT::kGenerate); }              // 6
      virtual generate(ref XT::Context n);             // 7
      var int[32] a_; // Template argument number      // 8
   }                                                   // 9

   virtual NotRef:generate(ref XT::Context n) {        // 10
      ref t_spec = convert[XT::TemplSpec](n.node());   // 11
      if t_spec.arg(a_).typenode()==null {             // 12
         XT::fatal(&n, "Unexpected non-type arg");     // 13
      } else if t_spec.arg(a_).typenode().is_ref() {   // 14
         XT::fatal(&n, "Unexpected ref type");         // 15
      } else { XT::Xromasome::generate(&n); }          // 16 Call base
   }                                                   // 17
} // End module XromasomeDemo                          // 18
```

Active components are implemented as objects that respond to a Xromasome interface provided by the framework. When the active component attaches itself to some aspect of the Xroma system, the method `phase()` (line 5) is invoked to determine at which stage the Xromasome wishes to take control. When that stage is encountered for the construct to which the Xromasome was attached, another method `generate()` (line 10) for template instantiation is invoked with the current context of translation. In our example the template argument is retrieved from the Xromazene, and after verification that it is a non-reference type argument, the default instantiation behavior accessible through the Xromasome interface is called.

7 Conclusion and Future Work

We presented an approach to generating software components which we call Generative Programming, and described Active Libraries, which implement this approach. Our approach has several advantages over conventional, passive libraries of procedures or objects: it allows the implementation of highly-parameterized and domain-specific abstractions without inefficiency. Even though we presented examples from the area of scientific computing and extensible translation, our approach should be applicable to other problem domains (business components, network applications, operating systems, etc.).

In summary, the ability to couple high-level abstractions with code transformation and implementation weaving enables the development of highly intentional, adaptable and reusable software for generating highly optimized applications or components. However, it is also clear that more work will be required, including research on generative analysis and design methods (e.g., the use of Domain Analysis and OO methods [7]), development of metrics and testing techniques for generative programs, further research on integrating ideas from related areas (e.g., domain-specific languages, generators, automatic program transformation and optimization of programs, metaobject protocols, and Domain Engineering), and, finally, the development of industrial strength generative programming tools.

References

1. D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, Amsterdam, 1988.
2. S. Chiba. A Metaobject Protocol for C++. In *OOPSLA'95*, pages 285–299, 1995.
3. C. Consel, L. Hornof, F. Nöel, J. Noyé, and E. N. Volanschi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [8], pages 54–72.
4. K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. PhD Thesis.* Technical University of Ilmenau, Ilmenau, 1998.
5. K. Czarnecki and U. Eisenecker. Meta-control structures for template metaprogramming. `http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm`.
6. K. Czarnecki and U. Eisenecker. Components and generative programming. In O. Nierstrasz, editor, *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 99, Toulouse, Frankreich, September 1999).* Springer-Verlag, 1999.
7. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications.* Addison-Wesley Longman, 1999. (to appear).
8. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
9. U. Eisenecker. Generative Programming (GP) with C++. In H. Mössenböck, editor, *Modular Programming Languages*, volume 1024, pages 351–365. Springer-Verlag, 1997.

10. D. R. Engler. Incorporating application semantics and control into compilation. In *USENIX Conference on Domain-Specific Languages (DSL'97)*, October 15–17, 1997.
11. D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *POPL'96*, pages 131–144, 1996.
12. A. P. Ershov. On the essence of compilation. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.
13. Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
14. Y. Futamura. Partial computation of programs. In E. Goto, K. Furukawa, R. Nakajima, I. Nakata, and A. Yonezawa, editors, *RIMS Symposia on Software Science and Engineering*, volume 147 of *Lecture Notes in Computer Science*, pages 1–35, Kyoto, Japan, 1983. Springer-Verlag.
15. R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.
16. R. Glück and A. V. Klimov. Metacomputation as a tool for formal linguistic modeling. In R.Trappl, editor, *Cybernetics and Systems '94*, volume 2, pages 1563–1570, Singapore, 1994. World Scientific.
17. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, pages 163–178. ACM, June 1997.
18. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196–196, Dec. 1996.
19. Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and implementation of metalevel architecture in C++ – MPC++ approach. In *Reflection'96*, 1996.
20. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).
21. G. Kiczales and et al. Home page of the Aspect-Oriented Programming Project. `http://www.parc.xerox.com/aop/`.
22. M. Leone and P. Lee. Lightweight run-time code generation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994.
23. R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering*, pages 183–192, Lake Tahoe, Nevada, 1997. IEEE Computer Society.
24. D. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, Reading (MA), USA, 1996.
25. T. Neubert. *Anwendung von generativen Programmiertechniken am Beispiel der Matrixalgebra.* Technische Universitet Chemnitz, 1998. Diplomarbeit.
26. M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *PLDI'97*, 1996.
27. S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable profiling and tracing for parallel, scientific applications using C++. In *Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134–145, August, 1998.

28. J. G. Siek and A. Lumsdaine. *Modern Software Tools in Scientific Computing*, chapter A Modern Framework for Portable High Performance Numerical Linear Algebra. Birkhauser, 1999.
29. C. Simonyi and et. al. Home page of the Intentional Programming Project. `http://www.research.microsoft.com/research/ip/`.
30. J. Stichnoth and T. Gross. Code composition as an implementation language for compilers. In *USENIX Conference on Domain-Specific Languages*, 1997.
31. W. Taha, Z.-E.-A. Benaissa, and T. Sheard. Multi-stage programming: axiomatization and type-safety. In *International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, Aalborg, Denmark, 1998. Springer-Verlag.
32. V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3):292–325, 1986.
33. T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
34. T. L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
35. T. L. Veldhuizen. Arrays in Blitz++. In *ISCOPE'98*, volume 1505 of *Lecture Notes in Computer Science*, 1998.
36. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
37. E. Willink and V. Muchnick. Weaving a way past the C++ One Definition Rule. Position Paper for the Aspect Oriented Programming Workshop at ECOOP 99, Lisbon, June 14, 1999. Available at `http://www.ee.surrey.ac.uk/Research/CSRG/fog/AopEcoop99.pdf`.

The Refinement Relation of Graph-Based Generic Programs

Extended Abstract

Karl Lieberherr[1] and Boaz Patt-Shamir[1,2]

[1] College of Computer Science
Northeastern University
Boston, MA 02115, USA
`lieber@ccs.neu.edu`
`http://www.ccs.neu.edu/home/lieber`
[2] Dept. of Electrical Engineering–Systems
Tel Aviv University
Tel Aviv 69978, Israel
`boaz@eng.tau.ac.il`

Abstract. This paper studies a particular variant of Generic Programming, called Adaptive Programming (AP). We explain the approach taken by Adaptive Programming to attain the goals set for Generic Programming. Within the formalism of AP, we explore the important problem of refinement: given two generic programs, does one express a subset of the programs expressed by the other? We show that two natural definitions of refinement coincide, but the corresponding decision problem is computationally intractable (co-NP-complete). We proceed to define a more restricted notion of refinement, which arises frequently in the practice of AP, and give an efficient algorithm for deciding it.

1 Introduction

What is Generic Programming (GP) [MS94]? According to the organizers of this Dagstuhl workshop, GP has the following important characteristics:

– Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.
– Lifting of a concrete algorithm to as a general level as possible without losing efficiency, i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.

GP is about parametric polymorphism and we think that non-traditional kinds of parametric polymorphism lead to particularly useful forms of Generic Programming. By non-traditional kinds of parametric polymorphism we mean that parameterization is over larger entities than classes. In this paper we focus on parameterization with entire class graphs and we outline how Adaptive Programming is a form of Generic Programming which attempts to satisfy the two characteristics mentioned above. We show the role of traversal strategies in Adaptive Programming by an analogy to Generic Programming and present new results about traversal strategies. We focus on the concept of graph refinement which is important when traversals are specialized. We show that the obvious definition of refinement leads to a co-NP-complete decision problem and we propose a refinement definition, called strong refinement, which is computationally tractable and useful for practical applications. The results are summarized in Table 1.

Table 1. Graph relationships for software evolution. $\mathcal{N}$ is a mapping of nodes of G_2 to nodes of G_1. $G_1 \leq_{\mathcal{N}} G_2$ if and only if $G_1 \preceq_{\mathcal{N}} G_2$. $G_1 \sqsubseteq_{\mathcal{N}} G_2$ implies $G_1 \leq_{\mathcal{N}} G_2$

Relationship	**Complexity**	**Symbol**
path-set-refinement	co-NP-complete	$G_1 \leq_{\mathcal{N}} G_2$
expansion	co-NP-complete	$G_1 \preceq_{\mathcal{N}} G_2$
strong refinement	polynomial	$G_1 \sqsubseteq_{\mathcal{N}} G_2$

A generic program P defines a family of programs $P(G)$, where G ranges over a set of permissible actual parameters. In this paper we let G range over directed graphs restricted by the program P. Those graphs are abstractions of the data structures on which the program operates. Given two generic programs P_1 and P_2, an important question is whether the programs defined by P_1 are a subset of the programs defined by P_2. We say that P_1 is a refinement of P_2. For example, the generic program P_1 "Find all B-objects contained in X-objects contained in an A-object" defines a subset of the programs determined by the generic program P_2 "Find all B-objects contained in an A-object." P_1 and P_2 are generic programs since they are parameterized by a class graph (e.g., a UML class diagram). Furthermore, the computations done by P_1 are a refinement of the computations done by P_2.

Formalizing the notion of refinement between generic programs leads to graph theoretic problems which have several applications. Refinement can be used to define "subroutines" in adaptive programs as well as to define common evolution relationships between class graphs.

1.1 Adaptive Programming (AP)

Adaptive Programming [Lie92,Lie96] is programming with traversal strategies. The programs use graphs which are referred to by traversal strategies. A traversal strategy defines traversals of graphs without referring to the details of the traversed graphs. AP is a special case of Aspect-Oriented Programming [Kic96], [KLM$^+$97].

AP adds flexibility and simultaneously simplifies designs and programs. We make a connection between GP (as practiced in the STL community) and AP (see Table 2). In GP, algorithms are parameterized by iterators so that they can be used with several different data structures. In AP, algorithms are parame-

Table 2. Correspondence between GP and AP

Kind	**Algorithms**	**Glue**	**Graphs**
GP(STL)	Algorithms	Iterators	Data Structures
AP	Adaptive Algorithms	Traversal Strategies	Class Graphs

terized by traversal strategies so that they can be used with several different data structures. Traversal strategies can be viewed as a form of iterators which are more flexible than ordinary iterators. For details on the parameterization mechanism in AP, see [Lie96,ML98].

AP can be applied directly to GP as implemented in the DJ tool [MOL99], [SFLL99]. A pair of a strategy graph and a class graph defines a traversal for objects defined by the class graph. Such a traversal is encapsulated by a traversal graph that is essentially the intersection (in the automata theoretic sense) of the strategy graph and the class graph. A traversal graph for a strategy graph with a single source and a single target represents an iteration from a source object to a set of target objects. Such an iteration can be used to drive generic algorithms as done in DJ where an adaptor maps traversal graphs to pairs of iterators markin g the beginning and end of the iteration. The benefit is that the iteration specification is robust under changes to the class structure.

A simple example illustrates how GP and AP can be integrated: we simulate a bus route in which several buses circulate and which has several bus stops along the way. We want to print all the people waiting at some bus stop. Therefore we defined a traversal strategy WPStrategy that only traverses the people at bus stops:

```
String WPStrategy = new String(
"from BusRoute through BusStop to Person");
```

This strategy has three nodes and two edges. The following program uses the Java Generic Library (JGL) to solve the problem in Java. First we compute the class graph by analyzing the Java class files (or the Java source code):

```
ClassGraph classGraph = new ClassGraph();
```

Method PrintWaitingPersons of class BusRoute solves the problem. The constructor of class TraveralGraph combines a class graph and a strategy graph into a traversal graph. Method start (finish) converts the traversal graph into an iterator positioned at the beginning (end) of the traversal. The two iterators are used to control the printing of the JGL method Printing.println.

```
class BusRoute {
  TraversalGraph WP = new TraversalGraph(classGraph,
        new Strategy(WPStrategy));
  void PrintWaitingPersons(){
    InputIterator beginDs = WP.start(this);
    InputIterator endDs = WP.finish(this);
    Printing.println(beginDs, endDs);
  }
}
```

Notice that the generic program is very robust under changes to the class graph. It does not matter whether the bus route is structured into villages each containing several bus stops or whether the bus route directly has several bus stops.

2 Traversal Strategies

Traversal strategies (also called succinct traversal specifications) are a key concept of AP. They were introduced in [LPS97,PXL95] together with efficient compilation algorithms. The purpose of a traversal strategy is to succinctly define a set of paths in a graph and as such it is a purely graph-theoretic concept. Since there are several works which demonstrate the usefulness of traversal strategies to programming [Lie96,PXL95,AL98,ML98] we are switching now to a mathematical presentation of the concepts underlying strategies without giving many connections to the practice of programming.

There are different forms of traversal strategies the most general of which are described in [LPS97]. In this paper we only consider a special case: positive strategy graphs. Positive strategy graphs express the path set only in a positive way without excluding nodes and edges. Positive strategies are defined in terms of graphs and interpreted in terms of expansions.

2.1 Definitions

A directed graph is a pair (V, E) where V is a finite set of *nodes*, and $E \subseteq V \times V$ is a set of *edges*. Given a directed graph $G = (V, E)$, a *path* is a sequence $p = \langle v_0 v_1 \dots v_n \rangle$, where $v_i \in V$ for $0 \leq i \leq n$, and $(v_{i-1}, v_i) \in E$ for all $0 < i \leq n$.

We first define the notion of an **embedded strategy graph**.

Definition 1. *A graph $S = (V_1, E_S)$ with a distinguished source node s and a distinguished target node t is said to be an* embedded strategy graph of *a graph $G = (V_2, E)$ if $V_1 \subseteq V_2$.*

Intuitively, a strategy graph S is a sort of digest of the base graph G which highlights certain connections between nodes. In the applications, a strategy graph plays the role of a traversal specification and the base graph plays the role of defining the class structure of an object-oriented program. For example, a strategy graph could say: Traverse all C-objects which are contained in B-objects which are contained in A-objects. This would be summarized as a graph with three nodes A,B,C and an edge from A to B and an edge from B to C. In this paper, the base graphs are just graphs without the embellishments usually found in a class structure. The edges of the simplified class graphs we use here represent directed associations between classes (sometimes also called part-of relationships). (In [LPS97,PXL95] it is shown how to generalize the concept of a strategy graph for general class graphs used in object-oriented programs.) To complicate matters, strategy graphs can also play the role of class graphs. In this case refinement between strategy graphs means refinement between class graphs in the sense that we make the object structures more complex while preserving their essential shape.

A strategy graph S of a base graph G defines a path set as follows. We say that a path p is an *expansion* of a path p' if p' can be obtained by deleteing some elements from p. We define $PathSet_{st}(G, S)$ to be the set of all $s-t$ paths in G which are expansions of any $s-t$ path in S.

Unlike embedded strategies, general strategies allow the node sets of the graphs S and G to be disjoint by using a "name mapping" between them.

Next we define the concept of a strategy graph independent of a base graph.

Definition 2. *A* strategy graph $\mathcal{T}$ *is a triple* $\mathcal{T} = (S, s, t)$, *where* $S = (C, D)$ *is a directed graph,* C *is the set of* strategy-graph nodes, D *is the set of* strategy-graph edges, *and* $s, t \in C$ *are the* source *and* target *of* $\mathcal{T}$, *respectively.*

The connection between strategies and base graphs is done by a name map, defined as follows.

Definition 3. *Let* $S = (C, D)$ *be a graph of a strategy graph and let* $G = (V, E)$ *be a base graph. A* name map *for* S *and* G *is a function* $\mathcal{N} : C \rightarrow V$. *If* p *is a sequence of strategy-graph nodes, then* $\mathcal{N}(p)$ *is the sequence of base graph nodes obtained by applying* $\mathcal{N}$ *to each element of* p.

We next define expansion in the presence of a name map.

Definition 4. *Let* V_1, V_2 *be arbitrary sets, and let* $\mathcal{N} : V_2 \rightarrow V_1$ *be a function. We say that a sequence* p_1 *of elements of* V_1 *is an* expansion under $\mathcal{N}$ *of a sequence* p_2 *of elements of* V_2 *if* $\mathcal{N}(p_2)$ *is a subsequence of* p_1, *where* $\mathcal{N}$ *is applied to each element in the sequence.*

With this definition, we define the concept of a path set.

Definition 5. *Let* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *be directed graphs, let* $\mathcal{N} : V_2 \rightarrow V_1$ *be a function, and let* $s, t \in V_2$. $PathSet_{st}(G_1, \mathcal{N}, G_2)$ *is defined to be the set of all paths in* G_1 *which are expansions under* $\mathcal{N}$ *of any* $s-t$ *path in* G_2.

The identity of s and t is assumed to be fixed, and we shall omit subscripts henceforth.

Using the terminology above, if the name map is the identity function $\mathcal{I}$, then G_2 is an embedded strategy for G_1. Note, for example, that $PathSet(G, \mathcal{I}, G)$ is exactly the set of all $s-t$ paths in G. (Exercise for the reader: Prove that $PathSet(G, \mathcal{I}, G) = PathSet(G, \mathcal{I}, H)$, where H is the directed graph consisting of the single edge (s, t).)

We now turn to the first definition of the graph refinement relations. For the case of embedded strategy graphs, we say that a strategy graph G_1 is a path-set-refinement of strategy graph G_2 if for all base graphs G_3 for which G_1 and G_2 are strategies, $PathSet(G_3, G_1) \subseteq PathSet(G_3, G_2)$.

Example 1. Strategy graph G_2: Nodes A,B. Edges (A,B). Strategy graph G_1: Nodes A,B,X,Y. Edges (A,X), (X,B), (A,Y), (Y,B). Source A, Target B. Name map is the identity map. G_1 is a path-set-refinement of G_2.

In the presence of name maps, the situation is more complex: First, we need the following technical concept (see Figure 1).

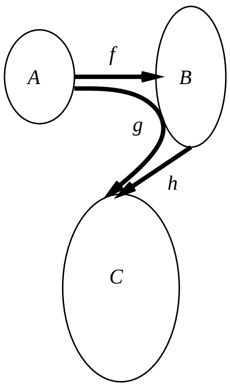


Fig. 1. Illustration for a function h extending g under f

Definition 6. *Let A, B, C be sets, and let $f : A \to B, g : A \to C, h : B \to C$ be functions. We say that h* extends *g* under *f if for all $a \in A$ we have $h(f(a)) = g(a)$.*

Definition 7. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed graphs, and let $\mathcal{N} : V_2 \to V_1$ be a function. We say that G_1 is a* path-set-refinement under $\mathcal{N}$ *of G_2, denoted $G_1 \leq_{\mathcal{N}} G_2$, if for all directed graphs $G_3 = (V_3, E_3)$ and functions $N_1 : V_1 \to V_3$ and $N_2 : V_2 \to V_3$ such that $\mathcal{N}_1$ extends $\mathcal{N}_2$ under $\mathcal{N}$, we have that $PathSet(G_3, \mathcal{N}_1, G_1) \subseteq PathSet(G_3, \mathcal{N}_2, G_2)$.*

Note that if $G_1 \leq_{\mathcal{N}} G_2$, then usually G_2 is the "smaller" graph: intuitively, G_2 is less specified than G_1.

We now define another relation for graph refinement, called "expansion." This relation is more useful for exploring properties of graph refinement. For the case of embedded strategy graphs, we say that a strategy graph G_1 is an expansion of strategy graph G_2 if for any path p_1 (from s to t) in G_1 there exists a path p_2 (from s to t) in G_2 such that p_1 is an expansion of p_2. In example 1, G_1 is an expansion of G_2.

The general definition of expansion for positive strategies is:

Definition 8. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed graphs, and let $\mathcal{N} : V_2 \to V_1$ be a function. We say that G_1 is an* expansion under $\mathcal{N}$ *of G_2, denoted $G_1 \preceq_{\mathcal{N}} G_2$, if for any path $p_1 \in PathSet(G_1, \mathcal{I}, G_1)$ there exists a path $p_2 \in PathSet(G_2, \mathcal{I}, G_2)$ such that p_1 is an expansion under $\mathcal{N}$ of p_2.*

We now show equivalence of the notions of "path-set-refinement" and "expansion".

Theorem 1. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed graphs, and let $\mathcal{N} : V_2 \to V_1$ be a function. Then $G_1 \preceq_{\mathcal{N}} G_2$ if and only if $G_1 \leq_{\mathcal{N}} G_2$.*

This theorem tells us that we can use the simpler definition of expansion instead of the more complex definition of path-set-refinement which involves quantification over general graphs.
Proof: See [LPS98].
The following problem arises naturally in many applications of strategies.

Graph Path-set-refinement Problem (GPP)
Input: Digraphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ with $s_1, t_1 \in V_1$, and a function $\mathcal{N} : V_2 \to V_1$.
Question: Does $G_1 \leq_{\mathcal{N}} G_2$ hold true?

Unfortunately, it turns out that deciding GPP is hard. To show hardness, we first consider a weakened version of GPP, defined as follows. Call an edge in a strategy *redundant* if its removal does not change the path sets defined by the strategy. For example, if there exists an edge from the source to the target, then all other edges are redundant, since all source-target paths are expansions of this edge anyway! More formally, an edge (u, v) in a strategy graph G is redundant if $G \leq_{\mathcal{I}} G - \{(u, v)\}$. We define the following decision problem.

Redundant Strategy Edge (RSE)
Input: A digraph $G = (V, E)$ with source and target nodes $s, t \in V$, and a distinguished edge $(u, v) \in E$.
Question: Is the distinguished edge (u, v) redundant?

Theorem 2. *RSE is co-NP-complete.*

Proof: See [LPS98].
A direct implication of Theorem 2 is that the problem of finding a strategy with minimal representation is hard. With regard to the main point of this paper, we have the following easy corollary.

Corollary 1. *GPP is co-NP-Complete.*

This corollary tells us that when we build tools for AP we cannot use the general definition of expansion since it would result in a slow design tool for large applications.
Proof: See [LPS98].

3 The Refinement Relation

In this section we define a more stringent version of the graph path-set-refinement relation, called the *strong refinement* relation. We argue that this relation is central to software engineering practices. We show that the path-set-refinement relation is a generalization of the strong refinement relation, and we give an efficient algorithm for deciding the strong refinement relation.

In this section we invoke a mathematical pattern called the *Tractable Specialization Pattern (TSP)* which has several applications in computer science. TSP is defined as follows: Given is a decision problem which has a high complexity but which we need to solve for practical purposes. We define a more strict version of the decision problem for which we can solve the decision problem more efficiently. The goal of the restricted version is that it does not disallow too many of the inputs which are occurring in practice. Fig. 3 shows two applications of TSP. The first is to graph properties in this paper and the second to context-free grammar properties in language theory [HU79]. The second column in Fig. 3 shows a decision problem with its complexity. The third column shows a stricter form of the decision problem with the hopefully lower complexity.

Table 3. Applications of the Tractable Specialization Pattern

Area	**Decision Problem**	**Stricter**
Graphs	path-set-refinement (co-NP-complete)	strong refinement (polynomial)
Grammars	ambiguous (undecidable)	LL(1) (polynomial)

We first consider the case of embedded strategy graphs. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed graphs with V_2 a subset of V_1. We say that G_1 is a strong refinement of G_2, denoted $G_1 \sqsubseteq G_2$, if for all $u, v \in V_2$ we have that $(u, v) \in E_2$ if and only if there exists a path in G_1 between u and v which does not use in its interior a node in V_2.

Example 2. Strategy graph G_2: Nodes A,B,C. Edges (A,B), (B,C). Strategy graph G_1: Nodes A,B,C. Edges (A,C), (C,B), (B,C)). Source A, Target C. Name map is identity map. G_1 is not a strong refinement of G_2. For the edge from A to B in G_2 there is no path in G_1 from A to B which does not go through C. However, strategy graph G_3: Nodes A, B, C, X. Edges (A,X), (X,B), (B,C) is a strong refinement of G_2.

The intuition behind the strong refinement relation is that we are allowed to replace an edge with a more complex graph using new nodes. In example 2, we replace the edge (A,B) by the graph (A,X),(X,B), where X is a new node and not one of A,B or C. Informally, G_1 is a strong refinement of G_2 if the connectivity of G_2 is exactly and “without surprises” in G_1. ”Without surprises” means that the nodes of G_2 can appear on paths only as advertised by G_2. For example, if G_2 has nodes A, B and C and an edge (A,B) but not an edge (A,C) then a path

A ... C ... B in G_1 is disallowed. We first need the following technical concepts to define strong refinement in the presence of a name map.

Definition 9. *Let* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *be directed graphs, and let* $\mathcal{N} : V_2 \to V_1$ *be a function. Given a path* p*, let first*(p) *and last*(p) *denote its first and last nodes, respectively. A path* p_1 *in* G_1 *(not necessarily an* $s - t$ *path) is* pure *if first*$(p) = \mathcal{N}(u)$ *and last*$(p) = \mathcal{N}(v)$ *for some* $u, v \in V_2$*, and none of the internal nodes of* p *is the image of a node in* V_2*.*

We define strong refinements as strategies whose pure-path connectivity is the same as the edge-connectivity in the super-strategy. Formally, we have the following definition.

Definition 10. *Let* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *be directed graphs, and let* $\mathcal{N} : V_2 \to V_1$ *be a function. We say that* G_1 *is a* strong refinement of G_2 under $\mathcal{N}$*, denoted* $G_1 \sqsubseteq_{\mathcal{N}} G_2$*, if for all* $u, v \in V_2$ *we have that* $(u, v) \in E_2$ *if and only if there exists a pure path in* G_1 *between* $\mathcal{N}(u)$ *and* $\mathcal{N}(v)$*.*

To justify Definition 10, we remark that the notion of strategies is particularly useful in evolution of software, where it is often the case that some crude concepts (and objects) are refined in the course of development. In such scenarios, refining an edge to a more complex structure is usually done with the aid of a strong refinement. It is important to check whether such an evolution leads to modifying the connectivity structure of the strategy, which is the question of deciding the strong refinement relation.

We call the relation *strong refinement* because we also use elsewhere the concept of *refinement*. Basically, refinement is strong refinement with "if and only if" replaced by "implies".

If G_1 is a strong refinement of G_2, the connectivity of G_2 is in "pure form" in G_1 and G_1 contains no new connections in terms of nodes of G_2. Basically, refinement also requires that the connectivity of G_2 is in pure form in G_1 but it allows extra connectivity. Strong refinement implies refinement but the converse does not hold. Refinement is an important abstraction in Adaptive Programming. When a class graph is a refinement of a strategy graph, the traversal paths better correspond to what the strategy graph says intuitively. Refinement is not discussed further in this paper but it is very useful in practice and has a polynomial decision algorithm.

Readers should not confuse the generic concept of refinement used to generalize path-set-refinement, expansion and strong refinement with the specific concept of refinement described above.

The following theorem states that the strong refinement relation is a subrelation of the path-set-refinement relation.

Theorem 3. *If* $G_1 \sqsubseteq_{\mathcal{N}} G_2$*, then* $G_1 \leq_{\mathcal{N}} G_2$*.*

Proof: See [LPS98].

The converse of Theorem 3 does not hold as demonstrated by example 3.

Example 3. We give an example of two graphs G_1 and G_2, where G_1 is an expansion of G_2 but G_1 is not a strong refinement of G_2. This proves that expansion does not imply strong refinement. An entire family of such examples is obtained by taking for G_1 the directed cycle for n nodes and for G_2 the complete graph for n nodes. As source we select the first node and as target the nth node. For n=3: G_1: Nodes A,B,C. Edges: (A,B), (B,C), (C,A). G_2: Nodes A,B,C. Edges: all ordered pairs. G_1 is an expansion of G_2 since G_2 is complete and therefore it has all the paths we want. G_1 is not a strong refinement of G_2 because for (C,B) in G_2 there is no path in G_1 from C to B which does not use A, i.e., there is no pure path from C to B in G_1.

We now give an algorithm to decide whether $G_1 \sqsubseteq_{\mathcal{N}} G_2$. This is done by a simple graph search algorithm defined as follows.

First Targets Search (FTS)
Input: Digraph $G = (V, E)$ a node $v_0 \in V$, and a set of *targets* $T \subseteq V$.
Output: A subset $T' \subseteq T$ of the targets, such that for each $t \in T'$, there exists a path in G from v_0 to t which does not contain any other target.

It is easy to implement FTS (using BFS or DFS) in linear time, assuming that we can test in constant time whether a node is a target. In Figure 2 we give a BFS-based algorithm, using a FIFO queue Q.

```
PROCEDURE FTS($G, v_0, T$)
  mark $v_0$ visited
  insert $v_0$ to tail of $Q$
  while $Q \neq \emptyset$
      remove $u$ from head of $Q$
      if $u \in T$ then add $u$ to output set        //...and don't explore this path further
      else for_all $v$ such that $(u, v) \in E$
          if $v$ is not marked visited then
              mark $v$ visited
              insert $v$ to tail of $Q$
          end_if
      end_if
  end_while
```

Fig. 2. *Algorithm for First Targets Search*

Running FTS can detect superfluous or missing connectivity in G_1, when compared to G_2 under $\mathcal{N}$. The algorithm for deciding the strong refinement relation proceeds as follows.

Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \mathcal{N} : V_2 \to V_1$.

1. Let $T \subseteq V_1$ be the image of V_2 under $\mathcal{N}$, i.e., $T = \{\mathcal{N}(v) \mid v \in V_2\}$.
2. For each node $v \in V_2$:
   (a) Perform FTS from $\mathcal{N}(v)$ with targets T in G_1. Let the resulting set be T_v, i.e., $T_v = \mathrm{FTS}(G_1, \mathcal{N}(v), T)$.
   (b) If there exists $u \in V_2$ such that $(v, u) \notin E_2$ and $\mathcal{N}(u) \in T_v$, or $(v, u) \in E$ and $\mathcal{N}(u) \notin T_v$, return "$G_1 \not\sqsubseteq_{\mathcal{N}} G_2$" and halt.
3. Return "$G_1 \sqsubseteq_{\mathcal{N}} G_2$."

The running time of the algorithm is $O(|E_1| \cdot |V_2|)$.

3.1 Related Work

This paper studies refinement relations between graphs. The Subgraph Homeomorphism problem (SH) is related but different from the problems studied here.

Definition 11. SH
instance*: graph $G = (V, E)$*
question*: does G contain a subgraph homeomorphic to H, i.e., a subgraph $G' = (V', E')$ that can be converted to a graph isomorphic to H by repeatedly removing any vertex of degree 2 and adding the edge joining its two neighbors?*

SH is NP-complete for variable H. See [FHW80] for general results. SH supports only limited graph refinements because only vertices of degree 2 may be removed.

[GJ79] mentions other refinement-style problems, such as graph contractability, graph homomorphism and D-morphism but none of those problems match our definition of graph refinement.

4 Conclusions

We have discussed how Generic Programming through parameterization of programs with entire graph structures (as opposed to only single classes) leads to more flexible programs. We introduced graph theory which is needed to better apply and understand this new form of generic programming, called Adaptive Programming.

We introduced the concept of refinement between graphs which has the following applications: It can be applied to 1. check efficiently whether one traversal is a subtraversal of another (path-set-refinement = expansion). 2. check whether one class graph is a generalization of another class graph so that the containment relationships are preserved (strong refinement). This kind of graph generalization relation is useful for automating the evolution of adaptive programs. 3. check whether an adaptive program defines a subset of the programs defined by another adaptive program (strong refinement). The results are summarized in Table 1.

Acknowledgements

We would like to thank Mira Mezini, Joshua Marshall, Doug Orleans, and Johan Ovlinger for the ideas they contributed to the paper. This work has been partially supported by the Defense Advanced Projects Agency (DARPA), and Rome Laboratory, under agreement number F30602-96-2-0239. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U. S. Government. See:
`http://www.ccs.neu.edu/research/demeter/biblio/graph-refine.html`
for further information about this paper including links to online versions of some related papers.

References

AL98. Dean Allemang and Karl J. Lieberherr. Softening Dependencies between Interfaces. Technical Report NU-CCS-98-07, College of Computer Science, Northeastern University, Boston, MA, August 1998.

FHW80. S. Fortune, John Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.

GJ79. Michael R. Garey and David S. Johnson. *Computers and Intractability.* Freeman, 1979.

HU79. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

Kic96. Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28A(4), December 1996.

KLM$^+$97. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer Verlag, 1997.

Lie92. Karl J. Lieberherr. Component enhancement: An adaptive reusability mechanism for groups of collaborating classes. In J. van Leeuwen, editor, *Information Processing '92, 12th World Computer Congress*, pages 179–185, Madrid, Spain, 1992. Elsevier.

Lie96. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X, entire book at `http://www.ccs.neu.edu/research/demeter`.

LPS97. Karl J. Lieberherr and Boaz Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997. `http://www.ccs.neu.edu/research/demeter/AP-Library/`.

LPS98. Karl Lieberherr and Boaz Patt-Shamir. Generic Programming with Graph Refinement. Technical Report NU-CCS-98-10, College of Computer Science, Northeastern University, Boston, MA, September 1998.

ML98. Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. Technical Report NU-CCS-98-3, Northeastern University, April 1998. To appear in OOPSLA '98.

MOL99. Joshua Marshall, Doug Orleans, and Karl Lieberherr. DJ: Dynamic Structure-Shy Traversal in Pure Java. Technical report, Northeastern University, May 1999. `http://www.ccs.neu.edu/research/demeter/DJ/`.

MS94. D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software–Practice and Experience*, 24(7), July 1994.

PXL95. Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.

SFLL99. Neeraj Sangal, Edward Farrell, Karl Lieberherr, and David Lorenz. Interaction schemata: Compiling interactions to code. In *TOOLS USA, Technology of Object-Oriented Languages and Systems*, Santa Barbara, CA, August 1999. IEEE Computer Society.

The Construction Principles of the Class Library Karla

Wolf Zimmermann[1], Arne Frick[2], and Rainer Neumann[1]

[1] Institut für Programmstrukturen und Datenorganisation Universität Karlsruhe, D-76128 Karlsruhe, Germany, {zimmer,rneumann}@ipd.info.uni-karlsruhe.de
[2] Tom Sawyer Software, africk@tomsawyer.com

Abstract. This article shows how to construct robust class libraries in the context of *flexibility*, implementation *efficiency* and *extensibility* as further design goals. A library is called *robust* if (1) no errors can be introduced into an application due to its use or inheritance from it, (2) error messages do not point deep into library code but into the user's application, and (3) statically checkable incorrect uses of library classes are recognized by compilers. The principles derived from a theoretical analysis have been applied practically in Karla, a robust class library of data structures and algorithms the authors designed and implemented, and that is being used for research projects and student education.
The main focus of this article is on the construction of hierarchies of abstract classes. The analysis shows that it is not possible in general to achieve all of the above design goals for robust libraries at the same time. The only solution that ensures robustness is to drop one of these criteria. Redesigning an existing library for robustness may lead to an exponential growth in the number of classes in the hierarchy of abstract classes. However, it is shown that this class explosion can be controlled by automatically generating the required additional classes.

1 Introduction

One of the main promises of object-oriented software design and programming has traditionally been to enable and enhance the reusability of designs and code. This goal has not been achieved so far—at least it could not be proven, since most existing methods for object-oriented design and programming do not provide a solid and formal basis for correctness proofs. Consider the methods Booch [1] or Rumbaugh [11] from the abundance of existing methods for object-oriented designs: they provide no support to assert or prove certain properties, since they do not formally define basic concepts like inheritance, polymorphism and genericity; and the implementation of, and the interaction between, these concepts opens robustness holes in almost all existing object-oriented programming languages. Robustness can therefore only be achieved with current technology by closely following design rules such as the ones in this article.

This work is based on our experience with the design and implementation of Karla, a robust class library for algorithms and data structures. To allow algorithms to work on different implementations of data structures, we made extensive use of inheritance and polymorphism. This led to several severe robustness problems with the initial design. Our initial investigation led us to the startling conclusion that no solution was known for the kind of problem we experienced: good library design was considered an artistic work for experienced specialists rather than an engineering task.

This article summarizes our results and method for designing robust class libraries. The remainder of this article is organized as follows: Section 2 gives basic definitions and defines our notion of correctness. In Section 3, four types of inheritance relationships are distinguished, based on the observation of [5] that the semantics of inheritance in existing programming languages can almost never enforce robustness under polymorphism. Section 4 extends the notion of robustness to include the concept of genericity often found in object-oriented programming languages. In Section 5, we study the construction of robust hierarchies of specializing and conforming classes. Section 6 introduces a specification that allows one to largely generate specialization hierarchies.

2 Basics

A class B inherits from class A if and only if definitions of A are visible in B. In this case, B is called a *subclass* of A. *Polymorphism* allows using an object of type A or any of its subclasses at a program point requiring an object of type A.[1] A parameterized class $A(T_1, \ldots, T_n)$ is called a *generic class.* Applications using generic classes must instantiate the *generic parameters* with concrete classes $U_1, \ldots, U_n$. This can be done by writing $A(U_1, \ldots, U_n)$, which means that T_i is instantiated with class U_i.[2] Each generic instance $A(U_1, \ldots, U_n)$ is a class.

We assume that classes are completely specified by invariants and pre- and postconditions (see also [10]). Pre- and postconditions are used to specify the functional behavior of each method. Furthermore we assume attributes to be encapsulated in the sense that they are invisible at the external interface of a class. A class A is called *locally correct* if and only if

(i) after creation of an object of class A, the invariant Inv_A holds, and
(ii) after calling a method m of an object of class A, the invariant Inv_A and the postcondition $Post_{m,A}$ of m hold, provided that the invariant Inv_A and the precondition $Pre_{m,A}$ of m are satisfied when calling m.

Any object of class A must satisfy its invariant Inv_A at any time except possibly during execution of a method of A. Local correctness is the only notion of correctness that can be supported by library designers. We now extend the notion of local correctness by the object-oriented concepts of inheritance, polymorphism, and genericity.

[1] Following the semantics of SATHER-K, we identify the notion of class and the notion of type.

[2] Generic parameters can be generic instantiations themselves.

A class library is *correct* if and only if each class is locally correct—even if it is inherited by other classes, used polymorphically, or used as an instantiation of a generic parameter. These cases are discussed in Sections 3 and 4, where we first analyze different notions of inheritance according to their robustness under polymorphism, and then discuss how robustness can be maintained when using generic parameters.

An *abstract class* is a class that cannot be instantiated because there exists at least one method which has only an interface description. Every non-abstract class is called *concrete*. We subsequently focus on hierarchies of abstract classes, although the discussion applies to concrete classes as well. The set of methods of a class A can be partitioned into *kernel* and *derived* methods. Kernel methods of A are *abstract* methods, which cannot be implemented within an abstract class A, but must be implemented in concrete subclasses. Derived methods are methods which can be implemented using at most kernel methods.[3] Invariants usually describe the possible states of objects, but may also specify parts of the functional behavior of kernel methods.

3 Inheritance Relations

In practice, the failure to achieve robustness in an object-oriented software component can often be witnessed by the fact that its reuse in an application leads to unexpected exceptions at runtime. At the root of this problem is almost always a robustness deficiency in the interaction between the inheritance model and the use of polymorphism in the semantics of the underlying programming language [5]. This leads to a distinction between four different possibilities to define inheritance using formal specifications. Minimal related work exists [2,9,12], which is furthermore limited to only one of the possibilities (conformance).[4] The four possible inheritance relations are shown in Fig. 1.

More formally, we say a class B *conforms to* a class A if and only if the following implications hold (Fig. 1a):

(i) $Inv_B \Rightarrow Inv_A$
(ii) For each method m of A: $Pre_{m,A} \Rightarrow Pre_{m,B}$ and $Post_{m,B} \Rightarrow Post_{m,A}$.

A class B is *more special than* a class A if and only if the following implications hold (Fig. 1b):

(i) $Inv_B \Rightarrow Inv_A$
(ii) For each method m of A: $Pre_{m,B} \Rightarrow Pre_{m,A}$ and $Post_{m,A} \Rightarrow Post_{m,B}$.

[3] Because abstract classes cannot take advantage of implementation details [5], derived methods are often inefficient and will typically be re-implemented in subclasses.

[4] Liskov and Wing [9] also define a “specialization.” However, upon close analysis it becomes clear that this is just a special case of conformance. Furthermore, their correctness proofs require the existence of execution histories of objects, leading to unnecessarily complicated proofs.

$$\begin{tikzcd}
Inv_A & Pre_{m,A} \arrow[r] \arrow[d] & Post_{m,A} \\
Inv_B \arrow[u] & Pre_{m,B} \arrow[r] & Post_{m,B} \arrow[u]
\end{tikzcd}$$

a. conforms to

$$\begin{tikzcd}
Inv_A & Pre_{m,A} \arrow[r] & Post_{m,A} \arrow[d] \\
Inv_B \arrow[u] & Pre_{m,B} \arrow[r] \arrow[u] & Post_{m,B}
\end{tikzcd}$$

b. more special than

$$\begin{tikzcd}
Inv_A & Pre_{m,A} \arrow[r] & Post_{m,A} \\
Inv_B \arrow[u] & Pre_{m,B} \arrow[r] \arrow[u] & Post_{m,B} \arrow[u]
\end{tikzcd}$$

c. covariant to

$$\begin{tikzcd}
Inv_A & Pre_{m,A} \arrow[r] \arrow[d] & Post_{m,A} \arrow[d] \\
Inv_B \arrow[u] & Pre_{m,B} \arrow[r] & Post_{m,B}
\end{tikzcd}$$

d. contravariant to

Fig. 1. Different Inheritance Relations.

A class B is *covariant to* a class A if and only if the following implications hold (Fig. 1c):

(i) $Inv_B \Rightarrow Inv_A$
(ii) For each method m of A: $Pre_{m,B} \Rightarrow Pre_{m,A}$ and $Post_{m,B} \Rightarrow Post_{m,A}$.

The specialization in [9] requires the equivalence of the preconditions. Therefore, their notion is a special case of both conformance and covariance.

A class B is *contravariant to* a class A if and only if the following implications hold (see Fig. 1d):

(i) $Inv_B \Rightarrow Inv_A$
(ii) For each method m of A: $Pre_{m,A} \Rightarrow Pre_{m,B}$ and $Post_{m,A} \Rightarrow Post_{m,B}$.

The actual type of polymorphic objects may be determined as late as at run-time, which can lead to violations of robustness if the actual type does not match the declared type. Robustness for polymorphic objects can only be guaranteed if the definition of inheritance is based on conformance, as the following argument shows. The definition of conformance assures the correctness of polymorphic variable use, because pre- and postconditions fit correctly. Assume that method m of a polymorphic object of type A is called, and that the actual type of the object is B. We need to assure that before the call, $pre_{m,B}$ holds, which is in fact the case, assuming that $pre_{m,A}$ was true. Therefore, we may safely assume that the call to m is legal. After the call, we know that $post_{m,B}$ holds due to the assumed correctness of the implementation of m in B. By definition, this logically implies $post_{m,A}$, which needed to be shown.

The remaining definitions of inheritance are not robust against polymorphism. In order to ensure correctness, the actual type of an object must be known. This naturally leads to the question of whether the other definitions of inheritance are irrelevant. It turns out that this is not the case, because they potentially allow for the reuse of code. We demonstrate this for the case of specialization.

Example 1. The robust modeling of directed graphs requires that acyclic directed graphs be a specialization of directed graphs. It is easy to see that not all edges which can be added to a graph can also be added to an acyclic directed graph: only edges that do not violate the acyclicity constraint may be added without violating the class invariant. Other examples are data structures (such as queues, lists, stacks, etc.) of bounded size, which are more special than their unbounded versions, because the insertion of items may raise an overflow on the size. A third example comes from relational databases. Integrity constraints and normal forms of relations lead to specializations.

These examples demonstrate the usefulness of the specialization. A robust class library should therefore provide at least conformance and specialization. From a library designer's viewpoint, specializations can be only considered in terms of their potential for reuse, excluding polymorphism for robustness reasons. However, the information on the inheritance relation between classes may help library users to design their programs.

Summarizing the results of this section, we observed that the conformance relation is the only notion of inheritance which is robust under polymorphism. The other three possible inheritance relations allow for the reuse of program code and design. A robust class library should therefore support each of the four inheritance relations to exploit the full potential of reuse. In practice, however, we have only observed conformance and specialization relationships.

4 Genericity

Genericity is mainly used for the construction of *container classes* $C(T)$. When instantiating a container class, the generic T is replaced by an arbitrary type (*unbounded genericity*). The definition of $C(T)$ is independent of T. Under robustness requirements, neither the definition of $C(T)$ nor any implementation of $C(T)$ is allowed to make any assumptions on the structure of T. Otherwise, the programmer might instantiate T with a type violating the assumptions of the library designer. This faulty instantiation leads to compile-time or run-time errors pointing deeply into library code, a situation to be avoided.

Example 2. Suppose there is class SET(T) defining sets with the usual operations. An operation max:T computing the maximum of the elements does not make sense unless there is a total order on the instantiations for the generic parameter T.

Requiring robust unbounded genericity restricts reuse potential. For example, it is impossible to sort sets generically unless it is assumed that any instance of the type parameter T has a relation $<$ defining a total order on the objects of T. A robust library should therefore, in addition, offer restricted instantiations (*bounded genericity*). A restriction on the instantiations of a generic parameter T of a container class $C(T)$ is defined by a type bound B. The definition of $C(T)$ and any implementation of $C(T)$ is allowed to make use of the properties specified in B. T can only be instantiated with types having at least the properties of B. The results of Section 3 provide an elegant way to express type bounds.

The type bound B must state minimum properties to be satisfied by any instance of T. The obvious way to do this is to define the type bound B itself as a class and restrict the possible instances of T to classes which conform to B. We denote this restriction by $C(T < B)$. The correctness proof of an implementation of $C(T < B)$ can then use Inv_B, $Pre_{m,B}$, and $Post_{m,B}$ for any method m of B.

Example 3. The class ORDERED_SET(T<ORDERED(T)) with the type bound

```
class ORDERED(T) is
    -- Inv: $\forall x, y, z : \mathsf{T} : conforms(\mathsf{T}, \mathsf{ORDERED(T)}) \Rightarrow$
    -- $(x \leq y \vee y \leq x) \wedge x \leq x \wedge$
    -- $(x \leq y \wedge y \leq x \Rightarrow x = y) \wedge$
    -- $(x \leq y \wedge y \leq z \Rightarrow x \leq z)$
    isLessEq(s:T):BOOL is abstract
end
```

The notions of specialization and conformance can be transferred to generic classes as follows. First, we define the notion of inheritance between generic classes. If a generic class B inherits from another generic class A, then A is instantiated by B, but may still use generic parameters of B. Such an inheritance is *valid* if and only if each valid instantiation of the parameters of B is a valid instantiation of the parameters of A. Then A is again a generic class within the generic parameters of B.

The generic class $B(T_1 < S'_1, \ldots, T_l < S'_l)$ *conforms to* (*is more specific than*) generic class $A(T_1 < S_1, \ldots, T_k < S_k)$, where the parameters of B are a superset of the parameters of A; i.e., all parameters of A get instantiated through inheritance. We say that (i) each valid instance of B is a valid instance of A, and (ii) each valid instance of B is a conformance subclass (specialization) of the corresponding instance of A.

5 Construction of Robust Class Hierarchies

Besides considering hierarchies of conforming classes that can be used polymorphically, it is also useful to consider hierarchies of specialized classes, which restrict the behavior of their base classes. This section summarizes the main results of [3,6], where the construction of robust class hierarchies has been discussed in detail; refer to those references for detailed proofs.

5.1 Specialization Hierarchies

The main result of [6] on specialization hierarchies states that they can be systematically constructed. A major possible problem with this approach is that the number of classes may grow exponentially, making it impractical to implement all these classes manually. The obvious solution is to try to generate most of the hierarchy based on some small specification. Of course, the generation process itself can be modified such that only classes required for a particular application are actually generated. The following theorems provide the foundation for the specification of specialization hierarchies.

Theorem 1. *Let A be a (generic) class with an invariant Inv_A and let P be a predicate defined over the instances of A. Let B be a class which contains at least the same methods as A, where objects of class B satisfy $Inv_A \wedge P$. Then B is more specific than, and covariant to, A.*

Theorem 1 is the basis for a systematic construction of specialization and/or covariance hierarchies.

Given a (generic) class A and predicates $P_1, \ldots, P_m$ on the instances of this class, we call the set of all conjunctions

$$H(P_1, \ldots, P_m) = \left\{ \bigwedge_{i \in I} P_i : I \subseteq \{1, \ldots, m\} \right\}$$

the *conjunctive closure* of $P_1, \ldots, P_m$ where $\bigwedge_{i \in \emptyset} P_i = \mathit{true}$.

Let $Q_1, Q_2 \in H(P_1, \ldots, P_m)$ be predicates with $Q_1 \Rightarrow Q_2$ and Q_1_A (Q_2_A) be the class of objects of A satisfying Q_1 (Q_2). We now know that Q_1_A and Q_2_A are specializations of A. We also know that Q_1_A is also a specialization of Q_2_A. The implication relationship between the two predicates obviously induces a specialization relationship between the corresponding classes. It is easy to see that specialization relationships are transitive. Therefore we can define a specialization hierarchy using the implications $H(P_1, \ldots, P_m)$. It may be that two elements $Q_1, Q_2 \in H(P_1, \ldots, P_m)$ are *equivalent*, which means that $Q_1 \Leftrightarrow Q_2$. In this case we do not need to introduce two classes that are specializations of each other. Therefore we look at equivalence classes of $H(P_1, \ldots, P_m)$. If $\overline{Q}$ is the equivalence class of $Q \in H(P_1, \ldots, P_m)$ or a representative of this class, then $\overline{H}(P_1, \ldots, P_m) := \{\overline{Q} : Q \in H(P_1, \ldots, P_m)\}$ defines a partial order relationship $\Rightarrow$. Knowing that $\overline{P_1 \wedge \cdots \wedge P_m} \Rightarrow \overline{Q}$ and $\overline{Q} \Rightarrow \overline{\mathit{true}}$ for every $Q \in H(P_1, \ldots, P_m)$, it is easy to see that $(\overline{H}(P_1, \ldots, P_m), \Rightarrow, \overline{P_1 \wedge \cdots \wedge P_m}, \overline{\mathit{true}})$ is a lattice. We can now say that Q_1_A is a specialization and covariant subclass of Q_2_A if and only if $Q_1 \Rightarrow Q_2$ and $\overline{Q_1} \neq \overline{Q_2}$.

Theorem 2. *Let A be a (generic) class and $P_1, \ldots, P_m$ predicates over A. Then the lattice induces*

$$(\overline{H}(P_1, \ldots, P_m), \Rightarrow, \overline{P_1 \wedge \cdots \wedge P_m}, \mathit{true})$$

a specialization and covariance hierarchy

$$(\mathcal{A}, \textit{more specific than/covariant to}, \bot, \top)$$

with $\mathcal{A} = \{Q\_A : \overline{Q} \in \overline{H}(P_1, \ldots, P_m)\}$, $\bot = P_1\_\ldots\_P_m\_A$ *and* $\top = A$.

Theorem 2 suggests the following design method for specialization hierarchies:

1. define a set of predicates on a class A.
2. compute the conjunctive closure of these predicates.
3. derive the specialization hierarchy from the conjunctive closure.
4. add some new methods to the specializations, which uses the stronger invariant.

Using this method to generate specialization hierarchies guarantees the specialization property because of Theorem 2.

Example 4. We consider again the class *DIRECTED_GRAPH*. This class may have the predicates *is_acyclic*, *is_tree*, and *is_weakly_connected*. Fig. 2 shows the conjunctive closure of these predicates and the induced specialization hierarchy. We might add the method *top_sort* to acyclic directed graphs, the method *children* and *parent* to trees. *AWCDG* is the class of acyclic, weakly connected directed graphs.

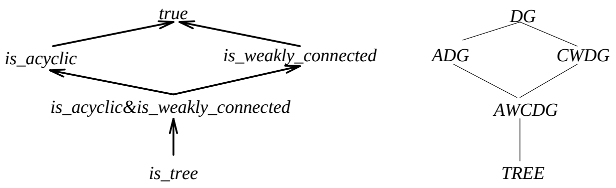


Fig. 2. A conjunctive closure and its specialization hierarchy.

The application of this method leads to many classes, but is sufficient to construct class libraries with robust class hierarchies. The next goal is to refine the method to allow for the construction of robust class hierarchies based on some specification that is much shorter than the source code of all classes. Such a specification must allow one to derive at least the following information:

(i) the signature, implementations and postconditions of all methods belonging to at least one class in the class hierarchy,
(ii) for each such method, the highest class in the specialization hierarchy where it occurs,
(iii) the set of predicates over which the specialization hierarchy is built according to Theorem 2,

(iv) the conjunctive closure of this set of predicates,
(v) the invariant *Inv* of the highest class in the hierarchy, and
(vi) for any class in the specialization hierarchy, the preconditions of the methods belonging to this class.

Section 6 discusses how to derive an appropriate specification. For now it suffices to know that this information is sufficient to generate any class A in the specialization hierarchy and the factorization A' of its subclasses. Suppose we want to construct a class A corresponding to predicate $Q = P_1 \wedge \cdots \wedge P_k$. Since $Inv_A = Inv \wedge Q$, it can be obtained from (iv). The methods (including their signatures, implementations, pre- and postconditions) belonging to A can be obtained from (i), (ii) and (vi). The specialization hierarchy can be obtained from (iii) and (iv). Theorem 2 implies that all postconditions are equal, and therefore, $Post_{m,A'} = Post_{m,A}$. The implementation of m is taken from the lowest class in the specialization hierarchy.

5.2 Hierarchies of Conforming Subclasses

Let $A(T_1, \dots, T_k)$ be a generic class, where the allowed instances of the generic parameters of A hold for a predicate P_1. Instead of giving new requirements for the instances of A (see Section 5.1), we analyze the inheritance relationship for the case that B only defines new restrictions on the generic parameters. Let $B(T_1, \dots, T_k)$ be a generic class, where the allowed instances of the generic parameters of B hold the predicate P_2. We further assume that $P_2 \Rightarrow P_1$. In particular B behaves like A within the set of valid instances. This means that each method $m \in B$ preserves Inv_A and guarantees the postcondition $Post_{m,A}$ provided that $Pre_{m,A}$ holds when calling the method. As already shown this can only be the case if each valid instantiation of the generic class B conforms to its corresponding instantiation of class A. This leads to the following

Theorem 3. *Let $A(T_1, \dots, T_k)$ be a generic class and $P_1(T_1, \dots, T_k)$ a predicate that reduces the set of valid generic parameters. Let $B(T_1, \dots, T_k)$ be a generic class, where its instantiations are also instantiations of the generic class $A(T_1, \dots, T_k)$, but for which holds the stronger predicate $P_2(T_1, \dots, T_k) \Rightarrow P_1(T_1, \dots, T_k)$. Then B conforms to A.*

Additional restrictions of the generic parameters may allow the introduction of new methods, which can make use of the additional type information. Furthermore this information often allows more efficient implementations of certain methods.

Example 5. Consider the class SET(T) defining sets. Restricting the generic parameter T leads to new operations:

– The maximum or minimum of the set can be computed if a total order on the type of elements exists.
– The complement of a set can only be computed if the universe of the underlying element is finite.

In addition, if the underlying type T is known to have a finite (or better, tiny) universe, e.g., characters, then sets can be implemented very efficiently using bit vectors.

We define the conjunctive closure $H(P_1, \ldots, P_m)$ and the equivalence classes $\overline{H}(P_1, \ldots, P_m)$ in analogy to the previous section. Those equivalence classes again form a lattice with the top element *true*, which means that there are no restrictions. The equivalence class $Q\_A(T_1, \ldots, T_k)$ which only allows instantiations of $(T_1, \ldots, T_k)$ that satisfy Q is shown to conform to class $A(T_1, \ldots, T_k)$. In detail $Q\_A(T_1, \ldots, T_k)$ conforms to each class $R\_A(T_1, \ldots, T_k)$ that only allows instantiations by $(T_1, \ldots, T_k)$ that satisfy R, if $Q \Rightarrow R$ holds.

Theorem 4. *Let $A(T_1, \ldots, T_k)$ be a generic class and $P_1, \ldots, P_m$ predicates over the instantiations of $(T_1, \ldots, T_k)$. Then the lattice*

$$(\overline{H}(P_1, \ldots, P_m), \Rightarrow, \overline{P_1 \wedge \cdots \wedge P_m}, \mathit{true})$$

induces a lattice $(\mathcal{A}, \mathit{conform\ to}, \bot, \top)$ of conforming subclasses with

$$\mathcal{A} = \{Q\_A : \overline{Q} \in \overline{H}(P_1, \ldots, P_m)\},$$

where $\bot = P_1\_\ldots\_P_m\_A$ and $\top = A$.

This theorem leads to a design technique similar to that of Section 5.1.

6 Specification

This section, which is essentially a summary of the results of [7], discusses the specifications for specialization hierarchies required in the class generation algorithm of Section 5.1. The specification will consist of the following parts:

(A) A description of all methods (signature and implementation) which will occur in some of the classes of the hierarchy
(B) A description from which the conjunctive closure can be derived
(C) A description from which invariants, pre- and postconditions can be derived.

From description (A) it is possible to derive (i). Since by Theorem 2 the postconditions of a method are the same in all classes in the hierarchy, this description also contains the postconditions from (C). Since the invariant *Inv* of the highest class in the hierarchy is also included in this description as part of (C), the required information (v) is also available. Part (B) specifies a set of predicates *PRED* over which the conjunctive closure is built. Any predicate $P \in \mathit{PRED}$ must be defined in the description in (A). In order to build the equivalence classes, (B) contains also a set of implications *IMPL*. An implication always has the form $P_1 \wedge \ldots \wedge P_k \Rightarrow P$, where $P_1, \ldots, P_k, P \in \mathit{PRED}$. To avoid creating classes based on conjunctions of contradictory predicates, it is reasonable to include in (B) also a set of contradictions *CONTR*. A contradiction is a set of predicates $X \subseteq \mathit{PRED}$ such that $\bigwedge_{P \in X} P \Leftrightarrow \mathit{false}$.

Example 6. We revisit the graph example (see Fig. 2). The predicates used for the conjunctive closure are $PRED = \{is\_acyclic, is\_wc, is\_tree\}$. The set of implications are: $IMPL = \{is\_tree \Rightarrow is\_acyclic, is\_tree \Rightarrow is\_wc\}$. If we add a method *is_sc* which returns *true* if and only if the directed graph is strongly connected, then we obtain $CONTR = \{\{is\_acyclic, is\_sc\}\}$.

From this information the conjunctive closure can be derived as follows:

1. Compute the directed acyclic graph $H := (2^{PRED}, \supset)$, where 2^{PRED} is the set of all subsets of $PRED$.
2. For any $X \Rightarrow P \in IMPL$ add edge (X_1, X_2) to H if $X \subseteq X_1$ and $X \cup P \subseteq X_2$.
3. Compute the strongly connected components of H and its reduced graph.
4. Remove any strongly connected component S containing an $X \in S$ such that there is a $Y \in CONTR$ with $Y \subseteq X$.

Step 1 models the conjunctive closure without equivalence classes. A set $X \subseteq PRED$ is interpreted as the conjunction of all predicates in X. Then it is $X \supset Y$ if and only if $X \Rightarrow Y$. If $X \Rightarrow P \in IMPL$ then X implies $X \wedge P$. Hence, any edge added in step 2 is still an implication. Therefore, any two sets X and Y in the same strongly connected component represent equivalent conjunctions. Therefore, if no contradictions occur, the reduced graph represents the conjunctive closure of $PRED$. A set Y represents a contradiction if there is an $X \in CONTR$ such that $X \subseteq Y$. Therefore, the strongly connected components which contain such a set are removed in step 4. Therefore, we have determined the information required by (iii) and (iv).

For the above specification for directed graphs, this algorithm leads to the conjunctive closure shown in Fig. 2.

In order to satisfy requirement (ii) for each method in description (A), the highest class in the specialization hierarchy where it occurs is specified by a conjunction of the predicates in $PRED$. Finally, it remains to show how requirement (vi) can be specified without considering any class in the specialization hierarchy. For this, observe that by the definition of specialization, it follows that if B is more special than A, and A contains a method m, then $Pre_{m,B} \Rightarrow Pre_{m,A}$. It is easy to see, that for any predicates P, Q such that $Q \Rightarrow P$ there is a predicate R such that $Q \Leftrightarrow P \wedge R$. We therefore add in the description (A) for any method m the precondition Pre_m of the highest class in the specialization hierarchy, and the description (C) is a *predicate table*, where for each predicate $P \in PRED$ and each method m a predicate R is specified such that $Pre_m \wedge R$ preserves P. R must be defined by methods of description (A). If these methods are free of side effects, then in any specialization class which preserves P, the precondition can be checked before; i.e., the implementation of m in such classes is preceded by a check of R.

Suppose method m should preserve $P_1 \in PRED$ as well as $P_2 \in PRED$. If the table entries are R_1 and R_2, then it is easy to see that the precondition $Pre_m \wedge R_1 \wedge R_2$ is sufficient to preserve $P_1 \wedge P_2$. Therefore, the predicate table is sufficient for deriving the preconditions required by (vi).

We conclude the discussion of the graph example by giving the complete specification according to the above discussion.

Example 7. Table 1 shows the predicate table for the directed graphs. Figure 3 specifies most of the methods belonging to some class in the robust class hierarchy. We omitted the specification of V and E and the implementations of the methods. It is not hard to include them. If a method m has return type $SAME$, then the return type of this method in a generated class A is A. E.g., the method *create* in class ADG has the signature $create : ADG(VERTEX, EDGE)$. There is also an unmentioned method *sym_clos* which computes the symmetric closure of a directed graph; i.e., for each edge, the anti-parallel edge is added. This method is used in the specification of *is_wc*. The specification requires a class SET with the usual operations and a class $LIST$. $v_1 \prec v_2$ denotes that v_1 precedes v_2 in a list. In the definition of *wcc* which computes weakly connected components, observe that the result type uses the specialization of weakly connected directed graphs. Hence, it is already specified by the signature that each component in the result is weakly connected.

The robust class hierarchy is generated over the set of predicates $PRED = \{is\_acyclic, is\_wc, is\_tree\}$. The following implications exist between these predicates: $IMPL = \{is\_tree \Rightarrow is\_acyclic, is\_tree \Rightarrow is\_wc\}$. There are no contradictions. The robust class hierarchy shown in Fig. 2 can be generated from these specifications. E.g., the invariant of class ADG is $Inv_{ADG} \equiv E \subseteq V \times V \wedge is\_acyclic$. The specification also tells that, for example, ADG has a method *topsort*. From the predicate table shown in Table 1 and the basic specification in Fig. 3 it is possible to derive the stronger preconditions in specializations. E.g., the precondition of the method *connect* in class $AWCDG$ is the conjunction of the preconditions in the basic specification, and of the entries in columns *is_acyclic* and *is_wc* and row *connect* of the predicate table. This conjunction yields the desired result:

$$(v_1 \in V \vee v_2 \in V) \wedge (v_1 \notin V \vee v_2 \notin V \vee \neg has\_path(v_2, v_1)).$$

Table 1. Predicate Table

	is_acyclic	*is_wc*	*is_tree*
$has\_path(v_1, v_2)$	*true*	*true*	*true*
$is\_art\_point(v)$	*true*	*true*	*true*
$is\_bridge(e)$	*true*	*true*	*true*
is_acyclic	*true*	*true*	*true*
is_wc	*true*	*true*	*true*
is_tree	*true*	*true*	*true*
wcc	*true*	*true*	*true*
create	*true*	*true*	*true*
$init(v)$	*true*	*true*	*true*
$add\_vertex(v)$	*true*	*false*	*false*
$connect(v_1, v_2)$	$v_1 \notin V \vee v_2 \notin V$ $\vee \neg has\_path(v_2, v_1)$	$v_1 \in V \vee v_2 \in V$	$Q(v_1, v_2)$
$del\_vertex(v)$	*true*	$v \notin V \vee \neg is\_art\_point(v)$	$v \notin V \vee \neg is\_art\_point(v)$
$del\_edge(e)$	*true*	$e \notin E \vee \neg is\_bridge(e)$	$e \notin E$
topsort		*true*	*true*

where $Q(v_1, v_2) = v_1 \in V \wedge v_2 \notin V \vee v_1 \notin V \wedge v_2 \in V \wedge idg(v_2) = 0 \vee (v_1, v_2) \in E$

```
class hierarchy $DG(VERTEX, EDGE)$
    invariant $E \subseteq V \times V$; -- invariant of the highest class
    methods
        $has\_path(v_1, v_2 : VERTEX) : BOOL$
            hierarchy $true$; -- contained in all classes
            pre $v_1 \in V \wedge v_2 \in V$; -- precondition in $DG$
            post $res = (v_1 = v_2 \vee \exists w : VERTEX : (v_1, w) \in E \wedge has\_path(w, v_1))$
        $is\_art\_point(v : VERTEX) : BOOL$
            hierarchy $true$; -- contained in all classes
            pre $v \in V$; -- precondition in $DG$
            post $res = |del\_vertex(v).wcc| > |wcc|$
        $is\_bridge(e : EDGE) : BOOL$
            hierarchy $true$; -- contained in all classes
            pre $e \in E$; -- precondition in $DG$
            post $res = |del\_edge(e).wcc| > |wcc|$
        $is\_acyclic : BOOL$
            hierarchy $true$; -- contained in all classes
            pre $true$; -- precondition in $DG$
            post $res = \neg\exists v : VERTEX : (v, v) \in E \wedge$
                      $\neg\exists v_1, v_2 : VERTEX : v_1 \neq v_2 \wedge has\_path(v_1, v_2) \wedge has\_path(v_2, v_1)$
        $is\_wc : BOOL$
            hierarchy $true$; -- contained in all classes
            pre $true$; -- precondition in $DG$
            post $res = \forall v_1, v_2 : VERTEX : v_1, v_2 \in V \Rightarrow sym\_clos.has\_path(v_1, v_2)$;
        $is\_tree : BOOL$
            hierarchy $true$; -- contained in all classes
            pre $true$; -- precondition in $DG$
            post $res = is\_acyclic \wedge is\_wc \wedge \forall v : VERTEX : v \in V \Rightarrow idg(v) \leq 1$
        $wcc : SET(WCDG(VERTEX, EDGE))$
            hierarchy $true$; -- contained in all classes
            pre $true$; -- precondition in $DG$
            post $\bigcup_{G \in res} G.V = V \wedge \bigcup_{G \in res} G.E = E \wedge$
                $\forall S : SET(WCDG(VERTEX, EDGE))$:
                $\bigcup_{G \in S} G.V = V \wedge \bigcup_{G \in S} G.E = E \Rightarrow |S| \geq |res|$
        $create : SAME$ -- specifies that the result type is the corresponding class
            -- in the class hierarchy
            hierarchy $true$; -- contained in all classes
            pre $true$; -- precondition in $DG$
            post $res.V.is\_empty \wedge res.E.is\_empty$
        $init(v : VERTEX) : SAME$
            hierarchy $true$; -- contained in all classes
            pre $true$; -- precondition in $DG$
            post $res.V = \{v\} \wedge res.E.is\_empty$
        $add\_vertex(v : VERTEX) : SAME$
            hierarchy $true$; -- contained in all classes
            pre $true$; -- precondition in $DG$
            post $res.V = V \cup \{v\} \wedge res.E = E$
        $connect(v_1, v_2 : VERTEX) : SAME$
            hierarchy $true$; -- contained in all classes
            pre $true$; -- precondition in $DG$
            post $res.V = V \cup \{v_1, v_2\} \wedge res.E = E \cup \{(v_1, v_2)\}$
        $del\_vertex(v : VERTEX) : SAME$
            hierarchy $true$; -- contained in all classes
            pre $true$; -- precondition in $DG$
            post $res.V = V \setminus \{v\} \wedge res.E = E \setminus \{e : \exists w : VERTEX : (v, w) \in E \vee (w, v) \in E\}$
        $del\_edge(e : EDGE) : SAME$
            hierarchy $true$; -- contained in all classes
            pre $true$; -- precondition in $DG$
            post $res.V = V \wedge res.E = E \setminus \{e\}$
        $topsort : SEQ(VERTEX)$
            hierarchy $is\_acyclic$; -- contained in $ADG$ and all its specializations
            pre $true$; -- precondition in $ADG$
            post $\forall v_1, v_2 : VERTEX : v_1, v_2 \in res \wedge v_1 \prec v_2 \Rightarrow v_1, v_2 \in V \wedge \neg has\_path(v_2, v_1)$
```

Fig. 3. Basic Specification.

The three parts of the specifications can be summarized as follows:

- Part (A), called the *basic specification*, specifies for any method m occurring in some class in the robust class hierarchy
  - the signature
  - a set of predicates $H_m \subseteq PRED$ which specifies the highest class A in the specialization hierarchy where m occurs
  - an implementation of m for A (alternatively, the method could be abstract),
  - a precondition Pre_m, and a postcondition $Post_m$ for A.

  Part (A) also specifies the invariant *Inv* of the highest class in the hierarchy.
- Part (B), called the *hierarchy specification*, specifies the set of predicates *PRED* over which the robust class hierarchy is constructed, a set of implications *IMPL* between these predicates, and a set of *CONTR* of contradictions.
- Part (C) specifies the predicate table as described above.

From this specification we can derive an initial specialization hierarchy which then can be transformed into a hierarchy of conforming classes.

7 Conclusion

There are different kinds of inheritance relationships between classes. Only one of them can be said to be robust under polymorphism, the conformance relationship. However, specialization is at least as important in practice. This leads to several questions: How can the right inheritance relationship between two classes be determined? How can robust class hierarchies be created?

This article answers these questions. We developed techniques for the design of class hierarchies. We showed how specialization hierarchies arise from the definition of predicates over the instances of a given class A. These predicates can be partially ordered using the implication relationship. It was also shown that this partial order induces a specialization hierarchy of the corresponding classes, where the root class of the hierarchy is the initial class A. Theorem 1 shows that specialization hierarchies can be seen as covariant hierarchies. It is still an open question whether these two kinds of inheritance relationships appear separately. We conjecture that the answer might be found by studying algebraic data structures.

A quite similar procedure led to a lattice of conforming subclasses. In this case, we examined generic classes and predicates over the instantiations of their generic parameters. It was shown that every valid instantiation of such a restriction leads to a conforming generic subclass, which means that all instantiations of the restricted generic class are conforming subclasses of the initial generic class (without the additional restrictions). We pointed out the use of such restrictions using a well known example. We then showed how a table-based specification of a specialization hierarchy can be constructed in a rigorous way. An algorithm for the transformation of this hierarchy into a structure of conforming classes was given in Section 5.

The advantages of this construction and transformation method are:

- The systematic approach helps to identify incomplete specifications. Furthermore, experience has shown that it helps to reduce the amount of specification ambiguities.
- the formal transformation step generates a robust class structure from the specification. Of course, this transformation can be used independent from the specification method; e.g., to reorganize class structures.
- The generation method can be modified in a way that only needed classes are generated.
- Hierarchies can be generated *on the fly*; e.g., by a library management tool.
- Class hierarchies are intuitively designed and understood as specialization hierarchies. The method therefore bridges the gap between design and robust use of class structures.

To our knowledge, there exists no comparable work in this specific field of software engineering. In [4] we already showed that reusability heavily depends on robustness. This article provides a *specification-based* approach to achieve reuse, in contrast to the more common idea of *class-based reuse.* Especially the field of compiler construction has shown that reuse of concepts, such as specifications, architectures and patterns, is rather more necessary than reuse of single components, such as a single class.

To investigate the scalability of our technique, we are currently implementing a specification-based prototype to generate classes, which will be subsequently used to further develop KARLA. Initial experience has shown that this technique works very well for the design of local class clusters such as the graph classes mentioned above. It appears to be more useful in general to develop *deep* class hierarchies, such as parts of libraries, rather than to develop frameworks or applications.

The techniques developed in this article provide the basis for future work on specification methods for class structures. A good starting point for the development of a rigorous and systematic software construction process is design patterns [8]. To be applicable for the generation of robust class libraries, however, they need to be more formally specified.

References

1. G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings Publishing Company, 1991.
2. K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings 18th International Conference on Software Engineering*, pages 258–267. IEEE, 1996.
3. A. Frick, R. Neumann, and W. Zimmermann. A method for the construction of robust class hierarchies. In *Proceedings of WOON' 96*, 1996.
4. A. Frick, W. Zimmer, and W. Zimmermann. On the design of reliable libraries. In *TOOLS 17—Technology of Object-Oriented Programming*, pages 13–23, 1995.
5. A. Frick, W. Zimmer, and W. Zimmermann. Konstruktion robuster und flexibler Klassenbibliotheken. *Informatik—Forschung und Entwicklung*, 11(4):168–178, 1996. A preliminary version of this article was published in *Softwaretechnik'95*, pp. 35–46.

6. Arne Frick, Rainer Neumann, and Wolf Zimmermann. Eine Methode zur Konstruktion robuster Klassenhierarchien. *Informatik—Forschung und Entwicklung*, 12(4):186–195, 1997.
7. Jozsef Frigo, Rainer Neumann, and Wolf Zimmermann. Generation of robust class hierarchies. In *TOOLS 23—Technology of Object-Oriented Programming and Systems*, pages 282–291, 1997.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software Components*. Addison-Wesley, 1995.
9. B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
10. R. Mitchell, I. Maung, J. Howse, and T. Heathcote. Checking software contracts. In R. Ege, M. Singh, and B. Meyer, editors, *TOOLS 17 – Technology of Object-Oriented Languages and Systemes*, pages 97–106, 1995.
11. H. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, 1991.
12. H. W. Schmidt and R. Walker. TOF: An efficient type system for objects and functions. Technical Report TR-CS-92-17, Department of Computer Science, The Australian National University, November 1992.

Exception-Safety in Generic Components
Lessons Learned from Specifying Exception-Safety for the C++ Standard Library

David Abrahams

Dragon Systems
`David_Abrahams@dragonsys.com`

Abstract. This paper represents the knowledge accumulated in response to a real-world need: that the C++ Standard Template Library exhibit useful and well-defined interactions with exceptions, the error-handling mechanism built-in to the core C++ language. It explores the meaning of exception-safety, reveals surprising myths about exceptions and genericity, describes valuable tools for reasoning about program correctness, and outlines an automated testing procedure for verifying exception-safety.

Keywords: exception-safety, exceptions, STL, C++

1 What Is Exception-Safety?

Informally, exception-safety in a component means that it exhibits reasonable behavior when an exception is thrown during its execution. For most people, the term "reasonable" includes all the usual expectations for error-handling: that resources should not be leaked, and that the program should remain in a well-defined state so that execution can continue. For most components, it also includes the expectation that when an error is encountered, it is reported to the caller.

More formally, we can describe a component as minimally exception-safe if, when exceptions are thrown from within that component, its invariants are intact. Later on we'll see that at least three different levels of exception-safety can be usefully distinguished. These distinctions can help us to describe and reason about the behavior of large systems.

In a generic component, we usually have an additional expectation of *exception-neutrality*, which means that exceptions thrown by a component's type parameters should be propagated, unchanged, to the component's caller.

2 Myths and Superstitions

Exception-safety seems straightforward so far: it doesn't constitute anything more than we'd expect from code using more traditional error-handling techniques. It might be worthwhile, however, to examine the term from a psychological viewpoint. Nobody ever spoke of "error-safety" before C++ had exceptions.

It's almost as though exceptions are viewed as a *mysterious attack* on otherwise correct code, from which we must protect ourselves. Needless to say, this doesn't lead to a healthy relationship with error handling! During standardization, a democratic process which requires broad support for changes, I encountered many widely-held superstitions. In order to even begin the discussion of exception-safety in generic components, it may be worthwhile confronting a few of them.

"Interactions between templates and exceptions are not well-understood." This myth, often heard from those who consider these both *new* language features, is easily disposed of: there simply are no interactions. A template, once instantiated, works in all respects like an ordinary class or function. A simple way to reason about the behavior of a template with exceptions is to think of how a specific instantiation of that template works. Finally, the genericity of templates should not cause special concern. Although the component's client supplies part of the operation (which may, unless otherwise specified, throw arbitrary exceptions), the same is true of operations using familiar virtual functions or simple function pointers.

"It is well known to be impossible to write an exception-safe generic container." This claim is often heard with reference to an article by Tom Cargill [4] in which he explores the problem of exception-safety for a generic stack template. In his article, Cargill raises many useful questions, but unfortunately fails to present a solution to his problem.[1] He concludes by suggesting that a solution may not be possible. Unfortunately, his article was read by many as "proof" of that speculation. Since it was published there have been many examples of exception-safe generic components, among them the C++ standard library containers.

"Dealing with exceptions will slow code down, and templates are used specifically to get the best possible performance." A good implementation of C++ will not devote a single instruction cycle to dealing with exceptions until one is thrown, and then it can be handled at a speed comparable with that of calling a function [7]. That alone gives programs using exceptions performance equivalent to that of a program which ignores the possibility of errors. Using exceptions can actually result in faster programs than "traditional" error handling methods for other reasons. First, a catch block clearly indicates to the compiler which code is devoted to error-handling; it can then be separated from the usual execution-path, improving locality of reference. Second, code using "traditional" error handling must typically test a return value for errors after every single function call; using exceptions completely eliminates that overhead.

"Exceptions make it more difficult to reason about a program's behavior." Usually cited in support of this myth is the way "hidden" execution paths are followed during stack-unwinding. Hidden execution paths are nothing new to

[1] Probably the greatest impediment to a solution in Cargill's case was an unfortunate combination of choices on his part: the interface he chose for his container was incompatible with his particular demands for safety. By changing either one he might have solved the problem.

any C++ programmer who expects local variables to be destroyed upon returning from a function:

```
ErrorCode f( int& result )             // 1
{                                      // 2
   X x;                                // 3
   ErrorCode err = x.g( result );      // 4
   if ( err != kNoError )              // 5
           return err;                 // 6
   // ...More code here...
   return kNoError;                    // 7
}
```

In the example above, there is a “hidden” call to `X::~X()` in lines 6 and 7. Granted, using exceptions, there is *no* code devoted to error handling visible:

```
int f()                                // 1
{                                      // 2
   X x;                                // 3
   int result = x.g();                 // 4
   // ...More code here...
   return result;                      // 5
}
```

For many programmers more familiar with exceptions, the second example is actually more readable and understandable than the first. The “hidden” code paths include the same calls to destructors of local variables. In addition, they follow a simple pattern which acts *exactly* as though there were a potential `return` statement after each function call in case of an exception. Readability is enhanced because the normal path of execution is unobscured by error-handling, and return values are freed up to be used in a natural way.

There is an even more important way in which exceptions can enhance correctness: by allowing simple class invariants. In the first example, if `x`'s constructor should need to allocate resources, it has no way to report a failure: in C++, constructors have no return values. The usual result when exceptions are avoided is that classes requiring resources must include a separate initializer function which finishes the job of construction. The programmer can therefore never be sure, when an object of class `X` is used, whether he is handling a full-fledged `X` or some abortive attempt to construct one (or worse: someone simply forgot to call the initializer!)

3 A Contractual Basis for Exception-Safety

A non-generic component can be described as exception-safe in isolation, but because of its configurability by client code, exception-safety in a generic component usually depends on a contract between the component and its clients. For example, the designer of a generic component might require that an operation which is used in the component's destructor not throw any exceptions.[2] The generic component might, in return, provide one of the following guarantees:

– The *basic* guarantee: that the invariants of the component are preserved, and no resources are leaked.
– The *strong* guarantee: that the operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started.
– The *no-throw* guarantee: that the operation will not throw an exception.

The *basic* guarantee is a simple minimum standard for exception-safety to which we can hold all components. It says simply that after an exception, the component can still be used as before. Importantly, the preservation of invariants allows the component to be destroyed, potentially as part of stack-unwinding. This guarantee is actually less useful than it might at first appear. If a component has many valid states, after an exception we have no idea what state the component is in—only that the state is valid. The options for recovery in this case are limited: either destruction or resetting the component to some known state before further use. Consider the following example:

```
template <class>
void print_random_sequence()
{
   std::vector<X> v(10);   // A vector of 10 items

   try {
      // Provides only the basic guarantee
      v.insert( v.begin(), X() );
   }
   catch(...) {}   // ignore any exceptions above

   // print the vector's contents
   std::cout "(" << v.size() << ") ";
   std::copy( v.begin(), v.end(),
         std::ostream_iterator<X>( std::cout, " " ) );
}
```

Since all we know about `v` after an exception is that it is valid, the function is allowed to print any random sequence of `X`s.[3] It is "safe" in the sense that it is not allowed to crash, but its output may be unpredictable.

[2] It is usually inadvisable to throw an exception from a destructor in C++, since the destructor may itself be called during the stack-unwinding caused by another exception. If the second exception is allowed to propagate beyond the destructor, the program is immediately terminated.

[3] In practice of course, this function would make an extremely poor random sequence generator!

The *strong* guarantee provides full "commit-or-rollback" semantics. In the case of C++ standard containers, this means, for example, that if an exception is thrown all iterators remain valid. We also know that the container has exactly the same elements as before the exception was thrown. A transaction that has no effects if it fails has obvious benefits: the program state is simple and predictable in case of an exception. In the C++ standard library, nearly all of the operations on the node-based containers `list`, `set`, `multiset`, `map`, and `multimap` provide the strong guarantee. [4]).

The *no-throw* guarantee is the strongest of all, and it says that an operation is guaranteed not to throw an exception: it always completes successfully. This guarantee is necessary for most destructors, and indeed the destructors of C++ standard library components are all guaranteed not to throw exceptions. The no-throw guarantee turns out to be important for other reasons, as we shall see.[5]

4 Legal Wrangling

Inevitably, the contract can get more complicated: a *quid pro quo* arrangement is possible. Some components in the C++ Standard Library give one guarantee for arbitrary type parameters, but give a stronger guarantee in exchange for additional promises from the client type that no exceptions will be thrown. For example, the standard container operation `vector<T>::erase` gives the basic guarantee for any `T`, but for types whose copy constructor and copy assignment operator do not throw, it gives the no-throw guarantee.[6]

5 What Level of Exception-Safety Should a Component Specify?

From a client's point-of-view, the strongest possible level of safety would be ideal. Of course, the no-throw guarantee is simply impossible for many operations, but what about the strong guarantee? For example, suppose we wanted atomic behavior for `vector<T>::insert`. Insertion into the middle of a vector requires copying elements after the insertion point into later positions, to make room for the new element. If copying an element can fail, rolling back the operation would

[4] It is worth noting that mutating algorithms usually cannot provide the strong guarantee: to roll back a modified element of a range, it must be set back to its previous value using `operator=`, which itself might throw. In the C++ standard library, there are a few exceptions to this rule, whose rollback behavior consists only of destruction: `uninitialized_copy`, `uninitialized_fill`, and `uninitialized_fill_n`.

[5] All type parameters supplied by clients of the C++ standard library are required not to throw from their destructors. In return, all components of the C++ standard library provide at least the basic guarantee.

[6] Similar arrangements might have been made in the C++ standard for many of the mutating algorithms, but were never considered due to time constraints on the standardization process.

require “undoing” the previous copies ... which depends on copying again. If copying back should fail (as it likely would), we have failed to meet our guarantee.

One possible alternative would be to redefine `insert` to build the new array contents in a fresh piece of memory each time, and only destroy the old contents when that has succeeded. Unfortunately, there is a non-trivial cost if this approach is followed: insertions near the end of a vector which might have previously caused only a few copies would now cause *every* element to be copied. The basic guarantee is a “natural” level of safety for this operation, which it can provide without violating its performance guarantees. In fact all of the operations in the library appear to have such a “natural” level of safety.

Because performance requirements were already a well-established part of the draft standard and because performance is a primary goal of the STL, there was no attempt to specify more safety than could be provided within those requirements. Although not all of the library gives the strong guarantee, almost any operation on a standard container which gives the basic guarantee can be made strong using the “make a new copy” strategy described above:

```
template <class Container, class BasicOp>
void MakeOperationStrong( Container& c, const BasicOp& op )
{
   Container tmp( c );                // Copy c
   op( tmp );                         // Work on the copy
   c.swap( tmp );                     // Cannot fail[7]
}
```

This technique can be folded into a wrapper class to make a similar container which provides stronger guarantees (and different performance characteristics).[8]

6 Should We Take Everything We Can Get?

By considering a particular implementation, we can hope to discern a natural level of safety. The danger in using this to establish requirements for a component is that the implementation might be restricted. If someone should come up with a more-efficient implementation which we’d like to use, we may find that it’s incompatible with our exception-safety requirements. One might expect this to be of no concern in the well-explored domains of data structures and algorithms covered by the STL, but even there, advances are being made. A good example is the recent *introsort* algorithm [6], which represents a substantial improvement in worst-case complexity over the well-established *quicksort*.

To determine exactly how much to demand of the standard components, I looked at a typical real-world scenario. The chosen test case was a “composite

[7] Associative containers whose `Compare` object might throw an exception when copied cannot use this technique, since the `swap` function might fail.

[8] This suggests another potential use for the oft-wished-for but as yet unseen `container_traits<>` template: automated container selection to meet exception-safety constraints.

container." Such a container, built of two or more standard container components, is not only commonly needed, but serves as a simple representative case for maintaining invariants in larger systems:

```
// SearchableStack - A stack which can be efficiently searched
// for any value.

template <class T>
class SearchableStack
{
public:
   void push(const T& t);                 // O(log n)
   void pop();                            // O(log n)
   bool contains(const T& t) const;       // O(log n)
   const T& top() const;                  // O(1)
private:
   std::set<T> set_impl;
   std::list<std::set<T>::iterator> list_impl;
};
```

The idea is that the list acts as a stack of set iterators: every element goes into the set first, and the resulting position is pushed onto the list. The invariant is straightforward: the set and the list should always have the same number of elements, and every element of the set should be referenced by an element of the list. The following implementation of the push function is designed to give the *strong* guarantee within the natural levels of safety provided by set and list:

```
template <class T>                               // 1
void SearchableStack<T>::push(const T& t)        // 2
{                                                // 3
   set<T>::iterator i = set_impl.insert(t);      // 4
   try                                           // 5
   {                                             // 6
      list_impl.push_back(i);                    // 7
   }                                             // 8
   catch(...)                                    // 9
   {                                             // 10
      set_impl.erase(i);                         // 11
      throw;                                     // 12
   }                                             // 13
}                                                // 14
```

What does our code actually require of the library? We need to examine the lines where non-const operations occur:

– Line 4: if the insertion fails but `set_impl` is modified in the process, our invariant is violated. We need to be able to rely on the strong guarantee from `set<T>::insert`.

- Line 7: likewise, if `push_back` fails, but `list_impl` is modified in the process, our invariant is violated, so we need to be able to rely on the strong guarantee from `list<T>::insert`.
- Line 11: here we are "rolling back" the insertion on line 4. If this operation should fail, we will be unable to restore our invariant. We absolutely depend on the no-throw guarantee from `set<T>::erase`.[9]
- Line 11: for the same reasons, we also depend on being able to pass the `i` to the `erase` function: we need the no-throw guarantee from the copy constructor of `set<T>::iterator`.

I learned a great deal by approaching the question this way during standardization. First, the guarantee specified for the composite container actually depends on stronger guarantees from its components (the no-throw guarantees in line 11). Also, I took advantage of all of the natural level of safety to implement this simple example. Finally, the analysis revealed a requirement on iterators which I had previously overlooked when operations were considered on their own. The conclusion was that we should provide as much of the natural level of safety as possible. Faster but less-safe implementations could always be provided as extensions to the standard components.[10]

7 Automated Testing for Exception-Safety

As part of the standardization process, I produced an exception-safe reference implementation of the STL. Error-handling code is seldom rigorously tested in real life, in part because it is difficult to cause error conditions to occur. It is very common to see error-handling code which crashes the first time it is executed ... in a shipping product! To bolster confidence that the implementation actually worked as advertised, I designed an automated test suite, based on an exhaustive technique due to my colleague Matt Arnold.

The test program started with the basics: reinforcement and instrumentation, especially of the global operators `new` and `delete`.[11] Instances of the components (containers and algorithms) were created, with type parameters chosen to reveal

[9] One might be tempted to surround the erase operation with a try/catch block to reduce the requirements on `set<T>` and the problems that arise in case of an exception, but in the end that just begs the question. First, `erase` just failed and in this case there are no viable alternative ways to produce the necessary result. Second and more generally, because of the variability of its type parameters a generic component can seldom be assured that any alternatives will succeed.

[10] The prevalent philosophy in the design of STL was that functionality that wasn't essential to all uses should be left out in favor of efficiency, as long as that functionality could be obtained when needed by adapting the base components. This departs from that philosophy, but it would be difficult or impossible to obtain even the basic guarantee by adapting a base component that doesn't already have it.

[11] An excellent discussion on how to fortify memory subsystems can be found in: Steve Maguire, Writing Solid Code, Microsoft Press, Redmond, WA, 1993, ISBN 1-55615-551-4.

as many potential problems as possible. For example, all type parameters were given a pointer to heap-allocated memory, so that leaking a contained object would be detected as a memory leak.

Finally, a scheme was designed that could cause an operation to throw an exception at each possible point of failure. At the beginning of every client-supplied operation which is allowed to throw an exception, a call to `ThisCanThrow` was added. A call to `ThisCanThrow` also had to be added everywhere that the generic operation being tested might throw an exception, for example in the global operator `new`, for which an instrumented replacement was supplied.

```
// Use this as a type parameter, e.g. vector<TestClass>
struct TestClass

{
   TestClass( int v = 0 )
     : p( ThisCanThrow(), new int( v ) ) {}
   TestClass( const TestClass& rhs )
    : p( ThisCanThrow(), new int( *rhs.p ) ) {}
   const TestClass& operator=( const TestClass& rhs )
    { ThisCanThrow(); *p = *rhs.p; }
   bool operator==( const TestClass& rhs )
    { ThisCanThrow(); return *p == *rhs.p; }
   ...etc...
   ~TestClass() { delete p; }
};
```

`ThisCanThrow` simply decrements a "throw counter" and, if it has reached zero, throws an exception. Each test takes a form which begins the counter at successively higher values in an outer loop and repeatedly attempts to complete the operation being tested. The result is that the operation throws an exception at each successive step along its execution path that can possibly fail. For example, here is a simplified version of the function used to test the strong guarantee:[12]

```
extern int gThrowCounter;                     // The throw counter

void ThisCanThrow()
{
  if ( gThrowCounter-- == 0 )
      throw 0;
}
```

[12] Note that this technique requires that the operation being tested be exception-neutral. If the operation ever tries to recover from an exception and proceed, the throw counter will be negative, and subsequent operations that might fail will not be tested for exception-safety.

```
template <class Value, class Operation>
void StrongCheck( const Value& v, const Operation& op )
{
   bool succeeded = false;
   for ( long nextThrowCount = 0; !succeeded; ++nextThrowCount )
   {
      Value duplicate = v;
      try
      {
         gThrowCounter = nextThrowCount;
         op( duplicate );                 // Try the operation
         succeeded = true;
      }
      catch(...)                          // Catch all exceptions
      {
         bool unchanged = duplicate == v; // Test strong guarantee
         assert( unchanged );
      }
      // Specialize as desired for each container type, to check
      // integrity. For example, size() == distance(begin(),end())
      CheckInvariant(v);       // Check any invariant
   }
}
```

Notably, this kind of testing is much easier and less intrusive with a generic component than with non-generics, because testing-specific type parameters can be used without modifying the source code of the component being tested. Also, generic functions like `StrongCheck` above were instrumental in performing the tests on a wide range of values and operations.

8 Further Reading

To my knowledge, there are currently only two descriptions of STL exception-safety available. The original specification [2] for the reference exception-safe implementation of the STL is an informal specification, simple and self-explanatory (also verbose), and uses the basic- and strong-guarantee distinctions outlined in this article. It explicitly forbids leaks, and differs substantively from the final C++ standard in the guarantees it makes, though they are largely identical. I hope to produce an updated version of this document soon.

The description of exception-safety in the C++ Standard [1] is only slightly more formal, but relies on hard-to-read "standardese" and an occasionally subtle web of implication.[13] In particular, leaks are not treated directly at all. It does have the advantage that it *is* the standard.

[13] The changes to the draft standard which introduced exception-safety were made late in the process, when amendments were likely to be rejected solely on the basis of the number of altered words. Unfortunately, the result compromises clarity somewhat

The original reference implementation [5] of the exception-safe STL is an adaptation of an old version of the SGI STL, designed for C++ compilers with limited features. Although it is not a complete STL implementation, the code may be easier to read, and it illustrates a useful base-class technique for eliminating exception-handling code in constructors.

The full test suite [3] used to validate the reference implementation has been used successfully to validate all recent versions of the SGI STL, and has been adapted to test one other vendor's implementation (which failed). As noted on the documentation page, it also seems to have the power to reveal hidden compiler bugs, particularly where optimizers interact with exception-handling code.

References

1. International Standard ISO/IEC 14882, *Information Technology—Programming Languages—C++*, Document Number ISO/IEC 14882-1998, available from `http://webstore.ansi.org/ansidocstore/default.asp`.
2. D. Abrahams, *Exception Safety in STLport*, available at `http://www.stlport.org/doc/exception_safety.html`.
3. D. Abrahams and B. Fomitchev, *Exception Handling Test Suite*, available at `http://www.stlport.org/doc/eh_testsuite.html`.
4. Tom Cargill, "Exception Handling: A False Sense of Security," *C++ Report*, Nov-Dec 1994, also available at `http://www.awl.com/cp/mec++-cargill.html`.
5. B. Fomitchev, *Adapted SGI STL Version 1.0*, with exception handling code by D. Abrahams, available at `http://www.metabyte.com/~fbp/stl/old.html`.
6. D. R. Musser, "Introspective Sorting and Selection Algorithms," *Software—Practice and Experience* 27(8):983–993, 1997.
7. Bjarne Stroustrup, *The Design And Evolution of C++*. Addison Wesley, Reading, MA, 1995, ISBN 0-201-54330-3, Section 16.9.1.

in favor of brevity. Greg Colvin was responsible for the clever language-lawyering needed to minimize the extent of these changes.

Segmented Iterators and Hierarchical Algorithms

Matthew H. Austern

Silicon Graphics Computer Systems
austern@sgi.com

Abstract. Many data structures are naturally segmented. Generic algorithms that ignore that feature, and that treat every data structure as a uniform range of elements, are unnecessarily inefficient. A new kind of iterator abstraction, in which segmentation is explicit, makes it possible to write hierarchical algorithms that exploit segmentation.

Keywords: Standard Template Library, multidimensional data structures, iterators

1 Introduction

A defining characteristic of generic programming is "Lifting of a concrete algorithm to as general a level as possible without losing efficiency; *i.e.*, the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm." The best known example of a generic library is the C++ Standard Template Library, or STL [1,2,3], a collection of algorithms and data structures dealing with one-dimensional ranges of elements.

For an important class of data structures, those which exhibit segmentation, the STL does not make it possible to write algorithms that satisfy the goal of abstraction without loss of efficiency. A generic algorithm written within the framework of the STL can't exploit segmentation. The difficulty is a limitation of the STL's central abstraction: iterators.

2 Iterators and Algorithms

The STL is mainly concerned with algorithms on one-dimensional ranges. As a simple example of such an algorithm, consider the operation of assigning a value to the elements of an array. In C, we can write this operation as follows:

```
void fill1(char* first, char* last, char value)
{
  for ( ; first != last; ++first)
    *first = value;
}
```

The argument `first` is a pointer to the beginning of the array: the element that `first` points to, `*first`, is the first element of the array. The argument `last` is a pointer just past the end of the array. That is, `fill1` performs the assignments `*first = value`, `*(first + 1) = value`, and so on up to but not including `last`.

The arguments `first` and `last` form a *range*, [`first`, `last`), that contains the elements from `first` up to but not including `last`. The range [`first`, `last`) contains `last` − `first` elements.

The function `fill1` is similar to `memset`, from the standard C library; like `memset`, it can only be used for assigning a value to an array of type `char`. In C++ [4,5] it is possible to write a more general function, `fill`:

```
template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last,
          T value)
{
  for ( ; first != last; ++first)
    *first = value;
}
```

This is an example of a generic algorithm written in the STL framework. Like `fill1`, `fill` steps through the range [`first`, `last`) from beginning to end. For each iteration it tests whether there are any remaining elements in the range, and, if there are, it performs the assignment `*first = value` and then increments `first`. The difference between `fill1` and `fill` is how the arguments `first` and `last` are declared: in `fill1` they are pointers of type `char*`, but in `fill`, a function template, they may be of any type `ForwardIterator` for which the operations in `fill` are well-defined. The type `ForwardIterator` need not be a pointer type; it need only conform to a pointer-like interface. It must support the following operations:

– *Copying and assignment*: Objects of type `ForwardIterator` can be copied, and one object of type `ForwardIterator` can be assigned to another.
– *Dereference:* An object of type `ForwardIterator` can be dereferenced. If `i` is an object of type `ForwardIterator`, then the expression `*i` returns a reference to some C++ object. The type of that object is known as `ForwardIterator`'s *value type.*
– *Comparison for equality*: Objects of type `ForwardIterator` can be compared for equality. If `i` and `j` are values of type `ForwardIterator`, then the expressions `i == j` and `i != j` are well-defined. Two dereferenceable iterators `i` and `j` are equal if and only if `*i` and `*j` refer to the same object.
– *Increment:* Objects of type `ForwardIterator` can be incremented. If `i` is an object of type `ForwardIterator`, then `++i` modifies `i` so that it points to the next element.

These requirements are known as the Forward Iterator requirements, or the Forward Iterator concept, and a type that conforms to them is known as a Forward

Iterator. Pointers are Forward Iterators, for example, and it is also possible to define Forward Iterators that step through singly or doubly lists, segmented arrays, and many other data structures. Since operators in C++ can be overloaded, a user-defined iterator type can ensure that expressions like `*i` and `++i` have the appropriate behavior.

The Forward Iterator requirements are part of the STL, and so are algorithms, like `fill`, that operate on Forward Iterators. The STL also defines several other iterator concepts. A Bidirectional Iterator type, for example, is a type that conforms to all of the Forward Iterator and that provides additional functionality: a Bidirectional Iterator `i` can be decremented, using the expression `--i`, as well as incremented. Similarly, a Random Access Iterator type conforms to all of the Bidirectional Iterator requirements and also provides more general arithmetic operations. Random Access Iterators provide such operations as `i += n`, `i[n]`, and `i - j`.

Pointers are Random Access Iterators, which means that they are Bidirectional Iterators and Forward Iterators as well.

Dispatching algorithms and iterator traits The reason for the different iterator concepts is that they support different kinds of algorithms. Forward Iterators are sufficient for an algorithm like `fill`, which steps through a range from one element to the next, but they are not sufficient for an algorithm like Shell sort, which requires arbitrary-sized steps. If Shell sort were implemented as a generic STL algorithm, it would have to use Random Access Iterators. An algorithm written to use Forward Iterators can be called with arguments that are Random Access Iterators (every Random Access Iterator is also a Forward Iterator), but not the other way around.

It's obvious what kind of iterator an algorithm like Shell sort ought to use, but sometimes the choice is less obvious. Consider, for example, the problem of reversing the elements in a range. It is possible to write a reverse algorithm that uses Forward Iterators, or one that uses Bidirectional Iterators. The version that uses Forward Iterators is more general, but the version that uses Bidirectional Iterators is faster when it can be called at all.

Both generality and efficiency are important, so the STL introduces the notion of *dispatching algorithms*: algorithms that select one of several methods depending on the kind of iterator that they are invoked with. The dispatch mechanism uses a traits class and a set of iterator tag classes, and it incurs no runtime performance penalty.

The iterator tags are a set of five empty classes, each of which is a placeholder corresponding to an iterator concept.[1] The traits class is a template, `iterator_traits`, whose nested types supply information about an iterator.

For any iterator type `Iter`, `iterator_traits<Iter>::value_type` is `Iter`'s value type and `iterator_traits<Iter>::iterator_category` is the iterator

[1] The five iterator concepts are Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator, and Random Access Iterator. This paper does not discuss Input Iterator and Output Iterator

tag type corresponding to the most specific concept that Iter conforms to. For example, iterator_traits<char*>::value_type is char and iterator_traits<char*>::iterator_category is the category tag for Random Access Iterators, random_access_iterator_tag. The traits mechanism is quite general, and has many different uses.

Given this machinery, writing a dispatching algorithm is simple. The algorithm, using iterator_traits, calls an overloaded helper function with the iterator's category tag as an argument. The compiler selects the appropriate helper function via ordinary overload resolution. For example, here is the skeleton of reverse:

```
template <class ForwardIter>
void reverse(ForwardIter first, ForwardIter last,
             forward_iterator_tag) {
  ...
}

template <class BidirectionalIter>
void reverse(BidirectionalIter first, BidirectionalIter last,
             bidirectional_iterator_tag) {
  ...
}

template <class RandomAccessIter>
void reverse(RandomAccessIter first, RandomAccessIter last,
             random_access_iterator_tag) {
  ...
}

template <class Iter>
inline void reverse(Iter first, Iter last) {
  typedef typename iterator_traits<Iter>::iterator_category
          category;
  reverse(first, last, category());
}
```

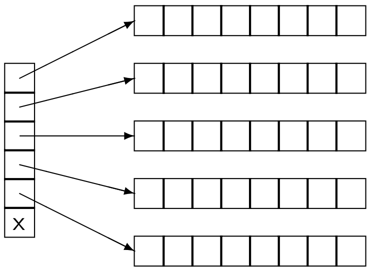
3 Segmented Data Structures

The STL defines five different iterator concepts, but all of them have one thing in common: each of them represents a position in a uniform one-dimensional range. Given a Forward Iterator i, for example, the next iterator in the range (if there is a next) is always ++i. As far as an algorithm that uses Forward Iterators is concerned, every increment operation is the same. Similarly, given a range [first, last) of Random Access Iterators, every iterator in the range is equal to first + n for some integer n. Again, the range is uniform and one-dimensional; every position within it is characterized by a single measure of distance.

Many data structures can be viewed as linear ranges (any finite collection can be arranged in *some* order), but there are data structures for which a different view is also appropriate. Often, the most natural index for an element isn't a single offset but a more complicated multidimensional description. One example is a segmented array, or a vector of vectors.

```
template <class T> struct seg_array
{
  typedef T value_type;
  typedef vector<T> node;
  vector<node*> nodes
  ...
};
```

Here is a simple pictorial representation:



Elements in the segmented array are contained in nodes, each of which is a vector; each element may be characterized by a node and an index within the node. The outer vector consists of pointers to nodes, and, to make it easier to detect the end of the array, the last entry in the outer vector is a null pointer. (Represented in the picture by an "x.")

This isn't an artificial example: segmented data structures are common. Within the STL, for example, the deque container is typically implemented as a segmented array, and hash tables [6] are typically implemented as arrays of buckets. Other examples of segmentation include two-dimensional matrices, graphs, files in record-based file systems, and, on modern NUMA (non-uniform memory architecture) multiprocessor machines, even ordinary memory.

The most natural way of assigning a value to all of the elements in a seg_array is a nested loop: loop through all of the nodes, and, for each node, assign a value to all of the node's elements.

```
vector<vector<T>*>::iterator i;
for (i = nodes.begin(); i != nodes.end(); ++i) {
  vector<T>::iterator j;
  for (j = (**i).begin(); j != (**i).end(); ++j)
    *j = value;
}
```

Instead of writing a new function to assign a value to every element in a `seg_array`, it is preferable to reuse the existing generic algorithm `fill`. It is merely necessary to define a Forward Iterator that steps through a `seg_array`, which, in turn, means defining a sense in which the elements of a `seg_array` can be viewed as a simple linear range. There is a straightforward linearization: an element's successor is the next element in the same node, or, if no such element exists, the first element in the next node. This definition can be used to write an iterator for `seg_array`, `seg_array_iter`:

```
template <class T> struct seg_array_iter
{
  vector<T>::iterator          cur;
  vector<vector<T>*>::iterator node;

  T& operator*() const { return *cur; }

  seg_array_iter& operator++() {
    if (++cur == (**node).end()) {
      ++node;
      cur = *node ? (**node).begin() : 0;
    }
  }
  ...
};
```

Using a `seg_array_iter` with `fill` is simple, but it changes the algorithm. Every time a `seg_array_iter` is incremented, it checks to see if it has arrived at the end of a node. Instead of a nested loop, the combination of `seg_array_iter` and `fill` is more like this:

```
vector<vector<T>*>::iterator node = nodes.begin();
vector<T>::iterator cur = (**node).begin();
while (node != nodes.end()) {
  *cur = value;
  ++cur;
  if (cur == (**node).end()) {
    ++node;
    if (node != nodes.end())
      cur = (**node).begin();
  }
}
```

As expected, the loop overhead is higher. On a 195 MHz SGI^{TM} Origin2000 system using the MIPSpro^{TM} 7.3 C++ compiler, for example, this complicated loop is almost twice as slow as the more straightforward nested loop. (Using a `seg_array_iter` is slightly slower still, but this appears merely to be an artifact of the compiler's "abstraction penalty.")

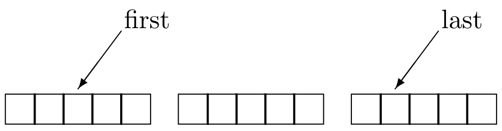
In this instance, the STL fails to achieve the goal of generic programming: the STL's generic algorithm `fill` is not as efficient as the original algorithm written for `seg_array`. The original algorithm uses a simple and easily optimized nested loop, while the generic algorithm expands into a complicated loop with multiple tests in every iteration.

The difficulty, fundamentally, is that, while the `seg_array_iter` class manages segmentation (it keeps track of a node and a position within the node), it does not expose the segmentation as part of its interface; segmentation is not part of the Forward Iterator requirements. The solution is a new iterator concept, one that explicitly supports segmentation.

4 Segmented Iterators[2]

The discussion of `fill` in Section 3 is not completely general, because it only addresses the special case of assigning a new value to every element of a segmented array. It does not consider the traversal of incomplete segments.

More generally `fill` operates on a range [`first`, `last`), where `first` is not necessarily at the beginning of a segment and `last` is not necessarily at the end of one. The general case is illustrated in the following diagram:



An algorithm that operates on a range [`first`, `last`) in a segmented data structure can be implemented using nested loops. The outer loop is from `first`'s segment up to and including `last`'s segment. There are three kinds of inner loops: a loop through the first segment (from `first` to the end of the segment), a loop through the last segment (from the beginning of the segment up to but not including `last`, and loops through every segment other than the first and the last (from the beginning of the segment to the end). One final special case is when `first` and `last` point into the same segment.

In pseudocode, a segmented version of `fill` might be written as follows:

```
hierarchical_fill(first, last, value)
{
  sfirst = Segment(first);
  slast  = Segment(last);

  if (Segment(first) == Segment(last))
    fill(Local(first), Local(last), value);
```

[2] Other names that have been used for the same concept include "NUMA iterator," "bucket iterator," and "cluster iterator."

```
    else {
      fill(Local(first), End(sfirst), value);
      ++sfirst;
      while (sfirst != slast) {
        fill(Begin(sfirst), End(sfirst), value);
        ++sfirst;
      }
      fill(Begin(slast), Local(last), value);
    }
  }
```

A segmented iterator can be viewed as an ordinary Forward Iterator; `*first` is an element in a data structure, and `first` can be incremented some finite number of times to obtain `last`. Additionally, though, a segmented iterator can be decomposed into two pieces: a *segment iterator* that points to a particular segment, and a *local iterator* that points to a location within that segment. Furthermore, it must be possible to inspect a segment iterator and obtain local iterators that point to the beginning and the end of the segment.[3]

A segmented iterator and its associated local iterator are both iterators, both have the same value type, and both can be used to traverse a segment within a segmented container. The difference is that the full segmented iterator can traverse a range that includes more than one segment, while a local iterator is valid only within a single segment. Incrementing a local iterator can thus be a very fast operation; it need not incur the overhead of checking for the end of a segment.

As an example, the `seg_array_iter` class from Section 3 is a segmented iterator. Its associated segment iterator type points to a `vector<T>` (incrementing a segment iterator moves from one vector to the next), and its associated local iterator type is an iterator that traverses a single vector; that is, it is `vector<T>::iterator`.

The distinction between segmented and nonsegmented iterators is orthogonal to the distinction between Forward Iterators, Bidirectional Iterators, and Random Access Iterators. The `seg_array_iter` class could easily be implemented as a Random Access Iterator; each segment has the same number of elements, so the operation `i += n` requires nothing more than integer arithmetic. A hash table where the buckets are implemented as linked lists could also provide segmented iterators, but, in contrast to `seg_array`, they could be at most Bidirectional Iterators. A segmented iterator type can be no stronger than the weaker of its associated segment iterator and local iterator types.

A segmented iterator has the same associated types as any other iterator (a value type, for example), and, additionally, it has an associated segment iterator type and local iterator type. Writing a generic algorithm that uses segmented

[3] STL iterators can be *past-the-end*. An iterator like `last` doesn't necessarily point to anything; it might, for example, point beyond the end of an array. Note that it must be possible to obtain a valid segment iterator even from a past-the-end segmented iterator.

iterators requires the ability to name those types. Additionally, a fully generic algorithm ought to be usable with both segmented and nonsegmented iterators. This is the same problem as with the generic algorithm `reverse`, in Section 2, and the solution is also the same: a traits class, which makes it possible to write `fill` as a dispatching algorithm.

The `segmented_iterator_traits` class identifies whether or not an iterator is segmented, and, for segmented iterators, it defines the segment and local iterator types. Finally, it is convenient for the traits class to contain the functions for decomposing a segmented iterator into its segment iterator and local iterator.[4]

The general definition of `segmented_iterator_traits` contains nothing but a flag identifying the iterator as nonsegmented:

```
template <class Iterator>
struct segmented_iterator_traits
{
  typedef false_type is_segmented_iterator;
};
```

The traits class is then specialized for every segmented iterator type. For `seg_array_iterator`, for example, the specialization is as follows:

```
template <class T>
struct segmented_iterator_traits<seg_array_iter<T> >
{
  typedef true_type                      is_segmented_iterator;
  typedef seg_array_iter<T>              iterator;
  typedef vector<vector<T>*>::iterator segment_iterator;
  typedef vector<T>::iterator            local_iterator;

  static segment_iterator segment(iterator);
  static local_iterator   local(iterator);

  static iterator compose(segment_iterator, local_iterator);

  static local_iterator begin(segment_iterator);
  static local_iterator end(segment_iterator);
};
```

Every such specialization for a segmented iterator type has the same interface: the types `segment_iterator` and `local_iterator`, and the functions `segment`, `local`, `compose`, `begin`, and `end`. This interface can be used by generic algorithms such as `fill`. Here is a full implementation:

[4] Putting these functions into the traits class is slightly more convenient than putting them into the iterator class, because it makes it easier to reuse existing components as segmented iterators.

```
template <class SegIter, class T>
void fill(SegIter first, SegIter last, const T& x, true_type)
{
  typedef segmented_iterator_traits<SegIter> traits;

  typename traits::segment_iterator sfirst
    = traits::segment(first);
  typename traits::segment_iterator slast
    = traits::segment(last);

  if (sfirst == slast)
    fill(traits::local(first), traits::local(last), x);
  else {
    fill(traits::local(first), traits::end(sfirst), x);
    for (++sfirst ; sfirst != slast ; ++sfirst)
      fill(traits::begin(sfirst), traits::end(sfirst), x);
    fill(traits::begin(sfirst), traits::local(last), x);
  }
}

template <class ForwardIter, class T>
void fill(ForwardIter first, ForwardIter last, const T& x,
          false_type)
{
  for ( ; first != last; ++first)
    *first = x;
}

template <class Iter, class T>
inline void fill(Iter first, Iter last, const T& x)
{
  typedef segmented_iterator_traits<Iter> Traits;
  fill(first, last, x,
       typename Traits::is_segmented_iterator());
}
```

The main function, `fill`, uses the traits mechanism to select either a segmented or a nonsegmented version. The segmented version, written using nested loops, uses the traits mechanism to decompose a segmented iterator into its segment and local iterators. The inner loop is written as in invocation of `fill` on a single segment.

Using the same system as the previous test, the segmented version of `fill` is no slower than the handwritten version using nested loops.

Multiple levels of segmentation can be used with no extra effort. Nothing in this discussion, or this implementation of `fill`, assumes that the local iterator is nonsegmented.

5 Conclusions

The most important innovation of the STL is the *iterator* abstraction, which represents a position in a one-dimensional range. Iterators are provided by containers and used by generic algorithms, and they make it possible to decouple algorithms from the data structures that they operate on.

Ordinary STL iterators describe a uniform range with no additional substructure. Even if a data structure has additional features, such as segmentation, it is impossible to access those features through the iterator interface.

Segmented iterators, an extension of ordinary STL iterators, make it possible for generic algorithms to treat a segmented data structure as something other than a uniform range of elements. This allows existing algorithms to operate more efficiently on segmented data structures, and provides a natural decomposition for parallel computation. Segmented iterators enable new kinds of generic algorithms that explicitly depend on segmentation.

Acknowledgments

This work is a collaborative effort by many members of the Silicon Graphics compiler group. I also wish to thank Ullrich Köthe and Dietmar Kühl for helpful suggestions.

References

1. A. A. Stepanov and M. Lee, “The Standard Template Library.” Hewlett-Packard technical report HPL-95-11(R.1), 1995.
2. D. R. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.
3. M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Addison-Wesley, 1998.
4. B. Stroustrup, *The C++ Programming Language*, Third Edition. Addison-Wesley, 1997.
5. International Organization for Standardization (ISO), 1 rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland, *ISO/IEC Final Draft International Standard 14882: Programming Language C++*, 1998.
6. J. Barreiro, R. Fraley, and D. R. Musser, “Hash tables for the Standard Template Library.” Technical Report X3J16/94-0218 and WG21/N0605, International Organization for Standardization, February 1995.

Theory and Generality of Complete Traversals

David R. Musser[1] and Arturo J. Sánchez-Ruíz[2] ⋆

[1] Computer Science Department, Rensselaer Polytechnic Institute,
Troy NY 12180, USA
musser@cs.rpi.edu
[2] Centro ISYS, Escuela de Computación, Facultad de Ciencias,
Universidad Central de Venezuela, AP 47642, Caracas 1041-A, Venezuela
arturo@acm.org

Abstract. Standard iteration mechanisms, even ones of such generality as those in the C++ Standard Template Library (STL), do not allow insertions to be made in a container during a traversal of the container. Since ad hoc ways of handling such iterations are tedious to program and error-prone, programmers should ideally have at their command an efficient generic solution. In earlier work we formally characterized the *complete traversal* problem and presented generic solutions by means of two generic algorithms and a container adaptor, all defined within the STL framework. Here we develop additional theory for reasoning about complete traversals and address the question of how general the complete traversal problem is by showing that it subsumes well-known graph (or relation) problems such as reachability and transitive closure.

Keywords: Generic Programming, Standard Template Library (STL), Iterators, Containers, Adaptors, Iteration Mechanisms, Closures.

1 Introduction

A *complete traversal* is an iteration scheme

$$\textbf{for all } x \textbf{ in } C$$
$$\mathcal{F}(x, C)$$

where C is a container (such as a set) and $\mathcal{F}$ is a function that might possibly modify C by inserting new elements into it, the iteration continuing until $\mathcal{F}$ has been applied once and only once to each of the elements currently in C, including those $\mathcal{F}$ has inserted. Standard iteration mechanisms, such as the iterators provided in the C++ Standard Template Library (STL), do not directly support complete traversals. In [7] we presented generic solutions to the complete traversal problem by means of two generic algorithms and a container adaptor, all

⋆ Partially supported by Postgrado en Ciencias de la Computación, CDCH-UCV (Ayuda Tipo A), and CONICIT. Current address: University of Massachusetts-Dartmouth, CIS Department, 285 Old Westport Road, North Dartmouth, MA 02745. Both authors would like to thank Eric Gamess for his contributions the initial phase of this work [7].

defined within the STL framework. In this paper we develop additional theory for reasoning about complete traversals and address the question of how general the complete traversal problem (CT) is. We show directly that CT subsumes well-known graph (or relation) problems such as reachability and transitive closure; these problems of course have many instances that could also be solved with complete traversals.

Let C be a container, x an element of C, and $\mathcal{F}$ be a (program)function. Denote by $\hat{\mathcal{F}}(x, C)$ the container of elements to be inserted into C by the call $\mathcal{F}(x, C)$. We formally define complete traversals in terms of a rewriting relation, as follows:

Definition 1.

1. *Given any containers C and D such that $D \subseteq C$ and a (program) function $\mathcal{F}$, let $x \in C - D$, $C' = C \cup \hat{\mathcal{F}}(x, C)$, and $D' = D \cup \{x\}$. We say that (C', D') is a* traversal successor *of (C, D) and denote this relation by $(C, D) \rightarrow (C', D')$.*
2. *(C, D) is said to be a* normal form*, or* irreducible*, if $C = D$.*
3. *A* traversal sequence *for a container C using a function $\mathcal{F}$ is any sequence $(C_0, D_0) \rightarrow (C_1, D_1) \rightarrow \ldots \rightarrow (C_n, D_n)$ starting from $C_0 = C$ and $D_0 = \phi$.*
4. *Such a traversal sequence is said to be* terminating *if (C_n, D_n) is irreducible (equivalently, if $C_n = D_n$).*
5. *A* complete traversal *of a container C using $\mathcal{F}$ is any terminating traversal sequence for C using $\mathcal{F}$.*

The container operations are appropriately interpreted depending on whether C is a unique or multiple container.

This terminology *unique* vs. *multiple* container is from the STL classification of associative containers: in a *unique* container, such as a set or map, objects in the container cannot have equivalent keys, whereas in a *multiple* container, such as a multiset or multimap, they can. In this paper we sometimes use ordinary set notation; e.g., writing "$x \in C$" where C is a container and x is a value it contains.

Can two complete traversals of the same container C and function $\mathcal{F}$ result in different final containers? Yes, as shown by the following example:

$$\begin{aligned} C &= \{1, 4\} \\ \mathcal{F}(x, X) &: \textit{insert } (x + |X|) \textbf{ div } 2 \textit{ into } X \end{aligned} \qquad (1)$$

where C is assumed to be a unique container and $|X|$ denotes the size of X. In Fig. 1 we show all possible results depending on the order used to apply $\mathcal{F}$. We see that, for this C and $\mathcal{F}$, different orders lead to different results.

Since we have cast the traversal successor relation as a rewriting relation, we can draw upon standard rewriting theory (e.g., [9,14]) for deriving conditions for uniqueness of complete traversal results. Given an arbitrary rewriting relation $\rightarrow$, the reflexive, transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$ and is referred to as *reduction.*

Order	Result
$\langle 1,4,3\rangle$	$\{1,3,4\}$
$\langle 4,1,2,3\rangle$	$\{1,2,3,4\}$
$\langle 4,1,3,2\rangle$	$\{1,2,3,4\}$
$\langle 4,3,1,2\rangle$	$\{1,2,3,4\}$

Fig. 1. All possible results associated with instance in (1)

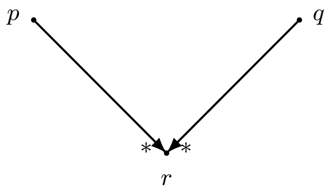


Fig. 2. Joinability Relation

Elements p and q are *joinable* if and only if there exists an r such that $p \rightarrow^* r$ and $q \rightarrow^* r$ (see Fig. 2).

A rewriting relation $\rightarrow$ is said to be *uniformly terminating*,[1] if and only if there is no infinite sequence of the form $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \ldots$. Thus with a uniformly terminating relation every reduction path is finite and every element has a *normal form*; i.e., the last element of a path. An element might have several distinct normal forms, however.

A rewriting relation $\rightarrow$ is *confluent* if and only if for all elements p, q_1, q_2, if $p \rightarrow^* q_1$ and $p \rightarrow^* q_2$ then q_1 and q_2 are joinable.

It follows directly from the definitions that if a rewriting relation is confluent, an element cannot have two distinct normal forms; that is, confluence is a sufficient condition for every element to have a unique normal form.

It is possible in some cases to use a more localized test. A rewriting relation $\rightarrow$ is *locally confluent* if and only if for all elements p, q_1, q_2, if $p \rightarrow q_1$ and $p \rightarrow q_2$ then q_1 and q_2 are joinable. (Note the difference from confluence: only one step is taken from p rather than arbitrarily many.)

If a rewriting relation is uniformly terminating and locally confluent, then it is confluent. This is called the "Diamond Lemma," because of the structure of its proof; see [9].

Applying this theory, we have:

[1] Other terms for this property are *finitely terminating* and *Noetherian.*

Theorem 1. *If a traversal successor relation is uniformly terminating and locally confluent, then every complete traversal of a container C using a function $\mathcal{F}$ results in a unique final container. We say that the complete traversal computation is* determinate.

Thus, in the determinate case, the order in which elements are chosen is not relevant, and we can speak of *the* complete traversal $CT(C, \mathcal{F})$ of C by $\mathcal{F}$. Uniform termination alone does not guarantee determinacy, as the instance in (1) shows.

Theorem 2. *If $\hat{\mathcal{F}}(x, C)$ does not depend on C, then the traversal successor relation is locally confluent.*

Proof. At any reduction step i, if $x_i, x'_i \in C_i - D_i$ and $x_i \neq x'_i$, let

$$(C_{i+1}, D_{i+1}) = (C_i \cup \hat{\mathcal{F}}(x_i, C_i), D_i \cup \{x_i\})$$
$$(C'_{i+1}, D'_{i+1}) = (C_i \cup \hat{\mathcal{F}}(x'_i, C_i), D_i \cup \{x'_i\})$$

Then (C_{i+1}, D_{i+1}) and (C'_{i+1}, D'_{i+1}) are immediately joinable to a common result: continue one step from (C_{i+1}, D_{i+1}) using x'_i to obtain

$$(C_{i+1} \cup \hat{\mathcal{F}}(x'_i, C_{i+1}), D_i \cup \{x_i\} \cup \{x'_i\})$$

and one step from (C'_{i+1}, D'_{i+1}) using x_i to obtain

$$(C'_{i+1} \cup \hat{\mathcal{F}}(x_i, C'_{i+1}), D_i \cup \{x'_i\} \cup \{x_i\})$$

By the assumption that $\hat{\mathcal{F}}$ does not depend on the container, these two results are identical. □

Corollary 1. *If $\hat{\mathcal{F}}(x, C)$ does not depend on C and the traversal successor relation is uniformly terminating, it is also determinate.*

Proof. Follows directly from Theorems 1 and 2. □

We now consider some sufficient conditions for uniform termination.

Theorem 3. *Let C and $\mathcal{F}$ be such that for any reduction sequence $(C, \phi) \to (C_1, D_1) \to \ldots$ there is some i such that $(C, \phi) \to^* (C_i, D_i)$ and for all $j \geq i$ in the sequence,*

1. *$\forall x_k \in C_j : \hat{\mathcal{F}}(x_k, C_j) \subseteq C_j$, if C is unique; or,*
2. *$\forall x_k \in C_j : \hat{\mathcal{F}}(x_k, C_j) = \phi$, if C is multiple.*

Then the traversal successor relation is uniformly terminating.

Proof. In both cases, we have $C_j = C_i$ for all $j \geq i$. Since $D_j \subseteq C_j$ we have $|D_j| \leq |C_i|$. Since $|D_j|$ is strictly increasing, eventually we must have $D_j = C_j$. □

Corollary 2. *If C is a unique container and X is a finite set such that $C \subseteq X$ and range($\hat{\mathcal{F}}$) $\subseteq X$, then the traversal successor relation is uniformly terminating.*

Proof. The length of any reduction sequence is bounded by $|X|$. □

The following theorem is the basis of an alternative approach to proving determinacy and characterizing the result of the traversal successor relation; we will use it in the next section.

Theorem 4. *Let C be a unique container, and C^* be a set, with $C^* \subseteq X$, where X is a finite set. Let $\mathcal{F}$ be a function such that range($\hat{\mathcal{F}}$) $\subseteq X$. Suppose that for every (C', D') such that $(C, \phi) \rightarrow^* (C', D')$,*

$$C' \subseteq C^* \subseteq C^U(C', D'),$$

where

$$C^U(A, B) = \bigcup \{C'' : (A, B) \rightarrow^* (C'', D'')\}.$$

Then $\rightarrow$ is uniformly terminating and confluent, and $CT(C, \mathcal{F}) = C^*$.

Proof. Uniform termination follows from Corollary 2. In order to prove confluence, let us consider a normal form (C', C') such that $(C, \phi) \rightarrow^* (C', C')$. By the hypotheses, we know that $C' \subseteq C^*$. On the other hand, since (C', C') is a normal form, we have $C^U(C', C') = C'$. Thus, $C^* \subseteq C'$, and we conclude that $C^* = C'$. This proves that all normal forms of (C, ϕ) are equal, and $CT(C, \mathcal{F}) = C^*$. □

2 Generality of Complete Traversals

As a measure of the generality of complete traversals, let us compare CT with some well-known problems. Many such problems are themselves instances of problems about binary relations (equivalently, directed graphs) such as the reachability problem and the transitive closure problem. We shall show that these two problems can in fact be viewed as instances of CT, in the sense that each can be performed by doing a complete traversal using an appropriately chosen container and function.

2.1 CT and the Graph Reachability Problem

We state this problem using graph terminology, although it could of course be stated in terms of a relation, as is the transitive closure problem in the next subsection.

The template shown in Fig. 3 describes reachability as an instance of CT. To prove the theorem stated in Fig. 3, we apply Theorem 4, taking $C^* = R$, $X = V$, and $\hat{\mathcal{F}}(x, C) = \{w \in X \mid (x, w) \in E\}$. The following two lemmas show that the remaining hypotheses of Theorem 4 are satisfied.

PROBLEM:	Given a finite directed graph and a subset S of vertices, find those vertices reachable from S.
REFERENCE:	[1], pp. 119.
INPUT:	A directed graph $G = (V, E)$, and a set $S \subseteq V$.
OUTPUT:	$R = \{v \in V : v$ is reachable from $u \in S\}$.
INSTANCE:	$C = S$ $\mathcal{F}(x, C)$: **for all** $(x, w) \in E$ insert w into C.
THEOREM:	$CT(\mathcal{F}, C) = R$.

Fig. 3. Reachability as an instance of CT

Lemma 1. *If $(C, \phi) \to^* (C_k, D_k)$ is any traversal sequence for C using $\mathcal{F}$, then $C_k \subseteq R$.*

Proof. By induction on k. If $k = 0$, then $C_k = S \subseteq R$. If $k > 0$, by the definition of $\mathcal{F}$ there is some $x \in C_{k-1} - D_{k-1}$ such that

$$\begin{aligned} C_k &= C_{k-1} \cup \hat{\mathcal{F}}(x, C_{k-1}) \\ &= C_{k-1} \cup \{w : (x, w) \in C_{k-1}\}. \end{aligned}$$

Now by the induction hypothesis, $C_{k-1} \subseteq R$ and every element w in the second set is in R. Thus $C_k \subseteq R$, as was to be shown. □

Lemma 2. *If $(C, \phi) \to^* (C', D')$, then every $y \in R$ is also in $C^U(C', D')$, where C^U is defined as in Theorem 4.*

Proof. By induction on $n = |\langle x, y\rangle|$, the length of a shortest path from x to y in G, where $x \in S$. For $n = 1$, $(x, y) \in E$ and thus $y \in S \cup \hat{\mathcal{F}}(x, S) \subseteq C^U(C', D')$. For $n > 1$, let $x = x_0, x_1, \ldots, x_{n-1}, x_n = y$ be such that $(x_i, x_{i+1}) \in E$ for $i = 0, \ldots, n-1$. Since $|\langle x, x_{n-1}\rangle| < n$, we can apply the induction hypothesis to conclude that $x_{n-1} \in C''$, for some $C'' \in C^U(C', D')$. Now, since $(x_{n-1}, y) \in E$, we have that $y \in C'' \cup \hat{\mathcal{F}}(x_{n-1}, C'') \subseteq C^U(C', D')$. This establishes the induction step and finishes the proof. □

Some problems which are instances of the reachability problem are the computation of ϵ-*closure*, *closure*(I) (where I is a set of LR(0) items), LALR(1) lookaheads by propagation, and loop-invariant code [1]. The CTS problem, and the manager's invitation list problem, presented in [7] to motivate complete traversals, are also instances of the reachability problem.

2.2 CT and the Transitive Closure Problem

Let A be a binary relation, i.e., $A \subseteq D \times D$, for some set D. We assume D is finite. The (irreflexive) *transitive closure* of A, denoted by A^+, is the binary relation on D such that $(x, y) \in A^+$ if and only if there exist $n > 1$ and $x_1, \ldots\ x_n \in D$ such that $x = x_1$, $x_n = y$, and $(x_i, x_{i+1}) \in A$ for $i = 1, \ldots, n-1$.

The template for the transitive closure problem can be seen in Fig. 4. To

PROBLEM:	Given a relation $A \subseteq D \times D$, find its transitive closure.
REFERENCE:	[2], pp. 8–9.
INPUT:	A relation $A \subseteq D \times D$.
OUTPUT:	A^+ as defined above.
INSTANCE:	$C = A$
	$\mathcal{F}(x, C)$: let $x = (a, b)$
	for all $y = (b, c) \in C$
	insert (a, c) into C.
THEOREM:	$CT(\mathcal{F}, C) = A^+$.

Fig. 4. Transitive Closure as an instance of CT

prove the theorem stated in Fig. 4, we again apply Theorem 4, this time taking $C^* = A^+$, $X = D \times D$, and $\hat{\mathcal{F}}(x, C) = \{z = (a, c) \in X \mid x = (a, b), y = (b, c) \in C\}$. As in the reachability case, we have to show that the hypotheses of Theorem 4 are satisfied.

Lemma 3. *If $(C, \phi) \rightarrow^* (C_k, D_k)$ is any traversal sequence for C using $\mathcal{F}$, then $C_k \subseteq A^+$.*

Proof. By induction on k. If $k = 0$, then $C_k = C = A \subseteq A^+$. If $k > 0$, by the definition of $\mathcal{F}$ there is some $(x, y) \in C_{k-1} - D_{k-1}$ such that

$$\begin{aligned} C_k &= C_{k-1} \cup \hat{\mathcal{F}}((x, y), C_{k-1}) \\ &= C_{k-1} \cup \{(x, z) : (y, z) \in C_{k-1}\}. \end{aligned}$$

Now, by the induction hypothesis, $C_{k-1} \subseteq A^+$ and, therefore, every pair (x, z) in the second set is in A^+. Thus $C_k \subseteq A^+$, as was to be shown. □

Lemma 4. *If $(C, \phi) \rightarrow^* (C', D')$, then every $(x, y) \in A^+$ is also in $C^U(C', D')$, where C^U is defined as in Theorem 4.*

Proof. By induction on $n = |\langle x, y \rangle|$, the length of a shortest path from x to y in the graph $G = (D, A)$. If $n = 1$, $(x, y) \in A \subseteq C^U(C', D')$. If $n > 1$, let $x = x_0, x_1, \ldots, x_{n-1}, x_n = y$, where $(x_i, x_{i+1}) \in A$, for $i = 0, \ldots, n-1$. Since $|\langle x, x_{n-1} \rangle| < n$, by the induction hypothesis we may assume $(x, x_{n-1}) \in C^U(C', D')$, and therefore that $(x, x_{n-1}) \in C''$ for some (C'', D'') such that $(C', D') \rightarrow^* (C'', D'')$. Since $(x_{n-1}, x_n) \in A$, we have $(x, x_n) = (x, y) \in C'' \cup \hat{\mathcal{F}}((x, x_{n-1}), C'') \subseteq C^U(C', D')$. This establishes the induction step and completes the proof. □

3 Related Work

A complete traversal, seen as a generic iteration mechanism that allows insertions into the container while the iteration is in progress, seems to be perceived in the literature as something negative that must be avoided. A good example of this perception is illustrated by the following *programming principle* [4]:

> "**Principle 14**: *Never modify a data structure while an associated* `Enumeration` *is alive*"

In this context, an `Enumeration` is the simplest mechanism for container traversal that is directly supported by Java. Other frameworks and languages that support the notions of containers and iterators, but do not support complete traversals are Karla (Karlsruhe Library of Data Structures and Algorithms) [5,6], Sather-K [8], Sather [12], JGL (Java Generic Library) [15], and the collection framework in the `java.util` package of Java 2 [17]. Our definition of complete traversals captures its non-deterministic nature and establishes conditions under which such traversals are determinate, and terminate. To the best of our knowledge, no such formal framework for the treatment of complete traversal problems has been previously developed.

When designing and implementing a generic component, the issue of assessing its level of generality arises quite naturally. STL designers approached this question by analyzing the class of types that model the set of concepts associated with the component. One says that a type models a concept if the former satisfies all the properties specified by the latter [3]. The wider the class of types, the more general the component is.

To illustrate this approach, consider the generic algorithm `sort`, which takes a container (defined by a pair of iterators), and a function object that implements an order on the container. The algorithm sorts the container *in situ* according to the given order. How general this component is? The STL specification by Stepanov and Lee [16], calls for the iterators to satisfy the properties of random access iterators, and for the function object to satisfy "the standard axioms for total ordering", and to induce an "equality relation". If `comp` is the function object in question, then `x` and `y` are considered equivalent if and only if `!comp(x,y) && !comp(y,x)` holds. As it was later pointed out by Musser and Saini [13], this class of orderings are known as strict weak orders. The point here is that `sort` is able to handle orderings that are more general than strict total orders, for which `!comp(x,y) && !comp(y,x)` implies equality, a particular case of equivalence.

In this paper, we have concentrated on the problems the components are aimed at, and used the notion of reducibility, well-known in the context of computability and complexity theory [11], to establish a relationship between the complete traversal problem and other problems. The intuition behind our rationale is the same, namely, if problem P is an instance of problem Q (equivalently, if P can be reduced to Q) then we say that Q is at least as general as P. In our case, Q is CT. Even though this is a standard technique, we are not aware of it being used to assess the generality of a generic component.

4 Future Work

We are currently working on proving that other well-known problems can be expressed as complete traversals. Chief among them is the problem of computing the closure of a set under a set of relations [11]. The connection between this problem, CT, and other problems that we have mentioned in this paper are depicted in Fig. 5.

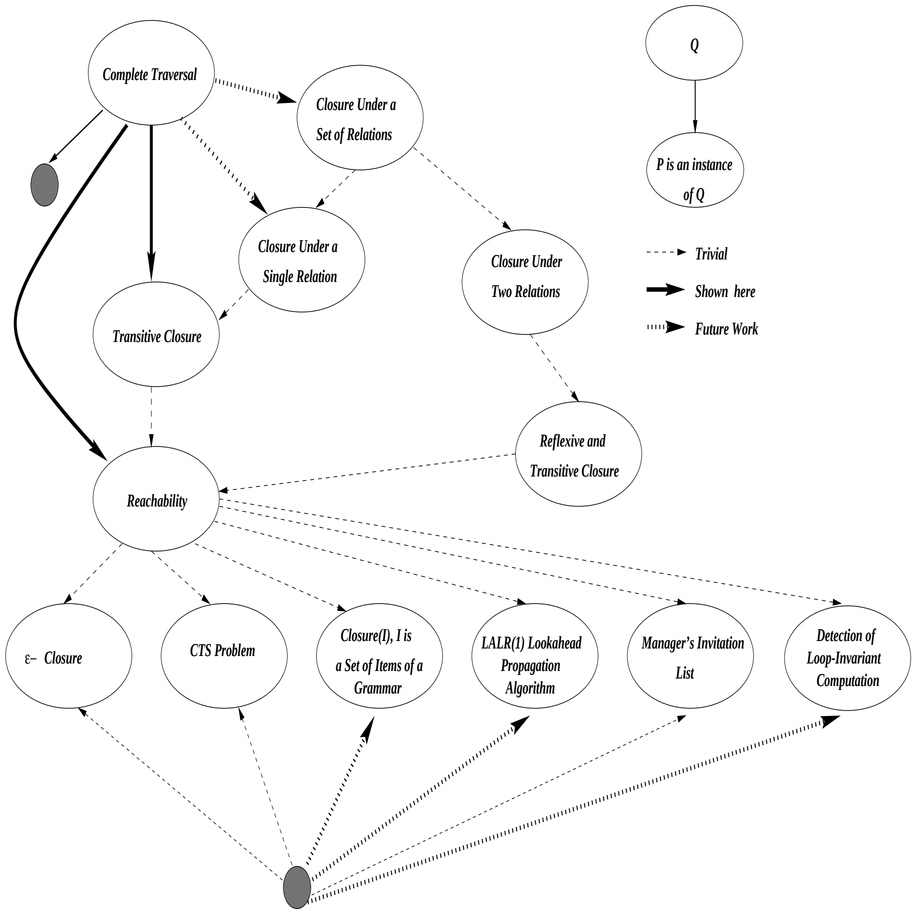

Complete Traversal
Closure Under a Set of Relations
Q
P is an instance of Q
Closure Under a Single Relation
Closure Under Two Relations
Trivial
Shown here
Future Work
Transitive Closure
Reflexive and Transitive Closure
Reachability
ε- Closure
CTS Problem
Closure(I), I is a Set of Items of a Grammar
LALR(1) Lookahead Propagation Algorithm
Manager's Invitation List
Detection of Loop-Invariant Computation

Fig. 5. CT and its relation to other well-known problems

Our formal characterization of complete traversals does not take into account deletions while the iteration is in progress. We would like to consider this possibility. We have conjectured, however, that there are no functions $\mathcal{F}$ such that the associated CT is determinate when deletions are allowed. We are also working on the characterization of fairness for the generic algorithms presented in [7] for infinite traversals, and on the characterization of CT when several iterators are being used to traverse the container.

From a more practical point of view, we are currently exploring one approach to assessing the usability of generic components across their application domain. Suppose that P is a well-known problem which is also known to be expressible as a complete traversal. This means that we have, at least, two solutions to it, namely S_1 obtained by instantiating one of the generic components presented in [7], and S_2 obtained by implementing well-known algorithms to tackle P. We are working on comparing these two solutions with randomly generated problems. A first candidate for P is the transitive closure problem, which have received a lot of attention in the literature [10].

Finally, we are also interested in finding problems P which are instances of CT, but not vice versa, and in finding those which cannot be expressed as instances of CT.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing.* Prentice-Hall, 1972.
3. M. H. Austern. *Generic Programming and the STL - Using and Extending the C++ Standard Template Library.* Addison-Wesley, 1999.
4. D. Bailey. *Java Structures—Data Structures in Java for the Principled Programmer.* WCB/McGraw-Hill, 1999.
5. A. Frick, W. Zimmer, and W. Zimmermann. *Karla: An Extensible Library of Data Structures and Algorithms. Part I: Design Rationale.* Karlsruhe University, Faculty of Computer Science, August 1994.
6. A. Frick and W. Zimmermann. *Karla: An Extensible Library of Data Structures and Algorithms. Part II: Usage for Beginners.* Karlsruhe University, Faculty of Computer Science, January 1995.
7. E. Gamess, D. R. Musser, and A. J. Sánchez-Ruíz. Complete traversals and their implementation using the standard template library. In Raúl Monge Anwandter, editor, *Proceedings of the XXIII Latinamerican Conference on Informatics*, volume I, pages 221–230, Valaparaíso, Chile, November 1997. CLEI.
8. G. Goos. Sather-K, the language. Technical report, Karlsruhe University, April 1996.
9. G. Huet and D. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal Languages: Perspectives and Open Problems.* Academic Press, New York, 1980.
10. Y. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM Transactions on Database Systems*, 18(3):512–576, Sept. 1993.
11. H. R. Lewis and Ch. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, second edition, 1998.

12. S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration abstraction in sather. *ACM TOPLAS*, 18(1):1–15, Jan. 1996. Available from `http://www.icsi.berkeley.edu/~sather/`.
13. D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.
14. David R. Musser. Automated theorem proving for analysis and synthesis of computations. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, New York, 1989.
15. ObjectSpace. Objectspace—JGL, the Generic Collection Library for Java. `http://www.objectspace.com/jgl`, 1997.
16. A. A. Stepanov and M. Lee. The standard template library. Technical Report HP-94-93, Hewlett-Packard, 1995.
17. Sun Microsystems. Java 2 Platform API Specification. `http://java.sun.com/products/jdk/1.2/docs/api/index.html`, 1999.

A Data Abstraction Alternative to Data Structure/Algorithm Modularization

Murali Sitaraman[1], Bruce W. Weide[2],
Timothy J. Long[2], and William F. Ogden[2]

[1] Computer Science and Electrical Engineering, West Virginia University
Morgantown, WV 26506-6109
murali@csee.wvu.edu

[2] Computer and Information Science, The Ohio State University
Columbus, OH 43210
{weide, long, ogden}@cis.ohio-state.edu

Abstract. Modularization along the boundaries of data structures and algorithms is a commonly-used software decomposition technique in computer science research and practice. When applied, however, it results in incomplete segregation of data structure handling and algorithm code into separate modules. The resulting tight coupling between modules makes it difficult to develop these modules independently, difficult to understand them independently, and difficult to change them independently. Object-oriented computing has maintained the traditional dichotomy between data structures and algorithms by encapsulating only data structures as objects, leaving algorithms to be encapsulated as single procedures whose parameters are such objects. For the full software engineering benefits of the information hiding principle to be realized, data abstractions that encapsulate data structures and algorithms together are essential.

1 Introduction

The dichotomy of data structures and algorithms is pervasive in computing, and this separation has been used routinely as a criterion for modularization in both structured and object-oriented software development. But the suitability of this criterion for decomposition has rarely been questioned from a software engineering perspective in the data structures and algorithms literature. Cormen, Leisersen, and Rivest, authors of probably the most widely used textbook on algorithms [2], admit to as much, as they set the following stage for their discourse on algorithms:

> We shall typically describe algorithms as programs written in pseudocode that is very much like C, Pascal, or Algol, ... not typically concerned with issues of software engineering. Issues of data abstraction, modularity, and error handling are often ignored to convey the essence of the algorithm more concisely.

Though it is arguably useful to understand the design and details of algorithms without concern for software engineering, Parnas has argued that the criteria to be used for module decomposition are at least as important in development of real software systems [8]. In the conventional data structure/algorithm decomposition, the code that stores "input" values into data structures, and the code that retrieves "output" values from data structures, resides in the calling module. The algorithm that transforms the inputs to outputs using the data structures, and which is typically a procedure whose parameters are these data structures, resides in a separate module. The major processing steps—input/output data structure handling and execution of the algorithm—have been turned into separate modules that communicate through data structures. Parnas has argued against this form of processing-based decomposition, noting that it makes it difficult to develop the modules independently, difficult to understand modules independently, and difficult to change modules independently. He has proposed "information hiding" as an alternative criterion for module decomposition [8]:

> ...it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions that are likely to change. Each module is then designed to hide such a decision from others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in processing.

While the information hiding criterion has had some impact on hiding decisions about the data structures involved in representing "container" abstract data types such as stacks and queues, it has had little effect on modularization that respects the classical dichotomy of data structures and algorithms. The objective of this paper is to expose the weaknesses of this widely used modularization along data structure/algorithm boundaries and to discuss an alternative based on the information hiding criterion. In this approach, data structures and algorithms are encapsulated together to produce new data abstractions. The resulting modularization arguably has desirable properties of module decomposition detailed in [8], including ease of independent module development, ease of use, and ease of change. In addition, the data abstraction interface approach provides performance advantages. For a given data abstraction, it becomes possible to develop alternative plug-compatible implementations that differ both in the data structures/algorithms used in computing the results, and in whether results are computed incrementally or in batch, allowing temporal flexibility. Through their interface specifications and support for alternative implementations, reusable data abstractions of this nature supplement (functional and performance) flexibility benefits of generic programming [7], with software engineering benefits.

The rest of the paper is organized as follows. Section 2 examines a classical example from the algorithms literature from a software engineering perspective. Section 3 presents a data abstraction alternative. Section 4 discusses performance benefits of the data abstraction-based modularization. The last section contains a discussion of related work and our conclusions.

2 A Conventional Modularization That Respects the Data Structure/Algorithm Dichotomy

To illustrate the ramifications of modularization based on separation of data structures and algorithms, we begin with an instance of a shortest path problem:

Given a graph, a source vertex, and a set of destination vertices, find the shortest paths between the source and the destinations, if such paths exist, and their costs.

The best-known solution to the single source shortest paths problem is Dijkstra's greedy algorithm for finding shortest paths to all (connected) vertices from a single source [2]. Details of the data structures used in this solution are given below.

```
procedure SSSP_Dijkstra
    inputs G, w, s
    outputs S, distances, predecessors
```

Input data structures: `G` is a graph represented by an adjacency list data structure. `w` is a weight function that returns the cost of the edge joining a given pair of vertices. `s` is the source vertex.
Output data structures: `S` is a set of vertices that can be reached from `s`. `distances` is a structure that contains the cost of the shortest path from `s` to every vertex in `S`. `predecessors` is a structure that holds the penultimate vertex on a shortest path from the source to each vertex.

In this modularization, responsibility for the algorithm is given to one module (procedure `SSSP_Dijkstra`) and responsibility for storing information into and retrieving information from input/output data structures is given to the calling module. This responsibility assignment thwarts independent module development, because both modules rely on details of the data structures. The significant coupling between the calling module and the procedure that encodes the algorithm is obvious in this example. To understand and reason about the procedure, the data structures in the calling module need to be understood; and to understand the reason for choosing a particular representation for a data structure in the calling module (e.g., adjacency list representation for graphs), it is essential to understand some details of the algorithm. Any changes to details of the data structures in the calling module may affect the called procedure, and any changes to the algorithm may affect the responsibilities of the caller. Every use of the procedure requires the calling module to set up non-trivial data structures. And even when a caller is not interested in all results (paths, costs, paths to all destinations), all the data structures must be set up, if the intent is not to modify the code of the reusable procedure.

Another fundamental problem in separating data structures from algorithms arises when different algorithms for solving a problem demand different data structures. In such cases, when a client desires to switch from one algorithm to another, significant changes may be needed to the calling code. It may be essential to use a different algorithm to improve performance or because of apparently minor changes to the problem requirements. Suppose that the shortest path problem were modified slightly as below:

Given a graph, a set of source vertices, and a set of destination vertices, find the shortest paths between the sources and the destinations, if paths exists, and their cost.

One solution to this problem is to call the single source shortest paths procedure repeatedly with different sources. If the set of sources is small, this indeed may be the right choice. But if the paths need to be computed from a larger set of sources, then the Floyd-Warshall algorithm that computes efficiently shortest paths from all source vertices to all destination vertices may be more appropriate. In the Floyd-Warshall algorithm, the following data structures are used:

```
procedure ASSP_Floyd_Warshall
    input G
    outputs distances, predecessors
```

Input data structures: `G` is a graph represented by an adjacency matrix.
Output data structures: `distances` is a $|V| \times |V|$ matrix that contains distances of the shortest path from every source to every destination vertex. `predecessors` is a $|V| \times |V|$ matrix holding the penultimate vertex on a shortest path from each source to each destination vertex.

The differences in the input/output data structures and their representations between Dijkstra's algorithm and Floyd-Warshall's algorithm are significant, and they demand considerable change in the client module to switch from one algorithm to the other.

3 Modularization Based on Information Hiding

A key design decision in classical examples, such as the ones discussed in the last section, is making a suitable choice for data structures/representations for a particular algorithm. Most algorithms work efficiently only when used with particular data structures and their workings are often intricately intertwined with those data structures. Rarely are the choices so independent that arbitrary combinations are meaningful. In addition, as illustrated above, when algorithms need to be changed for performance or for software evolution reasons, the data structures are likely to change as well.

The information hiding criterion suggests that the design decision of matching proper structures with algorithms is a difficult choice that is quite likely to change (e.g., if the client decides to use a different algorithm), and it therefore must be hidden inside a module. To use a module that computes shortest paths, it should not be essential for the calling module to understand how graphs are represented or what structures are used to store inputs and results. For the shortest path problems, the interface of the data abstraction must permit a client to supply information about a graph and get answers to questions about shortest paths. Use of a data abstraction essentially decouples an implementation of the abstraction from its use, in turn facilitating independent development, ease of use, and ease of change.

The basic idea is easily explained. The abstract state space for an ADT (abstract data type) for solving the class of shortest path finding problems includes the edges of the graph under consideration. Operations on the ADT allow a graph to be defined and questions about shortest paths to be asked. One operation allows a user to input graph edges, one at a time. Other operations allow a user to ask whether there is a path between two vertices and if one exists, to request the cost of the shortest such path. The ADT also includes an operation that gives the edges on a shortest path from a source to a destination, one at a time. Other than an object of the ADT under discussion, in this design the only other parameters to operations are values of simple types such as edge information, booleans, and real numbers. No complex data structures are passed as parameters, because suitable data structures together with algorithms that manipulate them are hidden in the implementation(s) of the ADT. Clients neither need to understand nor need to set up any fancy data structures. Instead they see only an ADT and operations to manipulate objects of that type.

One interesting new issue arises in designing an interface for a data abstraction such as the one described here. Unlike data abstractions that encapsulate classical "container" data structures, where, in general, there are few or no restrictions on the order in which the operations may be called, the data abstraction described here demands that all graph information be available before the queries begin. It is easy to specify such restrictions by embellishing the abstract state model in a formal specification of the data abstraction, as explained in the next subsection.

Formal Specification of a Data Abstraction

A formal specification for a data abstraction that captures the shortest path problems in a dialect of the RESOLVE notation [10], is given in Figure 1. We have used the name "least cost path" instead of "shortest path", so weights on edges are interpreted more generally as costs instead of distances.

```
concept Least_Cost_Path_Finding_Template (
        type Edge_Index,
        constant Max_Vertex: Integer
    )

    math subtype Edge is (
            v1: integer,
            v2: integer,
            id: Edge_Index,
            cost: real
        )
        exemplar e
        constraint
            1 <= e.v1 <= Max_Vertex and
            1 <= e.v2 <= Max_Vertex and
            e.cost > 0.0

    definition CONNECTING_PATH_EXISTS (
            graph_edges: finite set of Edge,
            v1, v2: integer
        ): boolean
        = (* true iff there is a connecting path from
               v1 to v2 in graph_edges *)

    definition IS_A_LEAST_COST_PATH (
            graph_edges: finite set of Edge,
            v1, v2: integer,
            s: finite set of Edge
        ): boolean
        = (* true iff s is the set of edges on a least
               cost path from v1 to v2 in graph_edges *)

    type Path_Finder is modeled by (
            edges: finite set of Edge,
            insertion_phase: boolean
        )
        exemplar m
        initialization
            ensures m = ({}, true)

    operation Insert_Edge (
            alters m: Path_Finder,
            consumes v1: Integer,
            consumes v2: Integer,
            consumes id: Edge_Index,
            consumes cost: Real
        )
        requires m.insertion_phase and
                1 <= v1 <= Max_Vertex and
            1 <= v2 <= Max_Vertex and
            e.cost > 0.0
        ensures m.edges = #m.edges union {(#v1, #v2, #id, #cost)}
            and m.insertion_phase = #m.insertion_phase

    operation Stop_Accepting_Edges (
            alters m: Path_Finder
        )
        ensures m.edges = #m.edges and
```

```
            m.insertion_phase = false

    operation Connecting_Path_Exists (
            preserves m: Path_Finder,
            preserves v1, v2: Integer
        ) returns result: Boolean
        requires not m.insertion_phase
        ensures result = CONNECTING_PATH_EXISTS (m.edges, v1, v2)

    operation Find_Least_Cost (
            preserves m: Path_Finder,
            preserves v1, v2: Integer
        ) returns result: Real
        requires not m.insertion_phase and
            CONNECTING_PATH_EXISTS (m.edges, v1, v2)
        ensures there exists s: set of Edge
            (IS_A_LEAST_COST_PATH (m.edges, v1, v2, s) and
             result = sum e: Edge where (e is in s) (e.cost))

    operation Get_Last_Stop (
            preserves m: Path_Finder,
            preserves v1, v2: Integer,
            produces stop: Integer,
            produces id: Edge_Index,
            produces cost: Real
        )
        requires not m.insertion_phase and
            CONNECTING_PATH_EXISTS (m.edges, v1, v2)
        ensures CONNECTING_PATH_EXISTS (m, v1, v2) and
                there exists s: set of Edge
                 (IS_A_LEAST_COST_PATH (m.edges, v1, stop, s) and
                  IS_A_LEAST_COST_PATH (m.edges, v1, v2,
                      s union {(stop, v2, id, cost)}))

    operation Is_In_Insertion_Phase (
            preserves m: Path_Finder
        ) returns result: Boolean
        ensures result = m.insertion_phase

end Least_Cost_Path_Finding_Template
```

Figure 1. `Least_Cost_Path_Finding_Template`

To use the template in Figure 1, a client needs to create an instance by picking a suitable value for `max_vertex` and an identification type for Edges (e.g., names). The module provides an ADT, named `Path_Finder`. The abstract state of the type has been modeled mathematically as an ordered pair: a finite set of (graph) edges and a boolean value that is true iff edges are allowed to be inserted. In the description of this mathematical model, we have used a mathematical subtype Edge [4,9]: values of this type are constrained to be such that edge weights are positive. The essential purpose of the subtype is to make the specification easier to understand.

The **initialization ensures** clause specifies that every newly-declared object of type `Path_Finder` contains no graph edges and is ready for insertion. Edges of the graph, for which least cost paths need to be found, are inserted one at a time to through calls to `Insert_Edge` operation. The operation takes as its parameters an object m, and information on the edge that is inserted. The operation has a pre-condition as specified in its **requires** clause: `m.insertion_phase` must be true. The operation alters the state of the machine m as specified in the post-condition and consumes the edge information. In the **ensures** clause, `#m` denotes the value of the object that is input to the operation and `m` denotes its value after the operation. (In the requires clause, all parameter names refer to input values.) The values of the consumed parameters are left unspecified. They have initial values of their types after the operation.

The three operations `Connecting_Path_Exists`, `Find_Least_Cost`, and `Get_Last_Stop` require `m.insertion_phase` to be false, i.e., all edges must be available. Through these requires clauses, the specification dictates conceptually the order in which the operations can be called. In particular, the `Stop_Accepting_Edges` operation, which ensures that `m.insertion_phase` is false, must be called before the path query operations. The last operation `Is_In_Insertion_Phase` can be used to determine the insertion status of the object.

The operation `Connecting_Path_Exists` returns true iff there is a connecting path between the two given vertices in the graph. In the ensures clause, we have used a mathematical predicate `CONNECTING_PATH_EXISTS` with the obvious meaning. A formal definition of this predicate should be included in the specification, but has been omitted here for brevity. In the specification of operations `Find_Least_Cost` and `Get_Last_Stop`, we have used another predicate `IS_A_LEAST_COST_PATH (e, v1, v2, s)`. This predicate is true iff the set of edges `s` constitutes a least cost path from `v1` to `v2` in the graph whose edges are in `e`. The `Get_Last_Stop` operation returns the penultimate vertex (and the corresponding edge) in a least cost path from `v1` to `v2`. By calling this operation repeatedly, with the returned vertex as the destination, all edges on a least cost path from `v1` to `v2` can be found, incrementally. Similarly, by calling the operation with suitable parameters, the least cost paths between different sources and destinations can be found.

The specification of the data abstraction serves as the contract between modules that implement the abstraction and modules that use the abstraction. Different implementations of the abstraction hide both the data structures and algorithms used in computing the results, as well as whether the results are computed in batch or incrementally. For calling modules, finding least costs paths using the data abstraction is as simple as using a more typical ADT such as a stack or a queue. Switching from one implementation of the data abstraction to another is just as easy.

4 Performance Ramifications

Unlike the conventional modularization in Section 2 in which algorithms are encoded as batch computing procedures, the data abstraction interface in Section 3 permits incremental inputs and outputs. This distinction may not make a difference for problems where no incremental input processing is meaningful because all known algorithms require knowledge of all inputs. For some other problems, all known algorithms may need the same execution time to compute full or partial solutions, and the distinction may not matter either. But there are other problems, such as those discussed in this paper and elsewhere [8,13], where such is not the case. In these cases, the separation of processing from input/output steps leads to computation of expensive and complete solutions, even when the applications may need only a subset of outputs.

For the least cost path problem, for example, conventional modularization leads to a procedure that computes all shortest paths from the source, though the caller may be willing to abandon computation once the shortest path(s) to desired destination(s) are found. This performance problem is a result of the rigid input/process/output computing that is introduced when an algorithm is designed as a single procedure. In this case, the caller has no communication mechanism to stop computation once questions of interest have been answered. But the incremental interface of the data abstraction for the problem provides the calling module this flexibility. This is clear from considering procedure `SSSP_Dijkstra` in Section 2 and operations `Find_Least_Cost/Get_Last_Stop` in Section 3.

Consider an implementation of `Least_Cost_Path_Finding_Template` using Dijkstra's greedy algorithm. This algorithm has the property that at any time during computation, for the destination vertices in a set S, the shortest paths and costs for those paths are available in the other structures. When one of the operations `Find_Least_Cost` or `Get_Last_Stop` is called, the implementation can stop computation as soon as S contains the desired destination, and store all intermediate results. This approach can offer significant performance savings on the average. In addition, if the intermediate results are stored in a transitive closure matrix, when operations `Find_Least_Cost` or `Get_Last_Stop` are called with different sources or destinations, the information in the matrix can be used without re-computation. Such an implementation is superior to both of the procedures outlined in the last section for applications that need partial results.

Though incremental/amortized cost computing is a key benefit of the data abstraction interface, it is important to note that the interface does not preclude batch computing implementations that compute all results. For example, a different implementation of the `Least_Cost_Path_Finding_Template` may use `Floyd_Warshall`'s algorithm and compute all shortest paths, when the operation `Stop_Accepting_Edges` is called. In other words, the data abstraction interface allows the plug-compatible implementation strategies that differ both in how and when results are computed.

The least cost path problem is an illustrative example that typifies how other classical algorithms can be packaged as data abstractions. We have discussed elsewhere, for example, data abstractions for classical algorithms such as sorting, graph algorithms such as finding a minimum spanning forest of a graph, and other optimization problems [13,12]. A data abstraction that encapsulates an ordering algorithm and related data structures, for instance, allows alternative implementations that order inputs incrementally as they are inserted, or in batch after all items are inserted, or in an amortized fashion during extraction of outputs, or using a combination of strategies. We have also documented the more general impact of both duration and limited storage capacity considerations on data abstraction interface design for a graph algorithm [11]. In every case, the result is a data abstraction that allows independent team development of modules, ease of use, ease of change, and performance flexibility and incremental computation.

5 Conclusions

Classical data structure/algorithm modularization continues to dictate software modularization, despite its disadvantages for software engineering. Even modern object-based approaches typically encapsulate data structures alone. They treat data structures such as queues, lists, trees, sets, and maps as objects, but leave algorithms as procedures/methods that manipulate these objects [1,5].

Typical object-oriented code for Dijkstra's algorithm for finding a shortest path, for example, is similar to that in [2], except that it might use encapsulated graph and set objects instead of unencapsulated graph and set data structures [3,6,14]. It is especially instructive to compare the above data abstraction solution to the shortest path problem with the approach used by Weihe in [14]. Weihe shares our objective of making algorithms more reusable, without sacrificing efficiency. His approach also allows performance "tuning" of a client to use a particular algorithm (e.g., by stopping computation early), by having interface operations that take smaller steps. However, without a data abstraction approach to the problem, considerable modifications to client code are essential to use different algorithms that provide "order of magnitude" performance improvements. The traditional modularization problems that preclude independent software development, originally catalogued by Parnas, remain.

The performance issues discussed in this paper have also brought into focus the need for "online" algorithms. In the data abstraction view, these algorithms become natural alternative implementations and provide additional performance flexibility to clients. However, the interfaces need to be designed carefully to allow use of such algorithms.

In this paper, we have illustrated how new kinds of data abstractions can be developed following the principle of information hiding. Unless such data abstractions are defined and implemented to replace the traditional data structure/algorithm modularization, the full potential of the information hiding principle for software engineering cannot be realized.

Acknowledgments

We would like to thank members of the Reusable Software Research groups at The Ohio State University and at West Virginia University for discussions relating to the contents of this paper. In particular, we would like to thank Jennifer Kannen, Jen Fox, and Susan Hohenberger for implementing and experimenting with the ideas in this paper, and Greg Kulczycki for his careful reading and formatting.

We gratefully acknowledge financial support from our own institutions, from the National Science Foundation under grants CCR-9311702, DUE-9555062, and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project number P116B60717, from the National Aeronautics and Space Administration under project NCC 2-979, from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office, and from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Education, NASA, the U.S. Department of Defense, or Microsoft.

References

1. Booch, G.: *Software Components With Ada.* Benjamin/Cummings, Menlo Park, CA (1987).
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to Algorithms.* The MIT Press, Cambridge, MA (1990).
3. Flamig, B.: *Practical Algorithms in C++.* Coriolis Group Book (1995).
4. Heym, W.D., Long, T.J., Ogden, W.F., Weide, B.W.: *Mathematical Foundations and Notation of RESOLVE*, Technical Report OSU-CISRC-8/94-TR45, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, Aug 1994.
5. Meyer, B.: *Object-Oriented Software Construction.* 2nd edn. Prentice Hall PTR, Upper Saddle River, New Jersey (1997).
6. Mehlhorn, K., Naher, S.: "LEDA: A Library of Efficient Data Structures and Algorithms" *Communications of the ACM 38*, No. 1, January 1995, 96–102.
7. Musser, D. R., Saini, A.: *STL Tutorial and Reference Guide.* Addison-Wesley Publishing Company (1995).
8. Parnas, D.L.: "On the criteria to be used in decomposing systems into modules." *Communications of the ACM 15*, No. 12, December 1972, 1053–1058.
9. Rushby, J., Owre, S., and Shankar, N.: "Subtypes for Specification: Predicate Subtyping in PVS," *IEEE Transactions on Software Engineering 24*, No. 9, September 1998, 709-720.
10. Sitaraman, M., Weide, B.W. (eds.): "Special Feature: Component-Based Software Using RESOLVE." *ACM SIGSOFT Software Engineering Notes 19*, No. 4, October 1994, 21–67.

11. Sitaraman, M: “Impact of Performance Considerations on Formal Specification Design,” *Formal Aspects of Computing 8*, 1996, 716–736.
12. Sitaraman, M., Weide, B.W., Ogden, W.F.: “On the Practical Need for Abstraction Relations to Verify Abstract Data Type Representations,” *IEEE Transactions on Software Engineering 23*, No. 3, March 1997, 157–170.
13. Weide, B.W., Ogden, W.F., Sitaraman, M.: “Recasting Algorithms to Encourage Reuse,” *IEEE Software 11*, No. 5, September 1994, 80–89.
14. Weihe, K.: “Reuse of Algorithms: Still A Challenge to Object- Oriented Programming,” *ACM SIGPLAN OOPSLA Proceedings*, 1997, 34–46.

Two Ways to Bake Your Pizza — Translating Parameterised Types into Java

Martin Odersky[1], Enno Runne[2], and Philip Wadler[3]

[1] Ecole Polytechnique Fèdèrale de Lausanne[†]
Martin.Odersky@epfl.ch
[2] University of Karlsruhe[‡]
enno.runne@ira.uka.de
[3] Lucent Technologies
wadler@research.bell-labs.com

Abstract. We have identified in previous work two translations for parametrically typed extensions of Java. The homogeneous translation maps type variables to a uniform representation, while the heterogeneous translation expands the program by specialising parameterised classes according to their arguments. This paper describes both translations in detail, compares their time and space requirements and discusses how each affects the Java security model.

1 Introduction

Java does not provide parameterised types, but is designed to make them relatively easy to simulate. Consider `java.util.Hashtable`, the standard library class whose declaration is shown in Figure 1. The `Hashtable` design uses `Object` as the type of both keys and values. Every reference type is a subtype of `Object`, and so may be used as a key or value. It is up to the user to recall what types have been used, and perform appropriate casts. For instance, the `HashtableClient` class uses a hashtable with keys of type `String` and values of type `Class`. Lookups in the hashtable return an `Object` value that must be cast to a `Class`.

Idioms like this are widespread in the Java library, including simple utilities like stacks and vectors, the event model introduced for Java 1.1 (where events may return values of arbitrary type), the JavaBeans library (which depends on events), and the collection classes planned for Java 1.2 (which provides generic collections, sets, lists, and hashtables).

In Pizza [OW97a], we have extended Java with features common in functional programming languages. The most important extension in Pizza is its type system, which supports parameterised types. Figure 2 shows how hashtables are defined in Pizza. Here, `Key` and `Data` are type parameters for the `Hashtable` class. Access code for parameterised hash tables no longer needs a type cast to recover the types of the retrieved elements.

[†] This work was completed while at the University of South Australia.
[‡] This work was completed while at the University of South Australia.

```
public class Hashtable {
    public Hashtable () { ... }
    public Object get(Object key) { ... }
    public Object put(Object key, Object value) { ... }
}

class HashtableClient {
    private Hashtable loadedClasses = new Hashtable();
    public Class loadClass(String name) {
        Class c = (Class)loadedClasses.get(name);
        if (c == null) {
            c = Class.forName(name);
            loadedClasses.put(name, c);
        }
        return c;
    }
}
```

Fig. 1. Hashtables in Java.

There have been also several other proposals to extend Java with parameterised types [MBL97,AFM97,Bru97], with roughly similar capabilities as the Pizza approach, as well as proposals to extend Java with virtual types [Tho97], [Tor97], with capabilities quite different to it [OW97b]. Pizza differs from these other proposals in that it admits basic types as type parameters and in that it has polymorphic methods in addition to parameterised types.

Generally, there is widespread agreement that a generic type system with its benefits for the construction of reliable software systems is a useful addition to Java. It is less clear how such a type system should be implemented in the confines of the Java Virtual Machine. In earlier work [OW97a] we outlined two translations from Pizza's generic types to Java. In this paper we present the two translations in detail, discuss their security aspects, and compare their resource consumption empirically.

The *homogeneous* translation maps a parameterised class into a single class that represents all its instantiations. For example, the homogeneous translation of the Pizza code for parametric hashtables in Figure 2 yields essentially the Java code in Figure 1. The homogeneous translation is similar to schemes for implementing parametric polymorphism in ML [Ler90,SA95]. The homogeneous translation underlies the current Pizza compiler whose use has become widespread since December 1996. This translation yields compact code that can run in standard Java-enabled browsers.

By contrast, the *heterogeneous* translation maps a parameterised class into a separate class for each instantiation. For example, the heterogeneous translation of the Pizza class `Hashtable<Key,Value>` replaces the instance `Hashtable<String,Class>` by the class `Hashtable$_String_$_Class_$`, whose

```
public class Hashtable<Key, Data extends Object> {
    public Hashtable() { ... }
    public Data get(Key key) { ... }
    public Data put(Key key, Data value) { ... }
}

class HashtableClient {
    Hashtable<String,Class> loadedClasses = new Hashtable();
    Class loadClass(String name) {
        Class c = loadedClasses.get(name);
        if (c == null) {
            c = Class.forName(name);
            loadedClasses.put(name, c);
        }
        return c;
    }
}
```

Fig. 2. Hashtables in Pizza.

body is generated by replacing each occurrence of `Key` by `String` and each occurrence of `Value` by `Class`. The heterogeneous translation resembles the translation scheme employed in the TIL compiler for Standard ML [HM95], the generic instantiation in Ada [oD80] and Modula-3 [CDG$^+$88], and the template expansion process in C++ [Str86]. However, unlike in C++, all type checking in Pizza is performed before expansion, not afterwards.

The heterogeneous translation poses the problem of keeping track of the different instances of generic types that are produced at compile-time or link-time. The Java environment admits an elegant solution to this problem: One can make use of Java's dynamic class loading capabilities to generate instances on demand at run-time. Agesen, Freund and Mitchell introduce a scheme based on this idea [AFM97]. They argue that the heterogeneous translation supported by their scheme offers several benefits:

First, since the heterogeneous translation preserves more type information than the homogeneous translation, it tends to give more freedom to the language designer. Language constructs such as checked type casts to parameterised types, generic array creation, or mixins [FKF98] require an explicit representation of type variables at run time and therefore are only feasible under a heterogeneous translation. (Consequently, since Pizza was designed to support both translations, these constructs are missing in Pizza.)

Second, the heterogeneous translation may lead to better run-time performance, trading off speed for code size. By specialising a generic type with its arguments, one can eliminate all type casts and boxing/unboxings introduced by the homogeneous translation. Furthermore, the specialisations might provide additional opportunities for subsequent optimisations.

While these arguments are plausible, they have not yet been verified empirically. We therefore extended the Pizza compiler to allow the heterogeneous translation as well as the existing homogeneous translation. Moreover the new translation allows heterogeneous and homogeneous code to be mixed, which is necessary to support Pizza's generic methods which are still translated homogeneously. We then compared both translations using a range of benchmarks, including the Pizza compiler itself. The results can be summarised as follows.

- As expected, the heterogeneous translation leads to somewhat faster code for member access in generic types, and to much faster code for generic array access. Except for arrays, the performance gains are modest, however. They do not generally exceed 5%. Furthermore, code involving polymorphic methods becomes much slower since the interface between homogeneous and heterogeneous code is complex.
- The increase in code size incurred by the heterogeneous translation can be substantial. For example, the footprint of the Pizza compiler increases by about 60%.
- With Sun's current JDK, the additional classes generated by the heterogeneous translation incur a significant class loading overhead, often large enough to wipe out the performance gains obtained by specialisation.
- In the course of our experiments we also discovered a severe security problem incurred by the heterogeneous translation. Unless the access checking in the Java Virtual Machine can be refined, this problem can be avoided only by restricting the source language to a point where the usefulness of parameterisation is very much in doubt.

The rest of this paper is organised as follows. Section 2 describes the homogeneous translation from Pizza to Java, and Section 3 describes its heterogeneous counterpart. Section 4 compares the two translations in terms of their security aspects. Section 5 compares them empirically in terms of their run-time and space consumption using a number of benchmark programs. Section 6 concludes.

2 The Homogeneous Translation

The homogeneous translation from Pizza to Java proceeds in three steps, which affect types, expressions, and declarations. First, Pizza types are mapped to Java types by erasing all type parameters, and by mapping all type variables to their upper bound. Then, type conversions are inserted in expressions where needed to avoid type compatibility errors. The effect of the first two steps is illustrated by comparing the Pizza program in Figure 2 with its translation in Figure 1.

The first step, type erasure, is defined recursively as follows:

1. The erasure of a parameterised type $C\langle T_1, ..., T_n\rangle$ is its class or interface name, C.
2. The erasure of an array type $A[\,]\ldots[\,]$ whose element type A is a type variable is the abstract class `pizza.support.array`.

```
abstract public class array {
    public abstract int length();
    public abstract Object at(int i);
    public abstract void at(int i, Object x);
    ...
}

public class booleanArray extends array {
    private boolean[] elems;
    public int length() { return elems.length; }
    public Object at(int i) { return new Boolean(elems[i]); }
    public void at(int i, Object x) {
       elems[i] = ((Boolean)x).booleanValue();
    }
    ...
}
```

Fig. 3. Array classes.

3. The erasure of every other type variable is the erasure of the type variable's bound, or `Object` if no bound was given.
4. The erasure of every other type is the type itself.

Arrays are treated specially in the second rule above. Without this rule, the array type `A[]` with type variable `A` would be mapped to `Object[]`. The problem is that `Object[]` is not convertible to arrays of primitive types except by copying the whole array, which loses sharing. Instead, generic arrays are mapped to the abstract class `pizza.support.array`, from whose definition we excerpt in Figure 3. This class provides a generic interface to an array. It has a method to return the length of the array as well as *getter* and *setter* methods for accessing individual elements. The translation maps accesses to the `length` field of a generic array to calls of the `length()` method, and it maps indexed accesses of generic arrays to calls of the getter and setter methods.

There are nine subclasses of `pizza.support.array`, one for each of Java's eight basic types, plus one for type `Object` which is the generic representation of all arrays of reference type. Each subclass implements the abstract methods in class `pizza.support.array` via a private array of the corresponding element type. As an example, Figure 3 gives the subclass for arrays of booleans.

The Pizza translation of expressions inserts type conversions as needed to make the resulting Java code legal. Type conversions between reference types are simply type casts. The Pizza type system guarantees that these type casts will always succeed at run-time, but they are still necessary to let the generated classes pass the byte-code verifier, which checks programs according to Java's typing rules. Type conversions between basic types and reference types work with Java's mirror classes for basic types. For instance, a `boolean B` is converted to an `Object` with `new Boolean(B)` and the reverse conversion from an `Object` `O` is `((Boolean)O).booleanValue()`.

Type conversions between basic arrays and the generic array class involve a wrapper class for the basic array. For instance, a `boolean[]` array `BS` is converted to `pizza.support.array` with `new booleanArray(BS)` and the reverse conversion from a generic array `A` can be achieved with `((booleanArray)A).elems`.

The last step of the translation deals with the problem that type erasure can destroy overriding and implementation relationships between Pizza methods. As an example, consider a parameterised interface `Iterator<A>` and a class that implements iterators of element type `String`:

```
interface Iterator<A> {
   A next();
   void append(A x);
}
class StringArrayIterator implements Iterator<String> {
   String[] a;
   public String next() { ... }
   public void append(String x) { ... }
}
```

The translation after erasing types is:

```
interface Iterator {
   Object next();
   void append(Object x);
}
class StringArrayIterator implements Iterator {
   String[] a;
   public String next() { ... }
   public void append(String x); { ... }
}
```

This translation has two problems. First, the implementation of method `next` in class `StringArrayIterator` has a different return type than its definition in interface `Iterator`, which violates a requirement of the Java language [GJS96, Sec. 8.4.6.3]. Second, the implementation relationship between the two append methods in the original Pizza source does not translate to a corresponding relationship in their translations: After translation, `append` in class `StringArrayIterator` takes a `String` as argument, and hence does not implement `append` in interface `Iterator`, which takes an `Object` as argument.

To correct these shortcomings, the translation inserts *bridge methods* to restore overriding relationships. Bridge methods have the type after translation of the implemented or overridden method. Their body simply forwards the call to the true implementing method, converting argument types as needed.

To avoid name clashes between bridge methods and other methods, and to avoid the problem with different return types in implementations, all methods whose type refers to a type parameter are coded with the name of their enclosing class. For instance, our iterator example would be translated as follows.

```
abstract class Iterator {
   Object Iterator$next();
   void Iterator$append(Object x);
}
class StringArrayIterator implements Iterator {
   String next() { ... }
   void append(String x); { ... }
   final Object Iterator$next() { return next(); }
   final void Iterator$append(Object x) { return append((String)x); }
}
```

The main strengths of the homogeneous translation are its simplicity and the compact size of generated code. Its main disadvantage is the run-time overhead introduced by bridge methods and type conversions. Type conversions from basic types or basic array types to their generic representations are particularly expensive since they involve the creation of a wrapper object. These costs are eliminated in the heterogeneous translation, which we discuss presently.

3 The Heterogeneous Translation

The heterogeneous translation expands the program by specialising generic classes according to their arguments, so that types become monomorphic. This expansion is performed on demand at load time, using a modified class loader which produces instance classes from class templates. The translation and expansion process resembles the one described by Agesen, Freund and Mitchell [AFM97], but there are two complications.

First, parameters in Pizza can be base types as well as reference types. Therefore, our class loader must be able to instantiate a parameterised class template with a base type, which generally requires changes to the byte codes. In Agesen *et al.*'s scheme, which only considers reference type parameters, a class file's constant pool is all that needs to be changed.

Second, Pizza has polymorphic methods, which are unaffected by the class loader expansion. Consider for instance the `zip` method in Pizza's `List` class which joins two lists into a list of pairs.

```
public class List<A> {
  public <B> Pair<A,B> zip(List<B> xs) { ... }
  ...
}
```

A specialisation of `List`'s parameter `A` to, say, `String` would leave the following residual function, which is still polymorphic:

```
  public <B> Pair<String,B> zip(List<B> xs) { ... }
```

Now, one might consider expanding such polymorphic functions as well as expanding parameterised types. This is difficult, however, because at the point where we load a class, we may not yet know the possible instances of its polymorphic methods. Since we can't change the code of a class after the class is

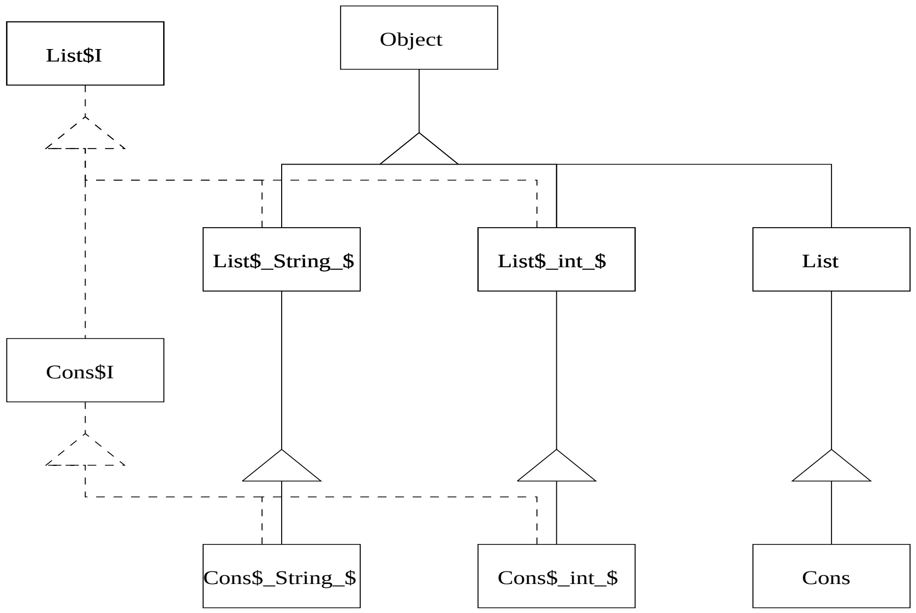


Fig. 4. Classes generated under the heterogeneous translation.

loaded, we can specialise generic methods only by placing each in a compiler-generated inner class of its own, which can then be specialised at each call. This approach would incur a heavy overhead at both compile-time and run-time.

Instead, our scheme leaves polymorphic methods as they are, using the homogeneous translation for all type variables which are not parameters. As a consequence, an object might now be accessed at the same time from code that is specialised and from code that is not. To allow this, we provide two *views* for every parameterised type. The homogeneous view maps the type's parameters to a uniform representation, namely the erasures of their bounds. The heterogeneous view is simply the specialised version of the class. The homogeneous view takes the form of a Java interface which all specialisations implement. This dual view would also provide the opportunity for more refined translation schemes, which mix homogeneous and heterogeneous translations in order to obtain a good balance between speed and code size.

We now describe the translation in more detail. For each parameterised class `C<A>`, three Java classes are generated.

– An *interface* `C$$I` which represents the homogeneous interface to the class,
– A *template class* `C$_$0_$` which the class loader will instantiate to obtain specialisations as needed,

```
public class List<A> {
    public A head;
    public List<A> tail;

    public List<A> append(A last) {
        return new List(head, tail == null
                              ? new List(last, null)
                              : tail.append(last));
    }

    public static <A> List<A> fromArray(A[] a) {
        List<A> xs = null;
        for (int i = a.length - 1; i >= 0; i--)
            xs = new List(a[i], xs);
        return xs;
    }

    public List(A x, List<A> xs) {
        head = x; tail = xs;
    }
}
```

Fig. 5. Generic list class.

– A *static base class* `C` which implements all static methods of `C`, and also provides methods to construct new instances of `C`.

If parameterised classes inherit from other parameterised classes, the inheritance relationship is translated to an inheritance relationship between corresponding components. For instance, Figure 4 shows the case of a parameterised class `Cons<A>` that extends `List<A>`. This figure shows the static base classes `List` and `Cons`, instance classes with `int` and `String` as the parameter type, and the homogeneous interfaces `List$$I` and `Cons$$I`. Class inheritance is expressed by solid lines and interface implementation is expressed by dashed lines.

To illustrate the roles of these classes and interfaces, consider the generic list class of Figure 5. Lists have `head` and `tail` fields. There is a method `append` that returns a new list with the given element appended to the elements of the current list. There is also a static polymorphic method `fromArray` which constructs a list from the elements of an array.

Homogeneous Interfaces

The homogeneous view of this class will map every occurrence of the type variable `A` to `A`'s upper bound, in this case `Object`. The view is represented by a Java interface, `List$$I`, given in Figure 6.

```
interface List$$I {
   List$$I List_append(Object x$0);
   List$$I List_tail();
   List$$I List_tail(List$$I x$0);
   Object List_head();
   Object List_head(Object x$0);
}
```

Fig. 6. Homogeneous interface for `List`.

The homogeneous interface contains one access method for every instance method whose type refers to a parameter of the class. Generic instance fields are represented by getter and setter methods in the interface.

Templates and Specialisations

A specialisation of class List with a concrete element type `X` is obtained simply by replacing the type parameter `A` with `X`. Instead of `List<X>`, which is not a legal class name in Java, we use `List$_X_$`.

In addition to all instance members of the original class, specialisations also contain implementations of all methods in the homogeneous interface. These implementations simply call the original method or load/store the original instance field, wrapping and unwrapping objects as required.

Specialisations are synthesised at run-time from a template class. Template classes are very similar to their instantiations, except that they use *place-holders* instead of concrete parameter types. Place-holders are named `$0`, `$1`, and so on, up to the number of type parameters of a class. As an example, Figure 7 presents the template class for lists.

Polymorphic Methods

Polymorphic methods are translated using a modified version of the homogeneous translation described in the last section. The erasure of a parameterised class is now its homogeneous interface. Accesses to fields and methods of such a class are translated to corresponding accesses in the homogeneous interface. Another change is that the actual instance type of a polymorphic method is made available at run time by passing implicit type parameters as additional parameters of type `Class`. For instance the method signature

```
<A> List<A> fromArray(A[] a)
```

becomes

```
List<Object> fromArray(Class A, pizza.support.array a)
```

```
public class List$_$0_$ implements List$$I {
   public $0 head;
   public List$_$0_$ tail;

   public List$_$0_$ append($0 last) {
      return new List$_$0_$(head, tail == null
                                  ? new List$_$0_$(last, null)
                                  : tail.append(last));
   }

   public List$_$0_$($0 x, List$_$0_$ xs) {
      super(); head = x; tail = xs;
   }

   /* Implementations of interface functions for homogenous access */

   public Object List_head() { return (Object)head; }
   public Object List_head(Object x$0) {
      head = ($0)x$0; return x$0;
   }
   public List$$I List_tail() { return tail; }
   public List$$I List_tail(List$$I x$0) {
      tail = (List$_$0_$)x$0; return x$0;
   }
   public List$$I List_append(Object x$0) { return append(($0)x$0); }
}
```

Fig. 7. Template class for `List`.

We have not yet explained how instances of a parameterised type are created from within a polymorphic method. The matter is trivial whenever the parameters are known types or type parameters of the enclosing class. In this case we simply invoke the corresponding constructor of the instance class (or its template). But if some parameters are local type variables of a polymorphic method, the correct instance is known only at run time, when the actual argument type is passed. In these situations, we have to resort to Java's reflection library for object construction. For example, the object allocation in method `fromArray` would involve the following three steps.

```
// Create class of object to be allocated:
Class list_A = Class.forName("LinkedList$_" + A.getName() + "_$);

// Obtain constructor given argument types:
Constructor constr = list_A.getConstructor(new Class[]{A, list_A});

// Invoke constructor:
return (LinkedList$$I) constr.newInstance(new Object[]{x, xs});
```

To make this scheme reasonably efficient, both class and constructor objects are cached in hashtables which are indexed by the argument type(s). There will be one such hashtable for every constructor of a parameterised class. Using caching, each constructor method will be looked up only once through the reflection interface. Caching was essential in our implementation since method and constructor lookup was very slow. With caching, we observed a slowdown of between 2 and 3 for programs that used polymorphic static methods exclusively, as compared to the same programs using instance methods. Without caching, the slowdown was about a factor of 500.

The Specialising Class Loader

The heterogeneous translation uses a customised class loader for expanding template classes with their actual type parameters. The class loader inherits from class `java.lang.ClassLoader`, overriding method `loadClass(String name, boolean resolve)`. Our new implementation of `loadClass` scans the given name for embedded type parameter sections, delimited by `$_` and `_$` brackets. If a parameter section is found, a template for the class constructor is loaded and expanded with the given parameter(s). Once loaded, template classes are kept in a cache for subsequent instantiations.

A classfile consists in essence of a *constant pool* which holds all constant information such as classnames, method descriptors and field types, a *fields* section and a *methods* section [LY96]. The code for all methods and for variable initialisers is stored in *code attributes* attached to the methods or the class.

Expansion of a classfile consists of two steps. The first step involves replacing in the constant pool all occurrences of a place holder name with the name of the corresponding actual type parameter. Quite surprisingly, this is all that is needed for expanding classes with reference type parameters.

If type parameters are basic types, a second step is required to adapt code blocks to the actual parameters. The Java Virtual Machine uses different bytecode instructions for accessing data of different type groups. For instance, a local variable of type `int` is accessed with the instruction `iload` whereas `aload` is used if the variable is of reference type. To help the class loader find all instructions that need to be adapted, the compiler generates a `Generic` attribute that gives the offsets of these instructions relative to the start of the code block.

4 Security Implications

The homogeneous and heterogeneous translations have quite different security implications. Under the homogeneous translation, a Pizza program is no more secure than the generated Java code after type erasure. Since type parameters are erased by the translation, we can not always guarantee that a parameterised type is used at run-time with the type parameter indicated in the source. As Agesen *et al.* [AFM97] argue, this constitutes a security risk as users might expect run-time type-checking to extend to fully parameterised types.

The heterogeneous translation does not have this problem, since all type information is maintained at run-time. Rather to our surprise, the heterogeneous translation nevertheless fits poorly with the security model of Java. We encountered two difficulties, one with visibility of instance methods, and one with visibility of type parameters.

Visibility of Instance Methods

Because the Pizza implementation mixes homogeneous and heterogeneous translations, all the heterogeneous instantiations of a class must be accessible via a common interface. For instance, in Section 3 there is a list interface (Figure 6) that is implemented by the list template (Figure 7). In Java, all methods implementing an interface are required to be public, and hence the Pizza translation works only when all instance methods are public.

If the template class has no superclass (other than `Object`), we could solve this problem by replacing the interface with an abstract class, since abstract classes do allow non-public members. Unfortunately, the template class may already have a superclass that depends on the type parameter, so the multiple inheritance provided by interfaces is essential.

If one were allowed to change the Java security model, the problem could be fixed by generalising interfaces to allow non-public methods. Since the JVM is currently being implemented in silicon, such a change seems unlikely.

Thus, the combination of homogeneous and heterogeneous translations of Pizza requires all instance methods of a parameterised class to be public, which severely restricts the utility of the Java visibility model. This problem would not arise if one adopted a pure heterogeneous translation, since then homogeneous interfaces would not be required. By contrast, the next security problem is inherent in the heterogeneous translation,

Visibility of Types

The JVM security model supports only two kinds of visibility for classes: package-wide and public visibility. It is not possible to use a class to reference objects outside a package unless the class is declared to be public. (This restriction holds even if the verifier is disabled, since the JVM specification [LY96] requires the virtual machine to throw an `IllegalAccessError` if a class refers to any other class that is in another package and not public.)

Consider an instantiation `C<D>` of a parameterised class `C` defined in package `P`, applied to a parameter class `D` defined in a different package `Q`. There are two possibilities: either class `D` must be public (in which case we can place the instantiation in package `P`), or else the body of class `C` must refer only to public classes (in which case we can place the instantiation in package `Q`).

Since classes usually refer to non-public classes in their package (otherwise, why have packages?), the Pizza compiler in its heterogeneous version limits itself to the first case: classes used as type parameters of public types defined in another package must be themselves be public. This rule severely restricts the utility of the Java visibility model.

Further, even if one were allowed to change the Java security model, it is not clear how to fix this problem.

5 Performance Evaluation

We initially believed that the heterogeneous translation would result in notably faster code than the homogeneous translation. We expected a certain loss at start time due to the class expansion, but assumed that than the execution — especially of classes parameterised with basic types — should be faster, as the heterogeneous translation does not need to convert between boxed and unboxed representations.

To evaluate the heterogeneous translation we compared the execution of several small benchmark programs. We used our own class loader in both cases. We ran each test at least 50 times. We ran all benchmarks on Sun's JVM 1.1 final on a Sun Ultra Sparc 1.

Micro Benchmarks

Our micro benchmarks try to measure one aspect of Java's execution in both translations. The code for these benchmarks is found in Appendix A.

The two versions of the *list-reverse* benchmark are designed to estimate the speedup of the heterogeneous translation for member access, as well as its slowdown for polymorphic methods.

The `List<int>` benchmarks show that polymorphic methods slow down significantly as every access to the list objects is made via the homogeneous interface. The instance reverse method shows only a slight speedup; we had expected a greater difference. We have smaller startup costs in the `List<String>` benchmarks as the class expansion is much easier for reference types.

The *cell* benchmark measures access of a variable of a parameterised type. This time, the heterogeneous translation yields a significant speedup for base types, which has to do with the fact that the variable accesses dominate all other costs. With a reference type, the heterogeneous translation shows a smaller speedup, since instead of an unboxing method call only an additional type cast is required. The cell benchmark was also used by Myers *et al.* to to test their implementation of parameterised types for Java [MBL97]. Our findings show a somewhat larger speedup for the heterogeneous translation compared to theirs (27% as opposed to their 14%).

The next set of benchmarks were designed to compare the efficiency of array accesses in both translations. We used two implementations of a reverse function for Pizza's `Vector` class. One is part of the class itself and accesses all data directly. This version does no runtime wrapping for the basic types in the homogeneous translation within the `Vector` class. But it has the array access overhead including wrapping and unwrapping in the `pizza.support.array` class.

Table 1. Micro benchmark results; times given in seconds

benchmark	iterations	homogeneous	heterogeneous	%
Cell<int>	1,000,000	4.26	3.10	73
Cell<Integer>	1,000,000	3.22	3.09	96
List<int> instance reverse	10,000	110.34	93.37	85
List<int> static reverse	10,000	110.48	403.77	365
List<String> instance reverse	10,000	109.77	94.76	86
List<String> static reverse	10,000	108.78	371.97	342
Vector<int> internal	10,000	78.71	21.60	27
Vector<int> external	10,000	125.81	42.55	34
Vector<String> internal	10,000	39.41	22.58	57
Vector<String> external	10,000	62.33	42.54	68
Hashtable<int, ..>	10,000	161.20	276.54	172
Hashtable<Integer, ..>	10,000	161.03	168.41	105

The second version is not local to the `Vector` class and has to access the data via methods. The homogeneous translation needs to introduce wrapping and unwrapping for basic types as well as for the arrays.

The vector benchmarks show a large speedup for `Vector<int>`. The homogeneous translation performs not quite as bad with a reference type as the element type, since then no runtime wrapping overhead is incurred. The differences between the two translations decrease in the external version of the benchmark since there array accesses are less frequent in the instruction mix.

A slightly more involved benchmark measured the efficiency of *hashtable accesses* under both translations. We measured the efficiency of the `get` operation for hashtables with keys of the basic `int` type. To our surprise, the heterogeneous translation performed much worse than the homogeneous translation on this benchmark. The reason for this effect is that a `get` operation on a hashtable involves several calls to methods `hashCode` and `equals` of `get`'s key parameter. In the homogeneous translation, a hashtable key will be boxed once, before it is passed as a parameter to `get`. But in the heterogeneous translation the key will be passed in unboxed form, and will then be boxed each time a `hashCode` and `equals` method is invoked. Hence, we have increased rather than reduced the number of type conversions that need to be performed. The same effect does not arise if the key parameter is of a reference type, since then all type conversions in `get` are widening casts from a reference type which do not translate into any bytecode instructions at all.

Macro Benchmark

As large benchmark we used the execution of the Pizza compiler itself. We translated it once homogeneously and once heterogeneously. Both versions then did the same job. They translated the compiler source and the sources of Pizza's API packages `pizza.lang` and `pizza.util` to heterogeneous classes.

We observed that under the heterogeneous translation the number of loaded classes increased by 85% and the total code size increased by 60%. Code size was measured as the number of bytes passed to the class loader's `defineClass` method. This includes both constant pool and code blocks of classes. We further observed a slowdown of 26% for the heterogeneously compiled compiler. This slowdown was rather unexpected. To isolate the different factors that might contribute to the slowdown we ran the compiler twice on the same data such that during the second run no more classes needed to be loaded. This showed that of the total slowdown 19 percentage points are attributable to increased class loading overhead and 7 percentage points are attributable to execution overhead. The additional class loading overhead is incurred both by the fact that many more classes need to be loaded, and by the fact the loading of the additional classes involves an extra expansion step.

A major factor in the execution overhead is the use of polymorphic methods such as `List.map`, `List.forAll`, which are heavily used in some parts of the Pizza compiler. The `map` method in particular is expensive since it involves the construction of parameterised object instances through the reflection library.

Table 2. Execution of the Pizza compiler

	loaded classes	size	system classes	duration in seconds
homogeneous	213	702 kB	28	93
heterogeneous	393	1251 kB	31	117

parameterised classes	171
templates used	16
size of the templates	53.1 kB

6 Conclusion

In summary, we found that at least for the current JVM implementation the heterogeneous translation of parametric polymorphism has not fulfilled its initial promise. A significant increase in code size did not yield a clear improvement in runtime efficiency. We also noted a severe incompatibility between the heterogeneous translation and Java's package based security model.

Sun's JVM has several peculiarities which affected the outcome of the benchmarks: The memory management is very inefficient, loaded classes are looked up in a linear array, which slows down execution as more classes are loaded, and all instructions are interpreted. It is unclear how the comparative performance of both translations would be affected with a different machine. A better memory management would work primarily in favor of the homogeneous translation, since it makes boxing operations more efficient. On the other hand, a better class loading scheme would primarily benefit the heterogeneous translation.

We also attempted to run our benchmarks on Microsoft's virtual machine for Java (JView 2.0 beta on Windows NT), but were not able to complete them, due to problems in Microsoft's implementation of the reflection library. *In the future, we hope to have additional data for both the latest Microsoft virtual machine and Sun's Hotspot machine.*

Acknowledgments

This work was supported by Sun Laboratories Academic Collaboration Grant "Measuring the Cost of Parameterised Types in Java." We thank David Detlefs and Guy Steele for their support, suggestions, and helpful criticisms.

References

AFM97. Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java language. In *Proc. ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, 1997.

Bru97. Kim Bruce. Increasing Java's expressiveness with ThisType and match-bounded polymorphism. Technical report, Williams College, 1997.

CDG$^+$88. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report 31, DEC SRC, 1988.

FKF98. Matthew Flatt, Shriram Krishnamourthi, and Matthias Felleisen. Mixins for java. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, January 1998.

GJS96. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Java Series, Sun Microsystems, 1996. ISBN 0-201-63451-1.

HM95. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, pages 130–141, January 1995.

Ler90. Xavier Leroy. Efficient data representation in polymorphic languages. In P. Deransart and J. Małuszyński, editors, *Programming Language Implementation and Logic Programming*, pages 255–276. Springer-Verlag, 1990. Lecture Notes in Computer Science 456.

LY96. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Java Series, Sun Microsystems, 1996. ISBN 0-201-63452-X.

MBL97. Andrew C. Myers, Joseph. A. Bank, and Barbara Liskov. Parameterised types for Java. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 132–145, January 1997.

oD80. United States Department of Defense. *The Programming Language Ada Reference Manual.* Springer-Verlag, 1980.

OW97a. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

OW97b. Martin Odersky and Philip Wadler. Two approaches to type structure, 1997.

SA95. Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc. 1995 ACM Conf. on Programming Language Design and Implementation*, (ACM SIGPLAN Notices vol. 30), pages 116–129, June 1995.

Str86. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

Tho97. Kresten Krab Thorup. Genericity in java with virtual types. In *Proc. ECOOP '97*, LNCS 1241, pages 444–471, June 1997.

Tor97. Mads Torgersen. Virtual types are statically safe. Note, circulated on the java-genericity mailing list, 1997.

A Code of Micro Benchmarks

Static List Reverse

```
static <A> List<A> reverse(List<A> xs) {
  List<A> ys = Nil();
  while (true) {
    switch (xs) {
    case Nil(): return ys;
    case Cons(A x, List<A> xs1): ys = Cons(x, ys); xs = xs1; break;
} } }
```

Instance List Reverse

```
class List<A> {

  ...

  List<A> reverse() {
    List<A> xs = this;
    List<A> ys = Nil();
    while (true) {
      switch (xs) {
      case Nil(): return ys;
      case Cons(A x, List<A> xs1): ys = Cons(x, ys); xs = xs1; break;
  } } }
}
```

Cell Access

```
class Cell<A> {
  A elem;
  Cell() {}
  void add(A elem) { this.elem = elem; }
  A get() { return elem; }
}
Cell<int> c = new Cell();
c.add(42);
for (int i=0; i < iterations; i++) int j = c.get();
```

Internal Vector Reverse

```
public class Vector<A> {
  ...
  public void reverseElements() {
    for (int i = 0; i < elementCount / 2; i++) {
      A e = elementData[elementCount - i - 1];
      elementData[elementCount - i - 1] = elementData[i];
      elementData[i] = e;
    } } }
```

External Vector Reverse

```
  public void reverseElementsInt(Vector<int> v) {
    for (int i = 0; i < v.size() / 2; i++) {
      int e = v.elementAt(v.size() - i - 1);
      v.setElementAt(v.elementAt(i), v.size() - i - 1);
      v.setElementAt(e, i);
    } }
```

Main Loop of Hashtable Benchmark

```
    for (int i=0; i < 1000; i++) table.put(i, "A"+i);
    for (int j=0; j < iterations; j++) {
      for (int i=0; i < 1000; i++) {
        String s = table.get(i);
      } }
```

SuchThat — Generic Programming Works

Sibylle Schupp[1] and Rüdiger Loos[2]

[1] Rensselaer Polytechnic Institute
schupp@cs.rpi.edu
[2] Universität Tübingen
loosr@acm.org

Abstract. This paper introduces SuchThat, a language for experiments in generic programming. SuchThat forms the synthesis of two independently developed languages: Aldes, a language for algorithm descriptions, and Tecton, a specification language for generic components. The SuchThat language is characterized by its support of algorithmic requirements and dependent genericity. We discuss the underlying design decisions and show that the semantic interpretation of overloading plays a key role in determining optimal instantiations.

1 Introduction

SuchThat [20] is a programming language for experiments in generic programming. This paper introduces SuchThat and discusses its major features, including the underlying design decisions. Discussing the design of a language for generic programming, however, cannot be separated from a discussion of generic programming itself. In a companion paper [18] we therefore elaborated on our definition of generic programming. We described generic programming as *requirements-oriented programming* and explained how our understanding of requirements extends the traditional concept, which is related to the partial correctness of an algorithm, its pre- and post-conditions, and the functional specification between input and output parameters. We also postulated that requirements have to become constituent parts of a language, instead of being meta-notions, used to program and not just to reason about programs. In summary of our argumentation given there, requirements-oriented programming reinterprets the traditional correctness requirements as minimal requirements and complements them with performance requirements. A language for requirements-oriented programming, then, allows algorithms to be expressed in terms of minimal requirements and guarantees that their implementations satisfy specified performance assertions.

SuchThat is designed to support requirements-oriented programming during all phases of programming, from design, specification, and verification to coding, compilation, linking, and execution. It has not been developed from scratch but rather glues together two independent languages: Aldes [12], a language for algorithm descriptions, and Tecton [6,16], a language for concept descriptions. Aldes, resembling Knuth's style of algorithm descriptions, serves as the implementation language of the computer algebra system SAC-2 [1], a system of algebraic algorithms with highly elaborated specifications. Tecton, a specification language for generic components, is characterized by a novel approach of defining concepts—that is, sets of many-sorted algebras—through refinements and replacements. Several components of generic libraries have been specified in Tecton [17,19]. Though both languages continue to be used independently, each benefits from its integration into SuchThat. When supplemented by Aldes algorithms, Tecton function descriptions become executable, as conversely Aldes algorithms become controllably generic through Tecton concepts. Their synthesis into SuchThat, at the same time, opens up new possibilities for generic programming. It also poses challenging problems for compiler construction. To properly treat all aspects of static and dynamic genericity, a non-traditional compilation model is necessary.

This paper is logically divided into three parts. The first part (sections 2-4) deals with requirements. First, we illustrate specification requirements in SuchThat. Second, we explain the problem of correctly instantiating a generic algorithm and show that Tecton's set-theoretical semantics serves as the base of requirements-oriented type checking. Third, we consider algorithmic requirements and describe their role in compiler construction and library design. The second part (sections 5-7) centers on overloading. In contrast to other programming languages or methodologies, overloading constitutes a semantic feature of generic programming and plays an instrumental role in determining the optimal instantiation. We first discuss overloading conceptually, then we look into dependent genericity and the aspects of static and dynamic genericity that affect overload resolution, hence optimal instantiation. In the last part, section 8, we outline the experimental SuchThat compilation scheme with which we tackle the tasks that the previous sections explained.

2 Specification Requirements

The basic units of SuchThat program specifications are *structures*, that is, algebraic abstractions which can be both parameterized and further refined by additional constraints, so-called *attributes.* For a first impression of SuchThat we list three SuchThat example programs and illustrate how the Tecton and the Aldes part are integrated into a SuchThat program. We also use the examples to demonstrate the design of generic algorithms as a repeated abstraction process from non-generic algorithms, where each iteration step lifts the assumptions made in the previous one up to a more abstract level. Then we introduce operators as structure-term constructors and explain how our operator-based approach allow users to compose their own structures.

2.1 Structures and Attributes

The goal of Aldes is to syntactically support a "natural" style of algorithm development. SuchThat continues and extends this style to the declaration part of an algorithm. It tries to syntactically follow algebraic notation and conventions, slightly formalized to disambiguate parsing. Parameters, variables, constants, operations, and substructures can be declared to belong to structures and attributes, which, in turn, are based on Tecton concepts.

As the first example of a SuchThat algorithm we choose the following non-generic version of the Euclidean greatest common divisor (gcd) algorithm.

```
Algorithm:  c := IGCD(a, b)
[Integer greatest common divisor.]
      uses  Integer, Gcd-domain.
    Input:  a, b ∈ integers.
    Output: c ∈ naturals such that c = gcd(a, b).
(1) [Basis.] if b = 0 then  c:= IABS(a); return .
(2) [Recursion.] c:= IGCD(b, IREM(a, b)) ||
```

The algorithm is based on two structures, `Integer` and `Gcd-domain`. Each structure corresponds to a Tecton concept, in these cases, to a concept of the same name. SuchThat structures and Tecton concepts are linked together through the `uses`-clause which imports Tecton concepts and provides access to concept definitions. The input and output parameters of the algorithm are declared as instances of the sorts `integers` and `naturals`, defined in the Tecton concept `Integer`. This concept also defines the `0` element and the two integer functions absolute value, `abs`, and remainder, `rem`, that are used in the algorithm body (for the full definition of the `Integer` concept and all other concepts discussed in this section, see [19]). After the parameter declarations and the keyword `such that` goes the functional specification, which states the `IGCD` algorithm as an implementation of the function `gcd`, defined in the concept `Gcd-domain`. Generally speaking, the distinction between a function and its algorithms, that is, between a map, from input to output parameters, and realizations of this map, is a key distinction for the understanding of SuchThat. To emphasize the distinction, we capitalize algorithm names and keep function identifiers in Tecton's lower case. The algorithm body, finally, essentially in Aldes syntax, consists of two computation steps, the recursion step and the base case. Its subalgorithms `IABS` and `IREM` refer to the corresponding function definitions `abs` and `rem` in the `Integer` concept.

Why is the `IGCD` algorithm not generic? As mentioned in the introduction, the parameters of a generic algorithm express minimal requirements, that is, the sufficient and necessary conditions for the partial correctness of the algorithm. The `IGCD` algorithm, however, only uses two subalgorithms, `IABS` and `IREM`, and a "0" element, hence does not intrinsically depend on the integer domain. Let's abstract from the integer domain and lift the algorithm to a (more) generic version.

As a first abstraction step we abstract from the integer remainder function that underlies the `IREM` subroutine and generalize the gcd algorithm to all domains providing a remainder function, so-called "Euclidean domains." What about the second function, the integer absolute value? Proceeding in a similar way, that is, lifting it to an absolute value function and generalizing the gcd algorithm to a ring with valuation, would be wrong. The IABS algorithm here is not used for its own sake, but as a way to norm the output of the gcd computation. As is well known, a greatest common divisor is unique only up to units. For a well-defined output, it is therefore necessary to pick one gcd in a deterministic manner. In case of integers, the units are ±1, so that the integer absolute value function is suited to selecting a representative. The general case, however, requires the introduction of a set of representatives of equivalence classes, called an *ample set* [14], together with a normalizing ample function, *representative*. The correct abstraction from the integral absolute value in Tecton reads is given by the following abbreviation:

```
Abbreviation: Ample-set is Set-of-representatives
    [with Unit-equivalence as Equivalence-relation].
```

With the two abstraction steps and the introduction of ample sets the IGCD algorithm can be generalized to a gcd algorithm for Euclidean domains:

```
Algorithm:  c := EGCD(a, b)
[Euclidean domain greatest common divisor.]
     uses Euclidean-domain, Ample-set[with REP as representative].
    Input:  a, b ∈ E, an Euclidean-domain.
    Output: c ∈ A, an Ample-set of E, such that c = gcd(a, b).
(1) [Basis.]  if b = 0 then  c:= REP(a); return .
(2) [Recursion.]  c:= EGCD(b, REM(a, b)) ||
```

As a side note we remark that SuchThat slightly extends Tecton's replacement lists. As seen in the replacement list [with REP as representative], SuchThat allows for replacing function descriptors by algorithm descriptors, which are unknown in the original Tecton language.

Returning to our gcd example, we have not yet reached the level of full genericity. Our previous version is based on the existence of a Euclidean remainder function. Greatest common divisors, however, already exist in domains where only a *pseudo*-remainder function is available (for a definition see [9]). These domains, called gcd-domains, are characterized by the fact that any two elements have a greatest common divisor. Since the pseudo-remainder function is an effective function, we can show that gcd-domains constitute both necessary and sufficient prerequisites for a gcd computation. Consequently, the generic GCD algorithm is stated in terms of gcd-domains [11]. In SuchThat syntax, the algorithm reads as follows:

```
Algorithm:  c := GCD(a, b)
[Gcd-domain greatest common divisor.]
      uses  Gcd-domain, Ample-set[with REP as representative].
    Input:  a, b ∈ G, a Gcd-domain.
    Output: c ∈ A, an Ample-set of G, such that c = gcd(a, b).
(1) [Basis.]  if b = 0 then  c:= REP(a); return .
(2) [Recursion.]  c:= GCD(b, PREM(a, b)) ||
```

2.2 User Defined Operators and Structures

Finding the right conceptual abstractions, as the examples have shown, is essential in generic programming. These abstractions, however, cannot be known at language design time. SuchThat therefore has to provide means that enable users to extend the language by the structures they need.

There are two cases to distinguish. The easy case is if users want to define just a "plain" structure. Suppose, for example, the structures `Ample-set` or `Gcd-domain` had not been predefined. All that users would have to do is to define the corresponding Tecton concept and to import it in their algorithm definition with the `uses` clause. Our program examples, however, show that there is also the need for composed or parameterized structures like the structure `A`, an `Ample-set of E (G)`. Although introduced in the `uses`-clause without parameters, the `Ample-set` structure later, in the declaration of the output parameter, gets parameterized by the domains of `E` and `G`, respectively. How do users parameterize structures?

Before looking into the SuchThat solution, let's step back for a moment to appreciate the idea of user-defined parameterization. In conventional programming languages, any kind and number of parameters of a data type have to be defined with the definition of the type itself, leaving no possibility to "add" parameters afterwards. In generic programming, however, users have to have the freedom of specifying a concept at different levels of details, i.e., with a varying number of parameters. They also have to be able to instantiate a concept like `Ample-set` in different ways, with gcd-domains as well as with other algebras. Such flexibility is hard to get in an object-oriented, inheritance-based framework. It requires multiple inheritance with several direct ancestors, which quickly leads to relations of unmanageable complexity.

The SuchThat approach to user-defined parameterization is operator-based. Users use operators , in our example the operator `of`, to compose structure-terms to new structures. To keep the notation in all situations "natural,"users can introduce their own operators. To this end, we extended the Tecton `Precedence` declaration by positional attributes `prefix`, `postfix`, `infix`, and `confix` (e.g., the subscript operator) and associativity attributes `left`, `right`, and `nonassociative` and consider to be an operator whatever is subject of a precedence declaration. Operators can be grouped into precedence classes, which can be increasingly ordered and prefixed by positional and associativity attributes. The following example lists the (user-defined) declaration of five precedence classes, with the lowest precedence for the operator `implies` and the highest for the equality operator; by default, an operator is assumed to be infix and left-associative.

```
Precedence: {implies} < {or, xor} < {and}
            < prefix{not} < nonassociative{=}.
```

This example declares classical (logical) operators with the standard positional and associativity attributes. If later concept definitions require new operators or modifications of the available ones, users simply provide a new precedence declaration. At the same time, each operator has to be defined before it can be used. Our example of a parameterized structure, `A, an Ample-set of E`, therefore is syntactically correct only if accompanied by a precedence declaration that casts the symbol “of” to the operator `of` and fits the operator appropriately in the hierarchy of precedence classes.

The SuchThat expression parser is table-driven, where the tables can be extended at run time of the parser. In contrast to DFA-based lexers like Lex that match the longest leftmost expression, we use a lexer that gives the leftmost operator of the longest sequence of matches of adjacent user defined operators (and returns an error if operators that are not related by precedence are used adjacent in an expression) [13].

Emphasizing operators as much as SuchThat does almost necessarily implies the support of operator overloading. As we said in the introductory section, however, overloading represents a constituent feature of SuchThat that goes beyond operator overloading. Section 5 will address the subject separately.

3 Generic Instantiation and Implications

When we lifted the integer instance of a gcd algorithm to a generic version over gcd-domains, we ended up with three gcd algorithms: the original `IGCD` algorithm, the `EGCD` algorithm over Euclidean domains, and the one over gcd-domains, `GCD`. Though the three algorithms are of different levels of genericity, the generic `GCD` algorithm by no means supersedes the less generic ones. Quite the opposite, the more specialized algorithms are expected to take advantage of the stronger assumptions that hold for their program parameters and to realize their functional specification more efficiently than the appropriately instantiated generic algorithm could do. Given integral input parameters, for example, it is more efficient to apply the `IGCD` algorithm than the `GCD` algorithm (instantiated with integers), because `IGCD` and its subalgorithm `IABS` normalize an integral greatest common divisor more efficiently than `GCD` does with its ample function. Generic libraries, therefore, are characterized by families of algorithms that implement the same given function at different degrees of efficiency and generality. In this subsection we discuss what kind of language support is needed to correctly and best use a generic library.

Consider, as an example, a library with the three gcd algorithms, `IGCD`, `EGCD`, and `GCD`. Consider further an electrical network and its transfer function, a rational function in the frequency with symbolic coefficients from the network elements. To reduce the function to lowest terms, we have to be able to cancel out the gcd of numerator and denominator, which are polynomials in several variables over the Gaussian integers. Can we use any of the gcd algorithms in our library?

Mathematically speaking, yes. Since the domain of Gaussian integers in particular forms a Euclidean domain, the ring of multivariate polynomials over them is known to be a gcd-domain. Hence it is correct to instantiate the generic pseudo-remainder algorithm for the polynomial part and the Euclidean gcd for its coefficients. If SuchThat is to automate derivations that determine the optimal instantiation, two formalisms are needed:

1. Given two algorithms, an inference method is required that derives one algorithm as an instance of the other one, provided it is semantically correct to do so.
2. Given an algorithm and a library, a decision procedure is required that selects the most appropriate algorithm, in case there is more than one legal instantiation.

Obviously, the formalisms we are asking for cannot be based on a monomorphic type or sort system—formal parameters sorts of a generic algorithm seldom are directly matched by the sorts of their actual parameters. At the same time, established polymorphic systems are not applicable either. Neither can we use the parametric polymorphism of functional programming nor the inclusion polymorphism of object-oriented programming. Parametric polymorphism, on the one hand, with its universally quantified type variables would result in incorrect instantiations, while inclusion polymorphism, on the other hand, is not available to our system of structures and concepts which negates rigid inheritance hierarchies. What we are asking for, in other words, is a new calculus of requirements-oriented type checking.

Tecton and its set-theoretical foundation provide a basis for this calculus. Tecton lemmas allow for claiming one concept as the specialization of another one. It is possible to state the lemmas

```
Lemma: Euclidean-domain implies Gcd-domain.
Lemma: Integer implies Euclidean-domain.
```

Combining these lemmas, the `Integer` concept can be proven to be an instance of the concept `Gcd-domain`. A lemma can be read as an abbreviation of a set-theoretical assertion; the first lemma, for example, claims that every many-sorted algebra in the Euclidean-domain concept also belongs to the GCD-domain concept. An equivalent statement in terms of requirements is that all sort and function descriptor requirements of Euclidean-domain imply those of Gcd-domain. Each Tecton lemma comes along with a proof obligation. Verified off-line, Tecton lemmas provide a mathematically provable foundation of requirements-oriented type checking.

In many cases, however, semantic correctness is not a sufficient criterion to select among algorithms. The Gaussian integers, for example, can be shown to be an instance of a Euclidean domain as well as an instance of a gcd-domain. From a purely semantic standpoint, therefore, the `GCD` algorithm is a choice as good as the `EGCD`, whereas in practice, of course, we want to get the most efficient instance. One can argue that for verifications with ambiguous results, the most specialized domain should serve as tie-breaker. But this is a view too simplistic. Neither can all structures be related via lemmas—sequence containers, for example, do not specialize each other, but simply are unrelated. Nor does the computing time of an algorithm exclusively depend on the computation domain.

In practice, several factors contribute to the actual complexity of an algorithm. Algorithmic requirements, the subject of the next section, are introduced to make these factors accessible and comparable.

4 Algorithmic Requirements

Algorithmic requirements can be considered under two aspects, depending on the point of view. From the standpoint of a compiler designer, as just discussed, algorithmic requirements support the instantiation process by defining the optimal choice among several correct algorithms. From the standpoint of a library designer, the requirements serve as an additional constraint on an algorithm which each implementation has to respect. Similar to the idea of a contract in object-oriented programming, algorithmic requirements give the users of a generic algorithm a guaranteed upper bound on the computing time of each algorithm. In this section we discuss complexity measures and profiling as two ways in SuchThat to determine and specify the actual run time behavior of algorithms.

With each computational domain we associate complexity parameters. Such a parameter could be dimension for matrices or degree for polynomials, which correspond to the input size of the traditional complexity measure, but could also, more fine-grained, be characteristic for fields or degree for algebraic extensions. Complexity parameters allow expressing the complexity of a function in terms of a concept and the complexity that characterizes a concept. In that sense, complexity parameters are part of concept definitions [7]. In SuchThat therefore all concept definitions are accompanied by complexity declarations.

While the specification of complexity parameters can be done statically, profiling takes place at run time. Profiling means measuring the actual computing time of an algorithm in terms of its input complexity parameters. It also includes breaking the total time down to the time spent in the major steps and subalgorithms. Profiles are based on complexity parameters. But there are also other important factors that contribute to the computing time or even dominate it. One factor is the underlying implementation—for example, the particular representation of data type or the particular allocation strategy—which adds hidden and often very fine grained computing time parameters. Another factor is hardware, that is, the instruction set of a processor or the memory hierarchy of an architecture. Especially in combination with an optimizing compiler, the time consumption of an algorithm can vary drastically. As precise as profiling is, however, it lacks the theoretical completeness of the analytic approach that characterizes complexity parameters. For SuchThat, we therefore decided to equally support profiling and complexity parameters.

Algorithmic requirements have to become part of an algorithm declaration and the requirements-oriented check. As part of an automated procedure, then, performance assertions can be checked similar to the way types are checked, with benefits similar to ones from type checking. It is possible to catch calls to subroutines that illegitimately increase their callers costs, to reject an instantiation request for violating performance requirements, or to detect simple user errors, like typographical errors, that result in inefficient instantiations. Algorithmic requirements are integrated into SuchThat through complexity classes we define for input and output parameters and the maximum computing times of algorithms.

Algorithmic requirements are a problem, both conceptually and with respect to their implementation, for which no final solution yet exists. In the last section we will present how we experimentally approach an implementation of algorithmic requirements for SuchThat. Before that, however, we have to understand overloading and overload resolution.

5 Overloading

When we discussed the instantiation of generic algorithms and the problem of the best instance, we assumed the requirements-oriented check to take a global view on the whole set of algorithm descriptors of a library. We also explained how semantic and performance requirements help to rule out inferior algorithms—but how does the compiler know in the first place what the set of potential candidates is? The answer to this is overloading. What we called `IGCD`, `EGCD`, and `GCD`, in reality share one identifier, `GCD`. Denoting different algorithms with the same identifier is a way to shift the choice of the best instance from the user to the compilation system. It determines, at the same time, which set of algorithm identifiers a compiler has to consider for the instantiation task. In this section we deal with the semantics of overloading. The process of overload resolution will be addressed in the following two sections.

Most languages allow overloading in a very restricted form, as the overloading of arithmetical operators or identifiers of basic procedures such as print routines. Languages that support operators, e.g. C++ , Ada, or Eiffel, typically allow each operator to be overloaded, although neither in Ada nor in C++ can positional and associativity attributes be changed. Regardless of the extent, however, to which traditional languages support overloading, it is considered as purely syntactic sugar. In this spirit, Strachey coined the term *ad-hoc polymorphism*, a good description for examples of overloading like the C++ `operator <<`, which combines the two unrelated meanings of a boolean shift operator and an I/O operator.

In SuchThat and generic programming in general, overloading has a semantic dimension. The distinction between functions and algorithms, as we have emphasized several times, is motivated by the fact that there is a one-to-many relation between a function and its algorithms, i.e., that there can be more than one algorithm implementing a function. As the flip-side of this one-to-many relation, consequently, each function defines the semantics of all algorithms realizing it. When these algorithms share their identifier, therefore, they syntactically indicate their common functional specification, which in turn semantically justifies the name *algorithmic overloading.*

6 Dependent Genericity

If overloading is a way to pass the control over the instantiation process on to the compiler, then overload resolution is the way to perform this instantiation. In the following two sections we elaborate on the specifics of overload resolution in the context of generic programming.

The general difference between the situation in generic programming and in traditional languages is that overload resolution in our context is not fully determined by the types (in Tecton terminology: sorts) of the parameters involved. In traditional overload resolution, the type information, possibly weighted as in the *implicit conversion sequences* of C++, is expected to allow for a disambiguous resolution. If more than one signature is left after the semantic analysis, an identifier is qualified as ambiguous and the whole expression rejected as unresolvable. In contrast, as we have seen in section 3, a two-level procedure applies in SuchThat, which allows the semantic analysis to return sets and consults algorithm requirements for disambiguation.

The semantic analysis itself is complicated by the kind of structures involved. In generic programming, many structures depend on parameters which are not structures again but special elements thereof. Matrices depend on their dimension, polynomials on the number of variables, fields on their characteristic, algebraic extensions on their minimal polynomial, to name a few. In most cases, these structures appear parameterized, which increases the number of their element parameters. More importantly, however, the behavior of a structure can vary, depending on the values of their element parameters. A field with finite characteristic, for example, has other mathematical properties than a field with characteristic 0, and a univariate polynomial allows for other operations than a multivariate polynomial—a phenomenon which in type theory is known as a dependent type. There are a few languages that implement dependent types [2,22].

Dependent types affect overload resolution in two ways. On the one hand, they ultimately determine the result of the resolution. If the value parameter of a program variable controls its type, then it controls as well, say, which gcd algorithm is the most appropriate. On the other hand, dependent types modify the process of overload resolution itself. Traditionally, overload resolution is part of the static analysis of a program. Value parameters of dependent types, however, typically are known at run time only. To keep overload resolution efficient, the SuchThat translator partitions the instantiation of generic constructs into structural parameterizations that are resolvable at compile time and dependent genericity where the resolution is delayed until run time.

7 Static versus Dynamic Genericity

At this point, we know various stages of overload resolution: a semantic and a performance check, a static and a dynamic analysis. We also have seen, that the semantic check takes place partly at compile time, partly at run time. The same holds true for the performance check, as we will see now.

Eventually, the family of algorithms that implement a function has to be ordered according to their algorithmic requirements. Such order, however, is more complex than the simple classification as good, better, optimal. Frequently, it is an interval of complexity parameters, where one algorithm outperforms another one (*piece-wise efficiency*). Resolving algorithmic genericity therefore means selecting the best algorithm for each interval. This best algorithm, in turn, depends on several parameters, not all of which are known at compile time. For instance the value parameters we dealt with in the previous section most typically also serve as complexity parameters. Piecewise efficiency therefore poses two problems: how to find trade off points and how to make the decision to switch, at run time, to a better candidate algorithm.

Object-oriented methodology suggests run-time polymorphism as solution. In C++, for example, one would introduce an abstract base class with derived classes, where the piecewise efficient algorithm implements the virtual function of the base class. The resulting library is modular and extensible—but not efficient enough. Virtual function calls are indirect, do not participate in inter-procedural analysis and therefore are not subject of optimizations that otherwise apply. Basing overload resolution, hence the entire instantiation process of SuchThat, on object-oriented polymorphism would be unacceptably slow.

8 The SuchThat Compilation Model

Algorithmic requirements, static and dynamic genericity, and the tasks for the compilation process we elaborated on ultimately require a new compilation model. Inspired by Fernandez, Franz, and Kistler [3,4,8], we decided on the following experimental compilation scheme that breaks down into seven steps:

1. The front end combines concept and algorithm descriptions in a portable way.
2. The static semantics is checked in the front end and maintained in attributed syntax trees of the algorithms; the syntax trees with semantic dictionaries form the output in a special encoding suitable to fast code generation.
3. The encoded syntax trees and dictionaries are kept in the generic library.
4. Static instantiation and code generation are done at link time.
5. The encoded syntax trees and dictionaries are passed on to the run time system.
6. The dynamic instantiation of algorithms and program optimization, based on profiling and actual complexity parameters, takes place in separate threads at run time.
7. The SuchThat run-time system also provides heap allocation and garbage collection.

The most innovative decision is to shift the classical compilation phases: code generation takes place at link time, code optimization at run time, so that static genericity gets resolved by the SuchThat linker and dynamic genericity by the SuchThat run-time system.

There are several benefits of this approach. The most important one is that algorithm instantiation is based on the actual sizes of complexity parameters. Other advantages include the portability of the front end and the library and, as demonstrated in [4], significantly better opportunities for run-time optimizations. Admittedly, this approach moves to run time some of the processing that traditional approaches have been assigned to compile time. The expectation, however, is, that the selection of the most appropriate algorithm brings orders of magnitude of speed up compared with conventional compiler optimizations.

9 Conclusion

We have introduced the programming language SuchThat as a tool to support generic programming. The fundamental design decision of SuchThat is to combine the powerful, yet simple and elegant specification language Tecton with the equally simple imperative language Aldes to describe algorithms. We showed that this combination on the one hand allows for specifications that are both more precise and more abstract than in traditional languages. It implies on the other hand that each efficiently executable program is an instance of an abstract algorithm that is based on formal requirements—a novel dimension in executable formal specification languages. Furthermore, we have argued that overloaded operations are the generic sublanguage to address parameterized subdomains consistently, based exclusively on their requirements and inherent laws, as in algebra.

We have accumulated to date experience in concept and algorithm descriptions, separately, supporting our design decisions. We want to extend both by upgrading a large library of algebraic algorithms. At the same time, we are experimenting with a combined implementation which supports both static checking of attributed types and run-time dependencies of structural and algorithmic requirements.

Acknowledgments

We thank our students involved in the project, in one part or another: Steve Cerniglia, Tobias Döhler, Holger Gast, Albrecht Haug, Peter Hrenka, Uwe Kreppel, Wolfgang Maier, Oriel Maute, Gor Nishanov, Volker Simonis, Quincy Stokes, Christoph Schwarzweller, and Roland Weiss.

References

1. George E. Collins and Rüdiger Loos. Specification and index of SAC-2 algorithms. Technical Report WSI-90-4, Wilhelm-Schickard-Institut für Informatik, 1990.
2. R.L. Constable, S.F. Allen, H.M Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Developement System.* Graduate Texts in Mathematics. Prentice Hall, 1986.

3. Mary Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1995.
4. Michael Franz. *Code-Generation On-the-Fly: A Key to Portable Software, Doctoral Dissertation No. 10497, ETH Zurich.* Verlag der Fachvereine, Zürich, 1994.
5. Holger Gast. With Scheme from SUCHTHAT to C++. Studienarbeit, RPI, Universität Tübingen, 1998.
6. Deepak Kapur and David Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report 92-20, RPI, Troy, 1992.
7. Deepak Kapur, David Musser, and Alexander Stepanov. Operators and algebraic structures. In *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire. ACM, 1981.
8. Thomas Kistler and Michael Franz. A tree-based alternative to java byte-codes. *International Journal of Parallel Programming*, 27(1):21–34, 1999.
9. Donald E. Knuth. *The Art of Programming*, volume 2. Addison-Wesley, 1997.
10. Rüdiger Loos. Algebraic descriptions as programs. *SIGSAM Bulletin*, 23:16–24, 1972.
11. Rüdiger Loos. Generalized polynomial remainder sequences. In Bruno Buchberger, George E. Collins, and Rüdiger Loos, editors, *Computer Algebra. Symbolic and Algebraic Computation*, pages 115–137. Springer-Verlag, 1982.
12. Rüdiger Loos and George Collins. Revised report on the algorithm description language ALDES. Technical Report WSI-92-14, Fakultät für Informatik, Universität Tübingen, 1992.
13. Oriel Maute. Lexikalische analyse von operatoren bei benutzerdefinierter operatormenge. Zusammenfassung, Studienarbeit, Universität Tübingen, Feb 1998.
14. David Musser. *Algorithms for Polynomial Factorization.* PhD thesis, University of Wisconsin, Madison, 1971.
15. David Musser and Alexander Stepanov. Algorithm-oriented generic libraries. *Software-practice and experience*, 27(7):623–642, Jul 1994.
16. David R. Musser. The Tecton concept description language. `http://www.cs.rpi.edu/~musser/gp/tecton/tecton1.ps.gz`, July 1998.
17. David R. Musser. Tecton description of STL container and iterator concepts. `http://www.cs.rpi.edu/~musser/gp/tecton/container.ps.gz`, August 1998.
18. David R. Musser, Sibylle Schupp, and Rüdiger Loos. Requirements-oriented programming. In this volume.
19. David R. Musser, Sibylle Schupp, Christoph Schwarzweller, and Rüdiger Loos. Tecton Concept Library. Technical Report WSI-99-2, Fakultät für Informatik, Universität Tübingen, January 1999.
20. Sibylle Schupp. *Generic Programming*—SUCHTHAT *One Can Build an Algebraic Library.* PhD thesis, University Tübingen, 1996.
21. Steffen Seitz. Compiling rewrite rules into algorithms. Master's thesis, Diplomarbeit, Universität Karlsruhe, 1985.
22. Stephen M. Watt, Peter A. Broadbery, Samual S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, and Robert S. Sutor. A first report on the $A^{\#}$ compiler. In *Proc. International Symposium on Symbolic and Algebraic Computation (ISSAC '94)*, pages 25–31, 1994.
23. Roland Weiss. SCM2CPP—an intelligent and configurable backend for SUCHTHAT. Diplomarbeit, Universität Tübingen, 1997.

Software Development in PVS Using Generic Development Steps[*]

Axel Dold

Universität Ulm
Fakultät für Informatik
D-89069 Ulm, Germany
dold@ki.informatik.uni-ulm.de

Abstract. This paper is concerned with a mechanized formal treatment of the transformational software development process in a unified framework. We utilize the PVS system to formally represent, verify and correctly apply generic software development steps and development methods from different existing transformational approaches. We illustrate our approach by representing the well-known divide-and-conquer paradigm, two optimization steps, and by formally deriving a mergesort program.

Keywords: formal verification, mechanized theorem proving, generic development steps, transformational software development

1 Introduction

The transformational approach to software development is widely accepted in modern software engineering. The idea is to start from an abstract formal requirement specification and to apply a series of correctness-preserving software development steps to finally obtain an executable and efficient program.

This paper is concerned with a fully formally mechanized treatment of the transformational development process in a unified framework. The specification system PVS is utilized as a vehicle to formalize, verify, and apply well-known generic software development steps and development methods of different kind and complexity. Applying formal methods to the transformational process is important since it greatly increases the confidence in transformations and their application. Experience has shown the benefits of formal approaches, since we have found errors in published (paper-and-pencil) proofs of transformations and their application.

Development steps from different transformational approaches (PROSPECTRA [4], KIDS [14], CIP [10], BM CALCULUS [2]) are integrated into our framework. They can be grouped into four categories: the first group of development steps consists of problem solving strategies transforming a (non-constructive) requirement specification into a functional specification. Among them, one can find

[*] This work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG) project *Verifix*.

"algorithm theories" [15] used in the KIDS system such as *generate-and-test*, *global search*, *divide-and-conquer*, a general scheme for backtracking algorithms [1] or simpler transformations such as *split-of-postcondition* [4]. In the second group there are transformations which modify and optimize functional specifications [10,2]. The third group consists of a library of implementations of standard data structures and operations (for example, different implementations for finite sets) while the fourth group is concerned with the translation of functional specifications into imperative programs.

In this paper we illustrate our approach by a formal representation of the general programming paradigm divide-and-conquer using a hierarchy of generic PVS theories, and by two selected optimization steps. These steps are then applied to derive a mergesort program from a formal requirement specification.

All generic development steps are represented within parameterized PVS theories which specify the semantic requirements on the parameters by means of assumptions and define the result of the transformation. Based on the semantic requirements on the parameters, correctness of the generic transformation step can be proved. Application of such a generic step to a given problem is then carried out by providing a concrete instantiation for the parameters and verifying that it satisfies the requirements. This paper extends [3] in several respects: first, the divide-and-conquer scheme is generalized to include n-ary decomposition and composition operators for an arbitrary n. Second, a hierarchy of divide-and-conquer theories is presented which enables a much more convenient application to specific problems. Third, the development of a sorting algorithm by applying several development steps is outlined, and fourth, the formal representation of two selected optimization steps is described.

We proceed as follows: the next section gives a brief introduction to PVS and presents the general form of requirement specifications. Sect. 3 is concerned with the representation of the divide-and-conquer paradigm while in Sect. 4 a formal treatment of two optimization steps is discussed. These transformation patterns are then applied in Sect. 5 to derive a mergesort program. Sect. 6 concludes. All PVS theories and proof scripts are available from the author.

2 Preliminaries

First, a brief introduction to PVS, the formal framework we are working in, is provided, then the general form of requirement specification is presented.

2.1 A Brief Introduction to PVS

The PVS system [9,8] combines an expressive specification language with an interactive proof checker that has a reasonable amount of theorem proving capabilities. The PVS specification language builds on classical typed higher-order logic with the usual base types, `bool`, `nat`, among others, the product type constructor `[A,B]` and the function type constructor `[A -> B]`. The type system of PVS is augmented with *dependent types* and *abstract data types*. A distinctive feature of the PVS specification language are *predicate subtypes*: the subtype {`x:A | P(x)`} consists of exactly those elements of type `A` satisfying predicate `P`. Predicate subtypes are used, for instance, for explicitly constraining the domains and ranges of operations in a specification and to define partial functions. Predicates in PVS are elements of type `bool`, and `pred[A]` is a notational convenience for the function type `[A` $\rightarrow$ `bool]`. Sets are identified with their characteristic predicates, and thus the expressions `pred[A]` and `set[A]` are interchangeable. For a predicate `P` of type `pred[A]`, the notation `(P)` is just an abbreviation for the predicate subtype {`x:A | P(x)`}. In general, type-checking with predicate subtypes is undecidable; the type-checker generates proof obligations, so-called *type correctness conditions* (TCCs) in cases where type conflicts cannot immediately be resolved. A PVS expression is not considered to be fully type-checked unless all generated TCCs have been proved. PVS only allows total functions, hence it must be ensured that all (recursive) functions terminate. For this purpose, a well-founded ordering or a *measure function* is used. The definition of a recursive function `f` generates a TCC which states that the measure function applied to the recursive arguments decreases with respect to a well-founded ordering. A built-in *prelude* and loadable *libraries* provide standard specifications and proved facts for a large number of theories. Specifications are realized as possibly parameterized PVS theories and theory parameters can be constrained by means of *assumptions*. When instantiating a parameterized theory, TCC's are automatically generated according to the assumptions.

Proofs in PVS are presented in a sequent calculus. There exists a large number of atomic commands (for quantifier instantiation, automatic conditional rewriting, induction etc.), and in addition PVS has an LCF-like strategy language for combining inference steps into more powerful proof strategies. The built-in strategy `GRIND`, for example, combines rewriting with propositional simplification using BDDs and decision procedures.

2.2 Requirement Specification

Initial requirement specifications are given as quadruples

$$P = (D : TYPE, R : TYPE, I : pred[D], O : [D, R \rightarrow bool])$$

where D and R denote the problem domain and range, respectively, I is a predicate on D describing admissible inputs, and O is the input/output relation of the problem. A solution to such a problem is either

1. a function $f : D \rightarrow R$ such that $\forall x : (I).O(x, f(x))$
2. or a function $F : D \rightarrow set[R]$ such that $\forall x : (I).F(x) = \{z : R \mid O(x, z)\}$.

In the first case, function f calculates *one* solution to the problem, while in the latter case, function F produces the set of all solutions to the problem.

3 The Divide and Conquer Paradigm

The well-known algorithmic paradigm *divide-and-conquer* [13,15] is based on the principle of solving primitive problem instances directly, and large problem instances by decomposing them into "smaller" instances, solving them independently and composing the resulting solutions. Typically, the smaller problem instances are solved in the same way as the input problem which results in a recursive algorithm. However, some of the subproblem instances could be solved in a different way. Fig. 1 illustrates the paradigm.

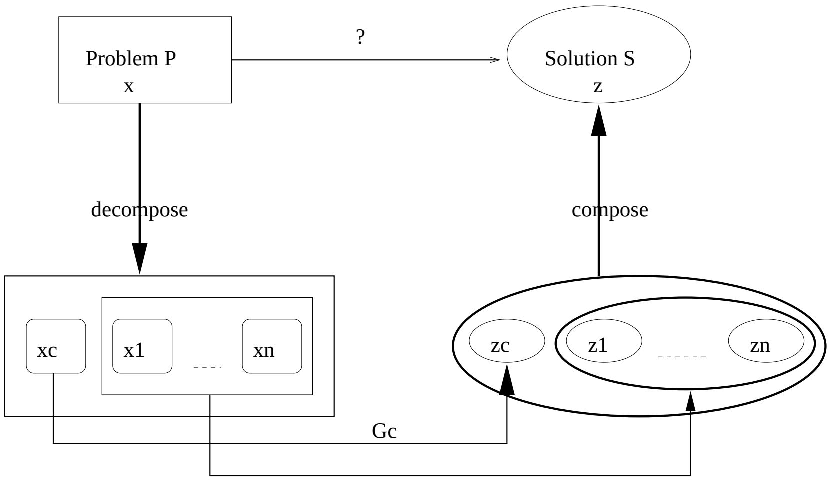


Fig. 1. Divide-and-Conquer Paradigm

3.1 The General Scheme

Our formalization follows [13]. However, our scheme is more general allowing the decomposition operator to produce an arbitrary but fixed number of subproblems.

Let (D, R, I, O) denote the problem specification, and (D_c, R_c, I_c, O_c) denote a "component problem" with a solution G_c, i.e. $\forall x_c : (I_c).O_c(x_c, G_c(x_c))$ Our most general theory uses a decomposition operator which decomposes a problem instance x into an arbitrary but fixed number N $(N \geq 1)$ of subproblem instances $x_1, \ldots, x_n$ and a component problem instance x_c. The component problem instance x_c is solved directly by G_c producing an output z_c, and the problem instances $x_1, \ldots, x_n$ are solved recursively producing a vector of solutions $z_1, \ldots, z_n$ which are then composed by the composition operator to form a solution to the input problem x. To ensure termination of this process a measure function on problem domain D is required. This completes the parameter list of the general scheme.[1]

```
% parameter list of general divide-and-conquer theory                      1
D:TYPE, R:TYPE, I:pred[D], O:[D,R -> bool],       % problem spec
Dc:TYPE, Rc:TYPE, Ic:pred[Dc], Oc:[Dc,Rc -> bool],% component problem
Gc         : [(Ic) -> Rc],                  % solution of (Dc,Rc,Ic,Oc)
N          : posnat,                        % number of subproblems
decompose  : [D -> [Dc, [below(N) -> D]]],  % decomposition operator
dir_solve  : [D -> R],                      % primitive solution
compose    : [Rc, [below(N) -> R] -> R],    % composition operator
primitive? : pred[D],                       % set of primitive problems
mes        : [D -> nat]                     % termination measure on D
```

Five assumptions specify the semantic requirements:

```
a1: ASSUMPTION I(x)  ∧  ¬ primitive?(x)                                    2
                 ⇒ ∀n. mes(decompose(x).2(n)) < mes(x)
a2: ASSUMPTION Ic(xc) ⇒ Oc(xc, Gc(xc))
a3: ASSUMPTION I(x)  ∧  primitive?(x) ⇒ O(x, dir_solve(x))
a4: ASSUMPTION I(x)  ∧  ¬ primitive?(x)  ∧  Oc(decompose(x).1, zc)
  ⇒ ∀v.(∀n. O(decompose(x).2(n), v(n))) ⇒ O(x, compose(zc,v))
a5: ASSUMPTION I(x)  ∧  ¬ primitive?(x)
                ⇒ Ic(decompose(x).1)  ∧  ∀n. I(decompose(x).2(n))
```

They state that

1. all subproblems created by the decomposition operator are smaller than the original problem with respect to the measure `mes`.
2. function Gc solves the component problem (D_c, R_c, I_c, O_c).
3. a primitive problem can be solved by `dir_solve`.
4. solutions to the subproblems $z_1, \ldots, z_n$ and a solution z_c to the component problem can be composed to build a solution to the input problem.
5. all subproblems generated by the decomposition operator satisfy the input conditions I of (D, R, I, O) and I_c of (D_c, R_c, I_c, O_c).

Based on these parameters, the generic divide-and-conquer algorithm can be defined by function `f_dc` in $\boxed{3}$:

```
f_dc(x:(I)) : RECURSIVE R =                                                3
 IF primitive?(x) THEN dir_solve(x)
 ELSE LET (xc,dcomp)        = decompose(x),
           subsolutions     = λn. f_dc(dcomp(n))
      IN compose(Gc(xc), subsolutions)
 ENDIF
  MEASURE mes(x)
```

One has to prove that `f_dc` is a correct solution of the problem (D, R, I, O):

[1] To increase the readability of PVS specifications the syntax has been liberally modified by replacing some ASCII codings with a more familiar mathematical notation.

Theorem 1 (Correctness of Divide-and-Conquer).

```
dc_correctness: THEOREM ∀x:(I). O(x, f_dc(x))
```

We prove the theorem by measure-induction using measure function `mes` with the usual $<$ ordering on `nat`:

```
% measure induction principle
% T: TYPE, M: TYPE, m: [T→M], <: (well_founded?[M])
measure_induction: LEMMA ∀p:pred[T]:
 (∀x. (∀y. m(y) < m(x) ⇒  p(y)) ⇒ p(x)) ⇒ (∀x. p(x))
```

After the built-in measure-induction strategy has been applied the PVS proof state looks as follows: [2]

```
dc_correctness:

{-1}   ∀y:(I). mes(y) < mes(x!1) ⇒ O(y, f_dc(y))
  |-------
{1}    O(x!1, f_dc(x!1))
```

The proof can be finished by expanding the definition of `f_dc`, case-analysis on `primitive?`, and application of the theory assumptions $\boxed{2}$.

3.2 A Hierarchy of Divide-and-Conquer Theories

In order to have more adequate support for application of the divide-and-conquer (DC) paradigm to specific problems we have developed a hierarchy of generic DC theories where the top most theory models the most general DC scheme presented above and lower theories in this hierarchy are specializations of the general scheme, see Fig. 2. Each child in this hierarchy is a partial instantiation or specialization of its parent theory. This hierarchy permits choice of the most specific theory in order to solve a given problem.

In theory `DC_ID` the component problem is given by $(D_c, D_c, TRUE, = [D_c])$ where $TRUE$ denotes the constant true predicate, and $= [D_c]$ denotes the equality relation on D_c, i.e. G_c is the identity on D_c. Theory `DC_BIN`, a refinement of `DC_ID`, uses a binary decomposition and composition operator where the type D_c of the (trivial) component problem is identified with domain D of problem (D, R, I, O). Theory `DC_1` specifies a binary decomposition and composition operator where the type D_c of the (trivial) component problem is in general different from D.

[2] The prover maintains a proof tree. The goal is to construct a proof tree which is complete, in the sense that all of the leaves are recognized as true. Each proof goal is a sequent consisting of a sequence of antecedent formulas (indicated by negative numbers) and consequent formulas (indicated by positive numbers). The intuitive meaning of such a goal is that the conjunction of the antecedents implies the disjunction of the consequents. Expressions of the form `x!i` denote constants introduced for universal-strength quantifiers.

The theories on the lowest level in the hierarchy define specific composition and decomposition operators for lists. Theory DEC_LST1 decomposes a (non-empty) list into its head and tail, theory DEC_LST2 splits a list into roughly two equal parts, while CMP_LST1 uses "cons" as its composition operator, and finally, theory CMP_LST2 composes lists by concatenation. In a similar way, the hierarchy can be extended for data types such as finite sets, balanced binary trees etc.

To illustrate a specific parent-child pair of the hierarchy theories DEC_LST2 and DC_BIN are explained in some more details. The parameters of DC_BIN are as follows:

```
% formal parameters of theory DC_BIN                                    4
D:TYPE, R:TYPE, I:pred[D], O:[D,R -> bool], % problem spec
decompose   : [D -> [D,D]],                 % decomposition operator
dir_solve   : [D -> R],                     % primitive solutions
compose     : [R,R -> R],                   % composition operator
primitive? : pred[D],                       % set of primitive problems
mes         : [D -> nat]                    % termination measure on D
```

There are four assumptions on the parameters. They are similar to assumptions a1,a3,a4,a5 of the general scheme $\boxed{2}$ specialized to binary decomposition and composition operators:

```
% assumptions for DC_BIN                                                5
a1: ASSUMPTION I(x) ∧ ¬ primitive?(x) ⇒ LET (x1,x2) = decompose(x) IN
                ⇒ mes(x1) < mes(x) ∧ mes(x2) < mes(x)
a3: ASSUMPTION I(x) ∧ primitive?(x) ⇒ O(x, dir_solve(x))
a4: ASSUMPTION I(x) ∧ ¬ primitive?(x) ⇒ LET (x1,x2) = decompose(x) IN
                O(x1,z1) ∧ O(x2,z2) ⇒ O(x, compose(z1,z2))
a5: ASSUMPTION I(x) ∧ ¬ primitive?(x)
                ⇒ I(decompose(x).1) ∧ I(decompose(x).2)
```

Note that there is no assumption for the component problem since this has already been specialized in theory DC_ID.

Consider now theory DEC_LST2. Its formal parameter list is as follows:

```
% formal parameters of theory DEC_LST2                                  6
T           : TYPE,                        % list element type
R           : TYPE,                        % problem range
I           : pred[list[T]],               % admissible inputs
O           : [list[T],R -> bool],         % input/output relation
dir_solve   : [list[T] -> R],              % primitive solutions
compose     : [R,R -> R]                   % binary composition operator
```

The problem domain is fixed by the type list[T], and the decomposition operator is defined by listsplit in $\boxed{7}$. Here, first_n returns the first n elements of a list, and cut_n cuts n elements from a list. Note that the primitive? predicate is determined by the choice of the decomposition operator: primitive instances are lists containing at most one element; the measure is given by the length of the list.

```
% decomposition operator and primitive predicate                          7

decompose(x:list[T]): [list[T], list[T]] =
 IF length(x) < 2 THEN (x,x)    % dummy value
 ELSE listsplit(x) ENDIF

listsplit(l:{l1:list[T] | length(l1) >= 2}): [list[T], list[T]] =
 (first_n(l, div2(length(l))), cut_n(l, div2(length(l))))

primitive?(x:list[T]): bool = (length(x) < 2)
```

The assumptions on the parameters are given by `a3,a4,a5` in $\boxed{5}$ using the specific operators above $\boxed{7}$. A specialization of theory `DC_BIN` is then obtained by importing the instantiated theory:

```
 IMPORTING
  DC_BIN[list[T],R,I,O,decompose,dir_solve,compose,primitive?,length]
```

TCCs are generated according to the assumptions $\boxed{5}$. The only non-trivial TCC is to prove `a1`, stating that the length of the lists produced by `listsplit` is smaller than the length of the input list. This requires additional lemmas for list functions `first_n` and `cut_n`.

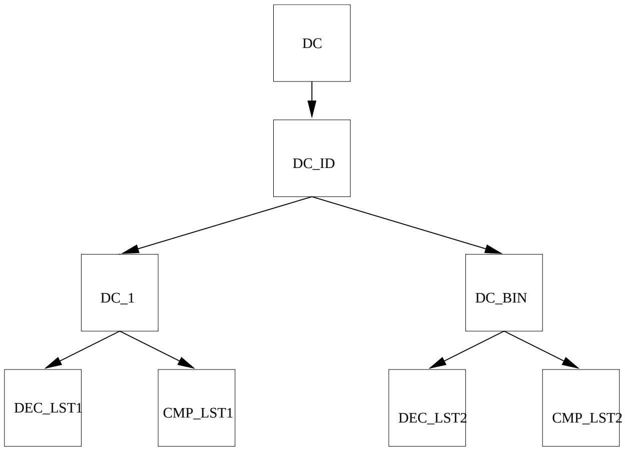


Fig. 2. A Hierarchy of Divide-and-Conquer Theories

In order to solve problems using the DC hierarchy the following general derivation strategy is utilized:

1. choose an appropriate DC theory from the hierarchy depending on the structure of the problem and/or the data types used in the specification

2. provide concrete instantiations for the formal theory parameters which are easy to find (i.e. determined by the problem)
3. set up new specifications for the parameters which are not obvious
4. solve the new problems in a similar way.

4 Optimization Steps

This section deals with the formal treatment of two examples of generic optimization steps: transformation of a linear recursive function into an equivalent tail-recursive one, and the introduction of a table to store and restore computed values.

4.1 Accumulation Strategy

Given a linear recursive function

$$f(x) = \text{ IF } B(x) \text{ THEN } H(x) \text{ ELSE } p(K(x), f(K_1(x)))$$

the goal is to transform it into an equivalent tail-recursive function. In a tail-recursive function recursive calls appear on the top level of expressions, i.e. they have the form $f(K(x))$ and do not occur within an expression. Wand [16] presents a transformation where the non-tail-recursive part of f (expression p) is captured as an extra continuation argument. This results in a tail-recursive function which is further transformed by replacing the continuation by an accumulator. A PVS formalization of this transformation has been carried out by Shankar [12]. Partsch [10] presents a transformation rule which combines both steps and directly converts the linear recursive function into a tail-recursive one. In the following a formal treatment of this rule is given. The basic idea of this transformation is to add a parameter that accumulates the function result. It is required that operation p has a neutral element e and is associative. The respective PVS theory takes 10 parameters:

```
% parameters of accumulation strategy                              8
D  : TYPE,          % domain of f
R  : TYPE+,         % range of f
B  : pred[D],       % termination condition
H  : [D -> R],      % base case solution
p  : [R,R -> R],    % associative operator
K  : [D -> R],      % non-recursive argument modifier
K1 : [D -> D],      % recursive argument modifier
e  : R,             % neutral element w.r.t. p
m  : [D -> nat],    % measure
f  : [D -> R]       % linear function to be transformed
```

There are four assumptions on the theory parameters. The first one asserts the associativity of `p`, the second one states the neutrality of `e` with respect to `p`. The third one ensures termination of `f`, while the fourth one constrains `f` to have a linear form. A tail-recursive variant is then given by function `f1`:

```
f1(r:R,x:D) : RECURSIVE R =                                        9
 IF B(x) THEN p(r, H(x)) ELSE f1(p(r, K(x)), K1(x)) ENDIF
  MEASURE m(x)
```

Theorem 2 (Correctness of Accumulation Strategy).

```
f_trans(x:D): {r:R | r = f(x)} = f1(e,x)
```

Note that the correctness is expressed by type constraints in the definition of `f_trans`. The proof is by establishing an invariant `invar` ($\boxed{10}$) which itself is accomplished by a straightforward measure-induction proof.

```
invar: LEMMA f1(r,x) = p(r,f(x))                                   10
```

4.2 Memoization

In a recursive function's computation inefficiencies often arise due to multiple evaluations of identical calls. This is especially the case if the recursion is nested or cascaded. The idea of the transformation described in this subsection is to store computed values in a table and to remember that this computation has been carried out. If the result of this computation is needed it is looked up in the table. We present a formal representation of a generalization of the transformation called *memoization* [10]. The transformation is applicable to recursive functions which have the form

$$f(x) = \text{IF } B(x) \text{ THEN } H(x) \text{ ELSE } E(x, f(K_0(x)), f(K_1(x)), \ldots, f(K_n(x)))$$

The respective PVS theory is parameterized as follows:

```
% parameters of memoization                                        11
D : TYPE,                          % domain of f
R : TYPE,                          % range of f
B : pred[D],                       % termination condition
H : [D -> R],                      % base case solution
N : nat,                           % N + 1 recursive calls
E : [D, [upto(N) -> R] -> R],      % expr. containing the recursive calls
K : [upto(N) -> [D -> D]],         % argument modifiers K0,K1, ..., KN
m : [D -> nat],                    % termination measure
f : [D -> R]                       % function to be transformed
```

There are two assumptions on the parameters: the first one constrains the input function f to have the specific form while the second one assumes that the arguments for the recursive calls decrease with respect to the measure `m`. The result of the transformation is given by function `f1` in $\boxed{12}$.

```
f1(x:D)(a:(invar?)): RECURSIVE {a1:(invar?) | P_inv(x,a,a1)} =     12
 IF isdef(a,x) THEN a ELSIF B(x) THEN add(a, x, H(x))
 ELSE LET final_map = concat(λn. f1(K(n)(x)))(N)(a) IN
  add(final_map, x, E(x, λn. final_map^(K(n)(x))))
 ENDIF
  MEASURE m(x)
```

f1 has an additional argument of type map (association list) (such that an invariant $\boxed{13}$ is preserved) which denotes the table in which the computed values are stored. The type map associates elements of an index set with values: empty denotes the empty map and add(m,i,e) adds a new association to m: value e is associated with index i. There are two functions isdef(m,i) for testing whether the map has a defined value for a given index, and ^ for retrieving a value associated with a given index (if there is a defined value). Function f1 is initially called with the empty map. If a result for some argument has already been computed then the value is looked up in the table otherwise it is calculated and then inserted into the table.

```
invar?(a:map): bool = ∀y:D. isdef(a,y) ⇒ a^y = f(y)                    13
P_inv(x:D,a1,a2:map): bool =
 isdef(a2,x) ∧ ∀y:D. isdef(a1,y) ⇒ (isdef(a2,y) ∧ a1^y = a2^y)
```

The invariant invar?(a) in $\boxed{13}$ states that if there exists an entry in a for some y then the value associated with y equals to f(y). P_inv(x,a1,a2) states that map a2 has a defined value for x and that a2 preserves all associations of a1. Function concat realizes the function composition $f_1(K_N(x)) \circ \cdots \circ f_1(K_0(x))$.

Theorem 3 (Correctness of Memoization).

```
correctness : THEOREM  ∀x:D. f(x) = f1(x)(empty)^x
```

Retrieving the value from the map f1(x)(empty) at index x equals to f(x). This theorem is trivial to prove using the type constraints of the range of f1 since the invariant directly states the correctness. However, some non-trivial TCC's are generated when type-checking this theory, since it has to be ensured that the invariant is preserved during computation. To prove this, additional lemmas have to be established which state that concat preserves the invariant and that the concatenation of maps can be split. Furthermore, it must be proved that there exists table entries for all $K_i(x)$, for $i \in \{0 \ldots N\}$ in the final map.

5 Derivation of a Sorting Algorithm

In the following the derivation strategy from Sect. 3.2 is applied to derive the well-known mergesort algorithm using different divide-and-conquer schemes. The problem specification of sorting a list of elements of some type T with respect to some total order $\leq$ can be stated as follows:

```
% specification of the sorting problem                                  14
% T:TYPE+; <=: (total_order?[T]); A,B: VAR list[T]
P_sort := (D:= list[T], R:= list[T], I:= TRUE, O:= λA,B. sorted?(A,B))

sorted?(A,B): bool = ordered?(B) ∧ perm?(A,B)
perm?(A,B): bool = ∀t: count(t,A) = count(t,B)
```

ordered? holds if all elements of a list are in nondecreasing order. Predicate perm?(A,B) holds if list B is a permutation of list A: the number of occurrences of any element in A and B agree.

In order to solve this problem an appropriate DC theory from the library has to be chosen. Since we decide to derive a mergesort algorithm we choose theory DEC_LST2 where the decomposition operator splits a list into roughly two equal length parts (see $\boxed{6}$ for the parameter list). Parameters T,R,I and O are determined by the sorting problem; dir_solve simply returns the input list since a list with at most one element is trivially sorted. The only creative work to do is to find an appropriate composition operator satisfying the assumption a4. Looking at this assumption one recognizes that the composition operator must "merge" two already sorted parts. Rather than inventing a merge function and proving its correctness, it is derived by first setting up a new specification and then selecting and instantiating another appropriate DC theory from the hierarchy:

```
% requirement specification of merge                                  15
P_merge :=
 (D1:= [list[T],list[T]],
  R1:= list[T],
  I1:= λ (d:D1). ordered?(d.1) ∧ ordered?(d.2),
  O1:= λ (d:D1,C:list[T]). perm?(append(d.1,d.2), C) ∧ ordered?(C))
```

To solve this problem a theory with a specific composition operator is selected: theory CMP_LST1 uses the list operator cons as its composition operator. The formal parameters of CMP_LST1 then have to be instantiated properly.

```
% formal parameters of theory CMP_LST1                                 16
T          : TYPE,                    % type of list elements
D          : TYPE,                    % problem domain
I          : pred[D],                 % input condition
O          : [D, list[T] -> bool],    % input/output condition
decompose  : [D -> [T, D]],           % decomposition operator
dir_solve  : [D -> list[T]],          % solution of primitive instances
primitive? : pred[D],                 % primitive instances
mes        : [D -> nat]               % measure on domain D
```

Note that the problem range R is fixed here and defined by list[T]. The main effort to solve this problem is to find an appropriate decomposition function. In the sequel, let A,B denote the input tuple of type D1. Obviously, the problem is primitive if either A or B are empty. In this case B respectively A is returned (dir_solve). A non-primitive problem instance is split according to the result of comparing the two head elements: either the tuple (car(A), (cdr(A), B)) is returned in case car(A) <= car(B) holds; otherwise the tuple (car(B), (A, cdr(B))) is returned. A decreasing measure on the domain D1 is given by the sum of A and B's length.

Most of the TCCs can simply be proved with the built-in grind strategy. The only non-trivial TCC (which requires additional proof effort) is the instantiated composition assumption stating that solutions to the subproblems are correctly composed by the composition operator cons. The proof requires some additional properties about list permutations. Due to space limitations we omit the details here. After all TCCs have been proved an operational specification of mergesort has been derived which is correct by construction. It has the form:

```
mergesort(x:list[T]): RECURSIVE list[T] =                              17
 IF length(x) < 2 THEN x
 ELSE merge(mergesort(listsplit(x).1), mergesort(listsplit(x).2)) ENDIF

merge(x:D1): RECURSIVE R1 =
 IF primitive?(x) THEN dir_solve(x)
 ELSE cons(decompose(x).1, merge(decompose(x).2)) ENDIF
```

This operational specification may be further improved by the optimization techniques presented in Sect. 4. For example, `merge` has a linear recursive form and the transformation from Sect. 4.1 can be applied (see $\boxed{8}$ for the formal parameters): for the expression `p` the list concatenation operator `append` can be instantiated which is associative and has the empty list as neutral element. The `cons` operator can be realized by the concatenation operator. Functions `K`, and `K1` are realized by selecting the first and second part of the decomposition, respectively, and the measure is given by the length of the list. The generated TCCs can be proved automatically using lemmas from the PVS list library. On the other hand, `mergesort` has a tree-like recursive form and therefore the memoization technique presented in Sect. 4.2 can be applied. This might be useful if in the list to be sorted elements occur more than once or if even the same subsequences occur more than once.

Fig. 3 illustrates the derivation process: `mergesort` is derived using `DEC_LST2` where the composition operator, i.e. merging two ordered lists, is derived using `CMP_LST1`. Then both `mergesort` and `merge` are further optimized.

6 Conclusions

In this paper we have presented an approach to rigorous formal mechanized treatment of the transformational software development process in a unified framework. This process comprises the formalization, verification, and application of generic software development steps. We showed that transformations from different existing approaches can be integrated into this framework. The approach has been illustrated by a formal representation of the divide-and-conquer paradigm, two optimization transformations, and a derivation of *mergesort* using these steps.

In most cases the built-in strategies are powerful enough to prove the correctness of the instantiation of the generic software artifacts automatically. However, sometimes additional knowledge about the application domain is required. In general, it is advisable and useful to establish specialized theories for specific application domains. For example, we have constructed a “sorting theory” which includes the elementary definitions for sorting problems as well as a set of proved facts. In addition, specialized proof strategies can be defined which make use of the proved facts. Such a sorting theory then provides a basis for the

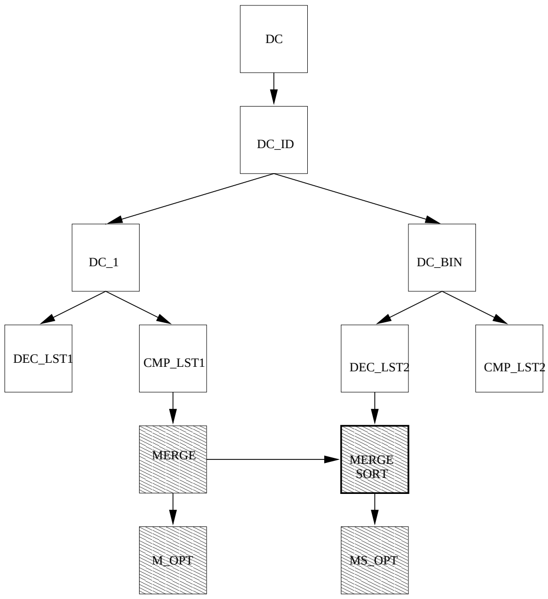


Fig. 3. Derivation of the Mergesort Algorithm

derivation of sorting algorithms. In summary, I believe that the set of generic software development steps together with domain-specific knowledge provide a powerful tool for the formal transformational software development process: specific design/implementation decisions are made by the choice of an appropriate generic theory, and the correctness of an instantiation may be proved with the help of domain-specific knowledge.

The work presented in this paper is part of an ongoing effort to rigorous formal representation of software development steps. Besides extending the existing set of formalized development steps and development methods, we are in the process of applying the approach to the construction of a non-trivial compiler implementation. Starting from a given compiling specification which is proved to be correct with respect to the language semantics, a series of development steps is applied to derive an implementation in the source language itself which is then correct by construction. The development steps involved include functional as well as data structure refinement. This work is carried out within the context of the German compiler verification project *Verifix*.

Related Work

The mechanization of transformational approaches has been considered by several researchers.

In [5], for example, program transformations for recursion removal are expressed as second-order patterns defined in the simply typed λ-calculus. Kreitz

[7] utilizes the NUPRL system which is based on a constructive type theory as a formal framework for representing existing approaches to program synthesis (such as KIDS [14], for example). Among other "algorithm theories" he presents a formal mechanization of a (binary) divide-and-conquer scheme. Rueß [11] formalizes the *divide-and-conquer* paradigm in his dissertation [11] using the calculus of constructions and has given a formal verification with the LEGO proof checker. More recently, Kolyang et al. [6] present a prototype development environment for implementing and applying transformations consisting of the tactical theorem prover Isabelle and a graphical user interface based on the toolkit TK. Their basic idea is to separate the soundness of transformations (stated by synthesis theorems) from the pragmatics of their application. They give a mechanization of the *global search* paradigm in Isabelle/HOL.

Acknowledgments

I would like to thank my colleague Martin Strecker for helpful comments on a draft version of this paper. The constructive criticisms and suggestions provided by the anonymous referees have greatly improved the paper.

References

1. Klaus Achatz and Helmuth Partsch. From descriptive specifications to operational ones: A powerful transformation rule, its applications and variants. Technical Report 96-13, Universität Ulm, December 1996.
2. Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM—Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.
3. Axel Dold. Representing, Verifying and Applying Software Development Steps using the PVS System. In V.S. Alagar and Maurice Nivat, editors, *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95, Montreal*, volume 936 of *Lecture Notes in Computer Science*, pages 431–445. Springer-Verlag, 1995.
4. Berthold Hoffmann and Bernd Krieg-Brückner (Eds.). *Program Development by Specification and Transformation—The PROSPECTRA Methodology, Language Family, and System*, volume 680 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
5. G. Huet and B. Lang. Proving and applying program transformations expressed with second-order-patterns. *Acta Informatica*, 11:31–55, 1978.
6. Kolyang, B. Wolff, and T. Santen. Correct and User-Friendly Implementations of Transformation Systems. In J. Woodcock M.-C. Gaudel, editor, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 629–648, 1996.
7. C. Kreitz. Metasynthesis—deriving programs that develop programs. Technical Report AIDA-93-03, Fachgebiet Intellektik, Technische Hochschule Darmstadt, 1993.
8. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

9. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga NY, 1992. Springer-Verlag.
10. Helmuth A. Partsch. *Specification and Transformation of Programs—A Formal Approach to Software Development.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.
11. H. Rueß. *Formal meta-programming in the calculus of constructions.* PhD thesis, Universität Ulm, Fakultät für Informatik, 1995.
12. N. Shankar. Steps towards mechanizing program transformations using PVS. *Science of Computer Programming*, 26(1–3):33–57, 1996.
13. Douglas R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8(3):213–229, 1987.
14. Douglas R. Smith. KIDS—a semi-automatic program development system. *IEEE—Transactions on Software Engineering, Special Issue on Formal Methods in Software Engineering*, 16(9):1024–1043, September 1990.
15. Douglas R. Smith and Michael R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14(2–3):305–321, 1990.
16. Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.

Model Checking Generic Container Implementations [⋆]

Matthew B. Dwyer and Corina S. Păsăreanu

Kansas State University,
Department of Computing and Information Sciences,
Manhattan KS 66506, USA
{dwyer,pcorina}@cis.ksu.edu

Abstract. Model checking techniques have been successfully applied to the verification of correctness properties of complex hardware systems and communication protocols. This success has fueled the application of these techniques to software systems. To date, those efforts have been targeted at concurrent software whose complexity lies, primarily, in the large number of possible execution orderings of asynchronously executing program actions. In this paper, we apply existing model checking techniques to parameterizable implementations of container data structures. In contrast to most of the concurrent systems that have been studied in the model checking literature, the complexity of these implementations lies in their data structures and algorithms. We report our experiences model checking specifications of correctness properties of queue, stack and priority queue data structures implemented in Ada.

Keywords: Model checking, temporal logic, assume-guarantee reasoning, generic containers

1 Introduction

The past three decades have seen the development of a large number of formal methods for specifying the intended computational results of a software component or system. Clearly, the practical utility of a formal method must be assessed with respect to its ability to describe and support reasoning about the correctness of realistic software systems and a number of efforts in this area have been made (e.g., [1,7]). Short of such case studies, a basic test of a formal method is its ability to address certain standard examples; one class of standard examples are container data types (e.g., stacks, queues, sets).

More recently, in the past decade, a number of formal methods researchers have focused on a class of specification formalisms and automated specification checking techniques referred to as *finite-state verification* (FSV) approaches. Rather than focusing on specification of the correct results of a computation, FSV methods specify correctness properties related to, perhaps internal, system

⋆ Supported in part by NSF under grants CCR-9703094 and CCR-9708184, by NSF and DARPA under grant CCR-9633388, and by NASA under grant NAG-02-1209.

behaviors. Such behaviors might include, for example, correctness of component coordination, safety of data access, and guarantee of progress in performing a computation. These techniques, in particular model checking, have enjoyed success in finding defects in and verifying properties of real hardware systems (e.g., [4,9,20]). Attempts have been made to capitalize on the success of model checking by applying it to selected software domains (e.g., communication protocols [16] and control systems [17]). Control software, typically, manipulate small amounts of data that influence system behavior. More general software systems, however, may manipulate large amounts of data and use that data to control system behavior. It remains to be seen whether FSV approaches will be generally effective for such data-intensive software.

In this paper, we assess a representative of the FSV approaches, namely linear temporal logic (LTL) [19] model checking using SPIN [16], in terms of its ability to support reasoning about the standard container implementations on which formal methods are often judged. Model checking is a technique for sound and complete reasoning about finite-state transition systems. To enable model checking of software, however, completeness is sacrificed for tractability and because software systems are, in general, not finite state. Our methodology [10] uses correctness preserving abstractions of system components to render model checking sound. We believe that such model checking complements traditional formal methods and testing as a software quality assurance technique. Like testing, model checking is an automated technique. Thus, unlike most traditional formal methods which require some user interaction, model checking can be applied by practitioners with little knowledge of proof construction strategies. Like traditional formal methods, sound model checking when successful provides a guarantee that the system satisfies the property being checked; testing is unable to guarantee the absence of errors with respect to a given property. While a failed test is a sure indication of a defect in the system, it provides no guidance to help locate the cause of the defect. In contrast, both proof-based methods and model checking produce a counter-example, which traces an example system execution that violates the property of interest. Analysis of such counter-examples often leads directly to the cause of the erroneous system behavior. Given these qualitative differences between model checking and other methods, our goal is not to compare classes of software validation approaches, but to assess the extent to which model checking can be applied to validate and detect defects in existing container implementations.

We use tools to automate the process of translating a program written in Ada to a safe finite-state model rendered in the input format of SPIN. For this reason, we selected implementations of generic queue, stack and priority queue data types written in Ada. We specified a variety of correctness properties of these abstractions in LTL and checked them using SPIN. Our results are encouraging and suggest that model checking tools can be effectively applied to detect defects in such implementations. We report these results and describe the process we followed to carry out this study.

In the following section we discuss relevant background material. The construction of finite-state models of the container implementations is outlined in Section 3. Section 4 describes the container implementations and Section 5 describes the correctness properties that we analyze. We then present, in Section 6, the results of our study. Section 7 describes this study's limitations and suggests directions for further study and Section 8 concludes.

2 Background and Related Work

In this section, we give a brief overview of temporal logic and model checking. We focus on the differences between this approach to system specification and verification and more traditional formal methods.

2.1 Traditional Formal Methods

Traditionally, formal methods have been designed to support the precise description of the intended results of a computation. A specification is written such that a conforming implementation is guaranteed to produce the correct result; a specification defines a sufficient condition for correctness. There are a wealth of formal methods including model-based (e.g., Z [8]), algebraic (e.g., Larch [18]), and trace-based (e.g., [15]) methods. While these methods differ in their formal underpinnings, each is expressive enough to describe computations over unbounded data domains, such as the naturals. Thus, each is capable of specifying container data structures. In fact, specifications for stacks and queues are often given as examples to illustrate a method [8,18,15].

Trace specifications [15] are particularly relevant to the work described in this paper. These are abstract specifications of the observable interface behavior of a software component. A trace specification defines the legal sequences of calls to operations in a component's interface and the computational effects of such sequences. Hoffman and Snodgrass distinguish three different components of such a trace specification: legality, equality, and value. The specifications we describe in Section 5 state necessary conditions for such trace components, concentrating primarily on the legality of a sequence of calls. Like the stack and queue specifications presented in [15], our stack and queue specifications are nearly identical, differing only in the names of operations and element ordering. We note that our specifications model invocation and return from a procedure as independent atomic actions, whereas traces model them as a single action. We do this to enable specification of properties of concurrent systems where multiple calls to a single procedure may co-execute.

2.2 Temporal Logic

For model checking, specifications are written in a propositional temporal logic. These logics are less expressive than the formalisms used in traditional formal methods (e.g., they cannot express properties of unbounded value sets). For this reason model checking focuses on specifications of necessary conditions for correctness, rather than complete specification of a system's computational effects.

We use linear temporal logic in our work because it supports unit-level model checking, through filter-based analysis [11,12], and it is supported by a robust tool, SPIN [16]. In LTL a pattern of states is defined that characterizes all possible behaviors of the finite-state system. We describe LTL operators using SPIN's ASCII notation. LTL is a propositional logic with the standard connectives `&&`, `||`, `->`, and `!`. It includes three temporal operators: `<>`p says p holds at some point in the future, `[]`p says p holds at all points in the future, and the binary p`U`q operator says that p holds at all points up to the first point where q holds. An example LTL specification for the response property "all calls to procedure P are followed by a return from P" is `[](call_P -> <>return_P)`.

2.3 Model Checking Software

In model checking, one describes software as a finite-state transition system, specifies properties with a temporal logic formula, and checks, exhaustively, that the sequences of transition system states satisfy the formula. In principle, model checking can be applied to any finite-state system. For software one cannot render a finite-state system that exactly models the software's behavior, since, in general, software will not be finite-state. Even for finite-state software the size of a precise finite-state model will, in general, be exponential in the number of independent components (i.e., variables and threads of control). For these reasons, we use abstracted finite-state system models that reflect the execution behavior of the software as precisely as possible while enabling tractable analysis.

Existing model checkers, such as SPIN, do not accept Ada source code. In fact, the semantic gap between Ada and a model checker input language, such as SPIN's Promela, is significant. To bridge this gap, and achieve tractable model checking, we perform an "abstract compilation" of Ada source code to a target program in Promela. This compilation is guided by the LTL formula to be checked with the result that target program is sound with respect to model checking of that formula. In this setting, positive model checks results for a specification imply conformance of the original Ada implementation to that specification. A failed model check result is a trace of the target program that violates the specified property called a counter-example. Analysis of the counter-example indicates one of two situations: (*i*) the implementation is defective with respect to the specified property, or (*ii*) the abstractions used in the compilation of the Promela program are too imprecise for checking the specified property. In the second case, the counter-example provides guidance as to which abstractions must be strengthened to enable a successful model check or the detection of a defect. This compilation process is discussed in more detail in Section 3.

2.4 Filter-Based Analysis

It is common in software validation and verification to reason about parts of an implementation in isolation (e.g., unit testing). For unit-level software model

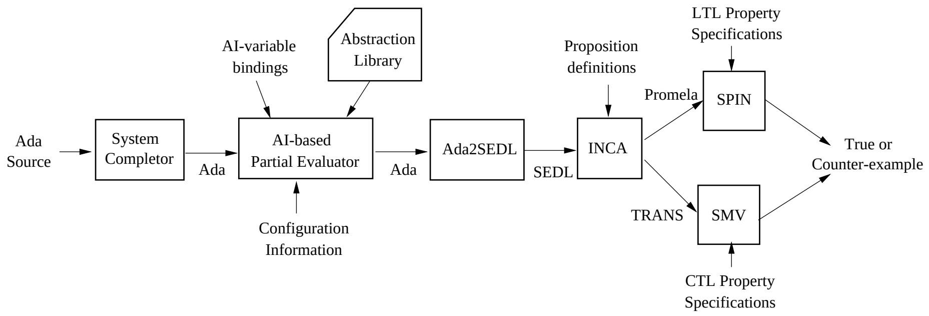


Fig. 1. Model Construction Process

checking, we adopt the assume-guarantee paradigm [22] in which a system description consists of two parts: a model of guaranteed behavior of the software unit and a model of the assumed behavior of the environment in which that unit will execute. In the work described in this paper, we define an Ada driver program that approximates the behavior of all possible contexts in which a software unit may be invoked; we discuss such driver programs in the next section. Model checking of properties of the software unit combined with the driver is sound with respect to any usage of the unit.

Many software components make assumptions about the context in which they will be used (e.g., that a called routine will eventually return). Such assumptions can be incorporated into LTL model checking using SPIN [11,12] as follows: given a property P and filters $F_1, F_2, \ldots F_n$ that encode assumptions about the environment, we model check the combined formula $(F_1 \&\& F_2 \&\& \ldots \&\& F_n) \rightarrow P$. We refer to the individual F_i as *filters* and to the combined formula as a *filter-formula.*

One concern with this method is that users may make invalid assumptions. We address this in two ways. First, we check that the assumptions are not inconsistent by checking that there exists an execution that satisfies the filters; with SPIN this is done by attempting to falsify a never claim for the conjoined filters. Second, when a user wishes to use the fact that a given software component satisfies P in the analysis of the larger system we model check $(F_1 \&\& F_2 \&\& \ldots \&\& F_n)$ on the sub-systems that use the component; this verifies that the context satisfies the assumptions.

3 Model Construction

In this section, we describe the process of compiling programs to descriptions suitable for model checking. We apply the methodology described in [12] which is supported by the toolset illustrated in Figure 1. Incompletely defined Ada source code is fed to a component which constructs a driver program to complete its definition. This completed program is then systematically abstracted and simplified using abstract interpretation and partial evaluation techniques. The resulting program is converted to the input language, called SEDL, of the INCA

```
procedure Driver() is
 choice : Integer;
 theObject_Type : Object_Type;
begin
 loop
  case choice is
   when 1 => Insert(theObject_Type);
   when 2 => theObject_Type := Remove;
   when 3 => null;
   when others => exit;
  end case;
 end loop;
end stub;
```

Fig. 2. Driver Source Code

toolset [2,5]. INCA accepts definitions of the states and events that form the propositions used in specifications and embeds those propositions into the model checker input. A tutorial on the use of this toolset is available [13].

3.1 Completing Partial Systems

A *complete* Ada program is constructed by defining a driver program and stub routines for referenced components that are external to a given partial Ada program. This is similar to a unit-testing approach, except that the stubs and driver are capable of calling the component's public operations in any possible sequence; this simulates any possible usage scenario that may arise for the component. Together, the stubs, driver and component under analysis form a complete Ada application.

To simplify the discussion, we assume that a partial system is encapsulated in a package with public procedures that can be called from outside the package and that we are only interested in the behavior of the partial system (not the environment). A driver program is generated that will execute all possible sequences of public procedure calls. The driver program defines local variables to hold parameter values. Figure 2 illustrates a driver for a partial system with an `Insert` procedure and a `Remove` function. Note that the `choice` and `theObject_Type` variables will be subsequently abstracted; `choice` will be abstracted to model nondeterministic choice of the case statement alternatives. Stub routines are defined in a similar way.

Ultimately, in order to generate a finite-state model from source code all layering of the container implementations must be removed (e.g., the calls to `Insert` and `Remove` in `Driver` must be inlined). The INCA tools perform inlining of non-recursive procedure calls.[1] The tools are not able to perform generic instantiations, so for our examples this was done by hand.

[1] A limitation of our current approach is that it does not treat recursive procedures.

3.2 Incorporating Abstractions

The methodology then proceeds by binding abstract interpretations [6] to selected program variables. A variety of sound abstract interpretations have been defined for different data types. Currently, a set of heuristics are applied to select the variables that are to be abstracted and the specific abstract interpretations to be used for each such variable [12].

The machinery of partial evaluation [14] is used to propagate these abstract variable definitions throughout the program's definition. In this way, the state space of the finite-state transition system constructed for the program can be safely reduced to a size that enables efficient model checking. For the container implementations we studied, almost all variables were defined over discrete ranges (e.g., sub-range types for array indices, booleans); for those variables no abstractions were applied.

The container implementations are parameterized by the type of contained data elements. The properties we check of those implementations, which are described in Section 5, do not specify behavior in terms of data element values, rather they refer to the identity of the data elements. Some properties are related to the ordering of two data elements. To construct a safe model that supports reasoning about such properties we use the *2-ordered data* abstraction [12]. This defines data elements in terms of their identity where two elements (named `d1,d2`) are distinguished from all others (named `ot`). For example, the `Object_Type` in Figure 2 is replaced by the enumerated type `(d1,d2,ot)` which implements this abstraction. Since container implementations only assign data elements and compare them based on identity, the 2-ordered abstraction allows us to treat containers as data-independent systems [23] and prove order related specifications under the assumption that `d1` and `d2` are input to the container at most once. These proofs generalize to any pair of user-defined data elements.

3.3 Specializing Source Code

Partial evaluation can also produce a specialized version of the software system when supplied with information by the user. For systems with dynamically sized data structures it is often the case that specialization is required to place an upper-bound on the number of dynamic objects to allow a finite-state model to be constructed. While this reduces the generality of any property proven of that model, in practice it preserves much of the effectiveness of model checking for defect detection.

The container implementations we consider in our study accept a single constructor parameter that pre-allocates storage for holding all data. We constructed drivers which invoked the constructor with small size values (e.g., 3) and checked properties of the resulting models.

4 The Containers

The Scranton Generic Data Structure Suite [3] is a publicly available collection of over 100 Ada packages that implement a number of variations of list, queue, stack, heap, priority queue, binary and n-ary tree data structures and algorithms on those structures. We selected three specific data structure implementations from version 4.02 of this collection as the subject of our study: `queue_pt_pt`, `stack_pt_pt`, and `priority_queue_lpt_lpt`.[2] The queue and stack are implemented using support packages which provide list and iterator implementations. The priority queue is implemented using underlying heap and iterator implementations. The stack, queue, and priority queue implementations are generic packages requiring a type parameter for the data to be stored in the container. The priority queue also requires a type for the priority of a contained data item as well as an ordering operation, `<`, on the priority type. The queue and stack implementations both allocate a fixed-size array of the appropriate object type for storage of data. The priority queue is implemented on top of an array-based heap implementation; three levels of generic package instantiation are required to reveal the structure of the concrete implementation. The queue and stack container types have associated iterator packages which support iteration in container-order (`Top_Down` iteration) or in reverse-container-order (`Bottom_Up` iteration). Iterators require a user defined call-back routine (`Process`) that is invoked for each datum stored in the container.

To denote the different models, we use `s`, `q`, `p` for the stack, queue and priority queue packages in isolation. The queue with forward and backward iterators required different models `qt` and `qb`, respectively. The stack with iterator required a single model `si`. Finally, each of these models can be scaled for a fixed maximum number of contained elements, e.g., `s(3)` is a stack of at most 3 elements.

5 The Properties

As discussed in Section 2, we cannot hope to specify the complete behavior of the container abstractions in LTL. Instead we focus on several crucial correctness properties of these abstractions. We group these in three areas: containment, order, and observation. Containment properties are related to whether a container implementation has knowledge of data items that have been put into it and not yet taken out of it. Order properties are related to the order in which data items can be removed from or observed in a container (e.g., that priority ordering, via `<`, is observed). Observation properties are related to the ability of operators to distinguish relevant features of the state of the container (e.g, whether a container is empty or not).

We wrote specifications in each of these categories for the three container implementations. In writing specifications one must select a level of abstraction at which to describe system behavior. For our study, we chose the package interfaces for the containers. Specifically, for our LTL specifications we define propositions that describe calls to container operations with specific input parameter values and returns from container operations with specific output parameters. We illustrate the syntax for the names of these propositions by way of example. For

[2] The naming scheme indicates whether the type of the contained data and the instantiated container itself is private, `pt`, or limited private, `lpt`.

the following procedure declaration:

```
 procedure And(x,y : in Boolean; r : out Boolean)
```

we would have a number of propositions, including:

`call_And` representing any call to `And`
`return_And` representing any return from `And`
`call_And(true,false)` representing any call to `And` with `x=true` and `y=false`
`call_And(,false)` representing any call to `And` with `y=false`
`return_And(false)` representing any return from `And` with `r=false`

Function return values are considered to be the last output parameter. With these propositions we can state that "Calling `And` with `true` parameters should return `true`" as

```
 [](call_And(true,true) -> !return_And U return_And(true))
```

This states that any call to `And` with `true` parameters must be followed by a return from `And` with the value `true` and that no other return from `And` can intervene between the the designated call and return. Our writing of specifications in LTL was greatly simplified by use of a specification patterns system [10]; the above specification is an instance of the *global constrained response* pattern.

Figure 3 illustrates a sampling of the specifications that were checked in our study; space limitations prohibit showing all of the specifications. There are 5 basic properties specified. Each of these properties has a slightly different intended semantics depending on the container being checked; we use `q`, `s`, and `p` in the property names to indicate queue, stack and priority queue versions, respectively. The containment specifications, (1), vary depending on whether forward or backward iteration is performed and on the name of the container insert(remove) operation `Enqueue` or `Push`(`Dequeue` or `Pop`) (e.g., (1fq) and (1bs)). For observation specifications, (2-3), only the names of the insert(remove) operations vary. For ordering specifications, (4-5), the modifications are more complex. The order of returned data for the queue and stack are reversed (e.g., (5q) (not shown) and (5s)). The order of dequeued data for the priority queue depends only on the priority value of a datum, thus, we specify the same return sequence regardless of insertion order (e.g., (5p21) and (5p12) (not shown)). There is an assumption about data items `d1` and `d2` that is important for reasoning about the order related properties; this assumption is discussed in the next section, In total there are 18 LTL specifications for variations of the 5 basic properties.

6 Analysis Results

A selection of the specifications we checked was given in Section 5. All model checks were performed using SPIN, version 3.09, on a SUN ULTRA5 with a 270Mhz UltraSparc IIi and 128Meg of RAM (machine 270) or on a SUN Enterprise 4000 with a 168Mhz UltraSparc II and 512Meg of RAM (machine 168). Figure 4 gives the data for each of the model checking runs; the transition system model used for the run is given.[3] We report the elapsed time for running

[3] Detailed description of the transition systems is published as a case-study at `http://www.cis.ksu.edu/santos`.

(1fq) If a datum is enqueued, and not dequeued, then forward iteration will invoke the iterator call-back with that datum, unless the iteration is terminated.
```
[](call_Enqueue(d1) && ((!return_Dequeue(d1)) U call_Top_Down) ->
   <>(call_Top_Down && <>(call_Process(d1) || return_Process(,false))))
```

(1bs) If a datum is pushed, and not popped, then forward iteration will invoke the iterator call-back with that datum, unless the iteration is terminated.
```
[](call_Push(d1) && ((!return_Pop(d1)) U call_Bottom_Up) ->
   <>(call_Bottom_Up && <>(call_Process(d1) || return_Process(,false))))
```

(2q) An enqueue without a subsequent dequeue can only be followed by a call to empty returning false.
```
[](call_Enqueue && (!return_Dequeue U call_Empty) ->
   <>(call_Empty && <>return_Empty(false)))
```

(3p) Between an enqueue of a datum and a call to empty returning true there must be a dequeue returning that datum.
```
[]((call_Enqueue(d1) && <> return_Empty(true)) ->
   (!return_Empty(true) U (return_Dequeue(d1))))
```

(4fq) If a pair of data are enqueued, and not dequeued, then forward iteration will invoke the iterator call-back with the first datum then the second, unless the iteration is terminated.
```
[]((call_Enqueue(d1) && ((!return_Dequeue(d1)) U (call_Enqueue(d2) &&
   ((!return_Dequeue(d1) && !return_Dequeue(d2) U call_Top_Down)))) ->
   <>(call_Top_Down && <>((call_Process(d1) && <>call_Process(d2)) ||
                          return_Process(,false))))
```

(5s) If a pair of data are pushed then they must be popped in reverse order, if they are popped.
```
[]((call_Push(d1) && (!return_Pop(d1) U call_Push(d2))) ->
   (!return_Pop(d1) U (return_Pop(d2) || [](!return_Pop(d1)))))
```

(5p21) If a pair of data are enqueued in reverse-priority order, then they must dequeued in priority order (`Priority(d1)>Priority(d2)`).
```
[]((call_Enqueue(d2) && ((!return_Dequeue(d2)) U (call_Enqueue(d1)))) ->
   (!return_Dequeue(d2) U (return_Dequeue(d1) || [](!return_Dequeue(d2)))))
```

Fig. 3. LTL Specifications

SPIN to convert LTL to the SPIN input format, to compile the Promela into a model checker, and to execute that model checker. The model construction tools were run on an AlphaStation 200 4/233 with 128Meg of RAM. The longest time taken to convert completed Ada to SEDL was for the model of a priority queue of size 3; it took 26.7 seconds. Generating Promela from the SEDL can vary due to differences in the predicate definitions required for different properties. The longest time taken for this step was for the model of the queue of size 2; it took 129.3 seconds.

Property	*Time*	*Result*	*Model*	*Machine*
(1fq)	0.2, 54:12.9, 7.6	false	qt(2)	270
(1fq_f)	0.9, 1:43:36.2, 9.4	true	qt(2)	270
(1bq)	0.1, 54:39.9, 7.6	false	qb(2)	270
(1bq_f)	1.0, 1:27:39.9, 13.9	true	qb(2)	270
(1fs)	0.2, 1:12.6, 0.1	false	si(2)	270
(1fs_f)	0.9, 2:01.6, 0.2	true	si(2)	270
(1bs)	0.2, 1:12.2, 0.2	false	si(2)	270
(1bs_f)	0.9, 1:53.1, 0.2	true	si(2)	270
(2q)	0.1, 26.1, 0.2	true	q(2)	270
(2s)	0.1, 23.6, 0.2	true	s(3)	270
(2p)	0.1, 21.2, 0.1	true	p(3)	270
(3q)	0.1, 14.0, 0.1	true	q(2)	270
(3s)	0.1, 14.1, 0.1	true	s(3)	270
(3p)	0.1, 10.4, 0.1	true	p(3)	270
(4fq)	6.9, 7:57:43.9, 12.6	false	qt(2)	168
(4fq_f)	1:59.5,11:41:56.9, 37.0	true	qt(2)	168
(4bq)	8.5, 7:44:51.6, 14.4	false	qb(2)	168
(4bq_f)	1:27.8, 9:40:58.7, 27.5	true	qb(2)	168
(4fs)	6.7, 7:08.5, 0.2	false	si(2)	270
(4fs_f)	1:24.6, 10:45.2, 1.1	true	si(2)	270
(4bs)	8.8, 7:12.1, 0.2	false	si(2)	270
(4bs_f)	2:00.8, 10:35.6, 0.7	true	si(2)	270
(5q)	0.1, 19.0, 0.1	false	q(2)	270
(5q_f)	11:57.5, 15:44.2, 2.6	true	q(2)	270
(5s)	0.2, 25.5, 0.1	false	s(3)	270
(5s_f)	9:34.9, 13:03.4, 3.6	true	s(3)	270
(5p12)	0.1, 11.9, 0.1	true	p(3)	270
(5p21)	0.2, 11.6, 0.1	true	p(3)	270

Fig. 4. Performance Data

Our container models make no assumptions about the way that the container operations are invoked (e.g., number of times, order of invocation). The iterator call-back `Process` is assumed to be data-independent. To boost precision in checking certain properties, we code assumptions about the required behavior of the driver or stub as a filter and then model checked the filter-formulae (denoted by "_f" in the table).

For example, analysis of the counter example provided by SPIN for specification (1fq) showed that the result is false because it is possible for the stub for `Process` to never return. Enforcing the reasonable assumption that the iterator call-back always returns, yields the filter-formula, (1fq_f):

```
[](call_Process -> <>return_Process) ->
[]((call_Enqueue(d1) && (!return_Dequeue(d1) U call_Top_Down)) ->
   <>(call_Top_Down && <>(call_Process(d1) || return_Process(,false))))
```

The same filter was used for other properties of type (1) and (4).

The use of filters in properties of type (5) was required since the 2-ordered data AI incorporated in the model is only guaranteed to be safe under the assumption of a single insertion of each data item into the container. For example, the filter-formula (5s_f) is:

```
([](return_Push(d1) -> [](!call_Push(d1))) &&
 [](return_Push(d2) -> [](!call_Push(d2)))) ->
[]((call_Push(d1) && (!return_Pop(d1) U call_Push(d2))) ->
   (!return_Pop(d1) U (return_Pop(d2) || [](!return_Pop(d1)))))
```

These results are consistent with previous work on filter-based analysis [12,21]. When filters are required they are relatively few and simple. For the most part (ignoring SPIN compile times), this study shows that the total time required to model check properties of container implementations is on the order of a few minutes. We discuss the compile-time issue in the next section.

7 Discussion and Future Work

While we believe that our results indicate the potential for model checking to be an effective quality assurance technique for a broad class of software systems, there are a number of clear limitations to our study.

7.1 Scaling Containers

We believe that the results of the previous section bode well for using model checking techniques for reasoning about software systems, but, there remain significant questions about its ability to scale to large systems. We used containers of size 2 and 3 for the bulk of our model checks. These are the smallest sizes that one would consider using for reasoning about order properties. When we increased container size to 4 for the priority queue, model generation and check times increased by a factor of 3. For stack and queue containers of size 4 the model generation tools ran for 10 hours before we stopped them. As one would expect there is a very rapid growth in the size of the state space as container size increases. Since the model generation tools expand parts of this state space they require significant amounts of memory. We believe that memory limitations were one cause of the significant slowdown in generation time with increasing size. Future experiments with large-memory machines will help address this question.

Our model generation tools (i.e. INCA) were designed for reasoning about synchronization properties of concurrent systems [5]. Such systems typically have little data that is used to control the pattern of inter-process synchronization. Given these requirements it was a reasonable design decision to encode all data local to a single process into the control flow of that process. Unfortunately, for data intensive systems like containers, this causes an enormous expansion in the program's control flow. This is why the Promela compile-times in Figure 4 are so large (nearly 12 hours for (4fq_f)). It is important to understand that this is an artifact of the model construction process and not an inherent limitation of SPIN. To illustrate this, we checked the queue properties of type (4) on a model that was hand translated from Ada to Promela (converting Ada variables into Promela variables). The effect was dramatic, for (4fq_f) Promela compile-time was 4.1 seconds and model check time dropped to 22.3 seconds. Future versions of model generation tools will need to define their mappings with such performance issues in mind.

A variety of different model checking techniques have been developed. Given the data-intensive nature of container implementations we wondered whether a different model checking technique might work better. In particular, whether SMV [20] and its use of OBDD-based encodings of transition systems might be effective in compactly representing the data state-space of the container models. INCA generates very efficient input models for SMV. We re-ran all of the priority queue property model checks using SMV 2.5 on machine `270`. Model generation time was 2.2 seconds and for the (5p12) and (5p21) properties model check time was 0.7 seconds. It appears that SMV may be more effective than SPIN for checking properties of this kind of system. Unfortunately, SMV's specification language cannot easily incorporate filter-formula and that is a significant limitation for assumption-based validation of partial software systems.

7.2 Dynamism in Implementations

Many container implementations do not pre-allocate storage for their contents, rather they dynamically allocate that storage as needed. To check properties of such implementations we need to incorporate a safe abstraction of allocation and reclamation of heap storage. We are investigating the use of a scalable bounded heap abstraction that allows allocation and deallocation of data and tracks the state of the heap. Whether the cost, in the size of the finite-state system model and consequent model check time, is prohibitive is the subject of future empirical study.

7.3 Defect Detection

While model checking is capable of verifying properties of software, its main benefit may be as a fault-detection technique. Fortunately, for many systems faults are exhibited in small system sizes where application of model checking is most cost-effective.

To illustrate this we seeded a fault in the priority queue implementation. In the `Insert` procedure from the heap package, presented in Figure 5, we deleted the `not` from the test of the `while` loop. We then re-ran the model checker for (5p21) and detected the defect in essentially the same time as reported in Figure 4. SPIN produced a counter-example that gives the changes in the values of the propositions along a path through the system on which the property does not hold. The simulation output generated by SPIN from the counter example with only the boolean variables for the predicates that appear in property specification (5p21) is given in Figure 6. It is easy to see from this counter-example, that the data is not dequeued in priority order (i.e., `Priority(d1)` > `Priority(d2)`).

```
procedure Insert (Heap : in out Heap_Type; Object: in out Object_Type) is
 Parent: natural := (Heap.Size + 1) / 2 ;
 Child : natural := Heap.Size + 1 ;
 begin
   if (Heap.Size = Heap.Max_Size) then
    raise Heap_Overflow ;
   else
    Heap.Size := Heap.Size + 1 ;
    Initialize (Heap.Data.all(Heap.Size));
    while (Parent>0) and then not ((Heap.Data.all(Parent)>=Object)) loop
      Swap (Heap.Data.all(Parent), Heap.Data.all (Child)) ;
      Child := Parent ;
      Parent:= Parent / 2 ;
    end loop;
    Swap (Object, Heap.Data.all(Child) ) ;
   end if;
 end Insert ;
```

Fig. 5. Insert procedure

```
53: proc 2 (driver_task) line 150 "pan_in" (state 143) [callEnqd2=1]
66: proc 2 (driver_task) line 165 "pan_in" (state 159) [callEnqd2=0]
119: proc 2 (driver_task) line 8179 "pan_in" (state 8916) [callEnqd1=1]
132: proc 2 (driver_task) line 8194 "pan_in" (state 8932) [callEnqd1=0]
191: proc 2 (driver_task) line 2227 "pan_in" (state 2402) [returnDeqd2=1]
```

Fig. 6. Reduced SPIN counter-example

This faulty result either indicates a defect in the implementation or some imprecision in the finite-state model. We use only one data abstraction (the *2-ordered data* abstraction) which is safe under the assumption that `d1` and `d2` are input to the container at most once [12]. However, in the counter-example, `d1` and `d2` are enqueued only once and thus, the assumption is not violated. Having eliminated the possibility of an imprecise abstraction the only conclusion is that the implementation has a defect.

7.4 Research Issues

Our study revealed several interesting research issues that we intend to explore in future work.

The properties we considered in this study are *necessary partial* specifications of system behavior. They are admittedly incomplete, but useful nevertheless. Our seeding, and subsequent detection, of faults in container implementations illustrates their utility. A more thorough study of the kinds of faults that can be revealed by such specifications would provide a better understanding of the breadth of applicability of model checking for fault-detection.

In [12], we use abstract interpretations of the behavior of container implementations in checking properties of systems that use containers. These AIs are finite-state, thus it would be possible to encode them as an LTL formula and to check that property against container implementations. This would essentially lift model checking results for containers up to model checking of applications and provide a practical illustration of the reuse of verification results.

8 Conclusions

We have applied existing model checking tools to validation of the correctness of common container data structure implementations. In doing this, we have illustrated how crucial correctness properties of these systems can be encoded in LTL. We have applied a methodology that incorporates techniques from abstract interpretation and partial evaluation to construct finite-state models from the source code of container implementations. We believe that this study demonstrates the potential of model checking as a practical means of detecting faults in general purpose software components. This work raises a number of research issues whose study may further expand the breadth of applicability of model checking to common types of software systems.

Acknowledgements

The authors would like to thank James Corbett and George Avrunin for making the INCA toolset available to us, for enhancing its predicate definition capabilities, and for many valuable discussions about model generation.

References

1. J.-R. Abrial, E. Börger, and H. Langmaack. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control.* Lecture Notes in Computer Science, 1165. Springer-Verlag, Oct. 1996.
2. G. Avrunin, U. Buy, J. Corbett, L. Dillon, and J. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, Nov. 1991.
3. J. Beidler. The Scranton generic data structure suite. |http://academic.uofs.edu/faculty/beidler/ADA/default.html—, 1996.
4. E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the future-bus+ cache coherence protocol. *Formal Methods in System Design*, 6(2), 1995.
5. J. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), Mar. 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 *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
7. D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical report, National Institute of Standards and Technology, Mar. 1993.
8. J. Davies and J. Woodcock. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.

9. D. Dill, A. Drexler, A. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, July 1992.
10. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999. to appear.
11. M. Dwyer and D. Schmidt. Limiting state explosion with filter-based refinement. In *Proceedings of the 1st International Workshop on Verification, Abstract Interpretation and Model Checking*, Oct. 1997.
12. M. B. Dwyer and C. S. Păsăreanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 1998.
13. M. B. Dwyer, C. S. Păsăreanu, and J. C. Corbett. Translating Ada programs for model checking : A tutorial. Technical Report 98-12, Kansas State University, Department of Computing and Information Sciences, 1998.
14. J. Hatcliff, M. B. Dwyer, and S. Laubach. Staging static analysis using abstraction-based program specialization. In *LNCS 1490.* Principles of Declarative Programming 10th International Symposium, PLILP'98, Sept. 1998.
15. D. Hoffman and R. Snodgrass. Trace specifications: Methodology and models. *IEEE Transactions on Software Engineering*, 14(9):1243–1252, Sept. 1988.
16. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
17. C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell.* Lecture Notes in Computer Science, 891. Springer-Verlag, Jan. 1995.
18. B. Liskov and J. V. Guttag. *Abstraction and Specification in Program Development.* The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1986.
19. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1991.
20. K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
21. G. Naumovich, L. Clarke, and L. Osterweil. Verification of communication protocols using data flow analysis. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Oct. 1996.
22. A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag, 1985.
23. P. Wolper. Specifying interesting properties of programs in propositional temporal logics. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 184–193, St. Petersburg, Fla., Jan. 1986.

Mizar Correctness Proofs of Generic Fraction Field Arithmetic*

Christoph Schwarzweller

Wilhelm-Schickard-Institute for Computer Science
Sand 13, D-72076 Tübingen
schwarzw@informatik.uni-tuebingen.de

Abstract. We propose the Mizar system as a theorem prover capable of verifying generic algebraic algorithms on an appropriate abstract level. The main advantage of the Mizar theorem prover is its special proof script language that enables textbook style presentation of proofs, hence allowing proofs in the language of algebra.
Using Mizar we were able to give a rigorous machine assisted correctness proof of a generic version of the Brown/Henrici arithmetic in the field of fractions over arbitrary gcd domains.

1 Introduction

Over the last several years generic programming has received more and more attention. Many programming languages nowadays include generic concepts like polymorphism in functional programming languages, or overloading or templates in C++; or they are even completely designed as a generic language like SuchThat [7]. Also generic libraries have been developed like the Ada Generic Library or the Stl. On the other hand the widespread use of generic concepts more and more entails the need for a thorough formal machine assisted verification of generic algorithms—especially to improve the reliability of generic libraries. This paper deals with the verification of generic algorithms in the field of computer algebra.

SuchThat is a programming language that enables generic programming in the field of computer algebra. SuchThat is a procedural language and can be seen as a successor of Aldes [4] from which it adopted its statement parts. The main feature of SuchThat is the possibility to express the specification of an algorithm (and hence the minimal conditions under which the algorithm works) in the language itself. To achieve this, SuchThat contains declarations that enable the user to introduce parameterized and attributed algebraic structures. Subsequently, algorithms are written based on these structures.

We consider the algorithm of Brown and Henrici concerning addition of fractions over gcd domains as an example. Let I be an integral domain, and let Q be the set of fractions over I. Based on algorithms for arithmetic operations in

* This paper is based on the author's Ph.D thesis [8], supervised by Rüdiger Loos.

I one obtains algorithms for arithmetic in Q. To be able to choose a unique representative from each equivalence class of Q, we assume that I is a gcd domain; that is, an integral domain in which for any two elements a greatest common divisor exists. We also assume that there is a constructor `fract` to build a fraction out of elements of `I` and selectors `num` and `denom` that decompose a fraction into numerator and denominator respectively.

The algorithm accepts normalized fractions as input, returning the result again as a normalized fraction. The point is that the normalized result is achieved not by executing ordinary addition of fractions followed by a normalization step, but by integrated greatest common divisor computations. This allows singling out trivial cases and, and, more important, taking advantage of the normalization of the inputs to achieve normalization of the output by, in general, cheaper gcd-computations. In SuchThat the algorithm is written as follows.

⟨ **BrHenAdd.sth 1** ⟩

```
global: let I be gcdDomain;
        let Q be Fractions of I;
        let Amp be multiplicative AmpleSet of I.

Algorithm: t := BHADD(r,s)
  Input: r,s ∈ Q such that r,s is_normalized_wrt Amp.
 Output: t ∈ Q such that t~r+s & t is_normalized_wrt Amp.
  Local: let 0,1,r1,r2,s1,s2,d,e,r2',s2',t1,t2,t1',t2' ∈ I;
           let 0 ∈ Q;

(1) [r = 0 or s = 0]
    if r = 0 then {t := s; return};
    if s = 0 then {t := r; return}.
(2) [get numerators and denominators]
    r1 := num(r); r2 := denom(r);
    s1 := num(s); s2 := denom(s).
(3) [r and s in I]
    if (r2 = 1 and s2 = 1) then
      {t := fract(r1+s1,1); return}.
(4) [r or s in I]
    if r2 = 1 then {t := fract(r1*s2+s1,s2); return};
    if s2 = 1 then {t := fract(s1*r2+r1,r2); return}.
(5) [general case]
    d := gcd(r2,s2);
    if d = 1 then {t := fract(r1*s2+r2*s1,r2*s2); return};
    r2' := r2/d; s2' := s2/d;
    t1 := r1*s2'+s1*r2'; t2 := r2*s2';
    if t1 = 0 then {t := 0; return};
    e := gcd(t1,d); t1' := t1/e; t2' := t2/e;
```

```
        t:= fract(t1',t2'). □
◇
```
Also defined in: 2, 11

For completeness we also give the SuchThat specifications of the necessary subalgorithms. Note that for the verification of Brown/Henrici addition we only need these so-called prototypes rather than the full subalgorithms. Consequently Brown/Henrici addition will be correct for every set of subalgorithms fulfilling these specifications regardless of how these algorithms are written in detail.

⟨ **BrHenAdd.sth 2** ⟩

```
   Algorithm: r1 := num(r)
   Input:     r ∈ Q.
   Output:    r1 ∈ I such that r1 = num(r). □

   Algorithm: r2 := denom(r)
   Input:     r ∈ Q.
   Output:    r2 ∈ I such that r2≠ 0 & r2 = denom(r). □

   Algorithm: r := fract(r1,r2)
   Input:     r1,r2 ∈ I such that r2 $\neq$ 0.
   Output:    r ∈ Q such that r = fract(r1,r2). □

   Algorithm  d := /(r1,r2)
   Input:     r1,r2 ∈ I such that r2≠ 0 & r2 divides r1.
   Output:    d ∈ I such that d = r1/r2. □

   Algorithm  c := gcd(a,b)
   Input:     a,b ∈ I.
   Output:    c ∈ I such that c ∈ Amp & c = gcd(a,b). □
◇
```
Also defined in: 1, 11

SuchThat is currently a preprocessor to C++. The static semantic checks of the SuchThat translator include extended typechecking, overload resolution and controlled instantiation of structure parameters. SuchThat programs can be instantiated with special domains in the usual way. For example the Brown/Henrici algorithm can be instantiated with the integers, polynomial rings or the Gaussian integers.

The correctness of this algorithm depends on fundamental properties of greatest common divisors. In the following we will show how to prove them (and the correctness of the algorithm) rigorously with machine assistance.

2 The Mizar System

Mizar [6] is a theorem prover based on natural deduction. Starting from the axioms of set theory,[1] up to now about 20,000 theorems from such different fields of mathematics as topology, algebra, category theory and many others have been proven and stored in a library. From our point of view one of the main contribution of the Mizar system is its special proof script language. This language is declarative and associates the natural deduction steps with English constructs, thus allowing one to write proofs close to textbook style.

In addition Mizar includes a kind of mathematical type system: the user can define so-called *modes*—that is, mathematical structures and objects he wants to argue about (for example `integral domain`, or `domRing` as it is called in Mizar, is such a mode). Consequently, we can write

```
let I be domRing;
let a be Element of the carrier of I;
```

Now `domRing` is defined as a commutative ring which satisfies the following predicate, in Mizar called an *attribute*.

```
definition
let R be comRing;
attr R is domRing − like means
  for x, y being Element of the carrier of R holds
  x ∗ y = 0.R implies x = 0.R or y  = 0.R;
end;
```

As a consequence it inherits all properties of (commutative) rings—especially, all already proven theorems concerning rings are applicable to `I`. So, if we already have proven the following theorem

```
T : for R being Ring
    for x being Element of R holds
    x * 0.R = 0.R;
```

where `T` is a label, we have only to write

```
a ∗ 0.I = 0.I  by T;
```

to prove $a \cdot 0 = 0$ in an integral domain `I`.

But this is just the platform we need to reason about generic algebraic algorithms: we argue in abstract algebraic domains, and we use only formal parameters that hold for every possible instantiation. Hence we prove the algorithms correct on the appropriate abstract algebraic level.

[1] To be more precise, Mizar uses the axioms of a variant of ZFC set theory due to Tarski.

Due to its natural proof script language, Mizar is well suitable not only for formalizing mathematics, but also for scientists writing generic (algebraic) algorithms: they can prove the correctness of their algorithms in Mizar in almost the same way they would prove them without machine assistance and need not in addition go deeply into a proof logic or the strategies of a special theorem prover.

In the following we give a short introduction to the Mizar language by using parts of our Mizar article `GCD.miz` as illustrations.[1] Each Mizar article starts with an environment which introduces mathematical concepts one wants to use in the rest of the article. This is done by referencing other Mizar articles in which the desired concepts have been defined. In our case:

⟨ **GCD.miz 3** ⟩
```
environ
 vocabulary
 BOOLE,VECTSP_1,VECTSP_2,REAL_1,LINALG_1,SFAMILY,GCD;
 notation
 TARSKI,BOOLE,STRUCT_0,RLVECT_1,SETFAM_1,VECTSP_1,VECTSP_2;
 constructors
 ALGSTR_1;
 theorems
 TARSKI,BOOLE,WELLORD2,SUBSET_1,ENUMSET1,VECTSP_1,VECTSP_2;
```
◇

Also defined in: 4, 5

The second part of a Mizar article is called the text proper. It includes new mathematical definitions and theorems as well as proofs for those. Based on the vocabulary introduced in the environment one can define new mathematical concepts in a very natural way. For example, to introduce the concept of divisibility in integral domains one can write

⟨ **GCD.miz 4** ⟩
```
reserve I for domRing;
reserve a,b,c for Element of the carrier of I;

definition
let I; let a,b,c;
pred a divides b means :Def1:
 ex c being Element of the carrier of I st b = a*c;
end;

definition
let I; let a,b;
```

[1] For more information on the Mizar language and the Mizar system see the Mizar home page at `http://mizar.org/`.

```
    pred a is_associated_to b means :Def2:
     a divides b & b divides a;
    antonym a is_not_associated_to b;
    end;
```
◇

Also defined in: 3, 5

In addition one can define functions in Mizar, for example the division function over integral domains. In contrast to introducing predicates this requires a correctness proof, that is an existence and a uniqueness proof. @proof is a construct that enables the user to postpone required proofs and to concentrate on the implications of the definitions.[1]

⟨ **GCD.miz 5** ⟩

```
    definition
    let I; let a,b;
    assume d: b divides a & b <> 0.I;
      func a/b -> Element of the carrier of I means :Def5:
      it*b = a;   :: it stands for the value of a/b.
    correctness @proof end;
    end;
```
◇

Also defined in: 3, 4

Unfortunately, the algebraic domains necessary to show the correctness of Brown/Henrici arithmetic, namely gcd domains and fractions, were not included in the Mizar library so far. So we first had to prove about 40 theorems about divisibility, gcd domains and fractions before we could start the entire correctness proof.

3 Verifying Generic Algebraic Algorithms

To prove correctness of generic algebraic algorithms, or to be more precise SuchThat algorithms, we first decompose the given algorithm using the Hoare calculus. We consider the algorithm with its input/output specification as a Hoare triple, on which we apply Hoare's rules in a backward manner. The key is that this allows us to eliminate the whole program code out of the given triple; that is, we end up with pure algebraic theorems. So we get a set of algebraic theorems[2] implying the correctness of this algorithm on the appropriate abstract level independent of any particular instantiation.

We implemented in Scheme a verification condition generator following this approach (see [8]). Of course this generator needs user support to compute a complete verification condition set; for example, loop invariants must be provided by

[1] Of course the Mizar Library Committee will not accept an article from which not all @proof's are removed.

[2] In fact this is a so-called verification condition set, see for example [2].

the user. For the Brown/Henrici addition algorithm the generator constructs 15 theorems, in this case all without user interaction. We give an example theorem which is already transformed into the Mizar language: it states that `t` computed in step `(5)` of the algorithm, where `d = gcd(r2,s2) = 1`, indeed is normalized and equivalent to `r+s`. The Mizar proof of this theorem can be found in the appendix.

⟨ **BrHenArith.miz 6** ⟩
```
theorem
(Amp is_multiplicative &
 r is_normalized_wrt Amp & s is_normalized_wrt Amp &
 not(r = 0.Q) & not(s = 0.Q) &
 r1 = num(r) & r2 = denom(r) & r2 <> 0.G &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
 not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 <> 1.G &
 d ∈ Amp & d = gcd(r2,s2,Amp) & d = 1.G &
 t = fract(r1*s2+r2*s1,r2*s2))
implies (t ~ r+s & t is_normalized_wrt Amp)
```
⟨ **proof of the theorem** : 13, 14, 15, 16 ⟩
◇
Also defined in: 7, 8, 9, 10, 12

To prove these automatically constructed theorems we also have to introduce the concepts used in the algorithm; that is, in addition to the algebraic domains—gcd domains and fractions—we must define in Mizar ample sets (or sets of representatives) over integral domains and the predicates `~` and `is_normalized_wrt`, as well as functions representing the subalgorithms of section one. Again we only give some examples, mainly to show how to define mathematical objects in the Mizar language.

⟨ **BrHenArith.miz 7** ⟩
```
definition
let I be domRing;
mode AmpSet of I -> non empty
  Subset of the carrier of I means
  (for a being Element of the carrier of I
   ex z being Element of it st z is_associated_to a) &
  (for x,y being Element of it holds
   x <> y implies x is_not_associated_to y);
existence @proof end;
end;

definition
let I be domRing;
attr I is gcd-like means
  (for x,y being Element of the carrier of I
   ex z being Element of the carrier of I st
   z divides x & z divides y &
   (for zz being Element of the carrier of I
    st (zz divides x & zz divides y) holds zz divides z));
end;
```
◇
Also defined in: 6, 8, 9, 10, 12

A gcd domain is simply an integral domains with the just defined attribute `gcd-like`. But before we can introduce mode `gcdDomain` in this way we have to prove the existence of such an object[1] in a so-called cluster definition.

⟨ **BrHenArith.miz 8** ⟩
```
   definition
     cluster gcd-like domRing;
   existence @proof end;

   definition
     mode gcdDomain is gcd-like domRing;
   end;
```
◇
Also defined in: 6, 7, 9, 10, 12

Now, having introduced gcd domains in Mizar, it is easy to define the remaining concepts; e.g., the predicate `is_normalized_wrt`.

⟨ **BrHenArith.miz 9** ⟩
```
   definition
   let G be gcdDomain;
   let u be Fraction of G;
   let Amp be AmpleSet of G;
   pred u is_normalized_wrt Amp means :Def20:
     gcd(num(u),denom(u),Amp) = 1.G &
     denom(u) ∈ Amp;
   end;
```
◇
Also defined in: 6, 7, 8, 10, 12

The main part of the proof of the verification conditions is the proof of the so-called theorem of Brown and Henrici [1], which shows how the sum of two normalized fractions can be computed in a normalized form with the reduction to lowest terms interleaved with the computation of the numerator and denominator of the sum. `r1` resp. `s1` are the numerators and `r2` resp. `s2` the denominators of the occurring fractions.

[1] Mizar does not allow empty modes.

⟨ **BrHenArith.miz 10** ⟩

```
theorem
for Amp being AmpleSet of I
for r1,r2,s1,s2 being Element of the carrier of I holds
(gcd(r1,r2,Amp) = 1.I & gcd(s1,s2,Amp) = 1.I &
 r2 <> 0.I & s2 <> 0.I)
implies

gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
    r2*(s2/gcd(r2,s2,Amp)),Amp) =
gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
    gcd(r2,s2,Amp),Amp)
@proof end;
```
◇

Also defined in: 6, 7, 8, 9, 12

We will not give the Mizar proof of this theorem here; it can be found in [8]. We only mention that the entire correctness proof (that is the introduction of gcd domains and ample sets is not included) took about 1000 lines of Mizar code. Note again that the verification proofs are done on an abstract algebraic level, hence that they are independent of any particular instantiation.

It is obvious that the subtraction algorithm follows by the same token as the addition algorithm. For multiplication Henrici and Brown provide the following algorithm:

⟨ **BrHenAdd.sth 11** ⟩

```
global: let I be gcdDomain;
        let Q be Fractions of I;
        let Amp be multiplicative AmpleSet of I.

Algorithm: t := BHMULT(r,s)
    Input: r,s ∈ Q such that r,s is_normalized_wrt Amp.
   Output: t ∈ Q such that t~r*s & t is_normalized_wrt Amp.
    Local: let 1,r1,r2,s1,s2,d,e,r1',r2',s1',s2' ∈ I;
           let 0 ∈ Q;

(1) [r = 0 or s = 0]
    if r = 0 or s = 0 then {t := 0; return}.
(2) [get numerators and denominators]
    r1 := num(r); r2 := denom(r);
    s1 := num(s); s2 := denom(s).
(3) [r and s in I]
    if (r2 = 1 and s2 = 1) then
      {t := fract(r1*s1,1); return}.
(4) [r or s in I]
    if r2 = 1 then {d := gcd(r1,s2); r1' := r1/d;
                    t := fract(r1'*s1,s2); return};
    if s2 = 1 then {d:= gcd(s1,r2); r1’ := r1/d;
                    t := fract(r1’*s1,r2); return}.
(5) [general case]
    d := gcd(r2,s2); e := gcd(s1,r2);
    r1’ := r1/d; r2’ := r2/e; s1’ := s1/e; s2’ := s2/d;
    ◇       t:= fract(r1’*s1’,r2’*s2’). □
```

Also defined in: 1, 2

The overall goal is the same as for addition: to make use of the normalization of the inputs to simplify and, in general, speed up the normalization of the output. Division of fractions is reduced to multiplication by the inverse fraction which does not introduce additional complexity.

The correctness proof of this algorithm is similar to the one of the addition algorithm. It is based on the following theorem which is also due to Brown and Henrici:

⟨ **BrHenArith.miz 12** ⟩

```
    theorem
    for Amp being AmpleSet of I
    for r1,r2,s1,s2 being Element of the carrier of I holds
    (gcd(r1,r2,Amp) = 1.I & gcd(s1,s2,Amp) = 1.I &
     r2 <> 0.I & s2 <> 0.I)
    implies
    gcd((r1/gcd(r1,s2,Amp))*(s1/gcd(s1,r2,Amp)),
        (r2/gcd(s1,r2,Amp))*(s2/gcd(r1,s2,Amp)),Amp) = 1.I;
◇   @proof end;
```

Also defined in: 6, 7, 8, 9, 10

Our verification condition generator constructs 12 theorems for the multiplication algorithm. Again each theorem is either straightforward or is proved by showing that the theorem of Brown and Henrici is applicable in this case.

To summarize, using Mizar we were able to give complete verification proofs of fraction field arithmetic based on Brown/Henrici’s algorithms. Furthermore we did this in a generic way so that correctness is ensured for fractions over arbitrary gcd domains.

4 Conclusions and Further Work

We have presented a new approach to bringing machine assistance into the field of generic programming. Thereby we focused on generic algebraic algorithms and their verification. Using the Mizar system we succeeded in verifying generic versions of Brown/Henrici arithmetic and of Euclid’s algorithm on the appropriate algebraic level; thus our proofs are independent of any particular instantiation and hold for all gcd-domains.

The emphasis is on the fact that algebraic proof in Mizar can be directly written in the language of algebra and need not be transformed into a more or less completely different proof language. In addition we provided a verification condition generator, which computes from a given SuchThat algorithm and user-supplied lemmata the theorems necessary to establish its correctness.

Another point concerns the instantiation of SuchThat algorithms. Consider again the Brown/Henrici addition algorithm. If it is called for example with the integers, then an obvious correctness condition is that the integers constitute a gcd domain. In addition assume that a (generic) Euclidean algorithm shall be used as a subalgorithm to compute greatest common divisors. The corresponding prototype requires a gcd domain which gives us another condition, namely that Euclidean domains are gcd domains. These kinds of correctness conditions arising for the use of generic algebraic algorithms also can be handled with the Mizar system and form the basis of the algebraic type checks in SuchThat.

Mizar's original purpose was to bring mathematics—including the necessary proof techniques—onto the computer and to build a library of mathematical knowledge. In fact, so far the library is nothing more than a collection of articles accepted by the Mizar proof checker: Reusing the knowledge is not supported as well as it needs to be for our purpose. Consequently, to build a verification system for generic algebraic algorithms around the Mizar system requires some further work. In this context we would like to mention two points:

– First of all, we need a tool for searching the Mizar library. At the beginning of a verification we have to look at which kinds of algebraic domains are already included in the library and which theorems about these domains have been proven.
  We did some experiments using Glimpse [5], a powerful indexing and query system: After indexing the files—the Mizar abstracts in our case—it allows one to look through these files without the need of specifying file names. It enables the user to look for arbitrary keywords, for instance `gcdDomain`, `VectorSpace` or `finite-dimensional`.
– Though the Mizar system provides a proof script language capable of expressing algebraic structures appropriately, reasoning about these structures sometimes is a bit cumbersome. For example to prove equations in integral domains we had to do each little step using explicitly the domain's axioms. To handle equational reasoning there are well known better methods, for instance *rewriting systems*; for a couple of algebraic domains there even exist canonical rewrite systems. It seems promising to extend Mizar by such procedural proof techniques.

Writing generic algebraic algorithms is a difficult but rewarding task: one has to look for abstract algebraic domains suitable for the method one wants to implement; in addition using the constructed generic algorithms with particular instantiations again raises nontrivial algebraic questions.

Consequently, writing correct generic algebraic algorithms requires a careful way of dealing with the underlying mathematical structures. We hope that our work is a first step toward supporting a rigorous development of provable correct generic algebraic algorithms.

Acknowledgments

I would like to acknowledge the exposition to the subject and the supervision of my thesis by Prof. Rüdiger Loos. The suggestion of the Mizar system came from Prof. Sibylle Schupp and Prof. David Musser. The latter also provided numerous technical and grammatical improvements. Finally I am grateful for the hospitality of the Mizar group in Białystok—in particular Andrzej Trybulec—in summer 1998.

References

1. George E. Collins, *Algebraic Algorithms, chapter two: Arithmetics*, Lecture manuscript, 1974.
2. Antoni Diller, *Z—An Introduction to Formal Methods*, 2nd ed., Wiley, New York, 1994.
3. D. Knuth, *Literate Programming*, The Computer Journal 27(2), 97–111, 1984.
4. Rüdiger Loos and George E. Collins, *Revised Report on the Algorithm Description Language ALDES*, Technical Report WSI-92-14, Wilhelm-Schickard-Institut für Informatik, 1992.
5. Udi Manber and Burra Gopal, *GLIMPSE—A Tool to Search Entire File Systems*, available from `http://glimpse.cs.arizona.edu`.
6. Piotr Rudnicki, *An Overview of the Mizar Project*, available from `http://web.cs.ualberta.ca:80/~piotr/Mizar`, June 1992.
7. Sibylle Schupp, *Generic Programming—SuchThat One Can Build an Algebraic Library*, Ph.D. thesis, University of Tübingen, 1996.
8. Christoph Schwarzweller, *Mizar Verification of Generic Algebraic Algorithms*, Ph. D. thesis, University of Tübingen, 1997.
9. Andrzej Trybulec, *Some Features of the Mizar Language*, available from `http://web.cs.ualberta.ca:80/~piotr/Mizar`, July 1993.

A Proof of the Theorem

In this appendix we present the Mizar proof of the theorem given in section three. We do so for two reasons: First we want to give an impression of how naturally proofs can be formulated in the Mizar language. Also we want to show how documenting proofs using literate programming tools (see [3]) can make even long proofs readable. The complete correctness proof of the example algorithm can be found in [8].

Note that the use of literate programming allows extraction of the Mizar proofs from our document giving the corresponding Mizar article. In general this leads to a proper Mizar article that can serve as input to the Mizar checker.

We start the proof establishing that `gcd(r1,r2,Amp)` and `gcd(s1,s2,Amp)` both equal `1.G`, which follows from the definition of `is_normalized_wrt`.

⟨ **proof of the theorem 13** ⟩
```
   proof
   M: now assume
   H0: Amp is_multiplicative &
       r is_normalized_wrt Amp & s is_normalized_wrt Amp &
       r1 = num(r) & r2 = denom(r) & r2 <> 0.G &
       s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
       d = gcd(r2,s2,Amp) & d = 1.G &
       t = fract(r1*s2+r2*s1,r2*s2);
   H3: r2*s2 <> 0.G by H0,VECTSP_2:15;
   H1: denom(t) = r2*s2 by H0,H3,F1;
   H2: num(t) = r1*s2+r2*s1 by H0,H3,F1;
   H4: gcd(r1,r2,Amp) = 1.G & gcd(s1,s2,Amp) = 1.G
◇      by H0,Def73;
```
Also defined in: 14, 15, 16
Scrap referenced in: 6

To show `gcd(num(t),denom(t),Amp) = 1.G` we apply the theorem of Brown and Henrici—which is stated as theorem GCD:40. This is done by extending the term `gcd(r1*s2+r2*s1,r2*s2,Amp)`—which in fact is nothing more than `gcd(num(t),denom(t),Amp)`—to the form the theorem requires. After this application the assumption `gcd(r2,s2,Amp) = 1.G` allows us to infer that the original term also equals `1.G`.

⟨ **proof of the theorem 14** ⟩
```
   H5:   gcd(r1*s2+r2*s1,r2*s2,Amp)
       = gcd(r1*(s2/1.G)+r2*s1,r2*s2,Amp)            by GCD:10
      .= gcd(r1*(s2/1.G)+s1*(r2/1.G),r2*s2,Amp)      by GCD:10
      .= gcd(r1*(s2/1.G)+s1*(r2/1.G),
             r2*(s2/1.G),Amp)                        by GCD:10
      .= gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
             r2*(s2/gcd(r2,s2,Amp)),Amp)                 by H0
      .= gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
             gcd(r2,s2,Amp),Amp)                by GCD:40,H4,H0
◇     .= 1.G                                        by H0,GCD:32;
```
Also defined in: 13, 15, 16
Scrap referenced in: 6

Next we show that `denom(t) = r2*s2` is an element of the ample set `Amp`. This follows from the assumption that `r` and `s` are normalized fractions. Note that we need `Amp` to be multiplicative to conclude `r2*s2` ∈ `Amp` at level `H6`.

⟨ **proof of the theorem 15** ⟩
```
    H8: r2 ∈ Amp & s2 ∈ Amp by H0,Def73;
    reconsider r2,s2 as Element of Amp by H8;
    H6: r2*s2 ∈ Amp by H0,GCD:def 9;
    H7: t is_normalized_wrt Amp by H6,H5,H2,H1,Def73;
```
◇

Also defined in: 13, 14, 16
Scrap referenced in: 6

It remains to show that `t~r+s`. To do so, we only have to take into consideration that `num(r+s) = r1*s2+s1*r2` and `denom(r+s) = r2*s2` hold and to substitute `r2 = 1.G` and `s2 = 1.G` respectively to get the desired equation `num(t)*denom(r+s) = num(r+s)*denom(t)`.

⟨ **proof of the theorem 16** ⟩
```
    H9: num(t)*denom(r+s) = (r1*s2+r2*s1)*denom(r+s)  by H2
                         .= (r1*s2+r2*s1)*(r2*s2)     by H0,F2
                         .= num(r+s)*(r2*s2)          by H0,F2
                         .= num(r+s)*denom(t)         by H1;
    H13: t ~ (r+s) by H9,Def76;
    thus thesis by H13,H7;
    end;  :: M
    thus thesis by M;
    end;
```
◇

Also defined in: 13, 14, 15
Scrap referenced in: 6

Language Independent Container Specification*

Alexandre V. Zamulin

Institute of Informatics Systems
Siberian Division of the Russian Academy of Sciences
630090, Novosibirsk, Russia
zam@iis.nsk.su

Abstract. An approach to the specification of STL components in terms of mutable objects is proposed in the paper. The approach is based on considering an object update as a transition from one algebra of a given signature to another of the same signature. Each object (iterator, container) belongs to a definite object type and possesses a state and a unique identifier represented by a typed address. Iterators are divided in different iterator categories. Containers are collections of iterators of the corresponding category. Transition rules of an Abstract State Machine are used as a means of container specification.

Keywords: generic programming, standard template library, formal methods, abstract state machines, object types, transition rules.

1 Introduction

The semantics of the Standard Template Library (STL) components is currently given only informally as a set of requirements stated partly in English and partly as C++ program fragments, in both the official C++ standard [1] and books such as [2]. As a result, the semantics remains incomplete and imprecise, heavily depending on reader's (and library implementor's) intuition and knowledge of C++. Therefore, a formal description of STL independent of a particular programming language is needed.

One way of formal STL definition would be the use of classical algebraic specifications, whose advantages are a sound mathematical foundation and existence of specification languages and tools [3]. However, such STL notions as *iterator* and *container* subsume the notion of *state* which can change when containers are searched or updated. The modeling of the state by classical algebraic specifications involves extra data structures representing the state which must be explicitly passed as function arguments and yielded as function results. As a result, the specification becomes very complex, and the differences between containers are difficult to capture.

* This research is supported in part by Russian Foundation for Basic Research under the grant 98-01-00682.

Another way is the use of Abstract State Machines [4,5] whose advantages are built-in notion of state and imperative style of specification by means of transition rules describing state transformations. Unfortunately, classical ASMs are too low-level and do not possess generic specification facilities. As a result, neither iterator categories nor containers can be conveniently specified.

In this paper we demonstrate the use of the third technique called Object-Oriented Abstract State Machines which combines the advantages of both approaches mentioned above. It permits the conventional specification of a needed set of data types and specification by transition rules of a needed set of object types. Due to space limitations, the technique is explained in the paper rather informally. One can find formal definitions of the necessary concepts in [6,7]. For the same reason, only vect or and list type containers are specified on the basis of their informal description given in [2].

The paper is organized in the following way. Data types defining immutable values and data type categories specifying sets of data types are introduced in Section 2, and generic data types constrained by type categories are described in Section 3. Object types defining mutable objects are discussed in Section 4. Transition rules used as a means of object type specification are described in Section 5. Iterator types and categories are specified in Section 6. A generic vector type and list type are specified in Section 7 and Section 8, respectively. Some related work is discussed in Section 9. Finally, some conclusions and directions of further research are outlined in Section 10.

2 Data Types and Type Categories

The specification of the state (instance algebra) of a dynamic system is a typical many-sorted algebraic specification. Therefore, any specification language whose semantics is a class of many-sorted algebras is suitable for our purpose. However, to make the specifications of static entities (viz. data types) and dynamic entities (viz. objects) as similar as possible, we prefer to work with type-structured algebraic specifications [6]. Thus, a type-structured specification is a set of data type specifications having the following form:

> **type** Data-type-name = **spec**
> [set-of-operation signatures]
> {set-of-data-axioms}

An operation signature has the following form: *operation-name: operation-profile*; where *operation-profile* is either T (constant profile) or $T_1, \ldots, T_n \longrightarrow T$ (function profile) where T, $T_1, \ldots T_n$ are type names. A special symbol "@" can be used as a type name in an operation profile. It denotes the data type being specified and corresponds to "Self" of [8]. In a particular algebra, a set of elements is associated with a data type name, and an element of the corresponding set (in case of a constant profile) or a function (in case of a function profile) is associated with an operation name.

A data axiom is a pair $t_1 == t_2$ where t_1 and t_2 are two terms of the same type. A model (algebra) of a given specification must satisfy all its data axioms. In such an algebra, a set of elements is associated with a data type name and a function is associated with an operation name. The set of elements associated with a data type name T in an algebra A is denoted by A_T and the function associated with a function name f in A is denoted by f^A. A function can be partial, therefore not e very term can be defined in the algebra. To express the definedness of a term of a type-structured specification, we use a special semantic predicate **D**, such that $\mathbf{D}(t)$ states that the term t is defined in an algebra A if and only if the interpretation of the term, t^A, produces some object in A. In a data type specification, the clause **dom** specifies the domain of a partial function: for a domain specification **dom** $t : b$, the predication $\mathbf{D}(t)$ holds in an algebra A if and only if b evaluates in A to $true^A$.

A *data type category* specification defines a set of operations common to a group of data types. A data type specification indexed with the name of a data type category inherits the specification of this category. The specification of the data type category resembles the data type specification. However, no algebra provides a single function for an operator introduced in a data type category specification. Instead, it is bound to particular functions in the implementations of data types indexed with the name of this data type category . Example:

typecat EQUAL = **spec** – *category of all data types with equality operation*
["=", "<>": @, @ ⟶ Boolean]
{**forall** x, y: @, **exist** z: @.
x = x == true; x = y == y = x;
x = z & z = y == x = y; x <> y == ¬(x = y)}.

A data type category can also be indexed with the name of another data type category. According to the terminology of object-oriented systems, an indexing category is called a *basic category* and an indexed category is called a *subcategory.* A subcategory inherits the specification of its basic category. Example:

typecat ORDERED = **spec** EQUAL - *the specification of EQUAL is inherited*
["<=", ">=", "<", ">": @, @ ⟶ Boolean]
{**forall** x, y: @, **exist** u: @.
x <=x == true; x <= u & u <= y == x <= y;
x <= y & y <= x == x = y; x < y == x <= y & ¬(x = y);
x >= y == y <= x; x > y == y < x}.

3 Generic Types

A data type can be generic, i.e., it can use type parameters in its specification. The specification of a generic address type can be used as an example. It is assumed that the computer memory is composed of units (bytes) enumerated by natural numbers. An address of a given type marks a location occupying several bytes. It is assumed that for each data type there is a function *Size* producing the number of bytes which a value of the type occupies.

type Addr(T) = **spec** ORDERED - *a set of location addresses of the same type, the specification of the type category "ORDERED" is inherited*
[create_addr: Nat ⟶ @; – *creates a new address of type T*
_+_, _–_: @, Nat ⟶ @; – *several addresses forward or back*
advance, retreat: @ ⟶ @; – *one address forward or back*
_[_]: @, Nat ⟶ @; – *another version of the "+" operation*
base_nat: @ ⟶ Nat; – *original address*
difference: @, @ ⟶ Nat; – *distance between two addresses*]
{**forall** x: T, k, n: Nat, v, v1: @.
v + k == create_addr(base_nat(v) + k); v – k == create_addr(base_nat(v) – k);
advance(v) == v + 1; retreat(v) == v – 1;
v[n] == v + n; base_nat(create_addr(k)) == k;
difference(v, v1) == (base_nat(v) – base_nat(v1))/Size(T);
v = v1 == base_nat(v) = base_nat(v1); v < v1 == base_nat(v) < base_nat(v1)}.

Generic functions in addition to generic types can also be specified. For example, the following generic function serves for converting an address of one type into an address of another type:

convert_addr: Addr(T) ⟶ Addr(T1)
{**forall** ad: Addr(T).
dom convert_addr(T, T1)(ad): size(T1) <= size(T) & mod(size(T1), size(T2)) = 0;
convert_addr(T, T1)(ad) == Addr(T1)'create_addr(base_nat(ad))}.

Note the clause **dom** in the above specification. It indicates that the function is partial and it is defined only for those address types where the size of a resulting location is shorter than the size of an argument location. At the right-hand side of the axiom the operation *create_address* is prefixed with the data type name to indicate the type of the address to be created.

4 Object Types

We distinguish between *data types* and *object types*. A data type defines a set of immutable values and operations over them. An object type defines a set of states of a potentially mutable object and a number of methods capable to observe and update the object's state.

An object type specification has the following form:

class Object-type-name = **spec**
[set-of-attribute-signatures; set-of-mutator-signatures; set-of-observer-signatures]
{set-of-data-or-object-axioms}.

An attribute or observer signature has the same form as a data type operation signature where both data type names and object type names can be used as profile components. A mutator signature is either just a mutator name or *mutator-name: mutator-profile* where *mutator-profile* is a sequence of data/object type names. The special symbol “@” denoting the object type being specified can also be used in any of the above signatures.

Intuitively, a tuple of attributes defines an object’s state, an observer is a function computing something at a given object’s state, and a mutator is a procedure changing an object’s state. Attributes are often called *instance variables*, and observers and mutators are often called *methods* in object-oriented programming languages.

A data-or-object-axiom is either a data axiom or an object axiom of the form $t_1 == t_2$ where t_1 and t_2 are transition terms explained below.

In a particular algebra A,

- an object type name T is mapped to a set of elements A_T; the elements are called *object identifiers*;
- an attribute name $at : T'_1, \ldots, T'_n \longrightarrow T'$ declared in the signature of an object type T is mapped to a (partial) function $at^A_T : A_T \longrightarrow (A_{T'_1}, \ldots, A_{T'_n} \longrightarrow A_{T'})$; such a function is called an *attribute function.* If $id \in A_T$, then $at^A_T(id)$ is an *attribute* of *id.*

Such an algebra extending the algebra of data types is called an *instance algebra.* It represents some state of a number of objects. An *object* is a pair (id, obs) where *id* is an object identifier and *obs* is a tuple of its attributes called the *object’s state.* An update of an object’s state as well as creation or deletion of an object leads to the transformation of one instance algebra into another.

Actually, any set can be used as the set of object identifiers. We assume in this paper that, for each object type, the addresses of the corresponding type are used as object identifiers. Therefore, any operation applicable to an address is applicable to an object identifier.

To define the interpretation of observer and mutator names, we need to introduce the notion of *object-oriented system.*

Let now *OID* be a set of object identifiers, and $|D(A')|$ be a set of instance algebras of the same signature satisfying the following conditions: (i) all algebras in $|D(A')|$ have the same algebra of data types A', and (ii) if T is an object type name and A and B are two instance algebras, then both A_T and B_T are subsets of *OID.* Then an object-oriented system $D(A')$ is:

1. set of object identifiers *OID*;
2. set of instance Σ-algebras $|D(A')|$;
3. set of *observers* each producing a value of the result type for a given instance algebra and a set of arguments if any;
4. set of *mutators* each producing a new instance algebra for a given instance algebra and a set of arguments if any.

There are several rules for creating terms of object types. A term denoting an identifier associated with an object of type T is a term of type T. Since addresses are used as object identifiers, such a term is also a term of type $Addr(T)$, and it is always regarded as an address when it is a subject of an address type operation. A special kind of term called *transition terms* is introduced to denote transitions from one algebra to another. The main rules for creating the terms are the following (we omit the interpretation where it is self-evident).

1. If $at : T_1, \dots, T_n \longrightarrow T'$ is an attribute/observer signature from the signature of an object type T, $t_1, \dots, t_n$ are terms of types $T_1, \dots, T_n$, respectively, and t is a term of type T, then $t.at(t_1, \dots, t_n)$ is a term of type T'. The term is interpreted by invoking the corresponding attribute/observer function.
2. If t is a term of type T, then $\mathbf{D}(t)$ is a term of type Boolean. **Interpretation**: $\mathbf{D}(t)^A = true^A$ if t is defined in A, and $\mathbf{D}(t)^A = false^A$ otherwise. The interpretation of this term allows us to check whether the argument term is defined in a given instance algebra.
3. If $m : T_1, \dots, T_n$ is a mutator signature from the signature of an object type T, $t_1, \dots, t_n$ are terms of types $T_1, \dots, T_n$, respectively, and t is a term of type T, then $t.m(t_1, \dots, t_n)$ is a transition term called a *mutator call.* This kind of term serves for indicating an update of a mutable object. The term is interpreted by invoking the corresponding mutator function.
4. If T is an object type name, f is the name of a function with the profile $T_1, \dots, T_n \longrightarrow T$, $t_1, \dots, t_n$ are terms of types $T_1, \dots, T_n$, respectively, then $f(t_1, \dots, t_n) := new(T)$ is a transition term. The interpretation of this transition term leads to the creation of a new object identifier with totally undefined attribute functions.

5 Transition Rules

Algebra updates are specified by means of special *transition rules.* A transition rule is a special kind of transition term. It is applicable only to a *dynamic* function (constant) which can evolve from one algebra to another. Among the functions defined above, an object attribute function is a dynamic function.

Basic transition rules. Let $f : T_1, \dots, T_n \longrightarrow T$ be a dynamic function signature, t_i, $i = 1, \dots, n$, be a ground term of type T_i and t be a ground term of type T. Then

$f(t_1, \dots, t_n) := t$ and $f(t_1, \dots, t_n) := undef$

are transition rules called *primitive update instructions.* The interpretation of the first transition rule in an algebra A transforms it into an algebra B so that $f(t_1, \dots, t_n) = t$ in B, and the interpretation of the second transition rule in an algebra A transforms it into an algebra B so that $f(t_1, \dots, t_n)$ is undefined in B.

Examples. Let x be a dynamic constant of type *Nat* and f be a dynamic function from *Nat* to *Nat.* The transition rule

$f(x) := f(x) + 1$

will transform an algebra A into an algebra B so that $f^B(x^A) = f^A(x^A) + 1$. A transition rule

$x := undef$

will make x undefined in the new algebra.

If $at : T'$ is an attribute signature from an object type T, o is a ground term of type T and t is a ground term of type T', then the transition rules
$o.at := t$ and $o.at := undef$
are interpreted as $at(o) := t$ and $at(o) := undef$.

If we have two dynamic constants, $o1$ and $o2$, of the same object type T, then the transition rule $o1 := o2$ will force both of them to have the same object identifier, which provides for *object sharing.*

Several rule constructors serve for constructing transition rules recursively from update instructions defined above. Here we describe in short the *sequence constructor*, *set constructor*, *guarded update*, and *loop constructors.* One can find more in [5,6].

Sequence constructor. If $R_1, \ldots, R_n$ are transition rules, then

$$\textbf{seq } R_1, \ldots, R_n \textbf{ end}$$

is a rule called *sequence transition rule.* Semantics: to execute a sequence of rules starting with an algebra A, execute the rules one after another.

Set constructor. If $R_1, \ldots, R_n$ are rules, then

$$\textbf{set } R_1, \ldots, R_n \textbf{ end}$$

is a rule called a *set transition rule.* Semantics: to execute a set of rules, execute all of them in parallel and unite the results.

Example: Let x, y, z be dynamic constants of type *Nat* and f be a dynamic function *Nat* to *Nat.* Then the execution of a set of rules:
set $f(x) := y, y := x, x := z$ **end**
will produce: $f^B(x^A) = y^A$; $y^B = x^A$; $x^B = z^A$.

A **guarded update** instruction is a rule of the form **if** g **then** R where R is a rule. Semantics: execute the rule if the condition evaluates to *true* and do nothing in the opposite case.

Loop constructors. The guarded update together with the sequence constructor gives us a possibility to define some loop constructors. If R is a rule and g is a Boolean term, then
while g **do** R and **do** R **until** g
are transition rules. Semantics:
$(\textbf{while } g \textbf{ do } R)^A = (\textbf{if } g \textbf{ then seq } R, \textbf{ while } g \textbf{ do } R \textbf{ end})^A$;
$(\textbf{do } R \textbf{ until } g)^A = (\textbf{seq } R, \textbf{ if} \neg g \textbf{ then do } R \textbf{ until } g)^A$.

A **massive update** permits the specification of a parallel update of one or more functions at several points. It has the following form:

forall $x_1 : T_1, \ldots, x_n : T_n.\ R$

where $x_1, \ldots, x_n$ are bound variables of types $T_1, \ldots, T_n$, respectively, and R is a transition rule having no free variables. The rule is interpreted by the parallel execution of all transition rules obtained by replacing variables with the ground terms of the corresponding types.

6 Iterator Types and Categories

In addition to data type categories, object type categories can also be defined. Example:

classcat Equal = **spec** — *category of object types with an equality operation*
[**observer** equal, not_equal: @ ⟶ Boolean]
{**forall** x, y: @, **exist** z: @.
x.equal(x) == true; x.equal(y) == y.equal(x);
x.equal(z) & z.equal(y) == x.equal(y); x.not_equal(y) == ¬x.equal(y)}.

A (data/object) type category can be generic, i.e., it can use type parameters in its specification. If $C(P_1, \ldots, P_n)$ is the heading of a generic type category and $T_1, \ldots, T_n$ are type names, then $C(T_1, \ldots, T_n)$ is a type category obtained by replacing each P_i with T_i in the specification of C. Iterator categories serve as examples.

1. Each iterator type of the following category has operations *advance* and *get_value* in addition to the operations of the category "Equal".

classcat InputIterator(T) = **spec** Equal
[**observer** advance: @; – *advances an iterator one position forward*
get_value: T] – *dereferencing operation for reading*

2. Each iterator type of the following category has operations *advance* and *put_value* in addition to the operations of the category "Equal".

classcat OutputIterator(T) **spec** Equal
[**observer** advance: @; – *advances an iterator one position forward*
mutator put_value: T] – *stores a new value in an iterator*

3. Each iterator type of the following category has a mutator *put_value* in addition to the operations of the category "InputIterator".

classcat ForwardIterator(T) = **spec** InputIterator(T)
[**mutator** put_value: T]

4. Each iterator type of the following category has the operation *retreat* in addition to the operations of the category "ForwardIterator".

classcat BidirectionalIterator(T: TYPE) = **spec** ForwardIterator(T)
[**observer** retreat: @] – *advances an iterator one position backward*

5. Each iterator type of the following category has several operations in addition to the operations of the category "BidirectionalIterator".

```
classcat RandomAccessIterator(T) = spec BidirectionalIterator(T)
 [observer plus, minus: Nat ⟶ @;
       difference: @ ⟶ Nat;
       less, greater, leq, geq: @ ⟶ Boolean]
 {forall i, i1: @, n: Nat.
  i.plus(n) == i+n; i.minus(n) == i–n;
  i.difference(i1) == i–i1;
  i.less(i1) == i < i1; i.greater(i1) == i > i1;
  i.leq(i1) == i <= i1; i.geq(i1) == i >= i1}.
```

Note that in the above specification, both i and $i1$ are interpreted as object identifiers in the left-hand side of the axioms, and they are interpreted as addresses in the right-hand sides of the axioms.

The same principle of genericity is used for defining generic object types. The specification of a vector iterator given below can serve as an example. We represent a vector iterator as an object possessing all methods of the RandomAccessIterator category and the attribute *value_stored*:

```
class VecIt(T) = spec RandomAccessIterator(T)
 [attribute value_stored: T]
 {forall i: @, x: T.
  i.put_value(x) == i.value_stored := x; i.get_value == i.value_stored;
  i.advance == i+1; i.retreat == i–1}.
```

A vector iterator thus specified belongs to the RandomAccessIterator category and consequently to the BidirectionalIterator category and so on up to the Equal category.

The specification of a list iterator is another example. We represent a list iterator as an object possessing all methods of the category "BidirectionalIterator" and the attributes *value_stored*, *pred* and *next*:

```
class ListIt(T) = spec BidirectionalIterator(T)
 [attribute value_stored: T;
       pred, next: @]
 {forall i: @, x: T.
  i.put_value(x) == i.value_stored := x; i.get_value == i.value_stored;
  i.advance == i.next; i.retreat == i.pred}.
```

A list iterator thus specified belongs to the BidirectionalIterator category and thus to the ForwardIterator category and so on up to the Equal category.

Note the different definitions of the operations *advance* and *retreat* in the above specifications.

7 Vector Types

Now we are ready to give the specification of a generic vector type. It is assumed that vector elements are iterators and they are numbered starting from zero.

```
class Vector(T) = spec Ordered
 [attribute comp: Nat ⟶ VecIt(T); – vector elements, initially undefined
      size: Nat; – current size of a vector, initial value is zero
  mutator empty_vec; – default constructor
      initialized_vec: Nat, T; – constructor 1
      copy: @; – copy constructor
      push_back: T; – insert an element at the end of a vector
      pop_back; – delete the last vector's element
      insert1: VecIt(T), T; – insert an element at the position indicated
      insertN: VecIt(T), Nat, T; – insert n elements at the position indicated
      erase1: VecIt(T); – remove the element indicated
      eraseN: VecIt(T), VecIt(T); – remove the elements between the iterators
      swap: @; - swap the contents of two vectors
  observer empty: Boolean; – is a vector empty?
      max_size: Nat; – maximal size of a vector
      _[_]: Nat ⟶ T; – fetch a vector's element
      front, back: T; – first and last elements of a vector
      begin, end: VecIt(T)] – starting and terminating iterators of a vector
 {forall x, x1: T, iv, iv1: VecIt(T), n, n1: Nat, v, v1: @.
  dom v[n]: n >= 0 & n <= v.size;
  dom v.initialized_vec (n, x): n <= v.max_size;
  dom v.push_back(x): v.size < v.max_size;
  dom v.insert1(iv, x): v.size < v.max_size & iv.geq(v.begin) & iv.less(v.end);
  dom v.insertN(iv, n, x): v.size+n ≤ v.max_size & iv.geq(v.begin) & iv.less(v.end);
  dom v.erase1(iv): iv.geq(v.begin) & iv.less(v.end) & v.size > 0;
  dom v.eraseN(iv, iv1): iv.geq(v.begin) & iv1.less(v.end) & v.size > 0 & iv.leq(iv1);
  v.empty_vec == v.size := 0;
  v.initialized_vec (n, x) == set v.size := n,
           forall i: Nat.if i < n then v.comp(i).value_stored := x, end;
  v.copy(v1) == v := v1;
  v.push_back(x) == set v.comp(v.size).value_stored := x; v.size := v.size + 1 end;
  v.pop_back == v.size := v.size - 1;
  v.insert1(iv, x) == forall i: Nat. set v.size := v.size + 1,
      if v.comp(i).geq(iv) & i < v.size
      then v.comp(i+1).value_stored := v.comp(i).value_stored;
      iv.value_stored := x end;
  v.insertN(iv, n, x) == forall i: Nat. set v.size := v.size + n,
      if v.comp(i).geq(iv) & i < v.size
      then v.comp(i+n).value_stored := v.comp(i).value_stored,
      if v.comp(i).geq(iv) & v.comp(i).less(iv.plus(n))
      then v.comp(i).value_stored := x end;
v.erase1(iv) == set forall i: Nat. if v.comp(i).geq(iv) & i <= v.size
      then v.comp(i).value_stored := v.comp(i+1).value_stored,
      v.size := v.size - 1 end;
v.eraseN(iv, iv1) == forall i: Nat.
      set if v.comp(i).geq(iv) & v.comp(i).less(iv1)
      then v.comp(i).value_stored := v.comp(i+iv1.difference(iv)).value_stored,
      v.size := v.size - (iv1.difference(iv)) end;
v.empty == v.size = 0;
v[n] == v.comp(n).value_stored; – n = 0 for the first component
v.begin == convert_addr(Vector(T), VecIt(T))(v);
v.end == v.begin.plus(v.size);
v.front == v.comp(0).value_stored; v.back == v.comp(v.size-1).value_stored }.
```

8 List Types

A list is ordered according to the use of constructor operations “push_back” (inserts an element at the end of a list), “insert”, and “remove”. All list elements are numbered starting with 1 for the first element. The number of the last element is equal to the number of the elements in the list. Due to space limitations, the specification of some operations is left to the reader. The import rule of Gurevich ASMs [5] is used in the specification for creating a new object identifier.

```
class List(T) == spec Ordered
 [attribute begin, end: ListIt(T);
      size: Nat; – current size of a list
  mutator empty_list; – empty list constructor
      initialized_list: Nat, T; – initialized list constructor
      copy: @; – copy constructor
      push_front: T; – insert an element at the beginning of a list
      push_back: T; – insert an element at the end of a list
      pop_front; – delete the first element of a list
      pop_back; – delete the last element of a list
      insert1: ListIt(T), T; – insert one element at the position indicated
      insertN: ListIt(T), Nat, T; – insert N elements at the position indicated
      erase1: ListIt(T); – remove the element indicated
      eraseN: ListIt(T), ListIt(T); – remove the elements between the iterators
  observ empty: Boolean; – is a list empty?
      front, back: T; – first and last elements of a list
      has: ListIt(T): Boolean; – an auxiliary operation checking
                  – the presence of an element in the list
      precede: ListIt(T), ListIt(T): Boolean] – an auxiliary operation verifying that
                  – the first iterator precedes the second one in a list
 {forall x, x1: T, n, n1: Nat, l, l1: List(T), i, i1: ListIt(T).
  dom l.pop_front: l.size > 0;   dom l.pop_back: l.size > 0;
  dom l.insert1(i, x): l.size > 0 & l.has(i);
  dom l.insertN(i, n, x): l.size > 0 & l.has(i);
  dom l.eraseN(i, i1): l.size > 0 & l.has(i) & l.has(i1) & l.precede(i, i1);
  dom l.front: l.size > 0;   dom l.back: l.size > 0;
  l.empty_list == import new: ListIt(T) in
      set l.begin := new, l.end := new, l.size := 0 end ;
           – an iterator beyond the list component iterators is created
  l.push_back(x) == import new: ListIt(T) in
      set new.pred := l.end, l.end.next := new, l.end.value_stored := x,
           l.end := new, l.size := l.size +1 end ;
  l.initialized_list(n, x) == seq l.empty_list, while l.size < n do l.push_back(x) end ;
  l.copy(l1) == seq sequence of transition rules end;
  l.push_front(x) == import new: ListIt(T) in set of transition rules;
  l.pop_front == set l.begin := l.begin.next, l.begin.next.pred := undef,
      size := size - 1 end;
  l.pop_back == set of transition rules;
  l.insert1(i, x) == import new: ListIt(T) in set i.next := new, i.next.pred := new,
      new.pred := i, new.next := i.next, new.value_stored := x, l.size := l.size +1 end;
  l.insertN(i, n, x) == seq l.insert1(i, x), if n > 1 then l.insertN(i.next, n-1, x) end ;
  l.erase1(i) == if l.size = 1 then l.pop_front – first element is deleted
      else set i.pred.next := i.next, i.next.pred:= i.pred, l.size := l.size - 1 end ;
  l.eraseN(i, i1) == set l.erase1(i), if i.next.not_equal(i1) then l.eraseN(i.next, i1) end ;
  l.front == l.begin.value_stored;  l.back == l.end.pred.value_stored;
  l.has(i) == Boolean term evaluating to "true" if l has i;
  l.precede(i, i1) == Boolean term evaluating to "true" if i precedes i1 in l}.
```

9 Related Work

We are not going to discuss here the approaches representing object states as elements of the same algebra. The works along this approach are heavily based on traditional algebraic specification methods. We can only repeat after F. Parisi-Presicce and A. Pierantonio that "the algebraic framework so far has been inadequate in describing the dynamic properties of objects and their state transformation as well as more complex notions typical of the object-oriented paradigm such as object identity and persist ency of objects" [10].

We will review several works considering object states as algebras. The first of them are object-oriented extensions of the prominent specification methods VDM and Z. These are VDM++, Object-Z, and Z++.

VDM++ [11] introduces the notion of a class definition as a template for a collection of objects possessing a number of instance variables (internal state) and methods (external protocol). However, no genericity is provided in the language, which does not allow its use for the specification of generic containers.

Object-Z [12] has practically the same features as VDM++ with the difference that it is based on Z. No notion of object type category and, respectively, constrained genericity exist in the language. Object creation is also not provided by the language. Thus, a specification of lists like the one given above is not possible.

Z++ [13,14] is another development based on Z which provides facilities comparable to those of Object-Z. Again, no notion of object type category and, respectively, constrained genericity exist in the language. The semantics of a generic class specification is also not reported.

The specification language Troll [15] should be mentioned as one of possible candidates. It is oriented toward the specification of objects where a method (event) is specified by means of evaluation rules similar to equations on attributes. Although the semantics of Troll is given rather informally, there is a strong mathematical foundation of its dialect Troll-light [16], with the use of data algebras, attribute algebras and event algebras. A relation constructed on two sets of attribute algebras and a set of event algebra, called *object community*, formalizes transitions from one attribute algebra into another. Although Troll possesses generic facilities, none of them is formalized in [16].

Finally, attention should be paid to the fundamental work [8] formalizing bounded parametric polymorphism similar to our constrained genericity. Here genericity is constrained by allowing only those type arguments which are subtypes of the parameter type. It is evident that our approach is more general since not every object type category can be defined as an object type. Another peculiarity of the work is that an object does not possess a unique identifier; it is just a tuple of methods, and object updates are simulated by method overrides generally producing new objects.

10 Conclusion

In this paper we have used the formalism of Object-Oriented Abstract State Machines for the formal definition of such basic STL components as iterators and containers. As a result, we have obtained strict and concise specifications of the components independent of any particular programming language. The imperative nature of the specifications, taking into account implicit state, permits us to hope that such specifications will be much more attractive for both implementors and users of STL than other formalisms.

Due to space limitation, not all containers are specified in the paper. Moreover, the specifications presented might not fully correspond to the original ideas of the STL authors since their informal definition could be misunderstood by the author of this paper. Therefore, the complete and consistent specification of the STL components remains the subject of further work.

Acknowledgment

The author thanks David Musser for a number of useful comments on the initial version of the paper.

References

1. ISO/IEC FDIS 14882. *International Standard for the C++ Programming Language*. American National Standards Institute (ANSI), X3 Secretariat, 1250 Eye Street NW, Suite 200, Washington, DC 20005, 1998.
2. D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.
3. M. Wirsing. Algebraic Specifications. *Handbook of Theoretical Computer Science*, Elsevier Science Publishers B.V., 1990, pp. 665–788.
4. Y. Gurevich. Evolving Algebras 1993: Lipary Guide. In: *Börger, E., ed., Specification and Validation Methods*, Oxford University Press, 1995, pp. 9–36.
5. Y. Gurevich. *May 1997 Draft of the ASM Guide*. University of Michigan (available electronically from `http://www.eecs.umich.edu/gasm/`).
6. A. V. Zamulin. *Typed Gurevich Machines Revisited*. Joint NCC&ISS Bull., Comp. Science, 5 (1997), pp. 1–26 (available electronically from `http://www.eecs.umich.edu/gasm/`).
7. A. V. Zamulin. *Object-Oriented Abstract State Machines*. Proc. ASM workshop, Magderburg, Germany, 21–22 September, 1998, pp. 1–21 (available electronically from `http://www.eecs.umich.edu/gasm/`).
8. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
9. P. Wadler and S. Blott. *How to make ad-hoc polymorphism less ad-hoc*. Conf. Record of the 16th ACM Annual Symp. on Principles of Progr. Lang., Austin, Texas, January 1989.
10. F. Parisi-Presicce and A. Pierantonio. *Dynamic Behavior of Object Systems*. In: Recent trends in Data Type Specification. LNCS, vol. 906, 1995, pp. 406–419.
11. E. H. Dürr and J. van Katwijk. *A Formal Specification Language for Object Oriented Designs*. In: Computer Systems and Engineering (Proceedings of CompEuro 92). IEEE Computer Society Press, 1992, pp. 214–219.
12. R. Duke, P. King, G.A. Rose, and G. Smith. *The Object-Z specification language*. In: T. Korson, V. Vaishnavi, and B. Meyer, eds., Technology of Object-Oriented Languages and Systems: TOOLS 5, Prentice Hall, 1991, pp. 465–483.
13. K. Lano and H. Houghton, eds., *Object-Oriented Specification Case Studies*. Prentice Hall (Object-Oriented Series), 1994.
14. K. Lano and H. Houghton. *The Z++ manual*. October 1994. Available from `ftp://theory.doc.ic.ac.uk/theory/papers/Lano/z++.ps.Z`.
15. T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kush. *Revised Version of the Modelling Language TROLL*. Technische Universität Braunschweig, Informatik-Berichte 94-03, 1994.
16. M. Gogolla and R. Herzig. *An Algebraic Semantics for the Object Specification Language TROLL-light*. In: Recent Trends in Data Type Specifications, LNCS, vol. 906, 1995, pp. 290–306.

Applications of the Generic Programming Paradigm in the Design of CGAL*

Hervé Brönnimann[1], Lutz Kettner[2], Stefan Schirra[3], and Remco Veltkamp[4]

[1] INRIA Sophia-Antipolis, France
Herve.Bronnimann@sophia.inria.fr
[2] ETH Zürich, Switzerland
kettner@inf.ethz.ch
[3] Max-Planck-Institute for Computer Science, Saarbrücken, Germany
stschirr@mpi-sb.mpg.de
[4] Dept. Computing Science, Utrecht University, The Netherlands
Remco.Veltkamp@cs.uu.nl

Abstract. We report on the use of the generic programming paradigm in the computational geometry algorithms library CGAL. The parameterization of the geometric algorithms in CGAL enhances flexibility and adaptability and opens an easy way for abolishing precision and robustness problems by exact but nevertheless efficient computation. Furthermore we discuss circulators, which are an extension of the iterator concept to circular structures. Such structures arise frequently in geometric computing.

1 Introduction

CGAL is a C++ library of geometric algorithms and data structures. It is developed by several sites in Europe and Israel. The goal is to enhance the technology transfer of the algorithmic knowledge developed in the field of computational geometry to applications in industry and academia.

Computational geometry is the sub-area of algorithm design that deals with the design and analysis of algorithms for geometric problems involving objects like points, lines, polygons, and polyhedra. Over the past twenty years, the field has developed a rich collection of solutions to a huge variety of geometric problems including intersection problems, visibility problems, and proximity problems. See the textbooks [15,23,18,21,6,1] and the handbooks [10,24] for an overview. The standard approach taken in computational geometry is the development of provably good and efficient solutions to problems. However, implementing these algorithms is not easy. The most common problems are the dissimilarity between fast floating-point arithmetic normally used in practice and exact arithmetic over the real numbers assumed in theoretical papers, the

* This work is partially supported by the ESPRIT IV LTR Projects No. 21957 (CGAL) and 28155 (GALIA), and by the Swiss Federal Office for Education and Science (CGAL and GALIA).

lack of explicit handling of degenerate cases in these papers, and the inherent complexity of many efficient solutions. As a result, many useful geometric algorithms have not yet found their way into the many application domains of computational geometry. Therefore, a number of institutions from the computational geometry community itself has started to develop CGAL [22]. The availability of a software library can make a tremendous difference. If there is no need anymore to implement the geometric algorithms from scratch, the effort of experimenting with the solutions developed in computational geometry is lowered.

The major design goals for CGAL include correctness, robustness, flexibility, efficiency and ease of use [7]. One aspect of flexibility is that CGAL algorithms can be easily adapted to work on data types in applications that already exist. The design goals, especially flexibility and efficient robust computation, led us to opt for the generic programming paradigm using templates in C++ [19], and to reject the object-oriented paradigm in C++ (as well as Java). In several appropriate places, however, we make use of object-oriented solutions and design patterns. Generic programming with templates in C++ provides us with the help of strong type checking at compile time, and also has the advantage that it doesn't cause runtime overhead. In the sequel we give examples of the use of the generic programming paradigm in CGAL.

2 Generic Programming in Geometric Computing

One of the hallmarks of geometry is the use of transformations. Indeed, geometric transformations link several geometric structures together [6,1]. For example, duality relates the problem of computing the intersection of halfplanes containing the origin to that of computing the convex hull of their dual points. The Voronoi diagram of a set of points is also dually related to its Delaunay triangulation, and this triangulation can be computed as a lower convex hull of the points lifted onto a paraboloid in one dimension higher.

For this kind of problem, the relevance of generic programming is obvious. Much like the problem of finding a minimum element in a sequence, the use of a geometric algorithm such as computing the convex hull can have many applications via geometric transformations. In this setting, the algorithm does not operate on the original objects but on their transformed version, and the primitives used by the algorithm must also be applied through the same transformation. This is achieved in CGAL through the use of *traits classes.*[1]

Another hallmark of geometric programming is the multiple choice of representations of a geometric object. Naturally, some representations are more suited to a particular problem. In computing Delaunay triangulations or minimum enclosing circles, for example, a circle will likely be represented implicitly as the circumcircle of three points. Other representations would include a pair of cen-

[1] In CGAL, the term 'traits class' is used in a more general setting than associating related type information to built-in types, which is the original usage [20].

ter and squared radius[2] (this is the default representation in CGAL). In most problems, a point would be represented by its coordinates, either Cartesian or homogeneous. Another choice, more suited to boolean operations on polygons for instance, might be as an intersection of two lines: such a representation is stable when performing an arbitrary number of boolean operations.

Finally, geometric computing has been successful in abstracting several paradigms of its own. For example, the sweep paradigm is used to compute arrangements of lines of segments, triangulations of polygons, and Voronoi diagrams. Randomized incremental algorithms have been abstracted in a framework [5,1] that uses few primitives. Geometric optimization problems have brought forth the generalized LP-type problems that have been so hugely successful in computing minimum enclosing circles and ellipses, distances between polytopes, linear programs, etc. In all three cases, one can write the skeleton of an algorithm that will work, given the appropriate primitives. The user then has to supply only those primitives via a traits class.

Algorithms in CGAL are generic, they work with a variety of implementations of predicates and representations of geometric objects. In the next subsection, we argue that generic programming is especially relevant to geometric computing.

3 Traits Classes and the CGAL Kernel

We briefly recall the main elements of the design [7] of the CGAL kernel [3]. The two basic representation classes in the CGAL kernel are `CGAL_Cartesian` and `CGAL_Homogeneous`. They provide basic geometric types like `Point_2` and `Segment_2`. Instead of a dense class hierarchy, these types are organized as a loosely coupled collection. For example, `PointC2` is a *model* for the abstract concept of a point (with internal representation as Cartesian coordinates). A model of a concept is an implementation that fulfills the requirements of this concept. A typedef declaration in the corresponding representation class `Cartesian` links `Cartesian::Point_2` with `PointC2`.

Genericity is extended by templating all these classes by the number type used to store the coordinates. This allows tailoring the kernel to the application, especially regarding robustness and precision concerns as described below in Section 6. The concept of a geometric kernel is currently extended to contain the predicates and constructions on the basic kernel objects. A kernel is a model of the concept of a geometric kernel as defined by the CGAL Reference Manual [3]. With this design, a model of a geometric kernel can be passed as a traits parameter to the algorithms of the CGAL basic library, as described in Section 5. This does not exclude users from providing an interface to their own library as a model of a CGAL kernel. For example, there are adaptations of the floating-point and the rational geometry kernel of LEDA [16].

[2] We use squared radius instead of radius, because the former is rational for a circle given by three points with rational coordinates, whereas the latter might be irrational.

4 Data Passing by Iterators and Circulators

A prominent example of the generic programming paradigm is the Standard Template Library (STL) accompanying the C++ standard. Algorithms interact with container classes through the concept of *iterators*, which are defined through a set of requirements. Each container class is supposed to provide a model for an iterator. The algorithms are parameterized with iterators, and can be instantiated with any model that fulfills the iterator requirements. So, containers and algorithms are kept independent from each other. New algorithms can be developed without knowing any container class, and vice versa.

For example, a range of points can be fed into an algorithm by providing two forward iterators `first` and `beyond`. They define a range `[first,beyond)`, if applying a finite number of times the operator `++` to `first` makes that `first == beyond`. The range refers to the points starting with `*first` up to but not including `*beyond`. The iterator `beyond` is said to point 'past the end' of the range. An application programmer need not copy the data from a favorite container into another container in order to feed it to the algorithm, as long as the container provides iterators. Inside the algorithm the points are addressed by advancing the iterator, independent of the particular container type.

In the following example, `min_encl_circle` is template-parameterized with a forward iterator:

```
template <class ForwardIterator>
Circle
min_encl_circle(ForwardIterator first, ForwardIterator beyond)
{ ... }
```

The concept of iterators in the STL is tailored for linear sequences. In contrast, circular sequences evolve naturally in many combinatorial and geometric structures. Examples are polyhedral surfaces and planar maps, where the edges emanating from a vertex or the edges around a facet form a circular sequence.

Since circular sequences cannot provide efficient iterators (for reasons discussed later), we have introduced the new concept of *circulators* in CGAL. They share most of the requirements with iterators, the main difference being the lack of a past-the-end position in the sequence. Appropriate adaptors are provided between iterators and circulators to seat circulators smoothly into the framework of the STL. We give a short introduction to circulators and discuss advantages and disadvantages thereafter.

The following example illustrates the use of a circulator in a generic function `contains` that tests whether a certain `value` exists in a circular sequence. As usual for circular structures, we use a `do-while` loop to reach all elements in the specific case `c == d`.

```
template <class Circulator, class T>
bool contains( Circulator c, Circulator d, const T& value) {
    if ( c != NULL) {
        do {
            if ( *c == value)
                return true;
        } while (++c != d);
    }
    return false;
}
```

Three circulator categories are defined: forward, bidirectional and random-access circulators. Given a circulator `c`, the operation `*c` denotes the item the circulator refers to. The operation `++c` advances the circulator by one item and `--c` steps a bidirectional circulator one item backwards. For random-access circulators `c+n` advances the circulator `n` times. Two circulators can be compared for equality.

Circulators have a different notion of reachability and ranges than iterators. A circulator `d` is called *reachable* from a circulator `c` if `c` can be made equal to `d` with finitely many applications of the operator `++`. Due to the circularity of the data structure this is always true if both circulators refer to items of the same data structure. In particular, `c` is always reachable from `c`. Given two circulators `c` and `d`, the range `[c,d)` denotes all circulators obtained by starting with `c` and advancing `c` until `d` is reached, but does not include `d` for $d \neq c$. So far it is the same range definition as for iterators. The difference lies in the use of `[c,c)` to denote all items in the circular data structure, whereas for an iterator `i` the range `[i,i)` denotes the empty range. As long as `c != d` the range `[c,d)` behaves like an iterator range and could be used in STL algorithms. For circulators however, an additional test `c == NULL` is required that returns true if and only if the data structure is empty. In this case the circulator `c` is said to have a *singular value*.

Supporting both iterators and circulators within the same generic algorithm is just as simple as supporting iterators only. This and the requirements for circulators are described in the CGAL Reference Manual [13].

The main reason for inventing a new concept with the same goals as iterators is efficiency. An iterator is supposed to be a light-weight object—merely a pointer and a single indirection to advance the iterator. Although iterators could be written for circular sequences, we do not know of an efficient solution. The missing past-the-end situation in circular sequences can be solved with an arbitrary sentinel in the cyclic order, but this would destroy the natural symmetry in the structure (which is in itself a bad idea) and additional bookkeeping in the items and checking in the iterator advance method reduces efficiency. Another solution may use more bookkeeping in the iterator, e.g., with a start item, a current item, and a kind of winding-number that is zero for the `begin()`-iterator and one for the past-the-end situation. Though the concept of circulators allows light-weight implementations, the CGAL support library provides adaptor classes that convert between iterators and circulators (with the corresponding penalty in efficiency), so as to integrate this new concept into the framework of the STL.

A serious design problem is the slight change of the semantic for circulator ranges as compared to iterator ranges. Since this semantic is defined by the intuitive operators `++` and `==`, which we would like to keep for circulators as well, circulator ranges can be used in STL algorithms. This is in itself a useful feature, if there would not be the definition of a full range `[c, c)` that an STL algorithm will treat as an empty range. However, the likelihood of a mistake may be overestimated, since for a container `C` supporting circulators there is no `end()` member function, and an expression such as `sort( C.begin(), C.end())` will fail. It is easy to distinguish iterators and circulators at compile time, which allows for generic algorithms supporting both as arguments. It is also possible to protect algorithms against inappropriate arguments with the same technique, though it is beyond the scope of CGAL to extend STL algorithms.

5 Generic Geometric Algorithms Using Traits Classes

Also at a higher level, CGAL uses traits classes. In algorithms, primitives are encapsulated in traits classes, so that application specific primitives and specialized versions can be plugged in. To illustrate this, consider the problem of computing the minimum enclosing circle of a set of points. This problem is generally motivated as a facility location problem, or in robotics, as how to anchor a robot arm so as to minimize its range under the constraint that it must reach all specified locations, modeled as points.

Suppose now that those points are at different heights and that the robot arm can slide along a vertical axis. To test whether a number of points in space can be reached by this robot arm, it is possible to compute the smallest enclosing vertical cylinder, and check whether the radius is not larger than the length of the robot arm. For the sake of simplicity we ignore possible collisions.

The smallest enclosing circle is calculated in the two-dimensional plane, but the points are three-dimensional. Of course we could make a version of the smallest enclosing circle algorithm that operates on three-dimensional points, but generates the two-dimensional circle. This is against the principle of code reuse, one of the hallmarks of modern programming. Alternatively, we could perform the projection explicitly, generate two-dimensional points, apply the two-dimensional algorithm, and maintain the correspondence between the three-dimensional and the two-dimensional points if needed. This requires extra bookkeeping and space, and is not so elegant.

In general, the smallest enclosing circle algorithm can treat points in an abstract way using predicates like `side_of_bounded_circle(p,q,r,test)` to test if the point `test` lies on the bounded side (inside), on the unbounded side, or on the boundary of the circle through `p`, `q` and `r` (assuming `p`, `q` and `r` are not collinear). So, the algorithm can be parameterized by point type and predicates. CGAL does this: the point type and predicates are put into what CGAL calls a traits class. In the following example, the algorithm `min_encl_circle` is template-parameterized with the ForwardIterator and the Traits class:

```
template <class ForwardIterator, class Traits>
Circle
min_encl_circle(ForwardIterator first, ForwardIterator beyond,
                Traits traits);
```

Rather than explicitly working with a CGAL point type, the algorithm internally works with `traits::Point_2`. Likewise, instead of using a global function for the `side_of_bounded_circle(p,q,r,test)` predicate, it uses a function object which is accessible through a member function of the traits class. This function object provides the `operator() (traits::Point_2 p, traits::Point_2 q, traits::Point_2 r, traits::Point_2 test)`.

```
template <class ForwardIterator, class Traits>
Circle
min_encl_circle(ForwardIterator first, ForwardIterator beyond,
                Traits traits) {
  typedef traits::Point_2  Point_2;  // local type for points
  for ( ; first != beyond; ++first ) {
      // ... using traits.side_of_bounded_circle()( p, q, r, test)
  }
  // return result
}
```

An application programmer can use any point type by defining an appropriate traits class `My_traits` with a proper `My_side_of_bounded_circle`:

```
struct My_side_of_bounded_circle {
 bool
 operator()(My_point *p, My_point *q, My_point *r, My_point *test)
 { ... }
};

struct My_traits {
  typedef My_point                  Point_2;
  typedef My_side_of_bounded_circle  Side_of_bounded_circle;
  Side_of_bounded_circle side_of_bounded_circle() const {
      return Side_of_bounded_circle();
  }
} my_traits;
```

The use of a member function to access the predicate allows one to pass additional data from a traits object to the predicate object in order to influence the predicate's behavior. The smallest enclosing circle algorithm can be invoked with this traits class:

```
vector<My_point> points(num);
Circle min_encl_circle(points.begin(), points.end(), My_traits());
```

In this way, specialized predicates can be used that work on 3D points, but evaluate the orientation on projections of those points, leaving the algorithm itself unchanged. CGAL uses traits classes for algorithms and for data structures throughout the basic library, and provides traits classes for common use, such as the two CGAL kernels and projections of them, or the LEDA geometry kernels. A default argument even hides the mechanism of the traits classes from inexperienced users.

6 Generic Programming Eases Efficient Robust Geometric Computing

Since the field of computational geometry has its roots in theoretical computer science, algorithms developed in computational geometry are designed for a theoretical machine model, the so-called real RAM model [23]. In a real RAM, exact computation with arbitrary real numbers is assumed (with constant cost per arithmetic operation). In practice, however, the most popular substitution for computation with real numbers in scientific computing is floating-point computation. Floating-point arithmetic is fast, but not necessarily exact. Depending on the type of geometric problem to be solved, implementations of theoretically correct algorithms frequently produce garbage, or crash, due to rounding errors in floating-point arithmetic. The reason for such crashes with floating-point arithmetic are inconsistent decisions leading the algorithms into states they never could get into with exact arithmetic [12,25]. Exact computation has been proposed to resolve the problem [28]. This approach is appealing, since it lets an implementation behave exactly as its theoretical counterpart. No redesign for a machine model that takes imprecise arithmetic into account is necessary. Fortunately, exact computation in the sense of guaranteeing correct decisions is, at least in principle, possible for many geometric problems. It is, however, also known to be expensive in terms of performance. In this section we discuss how parameterization opens the way for affordable efficient exact computation.

In many cases, geometric algorithms are already described in layers. Elementary tasks are encapsulated in geometric primitives. As described above, implementations of geometric algorithms in CGAL are generic with respect to the concrete implementation of geometric primitives. A traits class specifies the concrete types that are actually used. Most interesting with respect to the precision and robustness issue are geometric predicates, e.g., checking whether three points form a left turn. Naturally, the correctness of the overall algorithm depends on the correctness of the primitive operations.

CGAL provides generic implementations of such primitive operations that allow one to exchange the arithmetic. Both currently available CGAL kernels, `CGAL_Cartesian` and `CGAL_Homogeneous`, are parameterized by a number type. The actual requirements on the number type parameter vary with the primitive. For example, division operations are avoided in most primitives for the homogeneous kernel. Thus, a number type need not have a division operation in order to be used in an instantiation of a template for such a geometric primitive. For the majority of the primitives in CGAL, the basic arithmetic operations $+, -, *$, and $/$ suffice. For some other primitives, a number type must also provide a square root operation. A number type model must not only provide the arithmetic operations in the syntactically correct form, the available arithmetic operations must also have the correct semantics. Concerning the syntax of the operations on a number type, operator overloading is assumed. This makes the code highly readable.

Strictly speaking, a number type should be a model for the mathematical concept of a real number, just in accordance with the assumptions made in the abstract machine the geometric algorithm was developed for. However, there is no such valid model. Fortunately, in practice, the numerical input data for a geometric primitive are not arbitrary real numbers, but restricted to some subset of the real numbers, e.g., the set of real numbers representable by an `int` or a `double`. In such a situation, a concrete model for a number type is useful, if it provides the correct semantics for the relevant arithmetic operations restricted to the possible set of operands. Less strictly speaking, it suffices if the primitive computes the correct result.

C++ has several built-in number type models. There are signed and unsigned integral types and floating-point types. In general, they all fail to be valid models for the mathematical concepts of integers and real numbers, respectively. Due to rounding errors and over- or underflow, basic laws of arithmetic are violated. There are also various software packages [11,2] and libraries [16,14] that provide further models of number types, e.g., arbitrary precision integral types. A very useful model for geometric computations is the number type `leda_real` [4,16]. This type models a subset of algebraic numbers. `leda_real`s subsume (arbitrary precision) integers and are closed under addition, subtraction, multiplication, division, and k-root operations. Invisible to the user, this number type uses adaptive computation to speed up exact comparison operations.

In order to be useful as a library component, there should be a handy description of the set of permissible inputs, for which an instantiation of the template code of a geometric primitive works. For example, a description like 'Instantiated with `double`, this primitive gives the correct result for all those inputs where rounding errors do not lead to incorrect decisions during the evaluation of the primitive' is by no means useful. For number types guaranteeing exact decisions, a useful description is easy, however. Admittedly, there is still no satisfactory solution to this issue for potentially inexact number types.

Choosing a more powerful number type is only one way to get reliable primitive operations for CGAL algorithms. An alternative is to make special implementations for the primitives, which use evaluation strategies different from the default templates. A number of techniques for exact implementation of geometric primitives have been suggested, e.g., [8,27]. Alternative implementations can be provided within the CGAL kernel framework as explicit specializations of the template primitives for certain number types. Furthermore, primitives can be implemented independently from the CGAL kernel and can be used directly in traits class models. Finally, existing geometry kernels can be easily used with CGAL algorithms. For example, there are adaptations of the floating-point and the rational geometry kernel of LEDA.

The generic design offers a lot of flexibility. Figure 1 shows a bar chart with (typical running times for) various geometry kernels in a planar convex hull

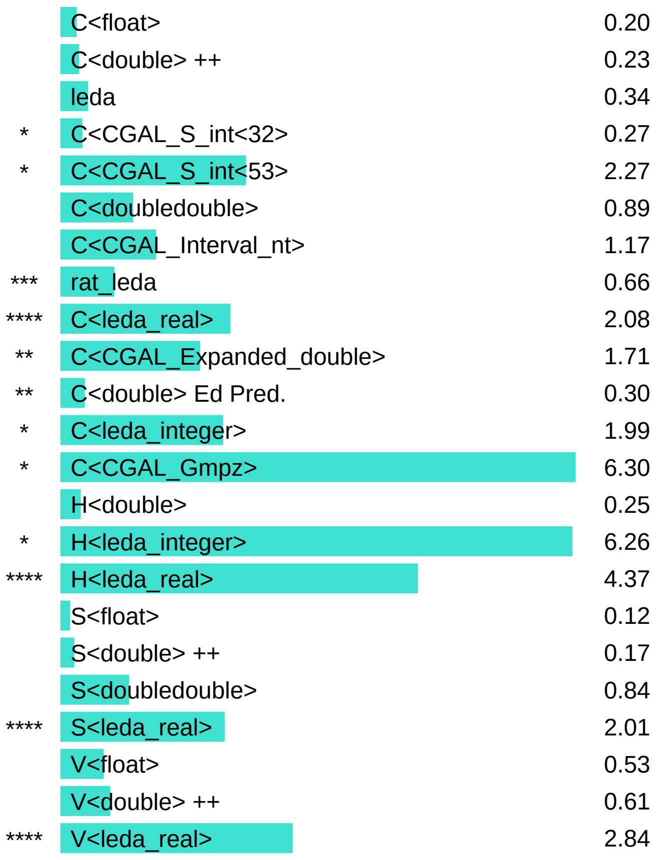


Fig. 1. Running times of a CGAL implementation of the Graham-scan algorithm with different geometry kernels. The kernels labelled `C< >` and `H< >` are instantiations of the Cartesian and homogeneous CGAL kernels resp., with the number type argument in angle-brackets. Both use reference counting [17]. The goal is to speed up copying operations at the cost of an indirection in accessing data. The kernels labelled `S< >` use Cartesian coordinates as well but no reference counting. The kernels labelled `V< >` are similar to the `S< >` kernels, but more Java-like: all access functions are virtual and not inlined. Finally, `leda` and `rat_leda` are adaptations of the floating-point and rational geometry kernels resp. of LEDA. In the `C<double>++` kernel, some primitives are specialized and use a slightly more robust computation in the primitives. Furthermore, the code for number type `CGAL_S_int<N>` is explicitly specialized. The primitives for `CGAL_S_int<N>` assume that the Cartesian coordinates, which are internally maintained as `doubles`, are integers with at most `N` bits. In the specializations, a static floating point filter [9] is used. The filter is based on the assumption of the integrality and the size of the numbers. If the filter fails, primitives are re-evaluated with arbitrary precision integer arithmetic. `C<double> Ed Pred` uses special primitives based on [27]. CGAL provides different inlining policies. Here, for all CGAL kernels, the level of inlining was increased with respect to the default in the current release.

computation. The geometry kernels differ not only in their efficiency, but also in their effectiveness with respect to exact geometric computation. The number of stars on the left in Fig. 1 is an indicator for the power of the kernels to produce correct results in the predicates. No star means potentially unreliable predicates. In the kernels with one star, the predicates are reliable, if the numerical data passed to the predicates are integers (of bounded size). The predicates in a kernel with two stars also make correct decisions for numerical data represented as double precision floating-point numbers.[3] The reliability of the kernels with one or two stars is based on the assumption that the numerical data passed to the predicates are exact. With cascaded computations, this assumption might not hold anymore. The `rat_leda` geometry kernel has three stars, because it gives correct result even with cascaded rational computations. Since the number type `leda_real` guarantees exact decisions for a superset of the rationals as described above, all kernels parameterized with number type `leda_real` have four stars. There are significant differences in the performance, ranging from fairly slow arbitrary precision integer arithmetic via fast, exact kernels based on adaptive evaluation to very fast, but potentially unreliable floating-point arithmetic. For further discussion of the various kernels and their performance we refer to [26].

7 Conclusions

The use of the generic programming paradigm is the main source of the flexibility of CGAL. Whereas in other geometry libraries there is a dovetailing between the geometric algorithms and the types on which they operate (including container types), in CGAL, algorithms are parameterized by the types on which they operate (not only by container types) such that these types can be exchanged. We have shown that the parameterization allows one to easily combine a CGAL algorithm with existing non-CGAL types, e.g., geometric types provided by a GIS or a CAD system. Furthermore, it allows one to apply a CGAL algorithm to a related geometric problem via geometric transformation. Finally, it allows an advanced user to tailor a CGAL algorithm to a particular practical context by exploiting features that do not hold in general, but hold in the given context. For example, this allows one to gain efficiency, especially for exact computation. The number of geometric algorithms in CGAL is still limited, so a major future task is to add generic implementations of further geometric algorithms. This includes finding suitable abstractions for the primitives they use.

References

1. J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry.* Cambridge University Press, UK, 1998. translated by H. Brönnimann.
2. K. Briggs. The doubledouble home page. `http://epidem13.plantsci.cam.ac.uk/~kbriggs/doubledouble.html`.

[3] Provided that neither overflow nor underflow occurs in the internal computation.

3. H. Brönnimann, S. Schirra, and R. Veltkamp, editors. *CGAL Reference Manuals.* CGAL consortium, 1998. `http://www.cs.uu.nl/CGAL`.
4. C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class `real` number. Technical Report MPI-I-96-1-001, Max-Planck-Institut für Informatik, 1996.
5. K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.
6. M. de Berg, M. van Kreveld, M. Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 1997.
7. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. Research Report MPI-I-98-1-007, Max-Planck-Institut für Informatik, 1998.
8. S. Fortune. Numerical stability of algorithms for 2D Delaunay triangulations and Voronoi diagrams. *Int. J. Computational Geometry and Appl.*, 5:193–213, 1995.
9. S. Fortune and C. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.
10. J. E. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* CRC Press LLC, Boca Raton, FL, 1997.
11. T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, June 1996.
12. C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3):31–41, March 1989.
13. L. Kettner. Circulators. In H. Brönnimann, S. Schirra, and R. Veltkamp, editors, *CGAL Reference Manual. Part 3: Support Library.* 1998.
14. LiDIA-Group, Fachbereich Informatik, TH Darmstadt. *LiDIA Manual A library for computational number theory*, 1.3 edition, April 1997.
15. K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms.* Springer-Verlag, Heidelberg, Germany, 1984.
16. K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User manual*, 3.7 edition, 1998. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.
17. S. Meyers. *More Effective C++*. Addison-Wesley, 1996.
18. K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1994.
19. D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.
20. N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.
21. J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.
22. M. H. Overmars. Designing the computational geometry algorithms library CGAL. *Applied Computational Geometry*, Lect. Notes in Comp. Science Vol. 1148, 1996, pages 53–58.
23. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.
24. J. R. Sack and J. Urrutia, editors. *Handbook on Computational Geometry.* Elsevier Science Publishers, Amsterdam, The Netherlands, 1999.
25. S. Schirra. Precision and robustness issues in geometric computation. In *Handbook on Computational Geometry.* Elsevier, Amsterdam, The Netherlands, 1999.
26. S. Schirra. A case study on the cost of geometric computing. *Algorithm Engineering and Experimentation.* Lect. Notes in Comp. Science Vol. 1619, pages 156–176.
27. J. R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.
28. C. K. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7(1-2):3–23, 1997.

Generic Programming in POOMA and PETE

James A. Crotinger, Julian Cummings, Scott Haney,
William Humphrey, Steve Karmesin, John Reynders,
Stephen Smith, and Timothy J. Williams

Los Alamos National Laboratory; Los Alamos, NM 87545

Abstract. POOMA is a C++ framework for developing portable scientific applications for serial and parallel computers using high-level physical abstractions. PETE is an general-purpose expression-template library employed by POOMA to implement expression evaluation. This paper discusses generic programming techniques that are used to achieve flexibility and high performance in both POOMA and PETE. POOMA's array class factors the data representation and look-up into a generic engine concept. PETE's expression templates are used to build and operate efficiently on expressions. PETE is implemented using generic techniques that allow it to adapt to a variety of client-class interfaces, and to provide a powerful and flexible compile-time expression-tree-traversal mechanism.

1 Introduction

POOMA (Parallel Object-Oriented Methods and Applications) is an object-oriented framework for developing scientific computing applications on platforms ranging from laptops to parallel supercomputers [1,2]. POOMA includes C++ template classes representing high-level mathematical and physical abstractions such as arrays, particles, and fields. POOMA objects can be used in data-parallel expressions, with the parallelism encapsulated in the underlying framework. Expression creation and manipulation facilities are provided by PETE, the Portable Expression Template Engine.[1]

This paper discusses generic programming techniques used to achieve flexibility and high performance in POOMA II and in PETE. POOMA II, currently under development, is a redesign of POOMA intended to further increase expressiveness and performance. POOMA arrays (the "II" will henceforth be understood) delegate data allocation and element access to a new *engine* class, allowing the array class to provide a uniform interface, including array expression

[1] This paper describes POOMA and PETE at the time of the Dagstuhl Seminar. Significant developments have occurred in the interim. The latest releases, along with examples and user-level tutorials, are available on the World Wide Web at `http://www.acl.lanl.gov/pooma` and `http://www.acl.lanl.gov/pete`. Certain important differences will be footnoted.

capability, for a variety of data formats. Using PETE, POOMA separates the representation of an expression from its evaluation, allowing POOMA to provide multiple expression evaluation mechanisms. The simplest mechanism inlines the entire evaluation in a manner similar to conventional expression-template array classes. Alternatively, an expression can be subdivided into expressions on subdomains of the arrays, and these sub-expressions can be evaluated independently by multiple threads.

PETE uses generic techniques to avoid assumptions about client-class interfaces and to provide a powerful and flexible expression tree traversal mechanism.

This paper is organized as follows. Section 2 discusses the engine abstraction. Section 3 gives a brief introduction to expression templates. Section 4 describes the use of PETE to add expression template capability to client classes, and the method by which PETE performs expression object manipulations, including evaluation. Section 5 concludes the paper with a discussion of POOMA's expression-engine, an engine that allows expressions to be used as arrays.

2 Arrays and Engines

Many scientific computing applications require data types with multidimensional array semantics, but with a variety of underlying data structures. Examples range from regular arrays having Fortran or C storage order, to banded or sparse matrices, to array-like objects that compute their elements directly from their indices. One could model these data types using an inheritance hierarchy. An abstract base-class would define the interface, and descendent classes would override virtual functions to deal properly with their internal data structures. Unfortunately, the cost of virtual function calls in the inner loop of an array expression is prohibitive. Compile-time techniques are required to satisfy the performance requirements of most scientific applications.

POOMA's `Array` class achieves the desired flexibility by factoring the data representation into a separate engine class. The `Array` provides the user interface and expression capability, while the engine provides data storage and mapping of indices to data elements.

All engines are specialization of an `Engine` template class:

```
template <int Dim, class T, class EngineTag> class Engine { };
```

For example, a *brick-engine*, which stores a contiguous block of data that is interpreted as a multi-dimensional array with Fortran storage-order, is declared:

```
class Brick {};
template <int Dim, class T> class Engine<Dim,T,Brick>;
```

The `Brick` class serves as a *policy tag* to choose a particular specialization of the `Engine` template. Requiring all engines to be specializations of a general `Engine` template allows for somewhat tighter type-checking, since functions that operate on engines in general can be given signatures of the form:

```
template <int Dim, class T, class ETag>
void foo(const Engine<Dim,T,ETag> &e);
```

rather than

```
template <class Engine>
void foo(const Engine &e);
```

Also, partial specialization on an engine-tag allows the `Array` template to provide reasonable default template parameters:

```
template <int Dim, class T=double, class EngineTag=Brick>
class Array;
```

The `Array` classes delegate individual element access to the corresponding engine classes, which return the appropriate element in the most efficient manner possible.

Note that the array is not the container of the data, it is a user interface for manipulating the data. As a result, a "`const Array`" is not what one might think. The following is completely legal:

```
Array<1> a(10);
const Array<1> & ar = a;
ar(4) = 3.0;
```

The assignment is allowed because it does not modify the `Array` object. This is similar to the distinction between an STL container and an STL iterator. POOMA's `ConstArray` class is a read-only array class that is analogous to the STL `const_iterator`. `ConstArray` is also the base class for `Array`. Not only is this natural from an implementation standpoint (`Array` extends `ConstArray` by adding assignment and indexing that returns element references), but it allows an `Array` to be passed as argument to a function that takes a `ConstArray`. For conciseness, this paper will focus on properties of the `Array` class. However, most of this discussion applies to `ConstArray` as well.

POOMA domain objects can be used to select a subset of an array. Rather than providing a special subarray class, POOMA makes further use of the engine concept: subscripting with a domain object creates a new `Array` object that has a different engine, a *view-engine.* View-engines reference a subset of the data owned by some other engine. For example, a brick-view engine, `Engine<Dim,T,BrickView>`, can be used to access any constant-stride, regular subset of points managed by a brick-engine. For example:

```
Array<1> a(20);                       // {a(0), a(1), ... a(19)}
Range<1> I(4,12,2);                   // {4,6,8,10,12}
Array<1,double,BrickView> b = a(I); // b(0) = a(4), ...
```

Here, the `Range<Dim>` object specifies a constant-stride, rectangular subset of points.

POOMA provides a traits class, `NewEngine`, to simplify the determination of the appropriate view type:

```
template <class Engine, class Domain> struct NewEngine { };
```

This template is specialized for the appropriate engine-domain pairs such that the trait `Type_t` is the type of the appropriate view engine. For example:

```
template <int Dim, class T>
struct NewEngine< Engine<Dim,T,Brick>, Range<Dim> >  {
  typedef Engine<Dim,T,BrickView> Type_t;
};
```

Thus, in the above example we could have written:

```
typedef Engine<1,double,Brick> Engine_t;
typedef Range<1> Domain_t;
typedef NewEngine<Engine_t,Domain_t>::Type_t ViewEngine_t;
ViewEngine_t b = a(I);
```

While using `NewEngine` doesn't appear to buy us much for this toy example, this idiom greatly simplifies much of the complex internal code in POOMA. Indeed, view-engine arrays are rarely explicitly declared. However, these are constructed automatically in many array expressions, such as the following centered-difference:

```
Array<1> a(10), b(10);
Interval<1> I(1,8);
a(I) = b(I+1) - b(I-1);
```

where `Interval<Dim>` is a unit-stride version of `Range`. Here the array class' `operator()` is overloaded to create views, and thus the terms `b(I+1)` and `b(I-1)` return temporary views of `b`. These view objects are then passed on to `operator-()`, which PETE uses to manufacture an expression-template object that represents this subexpression. This process will be described in the following section.

3 Expression Templates

Our goal is to write array expressions, such as

```
A = B + 2 * C;
```

and achieve the same efficiency as with explicit loops. Overloading the various operators to directly perform their operations cannot achieve this goal as evaluation will then happen in a pairwise fashion and involve multiple temporaries. Efficient evaluation requires the use of expression templates [3,4,5,6].

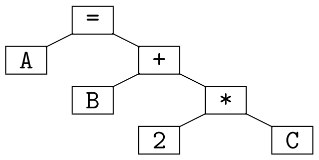


Fig. 1. Parse tree for the expression `A = B + 2 * C`.

Expression templates are a mechanism by which the parse tree for an expression is represented by a recursive template type. Both the type and an object of that type are manufactured by the overloaded operators as the expression is parsed. For instance, the expression `A = B + 2 * C` is represented by the parse tree shown in Fig. 1. In PETE, this parse tree would be encoded in an object of type[2]

```
Expression< TBTree<OpAssign, Array1,
            TBTree<OpAdd, Array2,
            TBTree<OpMultiply, Scalar<int>, Array3> > > >
```

where `TBTree<Op,E_l,E_r>` is a template class that stores an applicative template object, of type `Op`, and left and right subexpression, of types `E_l` and `E_r`. `Array1`, `Array2` and `Array3` are the types of `A`, `B`, and `C`. The `Expression` object can be used to efficiently evaluate the expression, as will be explained below.

4 PETE

PETE is a general-purpose generic library that allows client classes to use expression-template machinery for expression evaluation and manipulation.[3] Generic techniques are used to adapt to a variety of class interfaces and to minimize the impact on client classes. Unlike other expression template implementations, PETE separates the construction of the expression object from the operations that can be performed on that object, allowing functors to be written that perform general purpose parse-tree traversal at compile time.

PETE is an adaptable expression template library similar to that described by Furnish [6]. PETE can be used in two modes. The method demonstrated here uses a combination of inheritance and templates, similar to the approach used in Ref. [6], to enable the client class to be interpreted as a terminal (leaf node) in the parse tree.[4] For example, consider a simple one-dimensional vector class, `Vec3`, that wraps a `double[3]` array:

```
class Vec3 : public Expression<Vec3>  {
  double d[3];
  // . . .
};
```

PETE can also be used in a completely external manner, requiring no modifications to the client class.[5] There are advantages and disadvantages to each approach. The external approach will be discussed briefly at the end of the section.

[2] Many of the internal names have been changed in PETE 2.x (e.g. `TBTree` is now `BinaryNode`), but the basic functionality is identical.

[3] For example, with PETE 2.x expression template evaluation can be implemented for STL vectors with about 20 lines of code.

[4] This template idiom was first discussed by Barton and Nackman in [7].

[5] PETE 2.x can only be used in this mode. The disadvantage of this approach—explicitly writing the operators—is avoided by providing an easy-to-use tool for operator generation.

Unlike Furnish's Indexable base class, Expression makes no assumptions about the client's interface. Rather, Expression provides member functions that perform static up-casts to the daughter type:

```
template<class WrappedExpr>
class Expression {
public:
   typedef WrappedExpr Wrapped_t;
   Wrapped_t & peteUnwrap()
      { return *static_cast<Wrapped_t *>(this); }
   const Wrapped_t & peteUnwrap() const
      { return *static_cast<const Wrapped_t *>(this); }
};
```

Using these members, user-specified functions that operate on the terminals can directly invoke the client's methods.

When using PETE in this mode, the client class must provide two typedefs, Element_t and Expression_t, and a member function:

```
const Expression_t &makeExpression() const;
```

Element_t is the element-type for the container, for example double in the Vec3 class above. Expression_t and makeExpression allow the client class to specify an object of a different type to be used to generate the client's values. For example, some classes provide STL-like iterators rather than indexing. In this case, Expression_t would be the iterator type, makeExpression would construct and return an iterator object, and the iterator objects, not the client objects, would appear as terminals in the parse tree that PETE constructs. For clients that can efficiently produce their values directly (via indexing, internal cursor incrementing, etc.), Expression_t should be the client class itself.

Vec3 can be indexed directly, so the following must be added to its public interface:

```
   typedef double Element_t;
   typedef Vec3 Expression_t;
   const Vec3 &makeExpression() const { return *this; }
```

These are all that are required for PETE to build expression objects from expressions involving Vec3 objects. PETE's overloaded operators will recursively build an expression object, such as that described in Sect. 3, with Vec3 objects at the leaves. A bit more work is require to evaluate the expression.

PETE operates on expressions using a template metaprogram that recursively walks the parse tree, applying a generic *tag-functor* at the terminals and a generic *tag-combiner* at the non-terminals. For example, in order to evaluate an expression, a tag-functor would be used that returns a value from a container, and a tag-combiner would be used that combines values from its children with the appropriate arithmetic operator. Tag-functors are specializations of the class:

```
template <class FTag, class Expr>
struct TagFunctor {
      typedef ???? Type_t;
      static Type_t apply(const Expr &e, const FTag &f);
};
```

This class must define a static member `apply` that takes the tag-object `f` and operates on the expression object `e` in the appropriate manner. It must also export the return type, `Type_t` (indicated here by `????`).

At a minimum, the client class must provide a specialization of `TagFunctor` that returns the values of its elements. For example, in order to evaluate a `Vec3` object at index `i`, we define a tag indicating this purpose, and specialize `TagFunctor` to take the appropriate action:

```
struct EvalFunctor1 {
   int i_m;
   EvalFunctor1(int i) : i_m(i) {}
};

template <>
struct TagFunctor<EvalFunctor1,Vec3::Expression_t> {
   typedef Vec3::Element_t Type_t;
   static Type_t apply(const Vec3 &a, const EvalFunctor1 &f)
      { return a[f.i_m]; }
};
```

Note that the `EvalFunctor1` object carries the index with it.

Our final task in evaluating `Vec3` expressions is to write the assignment operator. The recipe for this is fairly straight-forward, but first the full expression-traversal functionality will be explained as it has a number of uses.

In order to evaluate the expression, the traversal needs to combine the values returned by the tag-functor using the appropriate operators. This is accomplished with a special tag-combiner. For binary operators, tag-combiners are specializations of

```
template<class A, class B, class Op, class CTag>
struct TagCombine2 {
   typedef ???? Type_t;
   static Type_t combine(A a, B b, Op op, const CTag &c);
};
```

This class must define a static member `combine` that takes the tag-object `c` and the operator object `op`, and operates on the expression objects `a` and `b` in the appropriate manner, returning some result. It must also export the return type. Note that `A` and `B` may be reference types—it is up to the various `TagFunctor` specializations to define their return types appropriately if reference semantics are needed.

For the purposes of evaluating the expression, PETE uses an empty tag structure `OpCombineTag`, specializing `TagCombine2` as follows:

```
struct OpCombineTag { };
template<class A, class B, class Op>
struct TagCombine2<A,B,Op,OpCombineTag> {
   typedef typename BinaryReturn<A,B,Op>::Type_t Type_t;
   static Type_t combine(A a, B b, Op op, OpCombineTag)
      { return op(a,b); }
};
```

Here the applicative template object `op` is applied to the subexpression values and the result is returned. `OpCombineTag` serves only as a tag; the tag-object is not involved in the combine operation.

`BinaryReturn` is a traits class that uses C++ operator semantics and promotion rules to compute the type that results when the binary operator `op` is applied to the subexpressions `a` and `b`. This class could be implemented by enumerating all possible specializations, but this quickly becomes unwieldy. Instead, the type is computed via a somewhat complex set of template classes. While this is an important example of generic programming in PETE, the details are beyond the scope of this paper. We note, however, that the scheme is designed to be user-extensible.

Finally, we come to the traversal of the parse tree. This is handled by the `ForEach` template class:

```
struct ForEach<Expr, FTag, CTag> {
   typedef ???? Type_t;
   static Type_t
      apply(const Expr &e, const FTag &ftag, const CTag &ctag);
};
```

`ForEach` defines an `apply` method that operates on the expression with the appropriate tag-functor or tag-combiner, depending on whether the expression is a terminal node or not. The default definition assumes that the expression is a terminal node, and thus applies the `TagFunctor` selected by the `FTag`:

```
template<class Expr, class FTag, class CTag>
struct ForEach {
   typedef typename TagFunctor<FTag,Expr>::Type_t Type_t;
   static Type_t
      apply(const Expr &expr, const FTag &ftag, const CTag &)
         { return TagFunctor<FTag,Expr>::apply(expr, ftag); }
};
```

The recursion is performed by specializations of `ForEach` for the various types of non-terminal nodes. For example, binary expressions are handled by specializing on `TBTree` as follows:

```
template<class Op, class A, class B, class FTag, class CTag>
struct ForEach< TBTree<Op, A, B>, FTag, CTag >
{
   typedef typename ForEach<A, FTag, CTag>::Type_t TypeA_t;
   typedef typename ForEach<B, FTag, CTag>::Type_t TypeB_t;
   typedef typename
      TagCombine2<TypeA_t, TypeB_t, Op, CTag>::Type_t Type_t;

   static Type_t apply(const TBTree<Op, A, B> &expr,
                       const FTag &f, const CTag &c) {
      return TagCombine2<TypeA_t, TypeB_t, Op, CTag>::
         combine(
            ForEach<A, FTag, CTag>::apply(expr.left(),  f, c),
            ForEach<B, FTag, CTag>::apply(expr.right(), f, c),
            expr.value(),
            c );
   }
};
```

The first two typedefs compute the types that will be returned by application of `ForEach::apply` to the left and right subexpressions of the `TBTree`, and the third specifies the type that *this* `ForEach::apply` will return. The `apply` method recursively calls `ForEach::apply` on the left and right subexpressions. The results are then passed, along with the `TBTree`'s operator object (the "value" stored by the `TBTree` node) and the combine-tag object, to `TagCombine2::combine`. Finally, the result of this call is returned. Note that, with a sufficiently aggressive compiler, all of this is expanded inline during template instantiation.

PETE also specializes `ForEach` for unary nodes, which are used to represent unary operators and element-wise function calls, and ternary nodes, which are used to represent "where"-expressions (PETE's version of the C++ "`?:`" operator). These are quite similar to the above.

Note that `ForEach` does a *post-order* traversal of the parse tree, visiting a `TBTree`'s "value" *after* visiting its children. This is the appropriate traversal for expression evaluation. One can write similar functors to do more general traversals. For instance, a functor that prints a representation of the expression (without building that representation in a buffer) requires an in-order traversal. Although PETE does not include more general traversal functors, they are not difficult to construct once PETE's `ForEach` functor is understood.

`ForEach::apply` could be written directly as a template function, as could the `combine` and `apply` methods of the tag-combiners and tag-functors. However, the `Type_t` traits that these classes export greatly simplify the rest of the code, so it seems more natural to put the implementation of the functor methods in the respective functor classes, providing template function wrappers where this simplifies the user interface. Such a wrapper is provided for `ForEach::apply`. This is handy since template functions can deduce their template parameters from their argument types:

```
template<class Expr,class FTag,class CTag>
inline typename ForEach<Expr,FTag,CTag>::Type_t
forEachTag(const Expr &e, const FTag &f, const CTag &c) {
   return ForEach<Expr,FTag,CTag>::apply(e, f, c);
}
```

Now that the evaluation process has been explained, we can write the assignment operator for the `Vec3` class:

```
template <class Expr>
Vec3 &operator=(const Expression<Expr> &exp)
{
  Expr e = exp.peteUnwrap();

  d[0] = forEachTag(e, EvalFunctor1(0), OpCombineTag());
  d[1] = forEachTag(e, EvalFunctor1(1), OpCombineTag());
  d[2] = forEachTag(e, EvalFunctor1(2), OpCombineTag());

  return *this;
}
```

This function is relatively simple: First the actual expression object is extracted from its base class. Then `forEachTag` is called, passing it the expression, an `EvalFunctor1` object that propagates the index to the terminal nodes, and an `OpCombineTag` object that serves only to choose the `TagCombine2` specialization that applies the `TBTree`'s operator to the results returned from its children.

With the definition of the assignment operator, our `Vec3` class has all of the features necessary to use expression templates. The following code snippet will now compile and run:

```
Vec3 a, b, c;
b[0] = 10; b[1] = 3; b[2] = 2;
c = 1;
a = b + 2 * c;
```

Handling of scalars is provided by PETE, which defines a `Scalar<T>` template that is PETE-aware. The final expression will compile to the equivalent of:

```
d[0] = b[0] + 2 * c[0];
d[1] = b[1] + 2 * c[1];
d[2] = b[2] + 2 * c[2];
```

which is exactly the desired result. (Note that this does require very aggressive optimization on the part of the compiler. This is true of all expression-template implementations—expression templates are only useful if all of these traversals are inlined.)

If we were instead dealing with a client class that provided an iterator interface, the example would only be slightly more complicated. As mentioned above, we would have to provide a TagFunctor specialization, say on EvalFunctor0, that dereferenced the iterator, and another specialization, say on Increment, that incremented the iterator. The assignment operator would then look something like

```
template <class Expr>
Vec &operator=(const Expression<Expr> &exp) {
  Expr e = exp.peteUnwrap();
  iterator i = begin();
  while (i != end()) {
     *i++ = forEachTag(e, EvalFunctor0(), OpCombineTag());
     (void) forEachTag(e, Increment(), NullCombineTag());
   }
  return *this;
}
```

Here the first forEachTag returns the value of the right-hand side for the current iterator position and assigns it via the local iterator. The second forEachTag increments the iterators on the right hand side. Its return value is ignored, as are the values returned by the increment tag-functors (this is the meaning of the NullCombineTag). Note that apply takes its Expr argument as a const reference. This is necessary since expression objects are often temporaries. Some tag-functor specializations, such as Increment, will have to cast away the constness in order to increment the iterators. Obviously this requires some care.

Evaluation is not the only use of ForEach. Another example is checking that all arrays in an expression have the same number of elements. With the proper specializations, the code to do this would look something like

```
Expr e = exp.peteUnwrap();
int size = forEachTag(e, SizeTag(), AssertEqTag());
```

Here TagFunctor<SizeTag,Expr>::apply would return the size of each terminal, and TagCombine2<A,B,Op,AssertEqTag>::combine would assert that the values were equal, and if they were, it would return the value.[6] In POOMA, this idea is extended to checking multi-dimensional domain conformance, returning the common domain as a result.

ForEach can also be used to construct a new expression tree from the original one. POOMA uses this capability to build expression trees whose terminal nodes contain views of the terminal arrays in the original tree.

As was mentioned earlier, PETE can be used without modifying the client class. Instead, the required functionality is put in an external traits class, MakeExpression. Unfortunately the assignment operator cannot be handled in this manner since C++ requires that it be a member function. As a result, a completely external implementation can only be achieved if the user is willing to forgo assignment for an external assign function. The biggest disadvantage of

[6] Life is actually a bit more complex due to interactions with Scalar<T>.

the external approach is that clients must define the appropriate overloaded operators to build the expression elements. PETE includes an example Perl script that is used by POOMA for this purpose. While this is more complicated to implement than the inheritance approach, it does avoid certain problems due to subtleties of the C++ template matching algorithm. POOMA employs a mixed approach, using `MakeExpression`, generated operators, and an overloaded assignment operator.

5 Expression-Engine

An expression object can be used to generate an optimal set of loops for evaluation, and it can be manipulated by PETE functors to query its properties, build new trees, etc. However, an expression cannot be passed to a function expecting an `Array`, which is a desirable capability. Implicit conversion from expressions to arrays will not work as these are not performed when matching templates. Even if they were, it would be undesirable to create a temporary to hold the result. POOMA's engine architecture provides an elegant solution to this problem. An engine object can be constructed that contains an expression object and uses functors to index the expression. POOMA implements this as:

```
template <int Dim, class T, class Expr>
class Engine<Dim,T,ExpressionTag<Expr> > ;
```

PETE's template machinery is used to efficiently calculate values when the array is indexed, avoiding unnecessary evaluation.

As a result, any template function that takes a `ConstArray` having an arbitrary engine type can be called with an array expression. For example, if we have the function

```
template<int Dim, class T, class ETag>
T trace(const ConstArray<Dim,T,ETag> &a) {
  T tr = 0;
  for (int i = 0; i < a.length(0); ++i) { tr += a(i,i); }
  return tr;
}
```

Then `trace(B+2*C)` sums the diagonal components of `B+2*C` without computing any of the off-diagonal values.

Adding an `Array` interface to an expression also simplifies the evaluation code. POOMA exploits the separation of expression construction and evaluation by deferring the evaluation to a separate set of template classes, called *evaluators*. The `Array` assignment operator constructs a new array whose engine contains the expression `*this = rhs`; i.e., an expression with the element-wise assignment operator at the root of the tree. This new array is then handed off to the evaluator. A detailed discussion of evaluators is beyond the scope of this paper. However, the simplest evaluators ultimately expand to the following loop (for a two-dimensional array):

```
for (int i = 0; i < n0; ++i)
  for (int j = 0; j < n1; ++j)
    expr(i,j);
```

where `expr` is the `Array` object passed to the evaluator. Again, it is critical that the compiler completely inline the indexing of the expression object; i.e. that no function calls remain in the inner loop. If this is accomplished, all of the usual compiler optimizations will be available and the resulting code should run as fast as hand-coded C.

6 Summary

This paper has presented generic techniques used in POOMA and PETE to achieve flexibility without sacrificing efficiency. The POOMA array-engine abstraction separates the representation of an array from its interface. This greatly simplifies the development of new array types as one need only build the appropriate engine. The `Array` class provides the interface, including the interaction with expression templates.

POOMA's expression templates are provided by a separate, reusable package, PETE, that makes extensive use of traits, template metaprograms, and other generic and template techniques to maintain container-independence and to simplify type computations. PETE can be used via inheritance, in which case one makes minor additions to the client's interface, and writes an assignment operator that takes an expression object and uses the `ForEach` functor to do the evaluation. PETE can also be used in a completely external mode, or in a mixture of the two.

POOMA further exploits the separation of the expression and its evaluation. Arrays can have expression-engines, allowing expressions to be passed to functions expecting `ConstArray` objects, with expression evaluation occurring when the array is indexed. Furthermore, arrays delegate evaluation of expressions to separate evaluator objects, allowing specialization of evaluators for certain execution environments and certain types of terminal engines.

Acknowledgments

We would like to thank Geoffrey Furnish and Todd Veldhuizen for many valuable discussions. The research described here was performed under the auspices of the U. S. Department of Energy by Los Alamos National Laboratory under contract No. W-7405-Eng-36.

References

1. J. Reynders et al. POOMA: a framework for scientific simulations on parallel architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.

2. William Humphrey, Steve Karmesin, Federico Bassetti, and John Reynders. Optimization of data-parallel field expressions in the POOMA framework. In *ISCOPE '97*, Marina del Rey, CA, December 1997.
3. Scott Haney. Is C++ fast enough for scientific computing. *Computers in Physics*, page 690, November 1994.
4. Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.
5. Scott W. Haney. Beating the abstraction penalty in C++ using expression templates. *Computers in Physics*, page 552, November 1996.
6. Geoffrey Furnish. Disambiguated glommable expression templates. *Computers in Physics*, page 263, May 1997.
7. John J. Barton and Lee R. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1994.

The STL Model in the Geometric Domain

Ullrich Köthe[1] and Karsten Weihe[2]

[1] Fraunhofer-Institute for Computer Graphics, Rostock
Joachim-Jungius-Str. 9, D-18059 Rostock, Germany
`koethe@egd.igd.fhg.de`
`http://www.egd.igd.fhg.de/~ulli`

[2] Universität Konstanz, Fakultät für Mathematik und Informatik
Fach D188, D-78457 Konstanz, Germany
`weihe@fmi.uni-konstanz.de`
`http://www.fmi.uni-konstanz.de/~weihe`

Abstract. Computational geometry and its close relative image analysis are among the most promising application domains of generic programming. This insight raises the question whether, and to which extent, the concepts of the *Standard Template Library* (*STL*) are appropriate for library design in this realm. We will discuss this question in view of selected fundamental algorithms and data structures.

1 Introduction

Throughout the last few years, *generic programming* has evolved as a new way of programming that is particularly well suited to algorithm-oriented software development. This is due to the specific role of algorithms and data structures in programs designed according to the ideas of generic programming. The conventional *structured-programming* approach puts emphasis on the algorithms, whereas data structures are treated as mere passive chunks of storage. On the other hand, *object-oriented programming* puts emphasis on data structures, and algorithms only appear as methods of data structures. In the latter approach, algorithms are nothing but "services" offered by data structures to access and modify the states of the objects.

In contrast, generic programming treats both algorithms and data structures as first-class citizens. Unlike object-oriented programming, the main design goal in generic programming is to *decouple* algorithms from data structures, and the main technique is to let data structures appear in a generic—and thus exchangeable—fashion inside algorithms. It is this loose coupling that makes the generic-programming approach promising for algorithm-oriented domains such as *computational geometry* and *computer vision*. Basically, we see three concrete reasons:

Complexity of design: Both data structures and algorithms are highly complex in these domains. Therefore, loose coupling between them is desirable, because separation makes the design much more manageable.

Maintainability: Experience shows that one of the main maintenance tasks in algorithmic software development is the adaptation of the underlying data structures to new, unforeseen requirements. If algorithms are strongly coupled to data structures, every such re-design also affects all algorithms. However, modifications of complex algorithms are expensive and highly error-prone and thus should be kept to a minimum.

Reusability: Many algorithms are potentially applicable to a wide range of data structures. However, if an algorithm is tightly coupled to a specific implementation of a specific data structure, this potential reusability cannot be realized.

At present, the *Standard Template Library* (*STL*, [6]) might be the best example for the application of the ideas of generic programming. Besides its specific purpose as a standard library of basic data structures and algorithms, the STL may be viewed as an "implemented repository" of detailed design concepts and techniques that are intended to support generic programming. When designing a generic library (in C++), it is natural to take patterns from the STL: the STL is the most mature fully generic library around, and its core has been integrated into the ANSI/ISO C++ standard. Hence, in the long run most C++ developers will become familiar with these concepts, and similar libraries will be easy to learn and use.

Familiarity with the STL will be very helpful for understanding the following discussion, see [6] for an introduction.

Iterators. The key concept of the STL for achieving loose coupling between data structures and algorithms is the *iterator*, which is among the basic design patterns in [1]. One of the fundamental innovations of the STL is to base the relationship between algorithms and data structures *exclusively* on iterators, i.e. algorithms do not see any data structures directly. This leads to a much more flexible design because uniform iterators can be implemented for a wide range of data structures, even if the data structures themselves are very different.

The STL restricts itself to linear iterators. A valid iterator object i is associated with a *sequence* (array, linked list, file, etc.) and a specific position $p(i)$ in this sequence. An iterator allows access to the item at position $p(i)$ in the sequence (through method `operator*`), and can be moved to another position. At the very least, an iterator can traverse its sequence step-by-step; that is, one position forward in every step. Depending on the underlying sequence type, an iterator could also offer methods for moving backwards (*e.g.*, in doubly linked lists) or for random offsets (*e.g.*, in arrays).[1]

From an algorithm's viewpoint, a sequence is given by a pair of iterators, i and j, which define a finite half-open range $[p(i), \ldots, p(j))$ of positions. Hence, traversing the items of a list amounts to moving iterator i forward until i equals j. The general pattern looks like this:

[1] This is reflected by the iterator categories of the STL.

```
template <typename Iterator>
void forEach(Iterator i, Iterator j) {
       for ( ; i != j; ++i )
          // do something with *i, the current item
}
```

In other words, i refers to the first position of the sequence and j to the position *one-past-the-end.* This is the natural view on many types of sequences (and thus will usually yield the most efficient implementations of iterators for these types). For instance, a one-past-the-end iterator j for a linked list might refer to the NULL item, and for a file it simply refers to the designated end-of-file position.

Geometry. This is one of the most promising application domains of generic programming. The question arises whether and to which extent the design concepts of the STL can—and should—be rigorously applied to the design of geometric algorithms and data structures. In this paper, we will try to give an answer to this question. First of all, we will see that the situation is much more complex than in the case of linear structures:

1. The items inside a container are not necessarily arranged in a linear fashion. For example, cyclic, multi-dimensional, and network-like data structures play an important role. These data structures offer non-linear access patterns, which in turn require new iterator categories.
2. In the geometric realm, data structures cannot always directly provide the iterators expected by a particular algorithm. There will be two kinds of mismatch:
   (a) The algorithm's navigation requirements may differ from what the data structure naturally provides. This requires some kind of adaptation.
   (b) The items of a container are usually assigned various *attributes.* Typically, a generic algorithm only operates on one or a few of them. However, it should be left open in the implementation of the algorithm which attributes are addressed (and how these attributes are implemented).
3. Many generic algorithms essentially realize non-trivial navigation patterns. It is favorable to interpret—and thus implement—such algorithms as iterators rather than subroutines, so that they can be passed to other algorithms.

We will analyze these aspects in view of various realistic use scenarios. Besides discussing new iterator categories, we will split the single level of indirection of the STL (the iterator) into several levels by introducing iterator adapters and so-called *data accessors.* Iterator adapters, although already present in the STL, will play a much more significant role in the geometric realm as a means of mapping different navigational patterns onto each other.

Data accessors [5] address Aspect 2(b). This concept allows one to strictly separate navigation over the items of a container from the access to item values. This means that an algorithm's view on a data structure is not formed exclusively by iterators anymore, which gives us the ability to vary access functionality independently of the navigation patterns, as is needed when different algorithms (or different instantiations of the same generic algorithm) shall access different attributes of the same items in the same container.

2 Iterators for Non-standard Cases

In this section, we discuss a few scenarios from the geometric realm, which deviate significantly from the STL-iteration model. It will turn out that the concept of half-open linear ranges is not really appropriate in these cases.

2.1 Cyclic Data Structures

Cyclic data structures play a prominent role in the geometric domain. Consider, for example, a polygon: a polygon is a sequence of points, connected by edges, where the last edge leads back to the first point. Thus, the polygon is a cyclic structure, and the choice of the first point is completely arbitrary. The same phenomenon can be observed with functions defined over an angle, such as sine and cosine, which are the basis for the most natural implementations of circles and ellipses. Again, the assignment of the angle 0 to a particular direction is completely arbitrary.

Although cyclic structures are still one-dimensional, we need an iterator concept beyond the STL model to describe them. This might seem surprising, but the reason for this lies in the lack of an explicit one-past-the-end position within a circular structure. Thus, the STL approach of specifying a range by a pair of iterators no longer works properly. This can best be seen by looking at the pair `[i, i)`. In the STL, this range uniquely specifies the empty range. However, for a cyclic structure a full revolution is the most likely interpretation, although it might just as well mean the empty range or even any number of consecutive revolutions. These different possibilities cannot be distinguished by the STL model.

One might try to solve the problem by introducing an auxiliary end element into the cycle. However, many algorithms operating on cycles want to choose the start/end positions arbitrarily, at runtime. In these cases a fixed end element would be counter-productive. Therefore, Fabri et al [2] introduced the *circulator* as a new concept to capture periodic behavior. As in the STL, we can define forward, bidirectional, and random access circulators, which basically provide the same set of functions as their STL counterparts. However, there are no limitations on how often a circulator can be incremented or decremented or on the admissible range of indexes for `operator[]`. Instead, we have the property that any circulator returns to itself after a number `n` of iterations (namely the number `n` of elements in the cyclic list):

```
Circulator i = ..., j=i;
int n = ...;
assert(advance(i, n), i == j);
```

All intermediate positions of the circulator refer to valid, referenceable items; no end-element exists.[2] To uniquely specify a range with circulators, an additional parameter is needed: the winding number. It tells an algorithm how many full revolutions to do in addition to a range specified by a circulator pair. Thus, a complete circulator range is given by a triple `[begin, end, winding]`. For example, the triple `[i, i, 1]` corresponds to exactly one revolution, while the triple `[i, i, 0]` denotes the empty range. However, many algorithms only want to do exactly one revolution, in which case it is sufficient to pass a single iterator that refers to the starting position of the desired revolution. Then the basic iteration pattern looks like this:

```
template <typename Circulator>
void oneRevolution(Circulator begin)
{
    if(!begin.isSingular()) {
        Circulator i = begin;
        do {
            // do something with *i, the current item
        } while(++i != begin);
}
```

Example: Planar graphs are an illustrative example of circulators. A graph is defined as a set of *nodes* and a set of *edges*. Every edge is associated with two *incident* nodes, i.e., the edge connects these nodes. In turn, every node has a list of all its edges (the *incidence list*). A graph is called *planar* if it can be drawn in the plane such that no two edges cross each other (except at a common incident node). Figure 1 gives an example of a planar graph, and Figure 2 shows a graph for which it can be mathematically proved that no crossing-free drawing is possible. As can be seen, the incidence lists of planar graphs are cyclic data structures without a natural end.

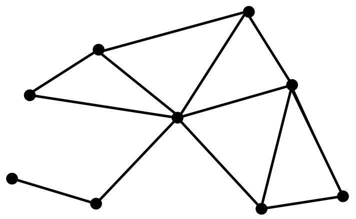


Fig. 1. A planar graph, embedded in the plane without crossings of edges.

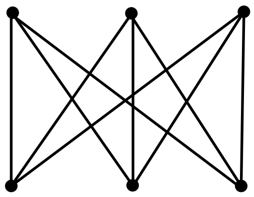


Fig. 2. A graph for which there is provably no embedding in the plane without crossings of edges.

[2] A singular circulator is introduced to denote the empty cycle. Its only valid operation is i.Singular() which returns `true` iff the circulator is indeed singular.

Many algorithms on planar graphs are based on iteration over incidence lists as a basic operation. For example, many algorithms jump from node to node via edges. Whenever a new node is reached, its incident edges are scanned in clockwise or counterclockwise order. However, such a scan typically starts with the successor of the edge over which the node was entered. A designated begin and end of the list of incident edges does not make sense, since begin and end are only determined at runtime. Hence, the circulator model is more natural than the STL model.

2.2 Multi-dimensional Data Structures

The linear iteration style of the STL can be interpreted as navigating in a 1-dimensional space. In computational geometry and computer vision we need to extend this design to higher dimensions [4]. For the sake of simplicity, we restrict the discussion to 2-dimensional data structures such as images and matrices, but the concepts discussed here can be applied to any dimension.

Since many algorithms explicitly require access to the 2-dimensional structure, we need to define an appropriate 2-dimensional iterator concept. To do so, a new problem must be solved: we must be able to specify the coordinate direction a navigation command (such as incrementation or offset addition) refers to. In [4], it is proposed to represent each coordinate by a data member of the iterator like this:

```
class ImageIterator {
    ...

    typedef ... MoveX;
    typedef ... MoveY;

    MoveX x;    // refer to x-coordinate
    MoveY y;    // refer to y-coordinate
};
```

Navigation commands are sent to one of the two data members so that the desired direction is always clear.[3] Both `MoveX` and `MoveY` support all navigation commands of a standard (usually random access) STL iterator:

```
ImageIterator iimage = ...;

++iimage.x;    // advance one step to the right
++iimage.y;    // advance one step down

iimage.x -= 20;    // go 20 steps to the left
```

[3] For convenience, it is also possible to send navigation commands directly to the iterator via a 2-dimensional distance object `Distance2D`, which basically generalizes `ptrdiff_t` to 2-D. However, this does not raise fundamental new issues and will not be discussed here (cf. [4] for details).

It should be noted that `MoveX` and `MoveY` are not themselves iterators since they do not support access to the actual data items, i.e., we cannot write `*iimage.x` or `*iimage.y`. Only the entire iterator has complete knowledge of the current position, so we always have to write `*iimage`.

Another important difference from the STL is found when we start to think about how to mark the borders of an iteration. A linear sequence has only 2 ends, which can be represented by two iterators. In contrast, an image of size $w \times h$ has $2(w+h+2)$ past-the-border positions.[4] It is not very realistic to pass iterators for all of these positions on every call of an algorithm. Yet, a complete representation of the image's boundary is necessary, because many algorithms cannot foresee where a given iterator will pass the border. Consider, for example, an algorithm that controls drawing on an image with the mouse. When the mouse leaves the image's range at an arbitrary, unforeseeable location, the algorithm must recognize it and stop drawing ("clipping").

The solution is to calculate new border markers as needed from a few markers at known positions. Of course, this requires that navigation commands are still applicable to past-the-border iterators which don't refer to valid pixels. For example, in an algorithm performing a row-major iteration, we need to mark the end of the first column and the end of the current row. If we advance the iterator to the next row, the marker at the end of the row needs to be advanced as well. To initialize this algorithm, we need at least three iterators: one to the upper left corner of the image, and two to the end of the first row and column respectively. However, if we use random access iterators, we can improve upon this by realizing that the two end iterators can be calculated from just one marking the position beyond the lower right corner of the image. Even simpler, we can then directly compare the `x` and `y` members of the two iterators. This results in the following basic navigation pattern:

```
template <typename ImageIterator>
void forEachPixel(ImageIterator upperleft, ImageIterator lowerright) {
    for(; upperleft.y != lowerright.y; ++upperleft.y) {
        ImageIterator i = upperleft;
        for(; i.x != lowerright.x; ++i.x) {
              // do something with *i, the current item
        }
    }
}
```

Figure 3 illustrates our convention of specifying a rectangular region by passing a pair of iterators to the upper left and beyond the lower right corners of the region. That is, lower right is outside the region, similarly to the end iterator in the STL.

Note that even in case of the `forEach()` algorithm it is necessary to use a 2-dimensional iterator. If we tried to use a linear iterator (such as a scan-order

[4] Past-the-border positions are those outside locations that have at least one direct or diagonal neighbor in the image.

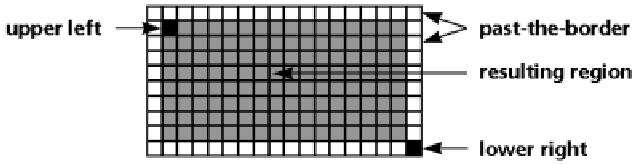


Fig. 3. Two image iterators at upper left and lower right (black squares) determine the region of interest (gray squares and upper left). There are 2(w+h+2) past-the-border locations (white squares and lower right).

iterator, which returns all pixels in row-major order), we would get incorrect subrange semantics. In image processing, a subrange usually means a subimage, which we can easily specify by moving upper left and lower right inwards by the appropriate amount. In contrast, a subrange of the scan-order iterator would correspond to a number of consecut ive lines plus two incomplete lines above and below them, which is hardly what we want.

The necessity of 2-dimensional access becomes even more apparent if we consider algorithms that involve a neighborhood around the current pixel. As a simple example, consider an algorithm that replaces a specific pixel by the average grey value of its 3×3 neighborhood (for simplicity, we omit bounds checking):

```
void avaragePixel(ImageIterator current)
{
    ImageIterator upperleft = current - Distance2D(1, 1);
    ImageIterator lowerright = current + Distance2D(1, 1);
    int sum = 0;

    for(; upperleft.y != lowerright.y; ++upperleft.y) {
        ImageIterator i = upperleft;
        for(; i.x != lowerright.x; ++i.x) sum += *i
    }
    *current = sum / 9;  // the average
}
```

Since images and matrices are usually random access data structures, one may ask why we are not simply using indexing instead of iterators. There are several reasons why iterators are more flexible:

– As in the STL, the iterator solution includes the indexing solution: a two dimensional indexing operator is supported by the `ImageIterator` via the syntax `i(10, 20)`.
– Iterators are easier to use with offsets relative to an arbitrary location. Instead of doing index calculations by hand, an algorithm can simply place the iterator on the reference position and use the relative indexes directly.

– It is easier to write iterator adapters if we already have iterators in the first place. Several interesting iterator adapters for the ImageIterator will be discussed in section 3.2.

2.3 Algorithms as Iterators

In our attempt to decompose functionality into orthogonal concepts, we realized that many geometric algorithms can be divided into a navigation part and a part that does not involve navigation (i.e., a part that does some computations with the current data items). Both types of functionality vary essentially independently of each other. Look, for example, at a graph traversal algorithm: many different algorithms use the same fundamental depth-first or breadth-first traversal patterns. When we encapsulate these patterns into independent building blocks, we can reuse one implementation with all of these algorithms. When we implement them in addition in the form of iterators rather than subroutines, it becomes much easier to pass these building blocks around, and we can even pass them to algorithms that don't know anything about graphs. For example, by passing a depth-first or breadth-first iterator to an algorithm like `print()`, we can print node information in either order without the need to implement a new algorithm.

The graph traversal example induces an interesting insight: iterators implementing algorithms do not always support the STL model of half-open ranges, because this requires that the end is known in advance. Often, the end is found only in the course of traversal, so that it must be reported via an `isEnd()` member function of the iterator. This method returns `true` if and only if all items of the sequence have been passed. We will call an iterator supporting the `isEnd()` function a *loose-end iterator*.

The most prominent example for loose-end behavior is an input stream: its end is only indicated by an EOF flag, which is set after the stream was closed. The STL (in its `istream_iterator`) chooses not to introduce a new category for this behavior, but experiences within the geometric realm suggest that this phenomenon might be so common as to justify a separate category. In Section 3.3 we show how the loose-end model can be adapted to the STL model.

Graph Traversal. There are various strategies for exploring a (connected) graph, for example *depth-first search* and *breadth-first search.* Such a strategy visits the nodes one after another, but the iteration is implemented by a navigation over the incidence structure of the graph (by jumping from node to node along edges).

It is quite natural to implement these strategies as a node iterator. Among other advantages, this makes it possible to visit all nodes once without the need for an additional list in which all nodes are maintained redundantly. The design of such an iterator class should incorporate the following insights:

1. It does not make much sense to specify the nodes to be visited as a range $[i \ldots j)$. In fact, only the start node i is known before the algorithm is executed, whereas $i+1, \ldots, j-1$ are unknown until the end of the algorithm.

2. In particular, the notion of subranges is not supported as it is for linear containers.
   There is one notable exception: it makes perfect sense to restrict such a strategy to a "subrange" that forms a prefix of the sequence of nodes visited regularly by the strategy. For example, depth-first search is often used to generate a path from some node s to another node t, which can be specified by a *closed* range $[s \dots t]$. A specification through a half-open range in STL style is not possible.
3. However, other kinds of prefixes cannot be specified by any kind of ranges (neither closed nor half-open nor open). For instance, breadth-first search proceeds in layers: first all nodes connected to the start node are visited (first layer), then all unvisited nodes connected to the first layer (second layer), and so on. Sometimes one is interested in iterating over all nodes in the layers $1 \dots k$ for some given k. Again, this forms a prefix; however, neither the last node of the prefix nor the first node of the rest is known, and thus no delimiter for the set of nodes to be visited can be specified.

A graph-search iterator class is easily and naturally implemented as a loose-end iterator as described above. In contrast, a one-past-the-end iterator does not make sense, because there is no designated one-past-the-end position—neither for the whole range of nodes nor for typical definitions of subranges.

3 Iterator Adapters

In the last section, we have identified a few scenarios in which a one-past-the-end iterator would be awkward or even meaningless. Nonetheless, it is often desirable to apply STL-style algorithms to such iterators. What we need here are *iterator adapters*, which adapt the syntax of their adaptees to the STL style. We will consider the individual scenarios from Section 2 in the same order in the following subsections.

3.1 From Cyclic Structures to the STL Model

In Section 2.1 we have seen that circular data structures are naturally described by circulators. The direct use of circulators requires special algorithms which know about the circulator's properties. This is okay as long as these algorithms are indeed most naturally implemented in terms of circulators, such as many algorithms on planar graphs (see Section 2.1). However, in some cases we will need to apply algorithms expecting an STL-compatible iterator to a cyclic data structure.

This can easily be achieved by wrapping circulators into STL-conforming adapters which encapsulate the special knowledge about circulator properties. In particular, these adapters hide the additional winding number, which says how many full revolutions the iterator is supposed to do. Two adapters compare equal if they point to the same location and their winding numbers are equal. This could be implemented as follows:

```
template <typename Circulator>
class CirculatorAdapter
{
    Circulator begin_, current_;
    int winding_number_;
  public:
    CirculatorAdapter(Circulator c, int winding_number)
    : begin(c), current_(c), winding_number_(winding_number)
    {}

    CirculatorAdapter<Circulator> & operator++() {
        ++current_;
        if(current_ == begin_)  ++winding_number_;
        return *this;
    }

    bool operator==(CirculatorAdapter<Circulator> const & c) const {
        return current_ == c.current_ &&
               winding_number_ == c.winding_number_;
    }
    ...
};
```

Given a circulator at an arbitrary position, it is now straightforward to construct an STL-conforming begin/end pair for exactly one circulation. In this case, begin and end just differ by the initialization of the winding number:

```
Circulator c = ...;
CirculatorAdapter<Circulator> begin(c, 0), end(c, 1);
```

Other ranges can be constructed similarly. Of course, we pay for compatibility with a slight runtime overhead (some additional checks), but in many applications this can be tolerated, considering the advantages.

3.2 One-Dimensional Subsets of Multi-dimensional Structures

Many algorithms operating on images are not interested in the entire image, but only in a one-dimensional subset of the pixels. There are many possibilities to define such subsets, and many algorithms don't care which subset is actually used as long as it is one-dimensional. Consequently, we implement these algorithms in terms of a linear iterator or circulator, and select the appropriate subset independently. Examples for useful linear subsets include single rows and columns, arbitrary straight lines, splines and even space-filling curves such as Hilbert's curve. We can also define many different circular structures such as circles, ellipses, rectangles, and the contours of arbitrary regions. These subsets are best implemented by means of adapters, mapping the two-dimensional image iterator to linear iterators or circulators respectively. Except under extremely tight performance constraints, it doesn't make sense to implement these one-dimensional views directly on top of the raw matrix, because this would require a separate implementation for each data structure, whereas the adapters can be reused for anything providing an ImageIterator.

To give an idea of the possibilities, we will describe an iterator adapter moving along an arbitrary straight line in the image. This LineIterator implements Bresenham's algorithm for traversing a line. That is, given a start point and an offset, the iterator will calculate the necessary increments and will advance by one pixel in the desired direction upon every call of operator++. A simple implementation of this adapter could look like this:

```
template <typename ImageIterator>
class LineIterator
{
    ImageIterator current_;
    float dx_, dy_, x_, y_;

  public:
    LineIterator(ImageIterator i, int xoffset, int yoffset)
    : current_(i), x_(0), y_(0), dx_(0), dy_(0)
    {
        int number_of_steps = max(abs(xoffset), abs(yoffset));
        if(number_of_steps > 0) {
            dx_ = (float)xoffset / number_of_steps;
            dy_ = (float)yoffset / number_of_steps;
            x_ = dx_ / 2.0;
            y_ = dy_ / 2.0;
        }
    }

    LineIterator & operator++()
    {
        x_ += dx_;
        if(x_ >= 1.0) {
            x_ -= 1.0;
            ++current_.x;
        }
        else if(x_ <= -1.0) {
            x_ += 1.0;
            --current_.x;
        }
        // likewise for the y coordinate
    }
    ...
};
```

Now we have several possibilities to mark the end of iteration. The most natural one is probably to implement the iterator as a loose-end iterator, because it knows its end anyway (note the variable number_of_steps in the constructor). This would correspond to the observation that many iterators implementing algorithms support the loose-end model (cf. Section 2.3). Alternatively, we can choose to provide iterator pairs. For example, we might create an end iterator by passing an `ImageIterator` at the line's target position to the `LineIterator`'s constructor. The working iterator compares equal to this iterator, when its `current_` member has reached the target position. However, this does not conform to the STL model, because we got a *closed* rather than a half-open interval (an at-the-end rather than one-past-the-end iterator). Additional effort is required to really construct a one-past-the-end iterator, which might indicate that the STL model is not the most appropriate one in this case.

The big advantage of these geometric iterators is that the number of algorithms can be reduced. For example, we only need one drawing algorithm for any kind of shape, because the shape is solely determined by the iterator. This pays off as soon as we have several algorithms that operate on different kinds of shapes.

3.3 Adaptation of Loose-End Iterators

Recall the loose-end iterator model from Section 2.3. A general template for adapting iterators with an `isEnd`-method to the STL style could look like this (brief sketch):

```
template <typename Iterator>
class IteratorAdapter
{
    Iterator iter_;
    bool is_dummy_;

  public:
    IteratorAdapter() : is_dummy_(true) {}

    IteratorAdapter(Iterator i)
    : iter_(i), is_dummy_(false) {}

    bool operator==(IteratorAdapter<Iterator> const & o) const {
        return (iter_.isEnd() || is_dummy_ ) ?
                              (o.iter_.isEnd() || o.is_dummy) :
                              (!o.is_dummy && iter_ == o.iter);
    }
    ...
};
```

4 Attribute Access

In [5], we proposed an extension of the STL guidelines, which we named *data accessors*. Basically, an object of a data-accessor class encapsulates the access to one specific attribute. It takes an iterator and a data accessor to read and write the value of an attribute for an item. For example, consider a sequence of type `struct Point`, where `Point` contains the `double` members `x` and `y`. Reading and writing attribute `x` for the item referred to by STL-style iterator `iter` is expressed by `(*iter).x`. This reveals the implementation of the abstract attribute `x` to all algorithms. In contrast, a data accessor `da` comes with methods `get` and `set` for reading and writing this value:

– Reading: `cout << da.get (it);`
– Writing: `da.set (it, 1.0);`

For example, when data accessors are incorporated into the design, the STL function `replace` would be declared like this:

```
template <typename ForwardIterator, typename DataAccessor,
          typename T>
void replace (ForwardIterator begin, ForwardIterator end,
              DataAccessor da, T old_value, T new_value);
```

To replace all x coordinates equal to 1.0 by 2.0 in a sequence `seq` of struct type `Point`, we could use the template class `MemberAccessor`, which was implemented in [5] and internally keeps a pointer-to-member [7]:

```
template <typename Str, typename T>
class MemberAccessor
  {
  public:
    MemberAccessor (T Str::*ptr)
          : i_ptr(ptr) { }
    template <typename Iterator>
       T get (Iterator it) const
          { return (*it).*i_ptr; }
    template <typename Iterator>
       void set (Iterator it, T value)
          { (*it).*i_ptr = value; }
  private:
    T Str::*i_ptr;
  };
```

This class can be applied as follows:

```
MemberAccessor<Point,double>  da (&Point::x);
replace (begin, end, da, 1.0, 2.0);
```

We refer to [5] for a more detailed and systematic discussion on the technical level[5] and to [8] for a discussion on the conceptual level in view of another

[5] This includes a critical discussion of another, more “STL-like” approach, which allows one to apply the STL function `replace` to a single item attribute instead of the item data as a whole: implementing an adapter class for iterators, whose sole purpose is to overwrite `operator*` such that it returns the attribute instead of the whole struct.

algorithmic domain: graph algorithms. In the next subsection, we will focus on another aspect, which has not been explicitly addressed in any previous work on data accessors but is at the heart of generic programming: to implement each component on the highest possible level of generality.

Remark. This concept even makes it possible to select the coordinate (*i.e.*, x or y) at run time, namely when the object `da` is instantiated. This aspect has been systematically investigated in [3].

4.1 Flexible Combination of Iterators and Data Accessors

Consider an algorithm such as the STL algorithm `remove_if`, which applies some predicate to all items of a sequence. In the STL design of `remove_if`, the predicate receives a value of the item type of the sequence for evaluation. The predicate is applied to an item by applying `operator*` of an iterator referring to this item and calling the evaluation method of the predicate with the result as the (unique) argument. However, the iterator object itself potentially yields more information than the result of `operator*`, and sometimes it is desirable to base the predicate on this additional information. We will first give an example where this additional information is necessary. Afterwards, we will show that data accessors allow generic implementations of such predicates.

For the sake of argument, suppose we are given a list of points as above, and we want to implement a predicate for items of this list that evaluates to `true` if the item is not the very last one and the x coordinate of this item is smaller than the x coordinate of the very next item. Hence, this predicate needs the values of both the current and the very next item. However, this predicate shall be implemented for algorithms such as `remove_if`, which do not "know" that the predicate accesses more than just the current position.

This example shows the limits of the STL version of `remove_if`: in many cases, it does not suffice to hand over the current item's value to the predicate. In fact, the predicate needs the current position to retrieve the next item's value. Of course, this position is given by an iterator.

However, it is desirable to implement such a predicate class as generically as possible. This class should cover all cases in which the value of some attribute of the current item is compared with the value of the same attribute for the very next item. So the following aspects should be left open in the implementation of the predicate class:

1. the comparison operation,
2. the type of the attribute,
3. the *name* of the attribute as a member of the struct,
4. even the fact that this attribute is simply implemented as a member of the (struct) item type.

Here is an implementation of this predicate class based on data accessors:

```
template <typename Iterator, typename Accessor, typename Comparator>
class SuccessorComparator
{
  Iterator    end_;
  Accessor    acc;
  Comparator  cmp_;

public:
  SuccessorComparator (Iterator end, Accessor acc, Comparator cmp)
  : end_(end), acc_(acc), cmp_(cmp)
  {}

  template <typename Iterator>
  bool operator() (Iterator it)
  {
    return it != end && cmp_ (acc_.get(it), acc_.get(it+1));
  }
};
```

Obviously, this implementation leaves open the above-declared aspects. On the other hand, leaving open the last two aspects might be next to impossible if data accessors (or an equivalent concept) are *not* used.

Remark. Again, a loose-end iterator would be more natural here: the first operand to the "`&&`" inside `operator()` would reduce to a call `it.isEnd()`, and there were no need for an additional end iterator object. This would allow us to make `Iterator` a template argument of `operator()` instead of `Successor-Comparator` itself, which means that one object of `SuccessorComparator` could serve all iterator types that are defined on the point list. In fact, `SuccessorCom-parator` itself only depends on `Iterator` because of the internal end iterator.

Conclusion

The STL was designed in view of fundamental, all-purposes algorithms and data structures. In this paper, we have seen that a specific application domain may suggest design decisions that are incompatible with the STL design. The question arises whether the design of a domain library should obey the STL guidelines or the needs of the domain. In the long run, the STL guidelines will be part of every C++ developer's background knowledge. This is a strong pragmatic argument in favor of the STL guidelines.

On the other hand, we hope that not only the syntax, but also the *spirit* of the STL will find its way into every C++ developer's mind. In our opinion, the spirit of the STL suggests focusing on the needs of the domain:

1. First focus on what the algorithms actually need from the data structures.
2. Then design the interface between data structures and algorithms according to these requirements.

3. Finally, if the interface offered by a data structure does not match the interface required by an algorithm, implement adapters to get the syntax right.

In our case this means: design your domain-specific library according to the domain-specific needs (as it was done in Section 2). If the STL algorithms are potentially useful for your data structures, provide appropriate adapters (as in Section 3).

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.L., *Design patterns, elements of reusable object-oriented software* Addison-Wesley, 1994.
2. Fabri, A., Giezeman, G.-J., Kettner, L., Schirra, S., Schönherr, S., *On the design of CGAL, the computational geometry algorithms library*, ETH Zürich, Department Informatik, Technical Report no. 291, 1998.
3. Gluche, D., Kühl, D., Weihe, K., "Iterators evaluate table queries," *ACM Sigplan Notices* **33**(1):22–29, 1998.
4. Köthe, U., "Reusable software in computer vision," to appear in B. Jaehne, H. Haussecker, P. Geissler (eds.), *Handbook of Computer Vision and Applications*, Vol. 3, Academic Press 1998.
5. Kühl, D., Weihe, K., "Data access templates," *C++ Report* **9**(7):15–21, 1997.
6. Musser, D., Saini, A., *STL tutorial and reference guide*, Addison-Wesley, 1995.
7. Stroustrup, B., *The C++ programming language (third edition)*, Addison-Wesley, 1997.
8. Weihe, K., "Reuse of algorithms—still a challenge to object-oriented programming," *Proceedings of the 12th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 34–48, 1997.
9. Weihe, K., "A software engineering perspective on algorithmics," `http://www.informatik.uni-konstanz.de/~weihe/manuscripts.html`, 1998.

Generic Graph Algorithms

Dietmar Kühl

Claas Solutions GmbH
dietmar.kuehl@claas-solutions.de

Abstract. Implementing non-trivial algorithms, like many graph algorithms, is generally expensive. Thus, it is desirable to reuse such implementations whenever possible. The implementations of graph algorithms normally cannot be reused mainly because the representations of graphs differ in many ways and the implementations normally assume one specific representation. This article presents an approach to overcoming this problem which is based mainly on special iterators and special means to access data associated with objects.

Keywords: graph algorithm, iterator, data accessor, algorithm object

1 Introduction

Implementing graph algorithms is generally a difficult enterprise due to the complexity of the algorithms. Thus, it is desirable that an implementation of a graph algorithm is widely applicable. That means, in an ideal world it should be possible to apply the implementation of a graph algorithm to an arbitrary graph representation.[1] Another aspect of the implementation of graph algorithms is extensibility: graph algorithms for complex problems often extend more basic algorithms. Thus, it should be possible to inject additional computations and control into the implementation of an algorithm.

In order to implement an algorithm independently from the representation of the graph an abstraction is necessary. Any abstraction, independent from the problem domain, should have some basic properties:

– It should be possible to implement the elements of the abstraction to interface any suitable representation.
– Using the abstraction for an implementation should not impose any overhead compared to one using the representation directly, at least for typical representations (for some representations an overhead cannot be avoided).
– The abstraction should be small: both the number of interfaces required by the abstraction and the work necessary to implement these interfaces (for a typical representation) should be small.

[1] Of course, this only makes sense if the graph and its representation are suited to the requirements of the algorithms: for example, it makes no sense to apply an algorithm for planar graphs to a non-planar graph. However, the implementation should not impose requirements beyond those imposed by the algorithm.

– For special problem cases the abstraction should provide a means for an implementation to exploit the special structure. For example, an abstraction for sequences can provide a possibility to take advantage of the fact that a special sequence provides random access to the elements.

The standard C++ library presents an example of an abstraction having all those properties, namely the iterators used by the Standard Template Library (STL [6]). Basically for everything which can be seen as a sequence it is possible to provide iterators. The programs using STL's iterators to implement algorithms on sequences are as fast as programs coding the algorithms on corresponding representations directly. The abstraction is fairly simple requiring just a few easily implemented operations for iterators. Using different requirements for iterators plus some supporting definitions (iterator traits), algorithms can take advantage of special structures of representations.

Since graphs are a more complex structure than sequences, the abstraction for graphs is also more complex than the abstraction for sequences. For the effective implementation of graph algorithms it is necessary to access the structure of the graph and to access data associated with the nodes and the edges separately. The graph's structure is accessed using several kinds of iterators and the details of accessing the data associated with an object are encapsulated by data accessors. The details of the various components are given below.

Often, graph algorithms are implemented as an extension of more basic algorithms. For example, a simple algorithm for finding a directed path between two nodes is an extension of a graph search algorithm such as depth first search or breadth first search: starting at one of the two nodes, it remembers the used edge when a new node is found and terminates the search algorithm when the second node is found. This, however, requires the possibility of injecting additional operations into the implementation of the search algorithm and the possibility for premature termination of the search algorithm. Support for such modifications can be achieved by implementing the algorithms as objects which are used in a manner quite similar to iterators: these algorithm objects iterate over the intermediate states of an algorithm's execution.

2 Theoretical Background and Related Work

There is a lot of literature on graph algorithms available: basic graph algorithms like graph search or simple flow algorithms are covered in most general introductions to algorithms, such as [2]. In addition to basic algorithms, more specialized topics are also covered in many books, for example the area of network flow algorithms is covered in [1] (algorithms from this book were used in experimenting with the approach described here). It is worth noting that the literature normally describes the algorithms in a generic fashion: the basic assumptions on the graph data structure used are described and the algorithms themselves are described in a data structure independent fashion. However, the pseudo code used is normally not directly suitable for an implementation because the abstraction used is too high level. As a result, implementations of graph algorithms are normally not generic.

An example of a graph algorithms library is LEDA (Library for Efficient Data structures and Algorithms, [5]). This library, implemented in C++, provides several graph algorithms in addition to algorithms for other areas like computational geometry, but it is basically impossible to use those algorithms on a different graph data structure than those also provided by LEDA. It is only generic in terms of the node and edge data associated with the nodes and edges of the graph data structure, which can be specified using template arguments.

A more recent development is GGCL (Generic Graph Components Library, [4]) which clearly spells out the requirements on the data structure to be used as input to one of the GGCL algorithms. The goal of this library is to be a generic library for graph like the STL is for sequences. The abstraction used is a whole graph resulting in a lot of work to get an existing data structure to conform to these requirements especially if the data structure was created before the requirements of the particular library are known. This library uses an approach quite similar to data accessors, called decorators, to access data associated with nodes and edges.

3 Data Accessors

The first hurdle that appears when trying to implement graphs in a representation-independent fashion is access to data used by the algorithm. Graph algorithms use some data associated with the nodes and some with the edges of the graph. The data to be accessed ranges from auxiliary labels used internally by the algorithm to data which is part of the problem. For example, most graph algorithms use some auxiliary label to indicate whether a node was already processed and an algorithm finding the shortest path between two nodes in a network uses the length of edges as part of the problem.

Here are some aspects which have to be taken into account when deciding how to access this data:

- Graph algorithms often use many different entities associated with nodes or edges. For example, algorithms dealing with network flows may access a lower and an upper bound of the flow, the cost of the flow, and the current flow for an edge plus several auxiliary fields.
- There may be several different representations of the same data with different performance trade-offs. For example, in network flow algorithms often both the free capacity and the current flow are used which are closely related: The free capacity is the upper bound minus the current flow. Thus, often only one of those values is represented while the other computed.
- The same data can be used under different names in different contexts. For example, what is a weight for one algorithm can be the cost or the length in the context of another algorithm. Since graph algorithms often use other algorithms to solve subproblems, it is necessary to avoid this naming problem.

– It should not be necessary to explicitly store data which can be computed from other data. For example, in the shortest path example the length of all edges might be the same (i.e., a unit length) or it might be possible to compute it from the coordinates of the nodes.

An approach to access data is to use an abstraction of the actual accesses. To do this, objects called *data accessors* are used (see [3] for detailed information on data accessors). Data accessors have methods to read and/or write data and encapsulate the details how the data is actually represented. The access methods take an iterator as an argument identifying the object whose associated data is to be accessed. If data is to be stored with this object, the corresponding new value is also passed.

Uses of a data accessor can for example look like as follows (using C++ as implementation language:[2])

`get(`*da*`,` *it*`)` Read the data associated with the object identified by `it` using the data accessor `da`

`set(`*da*`,` *it*`,` *val*`)` Associate `val` with the object identified by `it` using the data accessor `da`

In these examples `da` is a data accessor object, `it` is some appropriate iterator type, and `val` is on object of the type of data associated with objects by the data accessor `da`.

Basically, if the data associated with the objects in a graph is regarded as a table, an iterator identifies a row in the table. A data accessor identifies a column. Both together identify the data that is actually accessed. However, the representation of the table is kept open: a part of the columns can be organized "row-wise"; that is, the data is stored directly in the corresponding objects. Another part of the columns can be organized "column-wise"; that is, the iterator is used to find the data, for example in a hash table. Yet another part of the columns may be represented implicitly, if for example all entries in a column have the same value or because they are computed from other columns.

Normally, data accessors depend on the iterators with which they are used to access the actual data. For example, a data accessor using an STL-style iterator may access a component using the dereference operator to access the identified object and select a specific member of this object. Another data accessor may instead extract an index from the iterator and use this to index an array stored in the data accessor object. All of this gets a bit tricky if iterators are to be wrapped to modify the view of the graph (see the next section) because it is necessary to provide a way to extract the original iterator. However, this is more a technical problem than a conceptual one, with several more or less generic solutions.

[2] The methods are not implemented as class members but rather as global functions for technical reasons.

4 Iterators

For exploring the structure of a graph three kinds of iterators are used:

– Node iterators are used to access the nodes of a graph.
– Edge iterators are used to access the edges of a graph.
– Adjacency iterators are used to access the nodes adjacent to a node.

These three iterator types share the property that the data associated with the objects identified is accessed using data accessors. For nodes and edges it is obvious that the data accessed is the data associated with a node or an edge, respectively. For an adjacency iterator, the data accessor defines what is accessed: an adjacency iterator identifies a node whose adjacent nodes it explores. It also identifies a current adjacent node and the corresponding edge (the edge is important if there are parallel edges—multiple edges connecting the same two nodes—in the graph, which is possible in most algorithms).

The node and edge iterators are just iterators for some sequence, like STL iterators. For edges it is often useful to access the corresponding incident nodes. To allow the use of iterators for some sequence of edges as the implementation for edge iterators, the incident nodes are accessed using corresponding data accessors which return node iterators. For nodes it is necessary to find the incident edges and/or the adjacent nodes. This is done using adjacency iterators.

Adjacency iterators are special for graphs and are used to explore the local structure of a graph. An adjacency iterator iterates over the nodes adjacent to a fixed node. At the same time it also iterates over the edges incident to that node. Actually, for graphs with parallel edges it is often useful that adjacency iterators iterate over all edges incident to a node. A possible set of methods (in addition to general functions like constructors, assignment, and destructor) for an adjacency iterator could be the following (`ait1` and `ait2` are adjacency iterators):

`ait1.valid()` Return `true` if there still is an adjacent node
`ait1.next()` Advance the iterator to the next adjacent node
`ait1.cur_adj()` Return an adjacency iterator for the current adjacent node[3]
`ait1.same_node(ait2)` Return `true` if both iterators are positioned on the node (independent from the current node)
`ait1.same_edge(ait2)` Return `true` if both iterators are positioned on the same current edge

Using adjacency iterators, edges are always viewed as directed, namely from the fixed node to the current adjacent node. This view is, obviously, appropriate for directed graphs. Since the direction of edges does not matter for undirected graphs, it is also appropriate for undirected graphs.

[3] Recent discussions with developers using these concepts suggest that the current adjacent node should be accessed using a data accessor rather than a special method. Using data accessors improves the performance in some applications because storing many adjacency iterators can be done more efficiently in terms of memory.

5 Modifications to Graphs

Often, graph algorithms apply temporary changes to the graph structure. For example, in flow algorithms edges without free capacity are removed and instead edges with the opposite direction are inserted to indicate that it is possible to send flow in this direction. Another typical modification is to join two or more nodes forming a new node which is adjacent to all nodes which were previously adjacent to at least one of the original nodes. These modifications are only relevant for the execution of the algorithm: they only change the view of the graph taken by the algorithm. Also, it is sometimes necessary to undo the modification during the algorithm.

Using the abstraction formed by data accessors and iterators, it is possible to keep the original representation intact and only modify the view. This is realized by implementing new iterators which store internally the original iterators but apply additional logic when performing operations on them. For example, if an edge has no free capacity it can simply be ignored by an adjacency iterator which is built from the original adjacency iterator and a data accessor used to access the free capacity of an edge. Advancing the new iterator once may advance the underlying iterator multiple times until either an edge with free capacity is found or all edges are investigated.

6 Algorithm Objects

Graph algorithms are often slightly modified to suit specific needs. For example, during the execution of an algorithm certain intermediate data may be recorded (a typical example is the depth first number during the execution of a depth first search) or some data is collected to provide a "certificate" justifying the correctness of a result. Although the additional computations are essential in some situations, they are completely useless in others.

To provide the possibility to modify an algorithm, non-trivial algorithms should be implemented as *algorithm objects* which behave much like iterators. The major difference between normal iterators and algorithm objects is that the latter iterate over a set of intermediate states during an algorithm instead of iterating over a collection of objects. Together with appropriate access functions to access the current state of the algorithm, this can be used for example to record intermediate data, inject additional computations, modify the behavior of the algorithm, terminate the algorithm prematurely after the desired result is computed, preempt the algorithm, or to provide recovery points to avoid loss of computed results during time-consuming computations.

The basic idea is to split the execution of an algorithm into basic steps, providing the user fine control over the behavior. For example, a graph search algorithm may be implemented such that visiting a node is a basic step. The corresponding algorithm object would store all necessary data to continue the algorithm. In the case of a graph search algorithm this would be a container of nodes for which processing is not finished and a data accessor used to label visited nodes. Also, the algorithm object would provide methods to access important data about the current state, for example the newly found object.

This approach is, of course, not specific to graph algorithms. However, it is not (yet) widely applied.

7 Future Work

Up to now, only a few algorithms have been implemented based on the ideas in this article: the algorithms used to develop these approaches and some algorithms in research projects (where this abstraction was only secondary part of the research). What is missing is a library which implements at least all basic graph algorithms such that they are reusable. In this context it would be useful to address the problem of special graph properties: so far, the abstraction is well suited for general graphs but it does not take any special properties like planarity into account.

Also, the concepts presented in this article only address temporary modifications of graph. For permanent modifications additional problems arise because not all graph representations support all operations that modify a graph. For example, a representation of a planar graph cannot allow the insertion of an arbitrary edge because this might violate planarity. In this direction a suitable abstraction is still missing.

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B., *Network Flows*, Prentice Hall, 1993.
2. Cormen, T.H., Leiserson, C.L., Rivest, R.L., *Introduction to Algorithms*, MIT Press, 1990.
3. Kühl, D., Weihe, K., "Data Access Templates," *C++-Report*, 9, 1997, 18-21.
4. Lumsdaine, A., Lee, L.Q., Siek, J., *The Generic Graph Components Library*, 1999, `http://lsc.nd.edu/research/ggcl/`
5. Mehlhorn, K., Näher, S., "LEDA: a library of efficient data structures and algorithms," *Communications of the ACM*, 38, 96-102, 1995.
6. Musser, D., Saini, A., *The STL Tutorial and Reference Guide*, Addison-Wesley, 1995.

A Generic Programming Environment for High-Performance Mathematical Libraries[⋆]

Wolfgang Schreiner, Werner Danielczyk-Landerl,
Mircea Marin, and Wolfgang Stöcher

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
`FirstName.LastName@risc.uni-linz.ac.at`

Abstract. We report on a programming environment for the development of generic mathematical libraries based on functors (parameterized modules) that have rigorously specified but very abstract interfaces. We focus on the combination of the functor-based programming style with software engineering principles in large development projects. The generated target code is highly efficient and can be easily embedded into foreign application environments.

Keywords: functors, specifications, computer algebra, generic libraries.

1 Overview

We describe the preliminary results of a project that pursues the development of a programming environment for the construction of generic mathematical libraries. The project employs *functors* (parameterized modules) as the basis for constructing multi-layered software libraries for solving problems in various mathematical domains. Our work in this direction has been inspired by previous attempts on building generic computer algebra libraries that were seriously hampered by a lack of appropriate language support [7,12].

Module *parameterization* as the basic device for building large software systems has been pioneered by the functional language SML [1] and by special-purpose computer algebra languages [2,8,6]. Nevertheless, the main-stream in software engineering concentrates on the object-oriented notion of type *inheritance* for building reusable software [5]. However, it is difficult to model mathematically axiomatized structures by inheritance; parameterized structures offer a much more natural framework.

In industrial programming languages, parameterized program units have only found limited support. A major exception is the *template* feature in C++ which is the basis for the Standard Template Library [10]. However C++ templates do not provide means for specifying information about their parameters; thus the body of a template cannot be compiled (typed-checked) on its own. Another

⋆ Supported by the Austrian Science Foundation FWF grant P11414-ÖTE "HPGP — High-Performance Generic Programming"

problem is the inability of many C++ implementations to automatically share instantiated code among different users of a template library. Both problems make templates difficult to use in large-scale software development [3]. On the contrary, SML's functors are statically type-checked; however, they are compiled to function tables which are looked up at runtime with corresponding performance overheads. Both C++ and SML do not provide means for the specification of a component's behavior (apart from its syntactic interface and type signature).

The goal of our project is to provide an environment for the multi-user development of generic mathematical libraries without compromising the performance of the generated code. Our approach is based on the following concepts:

Low-level core language We utilize a low-level core language for writing the executable entities (functions, procedures) that are embedded in and exported by modules. This core language does not provide any powerful builtin concepts that would require a complex runtime system; it is essentially a simple subset of C. Our goal is to build all abstractions necessary for convenient programming in the generic programming environment itself.

Powerful abstraction facilities The environment provides powerful facilities of constructing generic software libraries. *Specifications* define the syntactic and semantic interface of modules; *functors* describe how new modules can be constructed from other modules; *modules* are constructed by application of functors to modules; *packages* group specifications, functors, modules, and other packages together.

Efficient implementation The implementation is carefully designed to make sure that abstraction is not punished by runtime overhead. Information about the implementation of an entity is transparently propagated through all levels of module abstractions such that ultimately the same machine code is generated than without using the abstraction. To avoid code explosion, equivalent functor instantiations do not generate duplicate code.

Support for project management Packages may be jointly developed and utilized by multiple persons in a shared environment. The system is designed and implemented such that code is never unnecessarily duplicated, that permissions are set correctly, and that dependencies between different units are automatically maintained.

The last two items raise issues that are crucial for the practical success of a generic programming environment. If generic abstractions are penalized by performance loss, by code explosion, or by management overkill, programmers are tempted to avoid them. We have taken considerable efforts to overcome these problems in order to develop efficient code that is independent of any builtin types and type constructors, and that does not hardwire module dependencies.

In the following section, we will sketch our approach to tackle these requirements. Our goals are not yet completely fulfilled, because the environment is currently restricted to first-order functors, i.e., functors cannot be components of modules or passed as arguments to other functors. However, after the current status of the development has been consolidated, we intend to further develop the system into a higher-order framework [9].

2 The Programming Environment

We are going to explain the environment in an informal style demonstrating its most important features on a case-by-case basis. A systematic description and user manual will be soon available in [11].

2.1 Packages

Packages represent the basic device for imposing *structure* on libraries; they allow to group related program units (specifications, functors, modules, and packages themselves) under a common header. Packages also restrict references to units outside the package boundaries by import and export interfaces; this enables the package maintainer to control inter-package dependencies.

A package is created by compiling a description such as

```
package Arithmetic<>Algebra
{
  use package Basic;
  import all of Basic<>Standard;
  export spec Ring, Field;
  export module IntRing
}
```

which introduces a subpackage `Algebra` of package `Arithmetic`. By default, the only units that may be referenced by units in `Algebra` are those defined within the package itself. The `use` directive, however, also gives access to the units exported by `Basic` (which must be visible in parent `Arithmetic`). The `import` directive adds all units of package `Standard` to the environment of `Algebra` as if they were defined within this package. Units of other packages may only refer to those units of `Algebra` that are listed in the `export` directive.

Our package concept is strictly more flexible and powerful than C++ namespaces or Java packages: explicit interfaces restrict the use of units from other packages; units may be used, imported, and exported under a different name (decoupling the name of a unit from references to the unit); units imported into a package may be as well exported from that package (up to the point of completely "virtual" packages that do not contain actual code but collect units from other packages). Since package descriptions may cyclically refer to each other, mutual references of units from different packages are allowed.

Compiling a package description creates the physical directory structure required for installing units within this package. Its essential purpose however is to set up the name space that maps unit references to file paths and that is used by every unit of this package to get access to other units. Logical unit references are therefore strictly separated from physical file locations, which is important for managing the development and evolution of large libraries.

As set up by the package description, the name space is fixed for all units in this package; this also helps to establish a uniform convention for external references across the source code of a package.

2.2 Specifications

General. A specification describes the syntactic and semantic interface of a module. A sketch of such a specification is shown below:

```
spec Standard<>Bool
{
  type Bool                                 // types

  constr b: Bool()                          // constructors
  constr b: Bool(c: Bool)                   // and destructors
  destr b: Bool()

  let type B builtin Bool                   // mapping to
  in                                        // builtin boolean
  {
    fun trueValue(b: Bool): B
    axiom type Bool truth trueValue
  }

  fun operator ==(x: Bool, y: Bool): Bool  // equality on Bool
  axiom type Bool equality operator ==
  ...
  fun operator !(b1: Bool) b2: Bool        // negation
  output (b1 == true)  && (b2 == false)
      || (b1 == false) && (b2 == true)
  ...
  proc operator =(out b1: Bool, b2: Bool)  // assignment
  output b1 == b2
  ...
}
```

This specification `Bool` in package `Standard` introduces a type `Bool` with associated constructors/destructors, values, functions, and procedures (an explanation follows below). A specification may also contain modules as in

```
spec Array
{
  module I: Index
  module E: Element
  axiom type I::Bool = E::Bool
  import I
  import E except Type
  ...
  fun operator ==(a: Array, b: Array) =
    maxIndex(a) == maxIndex(b) &&
    forall(i: Index) (0 <= i && i < maxIndex(a) => a[i] == b[i])
}
```

An entity *c* within module *M* may be referenced as `M::c`; the current environment inherits the contents of a module by an `import` clause (optionally with restricting qualifiers such as `except` or `only`).

Blocks. A block `let` S_1 `in` S_2 introduces in specification S_1 entities that may be used locally within specification S_2; however, only the contents of S_2 become part of the enclosing environment. Thus an implementation of this specification needs not provide an implementation of S_1; this construct therefore allows to express existential quantification in specifications.

Types. A type is identified by a name; by default, different names denote different types. Functions/procedures with the same name but with different argument number or argument types can coexist in the same scope ("overloading").

The construct `axiom type` T_1 `=` T_2 constrains two names T_1 and T_2 to denote equal types; without this constraint, specification `Array` would presumably not type-check because `I::Bool` and `E::Bool` would denote different types. The construct `axiom module` M_1 `=` M_2 makes all types in M_1 equal to the corresponding types of M_2 (which must satisfy the specification of M_1).

Constructors and Destructors. A constructor is a procedure associated to a type. The procedure is invoked by a declaration of a value/variable of this type. E.g,

```
var x: Bool(y)
```

calls the constructor `Bool` with argument `y` to initialize the `Bool` variable `x`. On exit from the scope, the destructor `Bool()` is invoked. Constructors may be also given explicit names which allows declarations such as `var x: Int‘one()`.

Special Axioms. The construct `axiom type` T `truth` t specifies T as a truth type by stating that the function t maps any T object to a corresponding machine ("builtin") boolean. This specification enables us to write for a T object x a core language statement

```
if (x) ... else ...
```

which the compiler translates to

```
if (t(x)) ... else ...
```

In a similar way, we detach all core language constructs from machine types.

The construct `axiom type` T `equality` e declares e as the equality function on T. This states that, if e yields true on two arguments of type T, these arguments cannot be distinguished by any user of the module being specified (even if the internal representations of these objects are different).

Parameter Modes. In procedure specifications, the parameter modes `in` (default), `inout`, and `out` specify whether the object denoted by the parameter is modified by the procedure, i.e., whether the object is used by the procedure and whether its value after the call equals its value before the call. A parameter mode does not yet imply a particular argument passing strategy; such a strategy is selected by the compiler at instantiation time from the mode and from the concrete parameter type. Thus in contrast to C++, if an `in` argument happens to be passed by value, just the argument's representation is copied but no constructor is invoked (the programmer is not able to modify the parameter).

Pre- and Post-conditions. Specifications may attach pre-conditions and post-conditions to functions and procedures. Such conditions embed expressions (of some truth type) that may be directly executable as in

```
fun operator /(x: Int, y: Int) z: Int
input y != 0 && y | x
output x == y*z
```

but may also embed universal or existential quantifiers as in

```
fun operator |(x: Int, y: Int) r: Bool
output r == (exists(z: Int) y == x*z)
```

In debugging mode, the executable (parts of) axioms are compiled to *assertions* that are implicitly invoked with every application of the corresponding entity.

Constructive Specifications. From function specifications like

```
fun operator <=(i: Int, j: Int): Bool =
  (i < j) || (i == j)
```

the compiler generates default implementations provided that the right-hand side represents an executable expression. In a corresponding functor definition, the programmer may request to use this default implementation, or she may explicitly give an (equivalent but presumably more efficient) implementation.

Parameterized Specifications. To increase its applicability, a specification may be also parameterized, as in

```
spec Arithmetic<>EuclidRing[Bool: Bool]
{
  include Ring[Bool]
  ...
}
```

which includes the content of another parameterized specification `Ring` (and is a syntactic shortcut for declaring a local module with this specification and importing its content).

Specification Checking. As sketched above, specifications define the interface of a module. The compiler can check whether a module confirms to the static signature of a specification (consisting of the names and the types of the exported entities); it is not able to verify whether the module actually satisfies the dynamic axioms. However, the main purpose of axioms is to rigorously specify the most important aspects of a behavior of a module such that, e.g., an implementation can be *falsified* by a counter example. In the form of *assertions and default implementations*, axioms also have some computational significance.

2.3 Functors

General. A functor constructs a result module from modules passed as arguments; both argument and result modules are constrained by specifications. E.g.,

```
functor Pair[X: Type, Y: Type]: Pair[X, Y]
{
  type X = X::Type
  type Y = Y::Type
  type R = Record[x: X, y: Y]
  type Pair new R
  ...
  constr p: Pair(x: X, y: Y) inline
  {
    constr repr p: R()
    constr (repr p).x: X(x)
    constr (repr p).y: Y(y)
  }
  reffun operator .x(inref p: Pair): X inline
  {
    return (repr p).x
  }
}
```

satisfies the specification

```
spec Pair[X: Type, Y: Type]
{
  axiom type X::Bool = Y::Bool
  type Pair
  constr p: Pair(x: X::Type, y: Y::Type)
  reffun operator.x(inref p: Pair): X::Type
  ...
}
```

for some modules X and Y that satisfy `Type` and have equal `Bool` types.

New Types. Above functor constructs a "new" type `Pair` based on the representation type `Record[...]`. There are no operations, constructors, or destructors defined on the new type; however, the expression

```
repr r
```

maps an object r of this type to an object of the underlying representation type. Thus a `Pair` constructor can be defined that first calls the `Record` constructor and then initializes the components by explicit constructor calls of the form

```
constr exp: Type(args)
```

Then we can create a value or variable of some instance of `Pair` as

```
type P = Pair[Int, Char]::Pair
var p: P(172, 'a')
```

where `Pair[Int, Char]` denotes the application of functor `Pair` to modules `Int` and `Char`, which creates a module that exports a corresponding type. The constructor associated to this type is used to declare the variable p.

This example demonstrates how type constructors like “Pair”, “Array”, or “Pointer” can be implemented by functors; consequently the core language does not provide any such concepts as “builtin”. The only restriction is that our environment does not yet allow functors with an arbitrary number of arguments, thus there is still the necessity of the builtin constructor `Record` shown above.

Reference Functions. The reference function `operator .x` may be used as

```
val x0: Int = p.x
p.x = x0
```

to get access to this component. The compiler parses this construction as

```
val x0: Int(operator .x(p))
operator =(operator .x(p), x0)
```

exhibiting that the result of the reference function is used as an output argument for procedure `operator =`. If `operator .x` is not visible in the environment (by import of `Pair[Int, Char]`), the qualification `p.[P]x` denotes the module P where `operator .x` is embedded. The `inline` tag in the function definition (not specification!) ensures that the code is inserted for every application.

Generic Literals. A literal such as “172” is essentially parsed as

```
operator 0d(#1, operator 0d(#7, operator 0d(#5)))
```

where `operator 0d` is a function and `#1`, `#7`, `#5` are values visible in the current scope. They are typically introduced by the import of a module that exports

```
val #0: Int
...
fun operator 0d(x: Int, y: Int)
```

which allows to use integer “literals” for type `Int`. Likewise, floating point literals (`1.72`) and string literals (`"abc"`) are not restricted to machine types (if defined by builtin operations, literal values are still computed at compile time).

Initializers and Terminators. In order to setup the state of the result module (represented by globally declared variables and values), a functor may contain one ore more initializer and terminator statements (similar to Java). Initializers are automatically executed before the module is executed, e.g.,

```
val #0: Int
init
{
  constr repr #0: I‘zero()
}
```

initializes the value `#0` by a call of constructor `zero` in representation type `I`. Different from C++, module initialization proceeds in dependence order, which ensures that the arguments of a functor instantiation are initialized before the result is initialized. Correspondingly, terminators ensure proper deinitialization.

Builtin Objects. Since any module hierarchy has to be ultimately based on machine objects, we need an interface to refer to these. A definition

```
type I builtin Int
fun operator +(x: I, y: I): I builtin operator +
```

defines a function `operator +` by the builtin operation `operator +` with the denoted type signature. This interface is only intended for the implementation of a fixed set of low level modules whose internals are subsequently hidden.

External Interfaces. In order to get access to code written in other programming languages, definitions like

```
fun operator +(x: I, y: I): I extern intSum
```

may be used. This generates an external reference to a function `intSum` for which an external object has to be provided and linked to the application program.

2.4 Modules

A module is composed by a nested functor application with references to other modules (possibly defined locally within the definition), e.g.

```
module Application<>Main: Main =
let
{
  module I: Int = Language::Int
  module B: Bool = Language::Bool
  module R: Rec = Pair[I, B]
} in
  Main[Language, R]
```

References to concrete modules (e.g., `Language`) are restricted to module definitions; functors can access modules only by their parameters. In the ultimate higher-order framework, we intend to restrict functor access in the same way such that a functor does only depend on its parameter interface (without "hardwired" functor dependencies).

We also allow *cyclic* module definitions such as in

```
module Cycle: Spec =
let
{
  module A: S1 forward
  module B: S2 = F1[A, G<>M]
  module A: S1 = F2[B, G<>N]
} in
  F3[A, B, G<>O]
```

where modules `A` and `B` depend mutually recursively on each other, i.e., each module can refer to the entities defined in the other module. Such cyclic dependencies are hardly good practice and should be avoided. However, in a couple of situations it is extremely hard to cope without them; we support this possibility for pragmatic reasons.

3 Implementation

File System Structure. An environment variable holds the names of the directories that represent the root of the package hierarchy. When a program unit located in package P is compiled, a subdirectory is generated in the corresponding location:

```
/P                  package P
  /spec
    /S                  specification P<>S
  /functor
    /F                  functor P<>F
      /instances
        /0                instance 0 of functor P<>F
  /module
    /M                  module P<>M
  /package
    /P0                 package P<>P0
```

The programming environment transparently maintains dependencies and automatically recompiles outdated units without any assistance from the programmer. This is an important issue in a generic context where many instances of a functor may exist and user-assisted management is practically impossible.

Symbol Tables. Every program unit is compiled to a symbol table which holds all information known about every exported entity. For a specification, this is the name and the type of the entity plus all (executable) axioms attached to it. For a functor, this also includes the definition of each used entity, i.e.,

– by a functor parameter (no definition is available yet),
– as a builtin object (implemented by the compiler),
– as an external object (implemented as an external reference),
– by explicit code (translated to C++),
– by constructive axioms (converted to explicit code),
– by another entity (from which it receives its definition).

The last case is relevant for entities that are defined as an `alias` of (i.e., by just renaming) another entity.

Instantiation. When a module definition is compiled, all denoted functor instantiations are resolved. If an up to date instance of the same functor with the same module arguments already exists, no code is generated for the module and the corresponding symbol table is read from file.

If a new instance of a functor has to be created, the symbols in the functor parameters (available from the functor symbol table) are temporarily linked to the corresponding symbols of the argument modules (available from the module symbol tables) thus resolving all definitions. A simplified duplicate of the functor symbol table is generated as the symbol table of the new module. Furthermore, the code generator produces C++ source files, from which the native compiler generates object code. For explicitly coded entities, the module symbol table does not contain the definition itself but just the information how this entity can be referenced (typically the external name).

Since instantiation transparently propagates all information that is available on the definition of a symbol, the efficiency of access to an entity is independent of the layers of abstractions that were put between the definition of the entity and its use: an entity defined as `builtin` remains known as builtin, every access to an entity defined with an `inline` tag is resolved by inserting this code. Unlike C++, the instantiation process is not limited by any object code boundaries!

Target Code. A functor instance is compiled to a set of C++ (essentially C) include and source files which are in turn compiled by the GNU g++ compiler to object code. The include files contain type definitions, function prototypes, and function inline code for use by other instances. The interface to this instance is exhibited to the C++ level by compiling a module with option `-export`. This generates an include file for use in the application source code and a makefile that defines the list of all object files that have to be linked to the application.

The entities of an exported module are given externally visible C++ names as sketched in the following export header file of an instance of functor `Int`:

```
#ifndef Int_h
#define Int_h

/* module header file */
#include "/test/module/Int/instances/0/Int_0.h"

/* module initialization/termination */
#define init_Int() (_init_Int_0())
#define term_Int() (_term_Int_0())

/* types and their constructors/destructors: */
#define _type_Int_Int          _0_Int_0      /* type Int */
  #define _constr_Int_Int_0(o) ((void)0)     /* constr: Int() */
  #define _constr_Int_Int_1(o, x) ((o) = (x))/* constr: Int(Int) */
  #define _destr_Int_Int(x)    ((void)0)     /* destr: Int() */

/* procedures, functions, values and variables: */
#define Int_print ...                        /* proc(Int) */
#define Int__op_assign(x, y) ...             /* proc(out: Int, Int)*/

/* contents of exported modules: */
  /* types and their constructors/destructors: */
  #define _type_Int__Bool_Bool ...           /* type Bool */
  ...
#endif
```

The generated object code is of the same efficiency as if generated from a non-generic C++ program. It can be also conveniently used with symbolic debuggers, because pragmas refer to the correct locations in the functor source code.

4 Conclusions

Lack of space prohibits us from describing in more detail the specification and functor language, its type system, the implementation of instantiation, and of code generation [4]. The system has been developed in a very restricted subset of C++; it is reasonably stable such that first major experiments have become possible. As an important practical aspect, the environment still lacks an integrated revision control system; major missing language features are higher-order functors and functors with a variable number of arguments.

We have built a package `Standard` that specifies and implements the programming interface (i.e., the basic types and constructors) known from most imperative languages. This package consists of seven subpackages, about 80 specifications, 30 functors, and four module descriptions; all exported datatypes have been axiomatically specified yielding a number of default implementations and a large number of assertions (the assertion generator is net yet completed).

Based on the `Standard` package and on an `Extended` package for higher concepts such as reference-counted type constructors, we have started to build up a library for polynomial arithmetic that is intended to serve as a first major test bed and demonstration application for the system. We hope to show that the environment will be suitable for engineering highly reusable computer algebra libraries while preserving the utmost runtime efficiency.

References

1. D. Berry. Lessons from the Design of a Standard ML Library. *Journal of Functional Programming*, 3(4):527–552, October 1993.
2. CA Group. The Scratchpad II Computer Algebra Sytem Interactive Environment Users Guide. Draft 1.2, TJ Watson Research Center, Yorktown Heights, NY, 1988.
3. O. Caprotti, H. Hong, et al. On C++ Polymorphism for the STURM Library. Internal Note 94-001, RISC-Linz, 1994.
4. W. Danielczyk-Landerl. Syntax Processing and Code Generation for a Generic Programming Language. Master's thesis, RISC-Linz, 1998. To appear.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
6. H. Gast, S. Schupp, and R. Loos. Completing the Compilation of SuchThat v0.7. Technical Report 97-12, Rensselaer Polytechnic Institute, December 1997.
7. H. Hong, A. Neubacher, et al. The STURM Library Manual — A C++ Library for Symbolic Computation. Technical Report 94-30, RISC-Linz, April 1994.
8. R. D. Jenks. *Axiom — The Scientific Computation System.* Springer, Berlin, 1992.
9. X. Leroy. Applicative functors and fully transparent higher-order modules. In *POPL '95*, pages 142–153, San Francisco, CA, January 22–25, 1995. ACM Press.
10. D. R. Musser and A. Saini. *STL Tutorial & Reference Guide.* Addison-Wesley, Reading, MA, 1996.
11. W. Schreiner et al. HPGP User and Reference Manual. Technical report, RISC-Linz, 1998. http://www.risc.uni-linz.ac.at/projects/basic/hpgp, to appear.
12. W. Windsteiger and B. Buchberger. GRÖBNER: A Library for Computing Gröbner Bases based on SACLIB. Technical Report 93-72, RISC-Linz, 1993.

Author Index