# TECHNIQUES IN ACTIVE AND GENERIC SOFTWARE LIBRARIES

A Dissertation

by

JACOB NYFFELER SMITH

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2010

Major Subject: Computer Science

# TECHNIQUES IN ACTIVE AND GENERIC SOFTWARE LIBRARIES

A Dissertation

by

JACOB NYFFELER SMITH

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

| | |
|---|---|
| Chair of Committee, | Jaakko Järvi |
| Committee Members, | Gabriel Dos Reis |
| | Thomas Ioerger |
| | Paul Gratz |
| Head of Department, | Valerie Taylor |

May 2010

Major Subject: Computer Science

# ABSTRACT

Techniques in Active and Generic Software Libraries. (May 2010)
Jacob Nyffeler Smith, B.S., The University of Texas at Austin;
B.A., The University of Texas at Austin
Chair of Advisory Committee: Dr. Jaako Järvi

Reusing code from software libraries can reduce the time and effort to construct software systems and also enable the development of larger systems. However, the benefits that come from the use of software libraries may not be realized due to limitations in the way that traditional software libraries are constructed. Libraries come equipped with application programming interfaces (API) that help enforce the correct use of the abstractions in those libraries. Writing new components and adapting existing ones to conform to library APIs may require substantial amounts of "glue" code that potentially affects software's efficiency, robustness, and ease-of-maintenance. If, as a result, the idea of reusing functionality from a software library is rejected, no benefits of reuse will be realized.

This dissertation explores and develops techniques that support the construction of software libraries with abstraction layers that do not impede efficiency. In many situations, glue code can be expected to have very low (or zero) performance overhead. In particular, we describe advances in the design and development of active libraries — software libraries that take an active role in the compilation of the user's code. Common to the presented techniques is that they may "break" a library API (in a controlled manner) to adapt the functionality of the library for a particular use case.

The concrete contributions of this dissertation are: a library API that supports iterator selection in the Standard Template Library, allowing generic algorithms to find the most suitable traversal through a container, allowing (in one case) a 30-fold

improvement in performance; the development of techniques, idioms, and best practices for `concept`s and `concept_map`s in C++, allowing the construction of algorithms for one domain entirely in terms of formalisms from a second domain; the construction of generic algorithms for algorithmic differentiation, implemented as an active library in Spad, language of the Open Axiom computer algebra system, allowing algorithmic differentiation to be applied to the appropriate mathematical object and not just concrete data-types; and the description of a static analysis framework to describe the generic programming notion of local specialization within Spad, allowing more sophisticated (value-based) control over algorithm selection and specialization in categories and domains.

We will find that active libraries simultaneously increase the expressivity of the underlying language and the performance of software using those libraries.

To Stephanie Hope & Kinsey Dorothea Danger Smith

# ACKNOWLEDGMENTS

I never had any intention of learning to program; in fact, I was threatened with harm about ten years ago when making fun of programming and programmers; so, I suppose, it is with a certain amount of auto-schadenfreude that I find my self at the end of a multiyear journey into computer science — especially software engineering and the development of software libraries.

I would like to thank my adviser Dr. Jaakko Järvi first and foremost. Little does he know that it was reading his papers (and similar papers from the OSL) and poring over the design of his Boost libraries that finally prompted me to apply for school. I want to thank Mat Marcus for sitting in a 512 $ft^3$ cell with Dr. Järvi and poring over my algorithms in order to make a large portion of this dissertation possible. Of course, working with Dr. Järvi would never have been possible without Dr. Thomas Ioerger's patience with myself (and Erik, my co-conspirator in generic programming); I will forever be grateful to Tom for suggesting that—just possibly—computational biology was not the field for *me*.

Dr. Gabriel Does Reis has, effectively, been the co-chair for this dissertation. I know I try his patience at almost all times, but he has been an invaluable teacher: each semester spent in his classes is, I feel, equivalent to whole Bachelor's and Master's degrees. I would also like to thank Dr. Paul Gratz for jumping on board this dissertation at such a late date and being so enthusiastic about its topic, regardless of the extremely short times I gave him to review and revise!

I would like to thank two different sources for support: Dr. James F. Leary (and the whole team at the old UTMB MCU) for his long-time support and encouragement and for telling me that if statistical software doesn't work, don't complain, learn to program and fix it! And, also, To Dr. Bart Childs for convincing me to come to

A&M. The second source are the members of Jaakko's lab (John Freeman, Xiaolong Tang), even though I don't think any one of us managed to actually work on a project together. Hand-in-hand I would like to thank everyone in Dr. Bjarne Stroustrup's lab (including Bjarne for being such a good sport): Luke Wagner for helping me to understand logic and language theory; Yuriy Solodkyy for our shouting matches and appreciation of books & coffee at the library; Dr. Damian Dechev and Peter Pirkelbauer for being too cool to be in such a geeky field. I owe a large debt to Yue Li not only for shepharding this dissertation to a close, but also for being a good friend my last year-or-so at TAMU.

Finally, I want to thank and send my love to my family for their infinite patience; to my wife, Stephanie, my daughter, Kinsey; to my brother for low-rents; and lastly, to my parents, who never once questioned why, after finishing my first degree in a little less than three years, I stayed in school for another ten.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

# LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

# CHAPTER I

## INTRODUCTION

More than forty years ago Doug McIlroy proposed that families of reusable software codes were necessary for developing large scale software [72]. He advocated a method where a family of software codes would be available for programmers such that rather than coding an algorithm by hand, a programmer could acquire the algorithm's code from a library of prewritten codes. This "software library" would provide not only the implementations of different algorithms, but also implementations of alternate versions of each algorithm. An implementation of an algorithm would be specialized for some property such as speed, efficiency, numerical robustness, or ease-of-use. After identifying a set of algorithms and data-structures necessary for an application, a programmer composes the codes found in a software library into a program.

McIlroy's vision for software libraries is widely embraced. All mainstream, general-purpose languages such as Java, C++, and C#, have a large number of software codes available in many software libraries, such as Java's Platform Library [102], C++'s STL [100], or C#'s *System.Forms* [26]. Software libraries provide a range of algorithms and data-structures, some as simple as finding the lesser of two values, some as complex as generating 3-D virtual worlds. There now exist many such software libraries, for almost any given task.

Software libraries provide two important benefits to programmers. First, a programmer does not need as much time or effort to code an application, because codes in the software library can be reused rather than redeveloped. Second, a programmer can write increasingly larger and more sophisticated applications, because software li-

---

The journal model is Science of Computer Programming.

braries provide abstractions which allow the programmer to reason about and express programs in terms of "higher level" concepts, rather than being occupied by "lower level" concerns such as details of implementation of particular algorithms. These two benefits of software libraries are essential to the development of modern software.

Even given the widespread use of software libraries and the advantages they provide, the software industry continues to struggle with low quality, high defect rates, and low productivity. The "software crisis" [72] identified forty years ago continues to hold its grip. Frustratingly, one aspect of this crisis is due to the use of software libraries, themselves. Using a software library requires a programmer to write "glue" code to adapt the software library's code to other codes in the application. The complexity and size of the glue code can become substantial as the size of the software increases. Furthermore, glue code often has a (hidden) computational cost: it is possible for glue code to have a deleterious effect on the performance of the whole program [75]. Similarly, the abstractions provided by a software library may hinder performance, caused by, e.g., redundant checks of data validity and inefficient "general case" implementations.

The size and complexity of the glue code, the negative impact to the performance of a program because of the glue code, or from the use of library abstractions, may force a programmer to write custom codes to re-implement the algorithms and data-structures provided by a software library. Rewriting requires both time and expertise on the part of the programmer, and increases the probability of defects in the program. Furthermore, if rewriting requires development of new codes "in house" rather than relying upon a library vendor, then there is a corresponding increase of maintenance costs to cover the new codes.

This dissertation will explore techniques that support the construction of software libraries with abstraction layers that do not impede efficiency, and where the

necessary glue code has low performance overhead. In particular, this dissertation will describe advances in the design and development of *generic* libraries, software libraries targeted to solve a class of similar problems in terms of *active libraries*, software libraries that take an active role in the compilation of the user's code [109]. All of the methods described in this dissertation are motivated by efforts to construct libraries to solve practical research and development problems. For each presented technique, we describe its application to at least one problem domain and a software library associated with that domain. The presentation of each technique will also discuss the difficulties experienced in the use of "traditional" software library designs to support the associated problem domains.

The next section will expand upon the background of software libraries and introduce the new techniques for the design of libraries this dissertation presents.

## A. Background

Software libraries are tools; we view the design of software libraries as an engineering effort separate from the initial development of the algorithms and data-structures which make up a software library. Determining the best way to implement, collect, and compose algorithms and data-structures into a coherent software library depends upon the goals of that library. For instance, a software library may be implemented for clarity of code, for efficiency of operations, for numerical robustness, or for coverage of a specific problem domain.

The traditional view of a software library is as a collection of functions and data-structures encapsulating codes [117]. The role of a traditionally designed library is passive; that is, the library is essentially an opaque object with an interface through which the user can gain access to functionality of the library. Such an abstraction

boundary provides a host of benefits: modularity and increased programmer productivity, simplified reasoning about program behavior, separate compilation, etc. A traditionally designed software library respects the abstraction boundary its interface defines: the library provides a number of entry points that the user can call, but does not include codes and data-structures which can observe the manner in which a user calls the entry points. A traditionally designed library is thus not designed to take advantage of the varying capabilities of client code of the library.

Some modern software libraries, such as Standard Template Library (STL) [93], Matrix Template Library 2 (MTL2) [90], and Blitz ++ [108] diverge from the architecture of traditionally designed software libraries: they observe, act upon, and interact with the user's code. They are *active libraries* [109, 117] because they participate in the interpretation, optimization, and transformation of the user's code during compilation [110]. In some cases, the user of an active library can be more accurately described as the client of the library, as the library transforms both the meaning and structure of the user's code.

This dissertation will focus on three aspects of active libraries:

1. *Permeable interfaces* which allow a library to observe the capabilities of the client's code so that, for example, the library can select alternate versions of an algorithm based on the capabilities of the user's code.

2. *Non-intrusive adaptation and composition* mechanisms allow components developed for a particular domain to be easily used with algorithms from different domains, with minimal performance impact.

3. *Library-directed transformation* of client code allows the user to write code in the most direct (obvious) way, while the library interacts with the compiler to provide extra checking or transformation of the user's code.

To clarify the use of permeable interfaces, non-intrusive adaptation and composition, and library-directed transformation in active libraries, we will discuss each these three aspects of active libraries in the next few sections. First, to give the feel of active libraries we describe a concrete example of the use of a particular library. MTL2 provides a number of classes representing various kinds of matrices such as symmetric, diagonal, and upper-triangular. MTL2 also provides several algorithms which implement matrix operations, such as addition and multiplication.

```
matrix<float, diagonal<> >::type M, N, P;
// initialize N, P
M = N * P;
```

Fig. 1. Example code fragment for the active library MTL2.

The code in Figure 1 shows the use of MT2 for the multiplication of a dense diagonal matrix `N` by the dense diagonal matrix `P`, where the result of the expression is assigned to the dense diagonal matrix `M`. The usual algorithmic complexity for matrix-matrix multiplication is cubic in the number of rows (for square matrices). Because MTL2 can observe that the matrices are diagonal, it can generate better code. The resulting executable code has *linear* algorithmic complexity in the number of rows [90]. Even further, the successor to MTL2, MTL4 [36, 35], can be configured to resolve the expression to a call to the Level 3 BLAS operation DGEMM [64][1].

## 1. Permeable interfaces

A library component that observes the capabilities of its inputs, i.e., the properties of the types of the inputs, has a *permeable interface*. There are a number of mechanisms

[1]This feature was added to improve performance; however, it was superseded by the fact that the MTL4 is second only to the Goto library [34], and vastly outperforms ATLAS, Lapack and other Fortran and C-based libraries [35].

implementing permeable interfaces, including, but not limited to, *traits classes* and *tagging* [77, 5], and techniques like *expression templates* [108, 106]. Permeable interfaces let a library designer reduce the number of explicit annotations required in the use of a library component. Below, we show code which compares use of MTL2, on the left, to functionally equivalent code using Lapack, on the right. Assuming that `A`, `B`, and `C` are all matrices, the codes effect a multiplication and assignment operation between the three matrices (source: [82]):

```
matrix C = A * B;
```

```
cblas_dgemm(ML_BLAS_STORAGE,
  CBlasNoTrans, CBlasNoTrans,
  A.rows(), B.cols(), A.rows(),
  1.0, A.raw_data(), A.data_stride(),
  B.raw_data(), B.data_stride(),
  0.0, C.raw_data(), C.data_stride());
```

The algorithmic complexity and run-time performance of the above codes are roughly the same [106, 35, 90]. Both codes use their inputs to select between alternate implementations of the multiplication-assignment algorithm. However, the MTL2 component observes the information needed for selecting a particular implementation of the algorithm from the types of its inputs — whereas the Lapack component requires that the client specify the information necessary for selecting a particular implementation of an algorithm by passing in explicit arguments. We observe that the MTL2 code, on the left, has a distinct advantage over the Lapack code, on the right, in terms of notational clarity. As a result, we can expect the code to be easier to maintain, and simpler to write.

## 2. Composition and adaptation

*Library composition* is the use of data-structures and functions from two or more libraries for a common task. For instance, the dense matrix data-structures defined by the MTL4 can be passed as arguments to Lapack algorithms which expect dense matrix data-structures defined by Lapack. The reuse of Lapack algorithms for MTL4 data-structures results in significant savings in development and debugging time. Ideally, such reuse has no cost either in terms of performance or maintainability to the user.

The prerequisite of library composition is that the data-structures and algorithms must be compatible. However, when the components of libraries are incompatible an *adaptation* can be used to make the components compatible. An ideal adaptation mechanism should be non-intrusive, i.e., the mechanism should not require the modification of the component being adapted. Non-intrusiveness is crucial since the source code may not be available, or changes to the component will cause other parts of the program to fail. In addition, adaptation mechanisms should not incur unnecessary performance costs. Such costs come from two sources: (1) the cost of the adaptation mechanism, itself; and (2) the costs of converting data-structures and adding missing functionality necessary to make the components compatible. The first cost can be influenced by programming language technologies and library design, while the latter cost is inherent to the adaptation.

Adaptation mechanisms which have a non-zero cost can lead to an overall decrease in the performance of a program. One possible reason for the decrease in performance is deeply stacked layers of adaptations. This is because each layer, by itself, has a small cost but the sum of the costs of the layers is large [75]. An example is the translation of names (identifiers) of functions. Each time a function is trans-

lated, for example, by calling another function, there is a cost in manipulating the call stack, which can be quite large compared to the cost of function, itself.

Some adaptations have an inherent cost because the adaptation must include additional functionality or data-structures to make components compatible. These adaptations can still provide important benefits both in terms of code re-use and total performance gains, if appropriately used. An example is the adaptation between the Parallel Boost Graph Library (PBGL) [41] and the Iterative Eigensolver Template Library (IETL) [104]. As shown by Breuer et al. in [11], data-structures from the domain of iterative eigensolvers can be adapted to be compatible to data-structures from the domain of parallel graphs. This means that the effort of writing parallel algorithms and data-structures for the domain of graphs (PBGL) can be re-used for the domain of iterative eigensolvers (IETL). This adaptation results in significant performance improvements for the IETL due to the increase in parallelism.

Without composition, the benefits of software libraries are diminished. Non-intrusive and low cost adaptation mechanisms increase the scope of possible compositions by removing the mismatch between semantically equivalent but syntactically differing data-structures and functions.

## 3. Library-directed transformation

We define *library-directed transformation* as the manipulation of data-structures representing the parse-trees of expressions [110]. Library-directed transformations consist of two parts: first, there must exist mechanisms to acquire representations of expressions as data-structures; and second, those data-structures should be manipulated for the purpose of improving run-time performance, providing stronger guarantees against memory leaks, etc.

An example of a mechanism to acquire representations of expressions as data-

structures is "expression templates" [35, 36, 77, 107]. Expression templates are implemented using a combination of parametrized data-structures and operator overloading. For instance, the following function could be part of an expression template library to build a data-structure representing multiplication.

```
template <typename E1, typename E2> // some types E1 and E2
mul_op<E1, E2>
operator * (E1 const& e1, E2 const& e2) { // any multiplication
  return mul_op<E1, E2>(e1, e2); // mul_op represents multiplication
}
```

Instead of immediately computing the product of the values "`e1`" and "`e2`", the operator "`*`" returns the data-structure "`mul_op`" which represents the multiplication of the arguments. Similar data-structures and functions can be written for addition, subtraction, division, general unary and binary operations, etc.

The manipulation of data-structures representing expressions can improve the run-time performance of a program. For example, an expression structure "`B*C+B*D`" for large, dense matrices "`B`", "`C`", and "`D`" can be transformed into the more run-time efficient expression "`B*(C+D)`", assuming distributivity of the multiplication over addition. One library that uses such transformations is MTL4 which uses loop unrolling, tiling, specialized multiplication operations, etc., to produce codes whose run-time performance is at least as fast, and sometimes much faster than equivalent FORTRAN77 codes [35].

Combining domain specific notations and library-directed transformations allows an active library to substitute for domain expertise [110, 107]. For instance, MTL4 uses the operators "`*`" and "`+`" for the multiplication and addition of any two kinds of matrices, i.e., dense, sparse, upper-triangular, etc. Expressions using those, and other, operators are optimized for run-time performance by MTL4. When the user writes

an expression such as "`C = A * B`" MTL4 produces efficient code for the expression dependent upon the types of the elements of the matrices, the storage specification of the matrices, if the matrices are upper- or lower- triangular, etc., where the user does not need to know about these optimizations.

## B. Dissertation statement

In this dissertation we view a software library as more than just a package of algorithms and related data-structures: we view the software library as a program which can benefit from abstraction. With this view in mind we can ask what data-structures and algorithms we can describe to support the development of software libraries. For instance, what algorithms and data-structures allows us to write permeable interfaces? More importantly, are these data-structures and algorithms general to any software library? Similarly, what features allow us to easily write — and easily maintain — the composition of library components? Can we generalize the notion of composition and write down best-case practices for any software library? Is it possible to write software libraries that participate in some of the roles that are classically only done by the compiler, i.e., compile-time transformations and optimizations? Are such libraries (or the notions of such libraries) portable across compilers?

In the preceding sections we have described a number of capabilities of active libraries. The techniques for implementing active libraries are relatively well established — STL, perhaps the first significant active library, is more than 15 years old. However, this dissertation argues that the tools available to active library writers — the algorithms and data-structures for writing software libraries — are still limited. Therefore, new techniques, clarification and augmentation of old techniques, and new language features are needed to aid the implementation of active libraries. The central

theme of all of the chapters is the enumeration of recipes for constructing software which observes the capabilities of the inputs to the libraries' components, and then performs transformations to the user's code and the library's implementations of algorithms based on those capabilities. The remaining chapters of this dissertation will then present advances in each of the three capabilities of active libraries for generic programming above discussed:

- Chapter II will describe a framework which generalizes the selection of associated datatypes from components. In particular, the framework is implemented to retrieve different categories of iterators from collections.

- Chapter III develops techniques, idioms, and best practices the use `concept`s and `concept_map`s, new language features in the up-coming revision of C++ called C++`0x`, for the construction of low- and zero- cost adaptations.

- Chapter IV describes an active library for computing the algorithmic differentiation of client code. This framework demonstrates the use of library-directed transformations in the Spad language [58].

- Chapter V formalizes the generic programming notion of (local) specialization within the language Spad. Specialization is an important language feature used in the construction of active libraries.

Each chapter will individually cover related work and background material; the dissertation will conclude in Chapter VI.

# CHAPTER II

# A PARAMETRIZED ITERATOR REQUEST FRAMEWORK FOR GENERIC LIBRARIES

The iterator abstraction is central to many generic libraries, such as the C++ Standard Template Library (STL). Generic algorithms are commonly specialized with regard to the kinds of iterators available. There is, however, no mechanism for selecting the kind of iterator that a container should provide for a particular algorithm. We propose a framework where an algorithm can query for the categories of iterators a container supports, and select the most appropriate iterator. This is a new axis of parametrization for STL-like generic libraries. The motivation for the framework comes from our work with the CCTBX and TEXTAL Protein Crystallography (PX) libraries. The models of the data in this domain provide multiple, complex iteration schemes, and the efficiency of many algorithms depends crucially on selecting a suitable scheme. The framework allows individual algorithms to access the preferred iteration scheme over the container it uses. We describe the framework with examples in the context of the STL and the PX libraries.

## A. Introduction

Generic programming emphasizes *algorithm specialization*, which essentially means providing many implementations for the same functionality. A specialization of a generic algorithm places more requirements on its inputs and can make more assumptions on them, possibly enabling a more efficient implementation. A simple example is finding an element in an unsorted versus in a sorted sequence: the former requires linear run-time with respect to the length of the sequence, the latter only logarithmic, since the assumption of sortedness allows an implementation to use a

binary search strategy.

In generic libraries, such as C++'s Standard Template Library [97] (STL), algorithm specialization is used for many algorithms whose inputs are types conforming to the STL's iterator concepts. For example, the `distance` function—which measures the distance between two iterators—is defined for all types that meet the requirements of the `InputIterator` *concept*. The least specialized version of `distance` is implemented by counting the number of times the first iterator is incremented to reach the second iterator. A specialization of this distance-computing algorithm—operating in constant time—is provided for `RandomAccessIterator`s. Due to the random access capability of the iterators, the implementation of this specialization is a simple subtraction. Typically, algorithm specialization takes place automatically: the generic library selects the best available specialization for the types of the inputs to the algorithm.

STL algorithms operate on sequences described as pairs of iterators. The source of a sequence is often a container which provides mechanisms (such as the `begin` and `end` functions) to present the contents of the container as a sequence. When using these mechanisms the container provides *the most powerful iterator types it can offer in order to enable the most efficient algorithms specializations*. This behavior occurs because of the expectation on the part of the writer of active libraries that the library will be composed by the end-user. For example, `v.begin()` for a `v` with type `std::vector` gives a `RandomAccessIterator`, whereas `l.begin()` for an `l` with type `std::list` is only capable of providing a `BidirectionalIterator`. The result is that the two lines below eventually invoke different implementations of the `distance` function. The first implementation is a constant time operation with respect to the distance between the iterators, and the second is linear:

```
distance(v.begin(), v.end());
distance(l.begin(), l.end());
```

Algorithm specialization along the hierarchy of iterator concepts is not the only opportunity for specialization. In particular, in a design where algorithms also operate on containers and not solely on iterators, the selection of the iteration scheme for a particular container can be subjected to specialization, and can result in performance gains. Note that in the active libraries built for many domains the norm is to pass data to generic algorithms as containers, not iterators; graphs [89] and matrices [90] serve as examples of such data. In Section 4 we describe data structures in the domain of *protein crystallography* where this is true as well.

When requesting a sequence from an STL container (with the `begin` and `end` member functions), the container typically provides iterators that conform to the most refined iterator concept possible, in terms of algorithmic performance. In STL containers there is a strong correspondence between the most-capable iterator a container provides and the most-refined iterator a container provides. Not all containers have such a strong correspondence between most-capable and most-refined: it might be more efficient to provide a *less capable* iterator scheme, because the possibility exists for the container to implement the less capable iterator in a more efficient way. For example, consider a generic image type that represents a two-dimensional raster image with an arbitrary number of channels, parametrized over the value type of the channels. Examples of concrete instances of such an image type include a one-bit black and white mask, an RGB or CMYK bitmap, or images with a larger number of color channels. Such an image type can be implemented as an array whose size is the product of the width, height, and the number of channels. Assume we lay out the image data in this array as a list of raster lines, where a raster line is a list of pixels,

and where a pixel is a list of values from each channel. Consider visiting each channel value and performing some independent operation on it. If the only iteration scheme provided by the image type directly models the hierarchy "raster line–pixel–channel value", a function visiting each channel value requires three nested loops:

```
raster_line_iterator rtr = image.begin();
raster_line_iterator rnd = image.end();
for ( ; rtr!=rnd; ++rtr ) {
  pixel_iterator ptr = rtr→begin();
  pixel_iterator pnd = rtr→end();
  for ( ; ptr!=pnd; ++ptr ) {
    channel_iterator ctr = ptr→begin();
    channel_iterator cnd = ptr→end();
    for ( ; cnd!=ctr; ++ctr ) some_operation (∗ctr);
  }
}
```

This hierarchy may not, however, be necessary for an operation performed with the channel values. In such a case, the iteration scheme will perform a non-trivial amount of unnecessary work, compared to certain to the computational cost of the function object `some_operation()`. This is because of the mismatch between the most refined iterator for images — which provide the most convenient iteration scheme for programmers, and the most-capable iterators, which provide the most efficient access to the underlying data. A more direct and efficient mechanism for visiting all channel values is to iterate over the underlying contiguous memory directly, ignoring the hierarchical structure:

```
channel_iterator ctr = begin<channel>(image);
channel_iterator end = end<channel>(image);

for ( ; cnd!=ctr; ++ctr )
    some_operation (∗ctr);
```

The `begin<channel>` and `end<channel>` functions are requests for a non-hierarchical iteration scheme (see Section C). This iteration scheme is considerably faster (we report timing results in Section 1).

Requesting a simpler iterator scheme for efficiency is the "dual" of algorithm specialization over iterator schemes. If equivalent algorithmic functionality can be provided with the same complexity guarantees with a simpler iterator, then it is preferable to use the simpler iterator as it will have improved performance. For example, the exact same code (the STL's `find` algorithm) has drastically different performance characteristics depending upon which iteration scheme is used. Experimental results can be found in Sections 2 and 3.

In this chapter, we propose a lightweight framework for algorithms to request a particular iterator scheme from a container or sequence source. The library consists of a small number of functions forming the API for the client of the library, and requires the algorithm and container implementations to follow a small set of conventions — a relatively light burden for library developers. The API is designed in such a way that it can be non-intrusively and retroactively added to a container library. The addition of this API to a container library helps to normalize the interface for the request of iterators for all container libraries which use the API.

The core of the framework is a set of tag `struct`s which we call *iterator tags*. The global functions `begin` and `end` parametrized with a tag and a container give access to the iterator schemes that a container provides. In essence, the proposed framework suggest a new degree of parametrization to STL-like generic libraries. We suspect that it is possible to identify a set of iterator tags that could be established similar to the iterator concepts in the STL. We do not suggest such a set in this dissertation, but describe a handful of useful iterator schemes. We identify situations where parametrizing the iteration scheme provides notable benefits. In describing the

framework, we assume some familiarity with *template metaprogramming*, as described, e.g., in [1].

The API we suggest allows the use of "the right iterator in the right place" [65] by providing a uniform method for requesting iterators from containers. The API allows the selection of the iterator category to be parameter of a generic function, allowing the library writer to defer decisions about the category of an iterator to the user. This is an especially important capability since, in many cases, making such a choice at the time of writing a function can create unsatisfactory trade-offs in the implementation of the function.

## B. Background and motivation

In the STL [100], access to the iterator of a container is provided through the member functions `begin` and `end`. Some containers support iteration backwards with the members `rbegin` and `rend`. Additionally, STL containers overload these functions for the case where the container is a `const` object. Any single container in the STL can thus provide up to four different iterator *types* (but essentially only two iteration *schemes*). For any particular STL container, these schemes are always the most capable iterators that the container can offer, for example, `std::vector` provides `RandomAccessIterator`s, and `std::list` `BidirectionalIterator`s [100]. The STL containers are thus closely tied to the iterator scheme they implement and to the iterator concept the return types of their `begin` and `end` functions provide. Even if a container could provide multiple iteration schemes over its data, the STL defines no generic interface for accessing them. An important side-effect is that the STL has become the *de facto* standard for defining the interfaces of containers and generic algorithms in active libraries such as the BGL, MTL (2, 4), etc.

In order to take advantage of alternative iterator schemes, we need a mechanism to access such schemes. A straightforward mechanism for doing this would be to provide a specific function name for each iteration scheme, as is already done with `rbegin` and `rend`. However, dedicating a specific function name for each iteration scheme does not work well with generic programming: function names become hard-wired in the implementations of generic algorithms. When writing a generic algorithm, the iteration scheme is not necessarily known. For example, the following code shows a function which is parametrized by the container type; the algorithm requests the iterators from the container and performs a search.

```
template <typename Ctr>
void fun (Ctr& ctr, ...) {
  typename iterator_traits<Ctr>::iterator f = ctr.begin(), l = ctr.end();
  // ...
  search(f, l, v);
  // ...
}
```

The particular iterator selected from the container `ctr` cannot change. For example, the end-user may want the container to be searched in reverse-order. This can only be accomplished in an indirect fashion: (1) rewrite the function; (2) reverse the container; or (3) build some proxy to forward calls to `begin` and `end` to `rbegin` and `rend`.

Using distinct member functions for each iterator scheme also goes against the principle of specialization. Algorithm specialization automatically selects the best available implementation for an algorithm, but gracefully degrades to a slower version if the requirements of the faster ones are not met. Similarly, we wish to allow a request for a particular iterator scheme, but settle for a less specialized one, if the exact requested one is not supported by a particular container.

The STL implements algorithm selection using *tag dispatching*: a fixed set of

tag `structs` which each correspond to a particular iterator concept. The tag of any iterator type can be accessed via the `iterator_traits` machinery. The expression `iterator_traits<Iter>::iterator_category` is guaranteed to denote the tag of the type `Iter`, if `Iter` conforms to one of the iterator concepts [100]. Similar to iterator categories, we use tags to refer to different iterator schemes. Our iterator request framework defines the global functions `begin` and `end` that take a container type as their function parameter, and additionally a type argument specifying the requested iterator tag. In this way the iterator tag is not a fixed part of the signature. The iterator tag can be, for example, a type parameter at the call site to the `begin` and `end` functions—a generic algorithm can itself be parametrized over the tag, allowing the caller of the algorithm to specify the tag requested in the interior of the algorithm. This allows clients of the generic algorithm to "reach through" the algorithm to specify functionality.

### C. Iterator request framework

The iterator request framework consists of a family of `begin` and `end` functions, a metafunction that computes the type of the iterators returned by the `begin` and `end` functions, and some helper functions and metafunctions. We first describe the metafunction `iterator`, shown in Figure 2, that specifies the type of the iterators for a given iterator tag–container pair. By default, a metafunction called `iterator<Tag, Container>::type` resolves to the member type `iterator` in the type `Container`. The default is thus to access the `iterator` member type in the STL containers. To make the iterator types of other iteration schemes accessible, a generic library must specialize the `iterator` template for the relevant iterator tag–container type pairs.

```
template < typename Tag, typename Container >
struct iterator {
        typedef typename Container::iterator type;
};
template < typename Arg >
struct const_ {
        typedef typename Arg::const_iterator type;
};
template < typename Tag, typename Container >
struct const_< iterator<Tag,Container> > {
        typedef typename Container::const_iterator type;
};
```

Fig. 2. The `iterator` and `const_` metafunctions for the framework.

The second metafunction, `const_`, is for convenience; it provides access to the type of the iterators implementing the constant version of the requested iteration scheme. The default is, analogously, the `const_iterator` member type of the container parameter. We provide `const_` to match the C++'s `const` qualifier.

In addition to the above metafunctions, the interface to the library includes the functions `begin`, `end`, and `const_begin`, and `const_end`. The first two functions provide both constant and non-constant access to the iterators. The latter two functions are included to aid in situations when a constant iterator scheme is needed, but where the current context is non-constant. The addition of the `const_*` form of the interface functions does not violate our principle of providing generic names for access to iterators: the `const` qualifier is a built-in language feature separate from the usual algorithmic or data-structure concerns, such as reverse, or random-access. All four functions are parametrized over both an iterator tag and a container type. Their return types are computed with the `iterator` metafunction discussed above, and shown in Figure 2.

Figure 3 shows all versions of the `begin` interface functions; the implementations

of the `end` functions are analogous. All the interface functions merely forward the calls to appropriate `tagged_begin` or `tagged_end` functions, implementing the three different versions of `begin` and `end` with only two "back-end" functions. This means less work for the container implementer.

```
template < typename Tag, typename Container >
typename iterator<Tag,Container>::type
begin ( Container& ctr ) {
        return tagged_begin(ctr,Tag());
}
template < typename Tag, typename Container >
typename const_<iterator<Tag,Container> >::type
begin ( Container const& ctr ) {
        return tagged_const_begin(ctr,Tag());
}
template < typename Tag, typename Container >
typename const_<iterator<Tag,Container> >::type
const_begin ( Container const& ctr ) {
        return tagged_const_begin(ctr,Tag());
}
```

Fig. 3. The set of `begin` functions for the framework.

The `tagged_begin` and `tagged_end` functions are shown in Figure 4. The default versions of these functions forward to the container's member functions `begin` and `end`, using the current STL convention. The second function is the same as the first, except that the computed return type is constant. It is up to the container or algorithm implementer to overload these functions to return the desired iterator for a particular iterator tag.

In sum, to add a new iterator scheme, one metafunction must be extended with a new class template specialization (`iterator`) and with four function template overloads (the const and non-const versions of the `tagged_begin` and `tagged_end`). Codes showing the implementations of the specializations and overloads are shown in the

```
template < typename Tag, typename Container >
typename iterator<Tag,Container>::type
tagged_begin ( Container& ctr, Tag ) {
        return ctr.begin();
}
template < typename Tag, typename Container >
typename const_<iterator<Tag,Container> >::type
tagged_const_begin ( Container const& ctr, Tag ) {
        return ctr.begin();
}
```

Fig. 4. The set of `tagged_begin` functions for the framework.

next section.

## D. Examples

In this section we demonstrate the use and benefits of the iterator request framework with three examples. The first one is the image example discussed in Section A, for which we present some run-time performance information; the second is in the context of the STL; and the third is our motivating example taken from the computational protein crystallography context.

### 1. Timings for images

In Section A we presented two alternative schemes of iterating over the pixels of an image. To demonstrate the importance of being able to select the most suitable iteration scheme for such image containers, we measured the performance difference of the two different iteration schemes, which we refer to as *hierarchical* and *linear*. Our implementation of the image was a wrapper around the `std::vector`. Access to the iterators are through the framework's `begin` and `end` functions, using either of the tags `hierarchical` or `linear`. Each of the hierarchical iterators stores a pointer

to its parent iterator and an integer to the offset from the parent's offset. That is, the raster iterator stores a pointer to the image and the raster-line it is representing. A pixel iterator stores a pointer to its parent raster-line iterator and an offset into that raster-line to the pixel. And the channel iterator stores a pointer to its parent pixel iterator and an offset into the pixel.

We assume that we have a data-structure representing an raster-image with an interface providing (at least) two iterator schemes. The first scheme is for access to hierarchical iterators through the functions `hierarchical_begin` and `hierarchical_end` and the second is for access to linear iterators through the functions `linear_begin` and `linear_end`. The implementation of the `iterator` metafunction and the functions for `tagged_begin` for hierarchical iterators are shown below.

```
struct hierarchical {};
template <typename Container>
struct iterator<hierarchical, Container> {
  typedef typename Container::hierarchical_iterator type;
};
template <typename Container>
struct const_< iterator<hierarchical, Container> > {
  typedef typename Container::const_hierarchical_iterator type;
};
template <typename Container>
typename iterator<hierarchical, Container>::type
tagged_begin ( Container& ctr, hierarchical) {
  return ctr.hierarchical_begin();
}
template <typename Container>
typename const_< iterator<hierarchical, Container> >::type
tagged_const_begin ( Container const& ctr, hierarchical) {
  return ctr.hierarchical_begin();
}
```

The first line of the code defines the iterator scheme tag for hierarchical iterators.

The next eight lines of code, lines 2–9, define the template specialization of `iterator` for any container that supports a hierarchical iterator, and access to that iterator via the member functions `hierarchical_begin()` and `hierarchical_end()`. The remainder of the code, lines 10–19, defines the functions `tagged_begin` and `tagged_const_begin` which, again, are defined for any container that satisfies the concept of a raster-image with member functions to gain access to the first and last hierarchical iterator. The definition of the functions `tagged_const_end` and `tagged_end` are similar.

| Size, Channels | 1 | 3 | 6 |
|---|---|---|---|
| 128×128 | 1.75 | 2.15909 | 2.24436 |
| 256×256 | 1.74011 | 2.15009 | 2.23694 |
| 512×512 | 1.73558 | 2.14746 | 2.24312 |
| 1024×1024 | 1.73084 | 2.15235 | 2.25039 |

Fig. 5. Iterator execution execution times. The ratios of the execution time of the hierarchical iteration scheme over the execution time of the linear iteration scheme, measured by timing the execution times of a function essentially equivalent to the STL's `fill`. The columns are the number of channels, and the rows are the number of pixels.

We measured the performance of iterating through images with varying width, height, and the number of channels using both the hierarchical and linear iteration schemes. Our test algorithm was a variation of the STL `fill` algorithm: we assigned the value "127" to each channel value in the image. The measured data are depicted in Figure 5, which shows the ratios of execution time of the hierarchical iterator scheme implementation to that of the linear iterator scheme implementation. The hierarchical iterator is generally about twice as fast as the linear iterator on a PowerBook5,6 G4 at 1.67GHz with 2GB of RAM.

### 2. Alternate run-time characteristics for `std::find`

In the above example with images, the selected iterator scheme affected the implementation of the algorithm. In this section, we show how changing the iterator scheme can affect the run-time performance, even run-time behavior, of the same piece of code. In particular, we take a linear deterministic algorithm and convert it into a Las Vegas algorithm. A Las Vegas algorithm is a randomized algorithm that terminates when some stopping condition is met or a certain number of iterations have occurred.

The input arguments of the STL `find` algorithm must be `InputIterator`s, at minimum. The `find` algorithm implements a straightforward sequential search. The run-time performance for `find` is dependent upon the distribution of the data in the sequence being iterated over, leading possibly to the worst-case behaviour being realized frequently. In such a case, a randomized algorithm, e.g., a Las Vegas algorithm, can possibly guarantee a better average complexity of iterator increments and dereferences.

With the iterator scheme selection framework we can parametrize a generic algorithm over the iteration scheme, allowing the client to choose the iteration scheme to be used in the `find` algorithm. In the following example, we associate the iterator tag `linear` with the sequential iterator scheme and the `las_vegas` with the Las Vegas iteration scheme:

```
template <typename Tag, typename Container>
void uses_find ( Container const& ctr,
  typename Container::value_type const& v ) {
  ...
  std::find(begin<Tag>(ctr), end<Tag>(ctr), v);
  ...
}
```

Note that the `uses_find` function does not need to change in order to change the

iterator scheme for `std::find`; the `Tag` type parameter tunnels through to `find`, as demonstrated by the following code fragment:

```
std::vector<int> a(1000,0), b(1000,1);
a.insert(a.end(), b.begin(), b.end());
a_function_that_calls_find<linear>(a, 1);
a_function_that_calls_find<las_vegas>(a, 1);
```

### 3. Alternate run-time characteristics for `std::find` II

In the above example we showed how to change the `find` algorithm from being deterministic to be being non-deterministic. In this section we show how to change the algorithmic complexity of `find` from linear to logarithmic with respect to the number of elements in the sequence. To do this we rely on the fact that sorted a random-access sequence of elements with a strict-weak ordering can be searched using a binary-search algorithm.

Our solution requires that a specialization of `find` exists which resolves to a call to `lower_bound`; such a version of `find` would need to be integrated into a library, e.g., the STL, and could be implemented like this:

```
template <typename Iter, typename Tag>
Iter find (Iter first, Iter last,
     typename Iter::value_type const& v, ra_sorted_iterator_tag) {
  return lower_bound(first, last, v);
}
```

Note that this implementation requires that the iterator category for sorted random-access iterators has a tag `ra_sorted_iterator_tag`. Because the iterator category tag is defined by a specialization of the STL's `iterator_traits` template metafunction, we must define a mechanism for retroactively 'changing' the `iterator_category` associated type of the container's iterator. This means we must have the following:

(1) a specialization of the iterator selection framework for sorted random-access iterators and containers; and (2) a way of retroactively changing the associated type `iterator_category` of an iterator.

We begin by defining a specialization of the `iterator` template metafunction for random-access iterators and a parametrized iterator scheme tag `ra_sorted`:

```
template <typename Tag> struct ra_sorted {};
struct ra_sorted_iterator_tag {};
template <typename Ctr, typename Tag>
struct iterator<ra_sorted<Tag>, Ctr> {
  typedef ra_sorted_iterator_proxy<
    typename iterator<Ctr::iterator, Tag>::type> type;
};
```

We do not show the `const` version of the iterator. The iterator scheme tag `ra_sorted` is instantiated with the iterator scheme tag of the underlying random-access iterator that will be extracted from the container. The template metafunction `iterator` is specialized over the parametrized tag `ra_sorted` and some container, and computes the type of the resultant iterator: essentially, this specialization of the template metafunction `iterator` wraps the container's random-access iterator into of a proxy object called `ra_sorted_iterator_proxy`. This proxy object implements a random-access iterator by owning a random-access iterator from the container, and forwarding calls made to the proxy object iterator to the container's random-access iterator. The iterator category of `ra_sorted_iterator_proxy` is `ra_sorted_iterator_tag`.

To finish this adaptation we must have appropriately defined versions of the functions `begin` and `end`. For instance, the back-end function `tagged_begin` could be implemented like this:

```
template <typename Tag, typename Container>
typename iterator<ra_sorted<Tag>, Container>::type
tagged_begin (Container& ctr, ra_sorted<Tag>) {
  return iterator<ra_sorted<Tag>, Container>::type(tagged_begin(ctr, Tag()));
}
```

The function `tagged_begin` takes a container and the instantiated `ra_sorted<Tag>` tag and returns the proxy iterator object. The function is implemented by constructing an proxy object with the result of calling the function `tagged_begin` recursively, where the tag is the parametrized tag `Tag`.

The following code fragment demonstrates the use of the `ra_sorted` parametrized iterator scheme:

```
std::vector<int> v;
... // v is sorted
a_function_that_calls_find<ra_sorted<linear> >(v, 1);
```

## 4. Protein crystallography

The impetus for the iterator request framework was from the design of generic algorithms for computational protein crystallography (PX). This section describes how without such a framework writing generic code leads to unacceptable trade-offs: the programmer must either depend on library-specific data structures, or accept an unreasonable loss of efficiency—a performance penalty of up to a factor of 30. We first briefly introduce the field of protein crystallography, followed by the discussion on implementing one of the key algorithms of PX libraries. We then demonstrate the use of our framework that avoids the above trade-off, achieving simultaneously generality and efficiency.

a. Background information on PX

In computational protein crystallography, one of the fundamental data structures is the *electron density* container. This is a container which represents the "presence" of an electron at a particular point in space—literally, the probability of an electron being at a given point in space. The data comes from bombarding a crystal of a protein with X-Rays [83, 48]. The electron density is used in algorithms and programs to help the crystallographer construct a model of the protein under investigation, to be used in drug discovery, determining novel structures, and so forth [83].

The libraries we primarily work with are the PX packages TEXTAL [49] and CCTBX [47]. TEXTAL is a tool chain that automatically builds protein models from electron density data [49]. CCTBX is a library of algorithms and other tools to aid in the development of PX software [47]. Within CCTBX and TEXTAL—as with other PX libraries not considered here [13, 18]—the electron density container has numerous different representations [83, 48, 47, 49], leading to numerous ways of iterating over the data.

In TEXTAL and CCTBX the iterators are also coordinates in 3D space, i.e., the type which represents an iterator with the normal semantics (increment, dereference), is also a type which implements the semantics of a 3-dimensional vector, or an offset, depending on the iterator. Access to the data in an electron density container is then provided either by dereferencing the iterator in the normal way, or by "passing" the iterator as a coordinate to the electron density container, usually by the `operator []`. The two concepts are mixed to provide programmers' the syntactic convenience of iterators and, on the other hand, coordinate access to the same data when that is more natural. There are a large number of coordinate systems—and their equivalent iterator schemes—for the electron density container which arise from the various

descriptions of the topology of the electron density data.

The large number of iteration schemes arises because there are four coordinate systems and three so called "symmetry" representations. The coordinate systems are known as the *linear-array*, *grid*, *fractional*, and *cartesian* system. Of these, the grid coordinate system—and its equivalent iteration scheme—is the default system for the CCTBX and TEXTAL libraries. The grid-coordinates essentially represent the data as a 3-dimensional array. The linear-array is a 1-dimensional array, and the underlying structure which holds the data. The fractional and cartesian coordinate systems are 3-dimensional systems which are used to conveniently represent the underlying topology, and "real" space, respectively. All of the coordinate systems and iteration schemes are necessary, and should thus be accessible to the client of these libraries [13, 18, 47, 49].

For each coordinate system there are three levels of increasing symmetry: *non-symmetric*, *translation-independent*, and *asymmetric*. The symmetry occurs due to the mechanism used to gather the PX data [83, 48]. Intuitively, the symmetries can be thought of a pattern used for tiling: non-symmetric means to not tile; translation-independent means to use use squares without any features (a checker-board without colors); and asymmetric means to use the smallest, non-repeating portion. Figure 6 depicts the symmetry classes with a simplified example. The four coordinate systems and the three symmetries amount to ten—we exclude the two higher symmetries for the linear-array coordinate system—different `RandomAccessIterators` possible for any electron density container.

Each iterator type is useful depending upon the algorithm in question. For example, since negative values are not well defined for electron density, some algorithms [49] set negative density values to zero; this can only be done with the linear-array or grid coordinate systems, because only those iterators provide mutability. Real-space refinement, a method to make the modeled protein fit better into the electron density

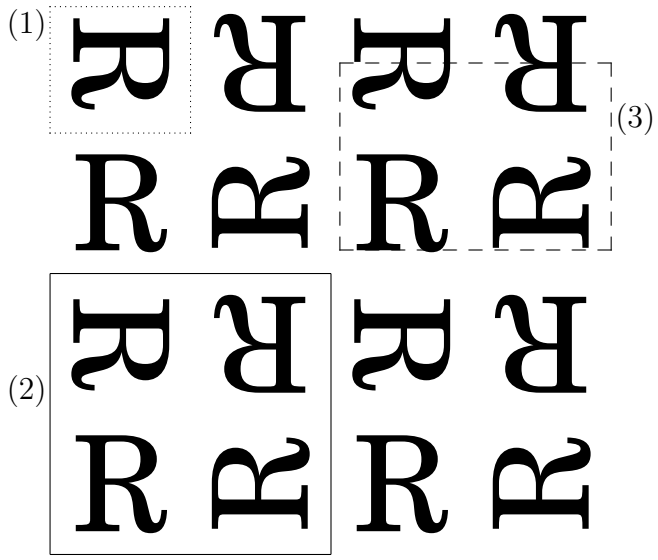



Fig. 6. The three types of symmetry. The image represents a finite subset of the infinite symmetric plane. Box 1 (dotted line $\cdots$) represents the asymmetric symmetry. Box 2 (using a solid line —) represents the translation-independent symmetry. Box 3 (using a dashed line $--$) represents a non-symmetric subset. In theory algorithms that operate on the asymmetric unit should be faster because they cover less data; in practice, the cost of discovering the proper symmetry operator to map the data back into the asymmetric unit more than offsets this.

data, is a critical technique in PX that relies heavily on the cartesian coordinates. This is because cartesian coordinates have an intuitive notion of distance and direction [33, 53]. Translation-independent and asymmetric symmetries, which are most naturally expressed in the fractional coordinate system, are useful in isolating a unique model [2].

b. Density interpolation algorithm

The density interpolation (DI) function is a critical algorithm used in PX. It computes a electron density value at an arbitrary coordinate in space based on the known stored electron density values surrounding that coordinate. The DI algorithm is invoked,

e.g., in the inner loop of the real-space refinement algorithm [33] that fits a model into the electron density data, and must therefore be efficient. This necessitates several different iterator schemes for accessing data in the electron density container. In particular, the non-symmetric symmetry can dramatically benefit from using the linear-array coordinate system.

The iteration selection framework gives access to many iterator schemes, which allows an implementation of an efficient DI algorithm in a generic fashion. The pseudo-code shown in Algorithm 1 outlines the computation of an interpolated electron density value at a given coordinate.

**input** : P *coordinate*; E *electron density*
**output**: V *linearly interpolated value*
grid-point ← convert-to-grid-coordinate-system (P);
grid-iters [8] ← get-iterators-surrounding (grid-point);
values [8] ← get-values-of (grid-iters);
distances ← get-distances-to-grid-iters (grid-point, grid-iters [0]);
V ← linearly-weight-values-based-on-dist. (distances, values);

Algorithm 1: Eight-point linear interpolation

The algorithm differentiates between a grid coordinate system *point*, and a grid coordinate system *iterator*. The former is some arbitrary vector in 3-dimensional space defined by the grid coordinate system. The latter is a dereferenceable and mutable iterator of the grid iterator scheme, pointing to a value stored in memory. To compute the interpolated density, the algorithm first converts the coordinate from its given coordinate system (cartesian, fractional, etc.) to the grid coordinate system. The algorithm then computes the coordinates for the eight grid coordinate system iterators which surround the given point. Then, the values of the iterators are acquired—this can be a non-trivial computation, potentially exploiting the asymmetric, translation-independent, or non-symmetric symmetries. The distance from the

point whose value is being interpolated to the "lower-left" grid iterator is calculated, and six weights are computed from the components of the distance as depicted in Figure 7.

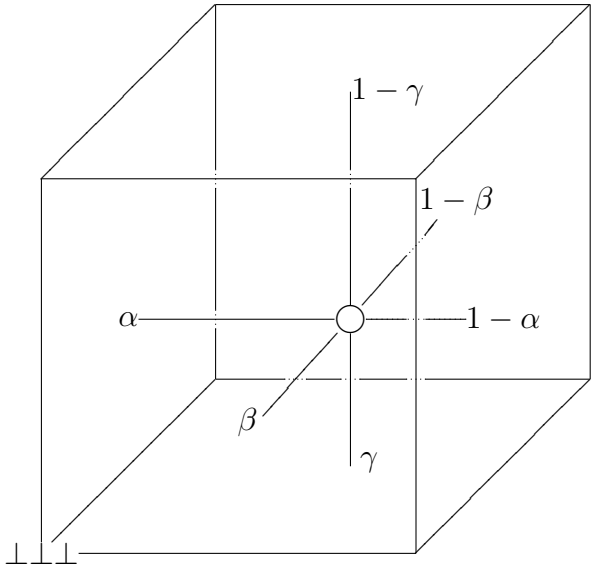



Fig. 7. The eight-point interpolation algorithm. (1) Given an arbitrary point in space (the sphere "◯"), (2) find the eight surrounding grid iterators/coordinates, and (3) measure the distance from the given point to the "lower-left" iterator (marked "⊥⊥⊥"). The distances—$\alpha$, $\beta$, $\gamma$—are used to weight the values of the eight coordinates to find the linearly interpolated value. Finding the eight surrounding points may require finding symmetric copies which are possibly "far away" in the the underlying data, necessitating costly computations. For the non-symmetric case no computations are needed, for translation-independent data the computations are a modulus operations, but for asymmetric data searching a list of 4×4 rotation-translation matrix operators is required. Because TEXTAL and CCTBX do not have efficient algorithms to find the operator, asymmetric algorithms are almost always slower than their translation-independent equivalents, even though there is less data.

c. Generic density interpolation using iterator selection

The most efficient way to compute the density interpolation is with the non-symmetric symmetry using the linear-array iterator scheme. Our benchmark for the fastest DI implementation is the TEXTAL function, `InterpolateDensity`, which makes these assumptions about the data. In addition, TEXTAL's `InterpolateDensity` is highly optimized: for example, it hand-unrolls all the loops that compute the weighted-average. To acquire the values of the grid coordinates surrounding the point to be interpolated, the TEXTAL code converts the lower-left grid coordinate to a linear-array iterator, then uses pre-computed offsets to find the other seven surrounding grid points, similar to Algorithm 2. By using precomputed offsets, the algorithm saves the recomputation of linear offsets into the underlying linear-array for the other seven grid iterators.

The TEXTAL `InterpolateDensity` code is dependent upon the particular selection of symmetry. Furthermore, different stages of the algorithm use different iterator schemes. TEXTAL encodes the concrete iterator types directly into the `InterpolateDensity` function, which is a non-generic function that only works with the TEXTAL's data structure representing the electron density map.

As described above, the interpolation algorithm consists of two parts: acquiring the values of the surrounding points, and computing the value in the current coordinate as a weighted average of the values of the surrounding points. The former part must be encoded differently for different symmetries, the latter part works for any symmetry. In order to make the code generic, we must first factor the symmetry-dependent value-acquisition code out of the symmetry-independent weight-averaging code. The weight-averaging code uses only the grid iterator scheme. The value-acquisition code uses different iteration schemes depending upon the symmetry. In

our implementation of the value-acquisition code in Figure 2 for non-symmetric data, we use the linear-array iteration scheme. To acquire a linear-array iterator we use our framework with the tag `linear`. We then convert the input grid coordinate to an offset with the function `linearize`.

**input** : E *Electron Density*; X *Grid Coordinate*; values *Values*
**output**: values *Values*

```
data        ←   begin<linear> (E) ;
data        +=  linearize(X);
values [0]  ←   *data;
values [1]  ←   *(data+1);
data        +=  stride (E,0);
values [2]  ←   *data;
values [3]  ←   *(data+1);
data        +=  stride (E,1);
values [6]  ←   *data;
values [7]  ←   *(data+1);
data        -=  stride (E,0);
values [4]  ←   *data;
values [5]  ←   *(data+1);
```

Algorithm 2: Part of our implementation of the electron density interpolation algorithm. Here, we acquire the values of the eight surrounding coordinates of a given point. We use the iterator selection framework to choose the linear-array iterator for this purpose (see line 3). This code uses pre-computed offsets stored in the electron density container, and accessed by the function `stride`. The iterator is offset by the linearized grid coordinate `X` to get to the value in the lower-left corner; the other seven values are calculated from pre-computed offsets into the linear-array. Using the linear-array iterator, instead of a grid iterator, dramatically boosts the speed of this implementation. Except for a change to line 3, this code is nearly identical to the corresponding part of TEXTAL's `InterpolateDensity` function.

The resulting code is nearly identical to the TEXTAL code, except for the call to acquire the alternate iterator scheme. Without the ability to specify the linear-array iterator scheme we would be unable to take advantage of the "pre-computed offsets" optimization described above. This very localized change allows us to write

both symmetry-independent and generic code, making the algorithm usable with any electron density container: the algorithm is parametrized over the coordinate type, the electron density container, and the symmetry type. This means that the client can pass in a point from any coordinate system (cartesian, fractional, grid, linear), and implicitly or explicitly specify any type of symmetry (non-symmetric, translation-independent, asymmetric), and get an interpolated value.

We compared our implementation to the density interpolation algorithm to that of TEXTAL's; to the comparable but less optimized standard density interpolation routine in CCTBX called `nonsymmetric_eight_point_interpolation`; and to the CCTBX's routine `basic_map::get_value` which is more general than the above two. The CCTBX's `basic_map::get_value` function supports different symmetries in the value acquisition through the use of virtual functions and overloading. Using a dual-processor, hyper-threaded, 3.04 GHz Pentium IV, with 2 GB of RAM, our generic C++ implementation of the algorithm for density interpolation runs about 5–10% faster than TEXTAL's `InterpolateDensity`. The speed increase is due to suggestions[1] for additional optimizations—which we implemented—in the TEXTAL code-base. Compared to CCTBX's `nonsymmetric_eight_point_interpolation` routine, our implementation is about 3–5 times faster, and it is about 30 times faster than CCTBX's `basic_map::get_value`. We have written the necessary adaptors to use our generic algorithm with TEXTAL's `emapT` C-`struct`, and with CCTBX's numerous electron density container implementations.

[1]Source: Dr. T. Ioerger and Erik McKee.

## E. Computing with capabilities

An important aspect to the iterator selection library presented is the use of tagging as an annotation feature. The basic idea is to allow the user to add annotations which augment data-structures with information about the semantic capabilities of those data-structures, e.g., the traversal order of hierarchical iterators, the required algorithmic complexities, etc. This means that part of the definition of the interface of a data-structure is the declaration of the various capabilities that a data-structure offers. Those declarations are then accessible to the routines of an active library. For instance, if the user annotates a data-structure with complexity guarantees then the interface of the data-structure must be defined to allow algorithms to both know that such an annotation exists, and how to access that annotation. In this way the capabilities of a data-structure can 'permeate' through the function-call boundary and be available to the software library.

The implementation of the annotation mechanism used in this chapter relies on type-traits and type-tagging—although it could prossibly be implemented more naturally using the proposed concepts feature [42]. The type-tags are then used as auxiliary parameters to an internally defined function. Because the mechanism used is ad hoc it would be an interesting direction of study whether a more general selection mechanism is possible. We can envision a formalism that essentially defines rules for computing with capabilities, and a type system that is harnessed to perform these computations. Such a system would share many properties with Gregor's *algorithm concepts* [37]: a mechanism which attaches a lattice of qualitative and quantitative annotations to data-structures, and a set of requirements onto algorithms, and then hijacks the function overload resolution mechanism to act as an algorithm selection mechanism. Algorithm concepts allow an arbitrary set of attributes to be defined,

along with a pre-order of those attributes. If the attributes satisfy the notion of a meet semilattice, then the algorithm selection mechanism can find a best-matching algorithm for a given set of inputs. This could potentially allow for more accurate information for algorithm selection.

The use of quantitative, e.g., run-time complexity, or memory-utilization polynomials, rather than qualitative, e.g., sorted, or random-access tags, is another important feature that could be considered. Such quantitative tags could be used to automatically annotate the run-time or memory complexity of an algorithm, or a call-site. Analyses for the automatic discovery of the worst case time complexity bounds exist, for an early example see Rosendahl [84]. More modern analyses employing much more sophisticated mechanism, i.e., abstract interpretation, the use of automatic theorem solvers, and other techniques, also exist; for example, see Ley-Wild [66]. Most such systems describe the run-time complexity of a few fundamental operations then analyze the control-flow of a function to find upper-bounds on the run-time (or memory) complexity. Such analyses could be used to deduce the run-time complexity requirements on the input parameters' interfaces for a routine. For instance, an routine such STL's `lower_bound` could require that the iterators passed to it have a run-time complexity for the function `advance` such that the composition of `advance`'s run-time complexity and `lower_bound`'s run-time complexity is never worse than $O(\log(N))$, i.e., `advance` must have complexity $O(1)$.

We realize this is a far-from-concrete proposal; this discussion should be seen as laying out future possible avenues of research. However, for this dissertation, we will not consider such ideas in greater depth.

## F. Conclusion and future work

The proposed iterator request framework allows containers to provide multiple iterator schemes in an easily accessible way. The iterator scheme is requested using a tag class, and not by a dedicated function name corresponding to the iterator scheme. Thus, the iteration scheme can be a parameter in a generic algorithm: a generic algorithm does not have to hard-wire the iterator scheme it uses. A single algorithm can even use several iteration schemes from the same container. Selecting a different iteration scheme becomes a simple change to the tag used to request the iterator.

The particular techniques of this chapter can be implemented in C++, as demonstrated above. First, we begin with the assumption that there exist families of related algorithms and data-structures. In languages such as C, members of a family are marked by mangling the names of the members, e.g., literal name mangling, namespaces, embedding in classes, etc. The name-mangling adds an annotation to the user about the semantics of the function or data-structure. This additional information is used implicitly by the functions that the data-structures are passed to. It is then up to the user to determine which function to call, based upon the capabilities of the data-structure being used.

Instead, for our technique, the additional information is added as an annotation that is accessible by functions or other data-structures (in C++ we use traits or concepts). The strategy is to then have a single function whose purpose is to select amongst a group of similar functions. This selection function takes, as inputs, the data-structures that are to be passed to the functions which actually implement the algorithm the user is interested in. However, the selection function operates on the capabilities of the inputs. The strategy we suggest for a selection function is to have a default implementation of an algorithm that assumes some minimal semantic ca-

pabilities of the data-structure. If, based upon the capabilities of the data-structures passed in, there exists a more efficient (faster, smaller footprint, etc.) implementation of the algorithm, then that implementation of the algorithm is selected instead. We suggest the use of a compile-time selection fnction based on function overloading and tag-dispatching to select amongst the versions of the run-time algorithm. The use of a compile-time selection function can minimize the impact to run-time performance, since the actual selection algorithm is performed during compilation and not at run-time.

In this chapter, we are proposing a method for adding to the information usually available to the algorithm which selects the run-time algorithm implemented by the function. Such extra information can encode information that is not usually considered, i.e., fast vs. safe, etc. Depending upon the exact extra capabilities allowed, the explicit access given to the user could violate the preconditions of the functions being selected. (Alternately, the algorithm which selects the algorithm to be run can guard against such selections.) In this paper, we do not give the user the ability to override correctness preconditions, and instead expose alternate, equivalent, and safe versions based upon the selected capabilities.

## 1. Readability

Using iterators in the interfaces of functions rather than the containers the functions operate on reduces readability [105] — an aspect of *minimum description length principle*. The minimum description length principle (MDL) describes the trade-offs of performance and generality versus readability and maintainability. MDL argues that as the need for generality and performance requirement of a function increases, the interface, i.e., number of arguments, increases dramatically. Conversely, as the number of arguments to a function increases, the ability of the programmer to correctly

utilize (and select correct inputs!) drops. The iterator selection mechanism develops an API that can both reduce the complexity of the interface (exchanging pairs of iterators for the container), while still maintaining flexibility for the selection of parameters within the function, through optional tag parameters.

The motivation for the framework comes from our experiences with implementing generic algorithms for the domain of computational protein crystallography. In that domain, the majority of the generic algorithms operate on containers, rather than pairs of iterators. Moreover, the containers support numerous iteration schemes, the choice of which has a dramatic impact on performance of the algorithms. The framework allowed us to write our algorithms in a fully generic way, and apply the most appropriate iteration scheme for each calling context. We found the framework crucial for writing efficient code for the complex data-structures in the domain.

The iterator selection framework is a generalization of the family of functions, such as `begin`, `end`, `rbegin`, `rend`, where a fixed signature is used to refer to an iterator scheme. Abstracting the iterator scheme in the `begin` and `end` functions is a new and beneficial axis of parametrization in generic libraries. Essentially, by varying the iteration scheme, we can get drastically different behavior and performance from the exact same algorithm or code.

In this chapter we discussed several iterator schemes and their corresponding tags in context of isolated examples. In future work, our next step is to continue to analyze the iterator tags and identify a set of generally applicable and "standardized" tags and their refinement hierarchy. We believe that with a full "concept analysis" a small number of useful tags, analogous to the STL's iterator hierarchy, can be developed.

Furthermore, we note that while we have presented a framework for iterators, the idea is more general. Permeable interfaces for active libraries in C++ rely on associated type and traits classes to attach properties to types. However, those mechanisms can

be inflexible: they do not allow either transient properties which hold only occasionally at run-time, or properties that hold only in some contexts. It is our viewpoint that active libraries should have APIs that are parametrized by (at least some) of the properties of the input types to their components. This will allow more expressive and powerful libraries.

# CHAPTER III

## MULTILAYER LIBRARY COMPOSITION

This chapter focuses on the definition, applications and limitations of *concepts* and *concept maps* in C++, with an emphasis on library composition. We report on two cases of data structure adaptation between different libraries. We compose an image processing library to a graph algorithm library by making use of a transparent adaptation layer, enabling the effortless application of graph algorithms to the image processing domain. We use the adaptation layer to realize a few key algorithms, and report little or no performance degradation when compared to algorithms implemented directly for the image processing domain. We then extend this adaptation to include a composition of a graph library with a linear algebra library. This multilayer composition implements several image analysis algorithms by directly following their presentation in the research literature in terms of graphs and matrices; we thus have high confidence in their correctness.

### A. Introduction

Modern software systems commonly make use of components from a variety of software libraries. Software libraries available to programmers are typically developed by different entities without centralized control. Consequently, different libraries' interfaces are seldom directly compatible. The cost and complexity of the code needed to combine libraries is significant, and can be prohibitively expensive: it may be easier to rewrite the needed components than to reuse them, or the performance overhead of the library composition mechanism may not be acceptable.

The language constructs and idioms for adaptation vary greatly between different programming languages, and can impact the cost of library composition. This

chapter discusses programming with C++ "concepts" [40], a set of extensions to the C++ template system which augment C++'s template system with constraints; this is an experimental system being considered for future versions of C++. We explore the applicability and limitations of these new features, particularly focusing on the use of concepts for library composition via non-intrusive component adaptation. At the moment, ConceptGCC [38] is the only compiler for ConceptC++. Note that in this chapter, we will refer to C++ extended with concepts as *ConceptC++*; *C++ 2003* will be used to denote the language as specified in its current standard [51].

The C++ standard library's collection of generic algorithms and data structures, formerly called the Standard Template Library (STL) [97], was the central use case that influenced the design of ConceptC++. Consequently, the first application of ConceptC++ was the STL. Naturally, the next step is to explore the applications of these new language features to broader domains. In this chapter we report on the use of concepts for adapting entire library interfaces, enabling the composition of whole generic libraries. Specifically, we (1) demonstrate how to adapt components in a non-intrusive and efficient manner using concepts, (2) guide programmers in the effective use of the C++ concepts feature, (3) report on the evaluation of the new features in ConceptC++ for the support of complex library composition, (4) evaluate the performance implications of the new features, (5) compare and relate the features to other adaptation mechanisms in C++ and in other languages, and (6) raise some issues and describe challenges faced when programming with concepts.

The structure of the chapter is as follows. Section B briefly summarizes the paradigm of *generic programming* that has motivated the design of ConceptC++, and introduces the main features of ConceptC++. Section 1 begins by demonstrating how one of these features, the *concept map* language construct, can be used to adapt generic components. Section C describes a complex library composition scenario,

where a transparent adaptation layer enables the use of an open-ended set of image types as inputs to a library of graph algorithms; we then show the composition of the graph algorithm library with an iterative eigensolver library, allowing the use of graph data-structures as inputs to iterative eigensolver algorithms. Performance of such adaptations is discussed in each relevant section. Section G relates concepts and concept maps to other adaptation mechanisms, such as instance declarations in Haskell and inheritance in object-oriented languages, and discusses concept maps' strengths and limitations. Conclusions follow in Section G.

## B. Background

The design of ConceptC++ has mainly been motivated by the desire to better support the paradigm of *generic programming*, as practiced, for example, in the design and implementation of active libraries such as Standard Template Library, Boost Graph Library (BGL) [89], Matrix Template Library [90] and other generic libraries in a variety of domains [28, 10, 3, 81]. The generic programming approach to library design has proven to support the production of efficient and reusable libraries. For example, the STL and the BGL are both large libraries providing extensive functionality, yet the interfaces to these libraries are quite small. Through careful consideration of the essential requirements for related classes of algorithms, the interface to a large number of library components has been made small and uniform. The generic programming paradigm, and generic libraries, are of interest in the context of library composition since adapting a particular widely applicable generic library interface to the requirements of another library may open up significant re-use opportunities. For example, the BGL, with a few dozens of lines of code, implements a transparent adaptation layer on top of some graph data structures of the LEDA library [74], making

the entire BGL usable for LEDA graphs without requiring any explicit wrapping or adaptation [89, §14.3.5].

### 1. From C++ 2003 to ConceptC++

In C++ 2003 templates are unconstrained. Generic C++ 2003 libraries, therefore, generally express constraints on type parameters of generic algorithms as part of algorithms' documentation and as names of template parameters. The STL established a systematic documentation style for this [93, 5], in which requirements on one or more types are collected into tables. These tables describe the functions and operators that the types must support. They can also require a set of other accessible types, called *associated types*. Naming conventions for template parameters are used to indicate the corresponding requirements table. These conventions, however, do not serve as first-class artifacts from the compiler's perspective: they only exist in the mind of the programmer.

Interfaces between components that are visible to the compiler are a key element in successful composition strategies and lead to a number of benefits. First, it is possible for the compiler to check and enforce the contract expressed by an interface to various extents depending on the language. Next, semantic properties captured in interface specifications can be leveraged for use in the optimization and transformation of codes. For instance, if the interface includes a binary operator, and an absorber element for that operator, i.e., multiplication and 0, then any expression of the form `a*b` where either `a=0` or `b=0` can be transformed to the expression `0`. Additionally, interfaces as first-class language constructs also offer a place to bundle related signatures and constraints. This gives programmers the ability to create coherent constructs about which readers can reason. This helps to reduce long-term maintenance costs of the code.

When introduced to generic programming with C++ 2003, programmers accustomed to object-oriented languages have difficulty when they fail to find component requirements in constructs analogous to the familiar abstract base classes. Such challenges are exacerbated by lengthy compiler diagnostics[1] that arise because the bulk of type checking of templates in C++ 2003 occurs late, at the time of their instantiation. ConceptC++, at last, changes this for generic programming in C++. The `concept` language construct gives an *explicit* representation of the syntactic requirements tables, and it is precisely these concept definitions that serve as the first class interfaces. Since concepts are analyzed by the compiler (in addition to the programmer), modular type checking of templates is possible; features such as concept-based overloading also become easier to use. For example, ConceptC++ can provide notably more informative compiler error diagnostics [40].

Note that the development of ConceptC++ was preceded by cleverly designed template libraries that emulated concepts. These libraries provide some support for expressing requirements tables programmatically, for enforcing constraints on template parameters expressed using those tables, and for rudimentary "type checking" of template bodies [73, 91]. These techniques are brittle and expert-friendly, and have not found their way to wide use.

The STL's requirements tables also specify semantic requirements as algebraic laws that implementations of required functions must satisfy as well as upper bounds for the algorithmic complexity of these functions. ConceptC++ supports expressing algebraic laws [44, §14.9.1.4] in concepts, as *axioms*. Type checking in ConceptC++, however, is not concerned with concepts' axioms (except for insisting that the expressions in axioms themselves are well-formed). Compilers and programmers can use

[1] Järvi et al. have experienced a 20MB error message from a single client error in the use of a, admittedly very complex, template library. [56]

axioms to justify optimizations, and they serve as a hook for auxiliary language tools. We do not further explore the usefulness of axioms in this dissertation.

## 2. Generic programming in ConceptC++

This section briefly describes the new language constructs in ConceptC++. More detailed description is available in the C++ concepts proposal overview [40], and in the current full specification of concepts [44]. The central new construct is `concept`. It defines a set of requirements on a type or tuple of types. We say that types that satisfy the requirements of a concept *model* that concept. For example, the following concept [42, §20.1.2] requires that the *less-than operator* (`<`), and the other comparison operators, are defined for objects of type `T`:

```
concept LessThanComparable<typename T> {
  bool operator<(const T& a, const T& b);
  bool operator>(const T& a, const T& b) { return b < a; }
  bool operator<=(const T& a, const T& b) { return !(b < a); }
  bool operator>=(const T& a, const T& b) { return !(a < b); }
}
```

ConceptC++ requires an explicit declaration, a *concept map*,[2] to establish that a particular type (or a parametrized class of types) models a concept. For example, the following definition states that the type `int` models the concept `LessThanComparable`:

```
concept_map LessThanComparable<int> { }
```

Another concept map makes the user defined type `name` a model of the concept `LessThanComparable`:

[2]The possibly more descriptive keyword "`model`" that was used in preliminary designs of ConceptC++, was replaced with the keyword `concept_map`, which occurs far less frequently in existing C++ code.

```
struct name { char* first; char* last; };
int namecmp(const name& n1, const name& n2) {
  int c = strcmp(n1.last, n2.last);
  if (c==0) return strcmp(n1.first, n2.first);
  else return c;
};
concept_map LessThanComparable<name> {
  bool operator<(const name& a, const name& b) {
    return namecmp(a, b) < 0;
  }
}
```

The two concept maps differ in how they satisfy the `LessThanComparable` concept's requirements. For `int`, the body of the concept map is empty; the built-in comparison operators for integers satisfy the four requirements of the `LessThanComparable` concept. For `name`, one of the required operations, the less-than operator, is defined in the body of the concept map. This suffices to make `name` a model of the `LessThanComparable` concept, since the concept's body provides *default implementations*, defined in terms of the less-than operator, for the other three required operations. A type can satisfy a requirement, for example the requirement for a less-than-operator, in any one of the following ways, in decreasing order of precedence: (1) the concept map can define the less-than operator explicitly, (2) the less-than operator can be defined as a member or non-member function (or in some cases as a built-in operation), (3) or a default implementation in the concept itself can provide a definition. In a well-formed concept map, each required operation is defined in at least one of these ways.

Explicit definitions of functions in the bodies of concept maps are a powerful tool for adaptation. For example, objects of the `name` class can be compared using the `namecmp` function, whose interface is similar to that of the `strcmp` function. The con-

cept map above adapts `name` to satisfy the requirements of the `LessThanComparable` concept, and defines the less-than operator in terms of the existing `namecmp` function. In this adaptation, the definition of the type `name` does not need to be modified, and the objects of type `name` do not need to be wrapped by other types, to be comparable with the less-than operator.

Definitions in concept maps do not introduce functions or type names into the global scope. For example, the less-than operator defined in the above concept map is only visible in generic definitions constrained by the `LessThanComparable` concept.

Concept maps can be made generic with templates. For example, the following concept map declares all instances of the standard template `pair` to be models of `LessThanComparable`—as long as the element types of the pair model the concept `LessThanComparable`. The constraints on the element types, the template parameters `T` and `U`, are stated in the *requires clause*, introduced with the `requires` keyword.

```
template <typename T, typename U>
  requires LessThanComparable<T> && LessThanComparable<U>
concept_map LessThanComparable<pair<T, U> > {
  bool operator<(const pair<T, U>& a, const pair<T, U>& b) {
    return a.first < b.first
      || (!(b.first < a.first) && a.second < b.second);
  }
}
```

Figure 8 shows a simple generic algorithm, `min_element`; the algorithm uses the `LessThanComparable` concept as a constraint. Constraints in the requires clause are assumed to hold during type checking of the template's body, and they are enforced at the time of template instantiation. The `ForwardIterator` concept that appears in the constraints of `min_element` is shown in Figure 9. This concept provides basic iteration capabilities. The indirection operator (`*`) gives the value that an iterator refers to.

```
template <typename Iter>
  requires ForwardIterator<Iter> && LessThanComparable<Iter::value_type>
Iter min_element(Iter first, Iter last) {
  Iter best = first;
  while (first != last) {
    if (*first < *best) best = first;
    ++first;
  }
  return best;
}
```

Fig. 8. The `min_element` generic algorithm.

The increment operator (`++`) advances an iterator to the next element. Equality comparison is used to decide when the end of a sequence is reached. Finally, iterators can be copied and assigned. Requirements for the equality operator (`==`) and the inequality operator (`!=`), as well as for copying and assignment, are not stated directly in the body of `ForwardIterator`, but are obtained through *refinement* of another concept `Regular`. A concept, `D`, is said to refine another concept, `B`, when all of `D`'s requirements are included in `B`'s requirements. The syntax for expressing refinement relationships resembles that used for expressing class inheritance relationships. The `Regular` concept (not shown, see [42, §20.1.7]) collects several common requirements supported by most types, including equality comparison, and the abilities to copy and assign objects. The associated type `value_type` denotes the type of values that the iterator refers to. A `requires` clause in the body of a concept can place additional constraints on the parameters or associated types of a concept. Here, `value_type` must model `CopyConstructible`, a self-explanatory concept from the proposal submitted by Gregor et al. [42] for the upcoming revision of C++. Examples of models of `ForwardIterator` include all pointer types and the iterator types of standard containers.

The `min_element` algorithm works for any sequence of values delimited by a pair of iterators, as long as the iterator type is a model of the `ForwardIterator` concept and the iterator's associated `value_type` is a model of the `LessThanComparable` concept. In the next few paragraphs, we illustrate how type checking works for the call to `min_element`, shown in Figure 10.

The C++ standard library specifies that `vector`'s `begin` and `end` member functions shall return types that satisfy the `ForwardIterator` concept. (In fact, these types satisfy stronger constraints, but this is not important for our purposes.) This satisfies the first requirement on types supplied to `min_element`. The second requirement, that the `iterator`'s associated `value_type` is a model of the concept `LessThanComparable`, is satisfied via the concept map for `LessThanComparable<name>` given earlier. As both these requirements are, collectively, met by the `vector<name>`'s iterator type and the `name` type, the call to `min_element` in Figure 10 passes type checking.

To illustrate the role of concept maps in type checking, and as adapters, consider the call to the less-than operator, "`*first < *best`", in the body of `min_element`. The type checker resolves this call to the `LessThanComparable` concept's less-than operator, looked up in the concept map `LessThanComparable<Iter::value_type>`. In the example in Figure 10, at template instantiation time the placeholder associated

```
concept ForwardIterator<typename Iter> : Regular<Iter> {
  typename value_type;
  requires CopyConstructible<value_type>;
  value_type& operator*(const Iter&);
  Iter& operator++(Iter&);
  Iter operator++(Iter&, int);
}
```

Fig. 9. The `ForwardIterator` concept (simplified from the one in the STL).

```
vector<name> names;
// fill names with values
name first_in_line = min_element(names.begin(), names.end());
```

Fig. 10. A call to the generic `min_element` function.

type `Iter::value_type` is bound to the `name` type; thus, the call to the less-than operator resolves to `LessThanComparable<name>::operator<(*first,*best)`, which is implemented in terms of `namecmp` in the `LessThanComparable<name>` concept map.

Indirections through concept maps are efficient. Recall that in C++ 2003's template compilation model distinct code is generated whenever a template is instantiated with different types—this compilation model is used in ConceptC++ as well, after type checking a template instantiation. Once the type checker has accepted that the types bound to the template arguments satisfy the template's constraints, code specialized for the particular template instance, in this instance, the function `min_element<vector<name>::iterator>` is generated. Consequently, the calls that depend on template parameters, such as the less-than operator call above, are statically resolved and subject to inlining and other compiler optimizations. This applies, in particular, to calls directed through functions in concept maps. Gregor and Siek give a more detailed account of type checking and compiling ConceptC++'s constrained templates [43].

A concept definition can be preceded with the keyword `auto`, signifying that no explicit concept map is necessary to establish an is-a-model-of relation between a type and a concept—it suffices that all functions and operations required by the concept are defined for the type. Concept maps can, however, also be written explicitly for `auto` concepts. Simple concepts, with only a few requirements, are typically defined

as `auto`; we could define `LessThanComparable` as:

**auto concept** LessThanComparable<**typename** T> { ... }

Throughout this chapter, we use ConceptC++'s syntactic shortcuts for succinct expression of constraints: instead of the keyword `typename`, a concept name can precede a template parameter in a template parameter list, or an associated type in the body of a concept. To demonstrate the former use of the shortcut, the signature of `min_element` in Figure 8 can be re-written as:

**template** <ForwardIterator Iter>
  **requires** LessThanComparable<Iter::value_type>
Iter min_element(Iter first, Iter last);

As an example of the latter use, lines 2 and 3 in the body of the `ForwardIterator` concept in Figure 9 can be replaced with the requirements clause declaration of "`CopyConstructible value_type;`".

## C. Cross-domain composition

When a concept map adapts a particular type to model a concept, the concept map implements the operations required by the concept in terms of the functionality provided by the type. In this section we move beyond the adaptation of individual types to the adaptation of entire library interfaces. An important aspect of adapting two or more library interfaces is that instead of one type modeling a concept, we define a modeling relationship between concepts defined for two entire library interfaces. This means that a collection of types from one library will model one or more concepts from another library.

We begin with a recipe showing such a modeling relationship between library interfaces in Figure 11. On the left are the data-structures $A_1, A_2, ...$ which all model

(shown by single arrows) the concept $A$, where the concept $A$ defines the interface of a library. A (templated; parametric) concept map is defined so that the concept $A$ models the concept $B$, shown by a double arrow. The concept $B$ defines the interface of another library; a collection of functions in the second library $f^1, f^2, ...$ take as inputs components which must satisfy the concept $B$. We summarize the process: first, data-structures $A_1$, $A_2$, etc. are defined to model concepts representing data-structures from the interface of a library **A**; then, a concept map (or many) is defined from the concepts of library **A** to the concepts of library **B**; and as a result, algorithms in library **B** can use data-structures from library **A**.

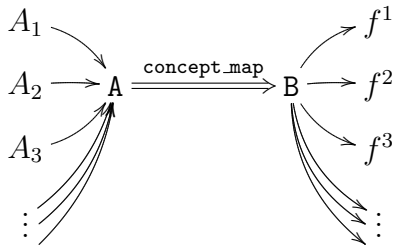



Fig. 11. Templated concept map definition describing a modeling relationship between two concepts. Data-structures $A_1$, $A_2$, etc. on the left of the figure are models of the concept `A`; the modeling relationship is represented with an arrow "$\rightarrow$". The concept `A` is adapted to model the concept `B` using a concept map that implements $B$'s required operations in terms of operations provided by $A$, shown with a double arrow "$\Rightarrow$". Any function $f^1$, $f^2$, etc. which takes as inputs components that must satisfy concept $B$, represented with an arrow "$\rightarrow$", can be passed data-structures that model the concept $B$.

We now explore a mapping between concepts that adapts abstractions from one domain to those of another domain. This example of cross-domain composition is from the domains of image processing and graph algorithms. Many image algorithms can be viewed as graph algorithms given a suitable representation of images as graphs [87, 24, 88]. In this section we present a partial composition of the Boost Graph Library (BGL) [89] and the Generic Image Library (GIL) [10] that enables

many image processing algorithms to be implemented as simple wrapper functions over BGL algorithms. We show the mapping from image-related concepts defined in the GIL to the graph concepts of the BGL. Concept maps are instrumental in such cross-domain compositions. The adaptation code involves relatively few lines of code, is transparent to the client, and comes with minimal performance cost.

Note that as part of the adaptation we define a handful of classes that are used as the associated types for the graph concepts. For example, one of the graph concepts requires an associated type `out_edge_iterator`, used by the graph algorithms to traverse the out edges of a node in a graph. The image concepts do not require or guarantee the existence of any types that can directly satisfy the requirements for this associated type, so we create a new small class for this purpose. Objects of this class maintain the state of iteration over a pixel's immediate neighbors. The newly created class does not intrude on the image library, and clients of the image library do not need to explicitly define objects of the iterator class, or even be knowledgeable of its existence. This arrangement serves as an example of the division of labor, in the case when stateful adaptation is needed, between concepts, concept maps, and traditional adaptation via the creation of new classes.

### 1. Background of GIL and BGL

The Generic Image Library is Adobe's open source image processing library, and also part of the *C++ Boost* collection of peer-reviewed C++ libraries (www.boost.org). The GIL defines concepts for raster images of any dimension, and provides generic implementations of basic image algorithms, such as copying, comparing, and applying a convolution. The GIL's algorithms operate on an open-ended set of image types that may vary in color-space, pixel type, storage order, and other image characteristics.

The Boost Graph Library [92] is a widely used library of generic algorithms for

manipulating graphs. The BGL defines concepts that describe different capabilities for graph data structures, such as *incidence graphs* that provide access to the outgoing edges of each vertex, *vertex list graphs* that additionally allow access to all vertices in the graph, and *edge list graphs* that add the ability to access all edges in the graph. The BGL also provides useful data structures modeling these concepts, many implemented in terms of STL containers (essentially as compositions of vectors, lists, and maps).

Neither the BGL nor the GIL are yet implemented using ConceptC++. We reimplement in ConceptC++ a subset of these libraries for our experiments. We omit support for mechanisms like the BGL's "named parameters" [89, §2]. The BGL describes its algorithms' requirements using STL-like concept documentation, and the GIL uses pseudo-code mimicking ConceptC++ to document its concepts; our concepts are translations of these documents into ConceptC++.

For the adaptation layer between the GIL and the BGL we defined concept maps for several GIL concepts, making those concepts models of various graph concepts in the BGL. Using the adaptation layer, many image processing algorithms can be implemented as thin wrappers over the BGL's graph algorithms. We describe the implementation of the adaptation layer, along with the implementations of multiple algorithms. In particular, we focus on the *flood-fill* algorithm. This algorithm modifies the color of a set of contiguous pixels that satisfy a predicate. The implementation of this algorithm performs a recursive search through neighboring pixels of an initial seed pixel. Applications of the flood-fill algorithm include transformation of a block of one color to another, insertion of a background texture (green screening), and image partitioning. We also report on using the adaptation layer to image *segmentation*. Graph-based image segmentation refers to a set of techniques for finding a partition of an image by representing the image as a graph, then finding a partition of the

graph using, e.g., edge weights or minimal cuts as the partitioning criteria [29, 88]. We chose to implement a basic segmentation algorithm based on pixel similarity. The third algorithm we describe is for finding minimal-energy paths between two points in an image. This can be accomplished with the Bellman-Ford [7] shortest path graph algorithm when the input image is represented as a graph.

## 2. GIL–BGL composition

In our adaptation of images to graphs, vertices correspond to pixel locations and each of the edges connect two vertices corresponding to neighboring pixel locations. Note that we are differentiating between a pixel's location and a pixel's chromatic value. The flood-fill algorithm is essentially a breadth-first graph search. The BGL's breadth-first search algorithm imposes several concept requirements on the types of its inputs, which now have to be satisfied by the image type. We could establish the image-to-graph correspondence directly with concept maps that adapt concrete image types to the BGL graph concepts. However, a broader adaptation for an open ended class of image types is achieved if we adapt generically all types that model GIL image concepts to model BGL concepts. Furthermore, the adaptation is not specific to breadth-first search and to flood-fill; many algorithms in the BGL use the same handful of concepts in their constraints.

The `breadth_first_search` function in our graph library is shown in Figure 12. For brevity, in all code examples we omit header includes, namespace prefixes of names from both the GIL and the BGL, and the prefix `std::` for names defined in the standard library. The `breadth_first_search` function is parametrized on the graph type, the type of queue used for storing references to vertices to maintain search state, a visitor type used for providing callback functions for various event points of the algorithm, and the type of color map used for tracking which vertices have already

been visited. The `breadth_first_search` function uses four concepts to constrain its template parameters. The `IncidenceGraph` concept specifies the requirements for the graph type: operations for enumerating out-edges of a given vertex, along with their incident vertices. The other concepts are `Buffer`, which describes the operations of the vertex queue; `BFSVisitor`, which specifies the dictionary of the callback functions; and `ColorMap`, which defines the interface to the data structure storing vertex visitation information.

```
template <IncidenceGraph G, Buffer Queue,
          typename Visitor, ColorMap CMap>
  requires BFSVisitor<Visitor, G>
    && SameType3<G::vertex_t, CMap::key_type, Queue::value_type>
void breadth_first_search (const G& g, const G::vertex_t& s,
                           Queue& Q, Visitor V, CMap Color);
```

Fig. 12. The signature of the `breadth_first_search` function in the BGL. The *same type* constraint guarantees that types of the function arguments are consistent, that is, that the type of the values in the queue argument and the key type of the color map, are the same as the type of the vertices in the graph.

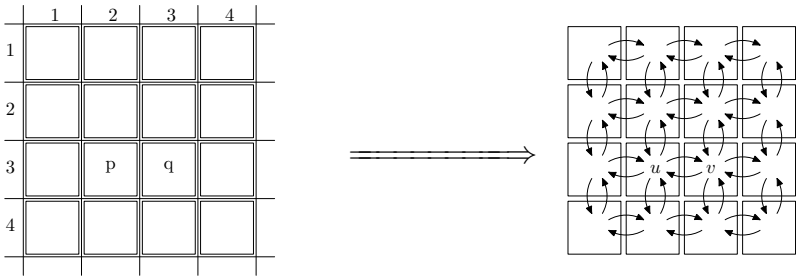
We focus on the `IncidenceGraph` concept, shown in Figure 13, in the description of the adaptation layer between images and graphs. The `Graph` concept, also in Figure 13, specifies associated vertex and edge types which, via refinement, become requirements of `IncidenceGraph`. Directly, `IncidenceGraph` requires the `out_edges`, `out_degree`, `source`, and `target` operations (lines 7–10). The `out_edges` function returns a pair of iterators that specify the sequence of edges emanating from a given vertex, and `out_degree` is for querying how many such edges there are. The associated type `out_edge_iterator` on line 4 has its expected meaning.

From the point of view of a type modeling a concept, operations specified in a concept are requirements that must be satisfied. From the point of view of an algorithm constrained by a concept, the operations are capabilities that can be relied

```
concept Graph<typename G> {
  Regular vertex_t; Regular edge_t; };
concept IncidenceGraph<typename G> : Graph<G> {
  ForwardIterator out_edge_iterator;
  requires SameType <edge_t, out_edge_iterator::value_type>;
  pair<out_edge_iterator, out_edge_iterator>
    out_edges (const vertex_t&, const G&);
  size_t out_degree (const vertex_t, const G&);
  vertex_t source (const edge_t&, const G&);
  vertex_t target (const edge_t&, const G&);  };
concept Point<typename P> : Regular<P> {
  IntegralLike value_type;
  const value_type& operator[](const P&, size_t);
  value_type& operator[](P&, size_t);
  size_t num_dimensions(P); }
concept ImageView<typename V> : Regular<V> {
  Locator locator; Point difference_type; Regular value_type;
  requires SameType <value_type, locator::value_type> &&
    SameType<difference_type, locator::difference_type>;
  const value_type& operator[] (const V&, const difference_type&);
  value_type& operator[] (V&, const difference_type&);
  difference_type dimensions(const V&);
  size_t size(const V&); };
```

Fig. 13. Concepts and concept-maps for graphs and images. Lines 3–10 shows the `IncidenceGraph` concept. The `Regular` concept, defined in the draft standard library of ConceptC++, describes a type that is "well-behaved", that is, it can be constructed, destructed, copied, assigned, and compared for equality. Furthermore, these operations adhere to fundamental laws, such as after assigning a value `x` to a variable `y`, then `y == x` returns `true`. The `ForwardIterator` (outlined in Figure 9) is also a standard concept. Lines 16–23 shows the capabilities provided by the GIL concepts that are relevant for the adaptation. The `IntegralLike` concept (not shown, see [42, §20.1.7]) is a concept. The `Locator` concept provides traversal capabilities through the values, i.e., pixels, of GIL images—we omit this concept since `Locator`s offer no generic way to determine the validity of the position of a locator. Our adaptation maintains current location in algorithms with values of the `ImageView`'s `difference_type`.

upon. In our example, the GIL concepts' capabilities are used to satisfy the BGL concepts' requirements. The concepts in Figure 13 on lines 3–10 describe the interface that the GIL imposes on images, and thus provides as images' capabilities. The `ImageView` concept on line 16 provides capabilities of a container. The associated type `value_type` ($3^{rd}$ declaration, line 17) is the type of the pixels. The `locator` ($1^{st}$ declaration, line 17) represents the position of a pixel in an image, we will refer to values of type `locator` as *pixel locators* to help distinguish their use. The `difference_type` ($2^{nd}$ declaration, line 17) represents the offsets between pixel locators, and can also be used to determine the position of a pixel in an image. The function `dimensions` (line 22) returns the extents of an image. The GIL `ImageView` concept has further requirements, such as an iterator for linear traversal over the sequence of pixels. These capabilities are not necessary for the adaptation to graphs, and are not shown.




(a) (b)

Fig. 14. The modeling relationship between images and graphs. A schematic showing the intuition of the image-to-graph modeling relationship: pixel locators for pixels '`p`' and '`q`' in the image (left) are vertices $u$ and $v$ in the graph (right), respectively. Given a pixel locator in the image, we compute the neighborhood of nearby pixel locators, represented as directed edges in the graph.

A schematic for the concept map from an incidence graph to a raster image is shown in Figure 14. Both pixel locators and the difference type of the image can be

used to represent the position of a pixel in an image, where the pixel locator can be thought of the location of a pixel and a reference to the pixel's color value. Because the pixel locator is more expensive to copy, we choose to use the difference type to represent the position of a pixel. The position of a pixel in an image models a vertex in the graph (the graph is on the right); for instance, the position of the pixel `p` is the vertex $u$. The value of the vertex that represents the pixel, i.e., its color, would thus be accessed through a property map. Given the position of a pixel such as `p` it is possible to compute the neighborhood of nearby pixel locations. This means that an edge $(u, v)$ exists in the graph if the position of the pixel `p`, represented by vertex $u$, is a neighbor of the position of the pixel `q`, represented by vertex $v$. A pair of pixel positions represents an edge; given an edge, all of the neighbors of the source of the edge can be computed as needed.

The `concept_map` that adapts `ImageView` to `IncidenceGraph` is shown in Figure 15. Lines 3–5 provide definitions for the associated types of `IncidenceGraph`. We represent vertices as the image's `difference_type`, a point type that specifies the coordinates of a pixel. Edges are pairs consisting of two vertices: the source and target. The `out_edges` function on lines 6–8 constructs a pair of edge iterators that denotes the sequence of out edges. The number of neighboring pixels, i.e., the number of out edges, for a given pixel is obtained as the `distance` between the beginning and end of the sequence of out edges, which gives directly the implementation for the `out_degree` function on lines 17–20. The `source` and `target` functions on lines 12 and 15 are trivial.

To arrive at a flood-fill algorithm, we make one additional adaptation: we use a color map tailored for flood-fill, instead of the BGL's default color map defined for `breadth_first_search`. The color map stores the search state: unseen, in progress, and processed vertices are respectively marked with white, gray, or black.

```
template <ImageView Img>
concept_map IncidenceGraph<Img> {
  typedef Img::difference_type vertex_t;
  typedef pair<vertex_t, vertex_t> edge_t;
  typedef out_edge_iterator_adapter<vertex_t> out_edge_iterator;
  inline pair<out_edge_iterator, out_edge_iterator>
  out_edges(const vertex_t& v, const Img& g) {
    out_edge_iterator first(v, dimensions(g)), last(num_dimensions(g));
    return make_pair(first, last);
  }
  vertex_t source (const edge_t& e, const Img&) {
    return e.first;
  }
  vertex_t target (const edge_t& e, const Img&) {
    return e.second;
  }
  size_t out_degree (const vertex_t& v, const Img&) {
    pair<out_edge_iterator, out_edge_iterator> iter = out_edges (v);
    return distance(iter.first, iter.second);
  }
}
```

Fig. 15. The `concept_map` adapting models of GIL `ImageView` to become models of BGL `IncidenceGraph`.

Only white vertices are added to the work queue. The default queue and visitor parameters of `breadth_first_search` that the BGL provides need no customization. With these adaptations, the generic implementation of flood-fill in terms of `breadth_first_search` is shown in Figure 16.

The queue and visitor parameters are those used in the BGL by default, but we need to specify them explicitly since our graph library does not implement the BGL's named parameters mechanism that provides support for default values for parameters.

The image segmentation algorithm mentioned in Section C can also be implemented in terms of a breadth-first graph search. The task of an image segmentation

```
template <ImageView Img, typename P>
  requires Predicate<P, Img::value_type>
void flood_fill(Img& img, const Img::difference_type& seed, P p,
                const Img::value_type& replacement) {
  if (!p(img[seed])) return;
  vector<Img::difference_type> buffer;
  breadth_first_search(img, seed, buffer, basic_bfs_visitor(),
                       color_map<Img, P>(img, p, replacement));
}
```

Fig. 16. The implementation of flood-fill using an adaptation from GIL to BGL.

algorithm is to partition an image into contiguous regions according to some criteria; its central building block is a generic partition algorithm that forms a single partition in an image; repeated invocations with different initial seed locations will segment the entire image. Implementation of the segmentation algorithm uses the same concept map adapters, but a different color map class. The default queue type suffices, but the BGL's visitor type is customized to respond to the event of discovering a vertex by adding that vertex to a data structure representing a partition.

To further evaluate the usability of cross-domain adaptation between images and graphs, we implemented the *backbone-healing* algorithm [32]. This is a practical image processing algorithm from the TEXTAL automatic model-building protein crystallography package [49], and is used in improving the results of image skeletonization. The algorithm can be implemented in terms of the Bellman-Ford shortest-path graph algorithm, which places stronger requirements on its input graphs than merely `IncidenceGraph`. The BGL's `bellman_ford` algorithm requires the capability to iterate over all edges and all vertices omitting the adjacency structure of the graph, and thus requires that its graph parameter type be a model of the `EdgeAndVertexListGraph` concept.

To allow GIL images to be used as inputs of the `bellman_ford` algorithm, a

slightly more capable adaptation is necessary, one that supports iteration over all vertices and all edges of a graph. We define two auxiliary classes, analogous to the `out_edge_iterator_adaptor` class discussed above, that masquerade traversal through an image's pixels and its neighboring pixel pairs as iteration over vertices and edges. With these classes, the adaptation between the GIL's `ImageView` concept and the BGL's `EdgeAndVertexListGraph` concept, is expressed with the concept map:

**template** <ImageView Img>
**concept_map** EdgeAndVertexListGraph<Img> { ... }

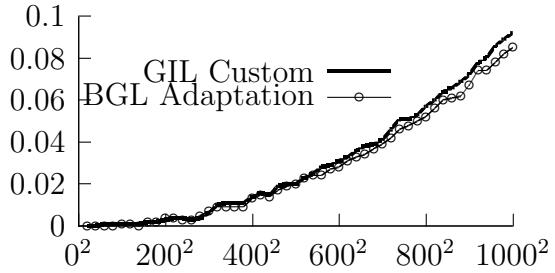
The implementation of the adaptation is similar to that between the `ImageView` and `IncidenceGraph` concepts; we experienced no notable difficulties in realizing it. We make the full code of the adaptions described above available.[3]

### 3. Performance results of the BGL to GIL adaptation

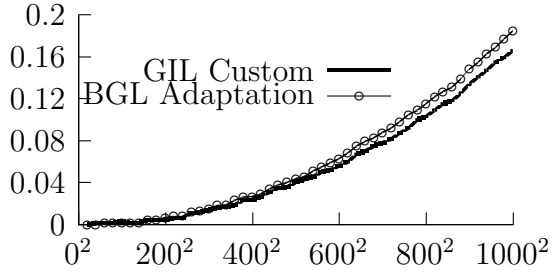
Adaptation mechanisms can have a negative impact on performance. Mitchell et al. [75] give a detailed analysis of a case where multiple inefficient adaptation layers had a major effect on the performance of a large software system. In our examples we have used adaptation freely, adding layers as appropriate to meet our design goals. In this section we explore the performance costs of adaptation implemented using concept maps.

We use the flood-fill, segmentation, and backbone-healing algorithms as test cases, and compare the execution times of two different programs for each algorithm. The first program for each algorithm is written directly in terms of the GIL concepts. Essentially, the flood-fill performs a breadth-first search tailored for images, and the segmentation algorithm a series of such searches. The backbone-healing al-

[3] parasol.cs.tamu.edu/groups/pttlgroup/programming-with-c++-concepts




(a)



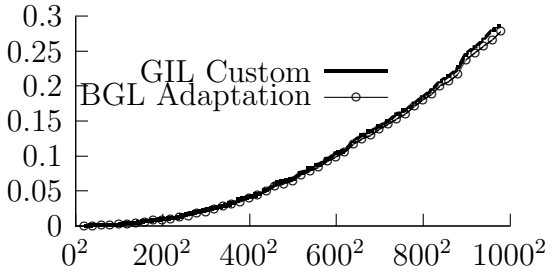

(b)

Fig. 17. The timing results for the function `flood_fill`. Results for the Intel architecture are shown in (a) and for the PowerPC are shown in (b). The $x$-axis is in seconds, and the $y$-axis is the area of the images in pixels. Each chart shows the timing results of executing two test programs, the first written directly for GIL concepts ("GIL Custom") and the second written to use the adaptation between the GIL and BGL ("BGL Adaptation").

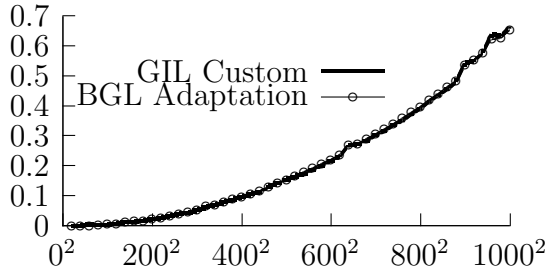
gorithm searches for all shortest paths using the Bellman-Ford algorithm that was directly written for GIL's image types. The second program for each algorithm uses concept maps to adapt GIL concepts to BGL concepts as described in Section 2, and uses BGL's `breadth_first_search` function for the flood-fill and segmentation algorithms, and `bellman_ford` for backbone-healing.

We compiled all of the test programs using the ConceptGCC [38] compiler's Alpha 7 Prerelease version[4] with the -O3 flag on two platforms: MacBook Pro (Intel

[4]conceptgcc (GCC) 4.3.0 20070330 (experimental) (Indiana University Concept-




(a)




(b)

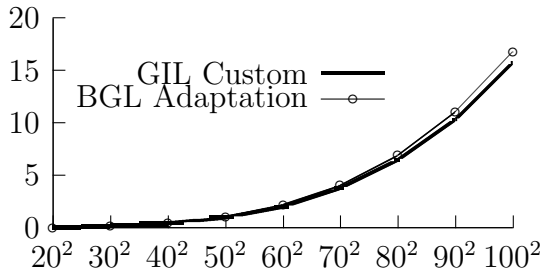
Fig. 18. The timing results for the function `segmentation`. Results for the Intel architecture are shown in (a) and for the PowerPC are shown in (b). The $x$-axis is in seconds, and the $y$-axis is the area of the images in pixels. Each chart shows the timing results of executing two test programs, the first written directly for GIL concepts ("GIL Custom") and the second written to use the adaptation between the GIL and BGL ("BGL Adaptation").

Core 2 Duo), 2.2 GHz, with 2 GB of RAM, and iMac G5 (PowerPC G5), 2.1 GHz, with 1 GB of RAM. The reported timings were obtained by executing the test programs ten times, and computing the average of the measured running times. The test sets for flood-fill and segmentation algorithms consist of 50 square images each, from the size of $20 \times 20$ pixels to $1000 \times 1000$ pixels. To make the image size directly determine the size of the problem, we use procedurally generated images where the number of reachable pixels is proportional to the image size. The test images consist of a

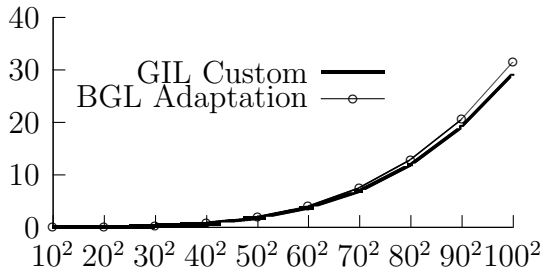
---

GCC Alpha 7 Prerelease), checkout 681

maze with vertical, horizontal, and diagonal lines, as well as corners and dead-ends. Figures 17(a) and 17(b) show the results for flood-fill, and Figures 18(a) and 18(b) the results for segmentation. The test set for the backbone-healing algorithm consists of ten square images, from the size of $10 \times 10$ pixels to $100 \times 100$ pixels. The test images are a topographical representation of a complex terrain with a non-trivial shortest path between the start and target pixels. The length of the optimal path grows linearly with the side lengths of the image, increasing the size of the problem accordingly; Figures 19(a) and 19(b) show the timing results.




(a)




(b)

Fig. 19. The timing results for the function `backbone_healing`. Results for the Intel architecture are shown in (a) and for the PowerPC are shown in (b). The $x$-axis is in seconds, and the $y$-axis is the area of the images in pixels. Each chart shows the timing results of executing two test programs, the first written directly for GIL concepts ("GIL Custom") and the second written to use the adaptation between the GIL and BGL ("BGL Adaptation").

For each algorithm we tested, performance of the two implementations is close to the same. The averaged (over different image sizes) *abstraction penalties*, defined as the ratio of an abstracted implementation over a direct implementation [52, §D.3], due to the adaptation were as follows: for the Intel architecture, flood-fill 0.92, segmentation 0.96, and backbone-healing 1.07; for the PowerPC architecture, flood-fill 1.11, segmentation 1.01, and backbone-healing 1.08. The implementation using cross-domain adaptation thus, in all cases, achieves performance roughly on a par with a hand written GIL algorithm. In two cases the implementation via adaptation was faster. This (close to) zero-overhead adaptation is due to C++'s template compilation model, where specialized code is generated for each different template instantiation. As discussed in Section 2, calls to functions defined in concept maps can be statically resolved, and often inlined, allowing the optimizer to see through adaptation layers.

Our tests compared the performance of algorithms written in terms of different data structures: images and graphs. To minimize noise, we were careful to ensure that the compared algorithms nevertheless employed a common strategy, for example for updating work lists. In a few cases, the control structure of the code differs because the direct implementation can take advantage of properties specific to images. Factors such as the use of auxiliary data structures, differences in cache locality, and the success of the compiler's inliner, all have an impact on the final observed performance. The experiments suggest that the composition mechanism itself incurs no significant penalties; other factors have a larger impact on performance. Furthermore, a generic algorithm in a widely used software library can be expected to be well tuned and tested; reusing such an algorithm even via a complex adaptation layer retains these benefits.

D. Multi-layer composition

In this section we extend the cross-domain composition presented in Section 2 to a *multilayer cross-domain composition*. This extension is *multilayered* because we add an adaptation layer from the domain of graphs to the domain of linear algebra, augmenting the adaptation layer from the domain of images to the domain of graphs. We show how by using the multilayer cross-domain composition of images-to-graphs-to-matrices we can directly express the architecture of an important class of image analysis algorithms.

The use of concepts for the construction of multilayer adaptations has many benefits over other methods. For instance, consider the construction of a multilayer library composition using object oriented techniques. Such a multilayer composition would use a hierarchy of interfaces and class adapters and wrappers, commonly integrated into a framework [71]. Such object-oriented systems suffer from problems such as hierarchy-hardening, fragmentation, and inflexible data representations [71]. A particular problem is that the client of the library composition must explicitly wrap their classes, and thus also objects, to make the classes compatible with the various layers. This wrapping must occur because object construction is hard-wired into the data-structures and algorithms and cannot be changed by the client. While such wrapping may be possible for some layers, other layers may not be accessible to the client. This means that some library compositions are impossible without exposing the internal representation of the adaptation to the client.

A solution utilizing concepts circumvents many of the problems associated with the traditional implementation of multilayered compositions. The use of concepts means that class and object wrapping are unnecessary due to the presence of retroactive modeling. Data representations are not hardwired to the algorithms or data-

structures of the library as concepts are defined for classes of types and not an individual type. Finally, multiple layers of adaptation can be effected transparently—the crux of this chapter.

The motivating example for this section comes from the class of *spectral clustering* algorithms, which are a class of popular, high-quality clustering algorithms [68]. In particular we implement the `ncuts` algorithm for bipartitioning images as described by Shi and Malik [88]. Spectral clustering algorithms were first developed for clustering and partitioning data represented by graphs, but have since been applied to image analysis. The purpose of such algorithms is to identify a partition of a data-set where each datum in a subset is more related to the other data in its subset than to the data in the rest of the data-set. Algorithms like `ncuts` create a bipartition of the data and not a general partition of the data into many parts. Should the data need to be partitioned into more than two parts the `ncuts` algorithm can be repeatedly run on the individual partitions of the data. It is a property of the `ncuts` algorithm that secondary cuts will always occur on strict subsets of the remaining edges in the graph representation of the data [88].

The Shi and Malik `ncuts` algorithm is an especially important example in the class of spectral clustering algorithms [68].[5] The `ncuts` objective gives results that tend to greatly outperform older clustering methods such as $k$-means clustering, or radial-basis classification [68]. Clustering can be used for image segmentation, which is an important step for the analysis of raster image data, for example in computer vision.

The algorithm described by Shi and Malik relies on a multilayer composition of images-to-graphs-to-matrices. A number of spectral clustering methods rely on

[5] As of May 11$^{\text{th}}$, 2009, the website citeseerx.ist.psu.edu reports 1061 citations of Shi and Malik paper [88] not including self-references.

the multilayer composition described by Shi and Malik, and thus it is applicable to a many image processing and analysis methods [68]. This means that once the multilayer composition has been implemented the composition can be reused for many different algorithms.

The goal of this section is to directly express the multilayer composition described by Shi and Malik as a library using the concept and concept map features of C++0x. Without transparent, retroactive adaptation either the client of the `ncuts` algorithm has to translate their image data-structures into compatible matrix data-structures, or the programmer of the `ncuts` algorithm has to translate the `ncuts` algorithm to work with image data-structures—and not matrix data-structures. In the first case the client of the `ncuts` algorithm must understand the particular details of the composition described by Shi and Malik, rather than just using the `ncuts` algorithm, in order to correctly transform their image data-structure into the needed matrix data-structure. Alternately, the library writer must become a domain expert in images, graphs, and matrices so that the `ncuts` algorithm can be translated from the domain of linear algebra to the domain of graph theory to the domain of image processing algorithms, preserving the meaning of the `ncuts` algorithm. Instead, with concepts, the composition from images-to-graphs-to-matrices is encapsulated in a library, using the architecture and compositions as specified by Shi and Malik. The result is that the `ncuts` algorithm is implemented using matrix concepts, the client of the `ncuts` algorithm passes an image data-structure to the `ncuts` algorithm, and the adaptation occurs transparently, automatically, and correctly.

The multilayer composition as described by Shi and Malik consists of three layers which we show schematically, on the right, in Figure 20. The top layer are the set of raster images. Images are defined to be graphs, where each pixel in an image is a vertex in the graph. Edges are added between vertices in the graph to represent some

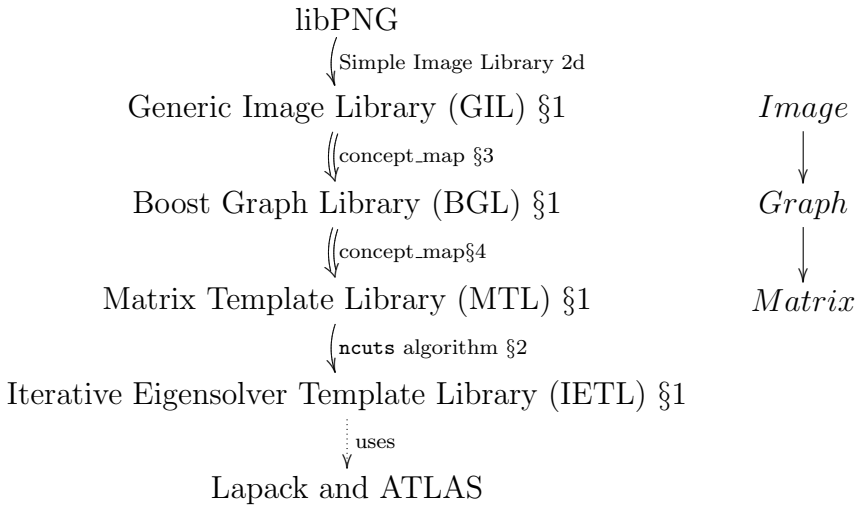



Fig. 20. A high-level view of the GIL-BGL-MTL multilayer composition. This composition is used in the implementation of the Shi and Malik `ncuts` algorithm. The single arrow from libPNG to GIL is the wrapping of the image data by a class. The double arrows from GIL to BGL and BGL to MTL represent the composition of library interfaces. The single arrow from MTL to the IETL is the use of the MTL concepts which the inputs of the `ncuts` algorithm must satisfy; in addition, the `ncuts` algorithm uses the IETL, which in turn uses the Lapack and ATLAS libraries.

relationship between pixels in the image, i.e., neighboring pixels, or similar intensity values, etc. Thus, the composition described by Shi and Malik matches closely to our definition of the image–graph composition, as described in Section 2. The second layer describes a graph as a matrix. The matrix is defined as an adjacency matrix [89], where there is an element in the matrix for each pair of vertices in the graph. The value of each element in the matrix is the label of the edge connecting the two vertices associated with that element; for Shi and Malik this label represents the degree of similarity between two vertices.

For this section we implement a subset of the Generic Image Library's (GIL) concepts [10], a subset of the Boost Graph Library's (BGL) concepts [89], and a subset of the Matrix Template Library's (MTL) concepts [90]. We implemented the

`ncuts` algorithm with the Iterative Eigensolver Template Library (IETL) [104], which requires the matrices used as inputs to its components be compatible both with the MTL, and the C-bindings for the Lapack [4] and ATLAS [116] FORTRAN back-ends. The particular concepts used from the GIL and BGL differ from those concepts used in Section 2: we must use the `VertexAndEdgeListGraph` concept instead of the `IncidenceGraph` concept; however, techniques and code written for that adaptation are used in the multilayer composition.

On the left in Figure 20 are the particular libraries we used or implemented for this section. At the top, we use the open source C-library *libPNG* [86] to read a raster bit-map image from disk. The data, a character buffer, is lightly wrapped by a class called `S2DImg::image`. This class satisfies the syntactic and semantic requirements of one of the GIL's raster image concepts. We provide a concept map to define a modeling relationship from the GIL's image concept to a graph concept in BGL and from the BGL's graph concept a matrix concept in MTL. The MTL matrix concept is used in the requires-clause of the `ncuts` algorithm, as the IETL requires this matrix concept for its algorithms. Finally, the IETL requires FORTRAN-compatible matrices, as it uses the Lapack and ATLAS back-ends.

### 1. Background of the MTL

In this multilayer composition we use the Matrix Template Library 2 (MTL) [90]. MTL is a high-performance C++ linear algebra library which utilizes domain specific notation, expression templates, and adaptation to low-level C and FORTRAN libraries. Other examples of libraries in this domain are Blitz ++ [108], Boost $\mu$BLAS [113], and MTL4 [35], etc. MTL represents matrices as two-dimensional containers — rows of elements or columns of elements — with two-dimensional indices to access elements in the matrices. The MTL supports a large number of different kinds of matrices:

upper, lower, diagonal, banded, sparse, dense, square, etc. The MTL has not been ported to ConceptC++. For this experiment we implement only the subset of the MTL needed for the multilayer cross-domain composition.

We also use the Iterative Eigensolver Template Library (IETL) [104], which consists of a number of algorithms and components useful for finding eigenvectors and eigenvalues of matrices. The IETL algorithms accept MTL-compatible matrices as input. The Shi and Malik algorithm uses the Lanczos method for iterative eigensolving; we note that the IETL only supports the Lanczos method through the use of a combination of the ATLAS [116] and Lapack [4] libraries [104]. Due to Lapack and ATLAS, certain memory layout requirements are imposed on the container which represent the matrices. Satisfying the memory layout requirement requires copying the sparse matrix data-structure which is the input to the `ncuts` algorithm into a Lapack- and ATLAS- compatible data-structure. This copying occurs within the `ncuts` algorithm so no further work is needed from the adaptation.

### 2. Implementation of the `ncuts` algorithm

The core of the `ncuts` algorithm is a formula, called an objective, which describes a minimization problem over the sum of the weight-labels of the edges representing a graph-cut. This minimization problem uses a metric of similarity measures between the two partitions described by the graph-cut. The metric is normalized, i.e., the measure of similarity of each part of the data-set is divided by a measure of similarity of the whole data-set. This normalization of the metric guarantees the measure of similarity of each datum in the partition is greater to the partition that it belongs to, than to the other partition in the bipartition of the graph. Finding such a normalized cut is an NP-complete problem [88], but can be approximated in polynomial time with an iterative eigensolver; Shi and Malik use the Lanczos algorithm in their paper.

Our implementation of the `ncuts` algorithm follows the description in the paper. That is, we implement the multilayer composition architecture described by Shi and Malik, and use their measure of similarity — a product of the normalized similarity of intensity between two pixels and the normalized distance used in calculating a neighborhood for each pixel. The neighborhood of pixels for a given pixel `p` is defined to be every pixel `q` within some chosen distance of `p`, and where the pixel `q` is also within the boundary of the image. We try to keep various other parameters as similar as possible. We differ from Shi and Malik's implementation in that we express the multilayer composition directly, using concepts, rather than translating the data from an image data-structure, to a graph data-structure, to a matrix data-structure.

Details of the measure of similarity, the derivation of the algebraic representation of the `ncuts` algorithm, and proof that the `ncuts` objective is NP-complete can be found in the article by Shi and Malik [88].

## 3. GIL–BGL composition for MTL

In this section we describe the modeling relationship used for the GIL-to-BGL adaptation in the multilayered composition which mimics the Shi and Malik architecture. In Section 2 we saw that for any given pixel locator the neighboring pixel locators were the four cardinal neighbors; the multilayered composition cannot make this same assumption. For the normalized cuts algorithm we must generalize the modeling relationship from images to graphs: one where a pixel locator is related to any pixel that is within some radius set by the user of the algorithm. This means that a pair of types representing an image and a radius will model the graph concept.

For this composition we use the a standard `tuple<...>` type containing a pointer to the image and a value representing the radius. The image type must model the GIL concept `ImageView`, and the radius's type (this is a dynamic value in this composition)

must model the associated `difference_type` of the `ImageView` concept. The first value of the tuple is a pointer so that the image is not copied into the tuple, since this is a potentially expensive operation.

A concept map is then defined from the tuple of a pointer to image type bound by the `ImageView` concept and a radius value to the concept `VertexListGraph`, as shown in Figure 21. Lines 4–9 define the associated types for the concept `VertexListGraph` in terms of helper types instantiated with associated types of the concept `ImageView`. The `edge_descriptor` is the same as the definition of the `edge_descriptor` in Section 2. The `out_edge_iterator` associated type is defined on line 7; it uses a new helper type instantiated with the `vertex_descriptor` associated type (the `difference_type` of the image). This version of the type `out_edge_iterator` iterates over all of the 'safe' neighbors of the current vertex. By 'safe' neighbors we mean all those neighbors within the window defined by the value radius (the second value of the tuple) that are also within the bounds of the image. The `vertex_iterator` is defined on line 9 and iterates over all of the pixel locations within the image.

The function `out_edges` is defined on line 10: it makes a call to another function called `out_edges` that builds a neighborhood of legal pixel locations for `v` based on the size of the neighborhood radius value `radius_m` and the bounds of the image. The functions `target` and `source` are similar to the functions defined in Section 2. The function `vertices`, defined on lines 26–29, builds a pair of iterators which traverse every pixel locator in the image. The function `num_vertices` is defined as a call to the `size` of the image.

## 4. BGL–MTL composition

In this section we describe the adaptation from a BGL graph concept to an MTL matrix concept. The basic idea is to represent a graph as an adjacency matrix:

```
template <GIL::ImageView View>
concept_map VertexListGraph<std::tuple<View*,
    GIL::ImageView<View>::difference_type>> {
  typedef typename GIL::ImageView<View>::difference_type vertex_descriptor;
  typedef std::pair<vertex_descriptor, vertex_descriptor> edge_descriptor;
  typedef ViewToVertexList::out_edge_iterator_adaptor<vertex_descriptor>
    out_edge_iterator;
  typedef ViewToVertexList::vertex_iterator_adaptor<vertex_descriptor>
    vertex_iterator;
  inline std::pair<out_edge_iterator, out_edge_iterator>
    out_edges (const vertex_descriptor& v,
      const std::tuple<View*, vertex_descriptor>& g) {
    return ViewToVertexList::out_edges(v, g); }
  inline vertex_descriptor
    source (const edge_descriptor& e,
      const std::tuple<View*, vertex_descriptor>&) {
    return e.first; }
  inline vertex_descriptor
    target (const edge_descriptor& e,
      const std::tuple<View*, vertex_descriptor>&) {
    return e.second; }
  inline std::size_t out_degree (const vertex_descriptor& v,
          const std::tuple<View*, vertex_descriptor>& g) {
    std::pair<out_edge_iterator, out_edge_iterator> oedges = out_edges(v, g);
    return utl::distance(oedges.first, oedges.second); }
  inline std::pair<vertex_iterator, vertex_iterator>
    vertices (const std::tuple<View*, vertex_descriptor>& g) {
    // create 0-vector, size, and last-position: Zero, dims, end_pos
    return std::make_pair(vertex_iterator(Zero, dims), vertex_iterator(end_pos)); }
  inline std::size_t
  num_vertices (const std::tuple<View*, vertex_descriptor>& g) {
    return size(*g.image_m); } }
```

Fig. 21. Neighborhood concept map. The concept map from the tuple of a pointer to an image type and radius to the concept `VertexListGraph`, where the image type is bounded by the GIL concept `ImageView` and the radius is bounded by the associated type `difference_type` of the concept `ImageView`. The definition of the associated types of the `VertexListGraph` relies on helper types instantiated with some of the associated types of the concept `ImageView`. The definition of the signature of of the `VertexListGraph` is similar to the concept map shown in Section 2.

an element in the matrix is the label between any two vertices in the graph, and the matrix is indexed by the vertices of the graph. Similar to the GIL-to-BGL adaptation we will define a modeling relationship from a tuple of a graph and two helper types to a matrix concept. The helper types must both model the BGL's `PropertyMap` concept, which represents a store of information accessed by keys, returning arbitrary values. Default—and automatically supplied—implementations for these property maps can be described, allowing the user to compose graphs to matrices without any intervention. The decision to use the general modeling relationship is done at the point of usage of the image with the `ncuts` algorithm.

BGL graphs do not directly represent properties (labels) such as edge-weights; instead, labels are supplied by an external entity called a *property map*. A property map takes a key, for example, an edge-descriptor, and returns a value, e.g., the weight of the edge. Likewise, we define a scalar index for each vertex in the graph with another property map. This property map takes a vertex-descriptor as a key and returns an integer so that if there are $N$ vertices, each vertex is given a unique integral value in the range $[0, N)$. Thus, the value at the $i^{th}$ by $j^{th}$ index of the matrix $M$ modeled by a graph is given by $M_{ij} = w(u, v)$ where the scalar index of the vertex $u$ is $i$, the scalar index of the vertex $v$ is $j$, and $w$ is the weight function for edges in the graph. In sum, this means that an adaptation from graphs to matrices is implemented as a tuple of a graph, an edge-weight property map, and an index property map to an adjacency matrix.

We show the overall architecture of the concept map from the GIL `ImageView` and its associated property maps for edge-weight and vertex-index to the MTL concept `SparseMatrix` in Figure 22. On the left is the composition from an image to a graph. This composition is defined through a modeling relationship from pair of a types (shown with dotted arrows) to the BGL concept `VertexListGraph` (shown with a

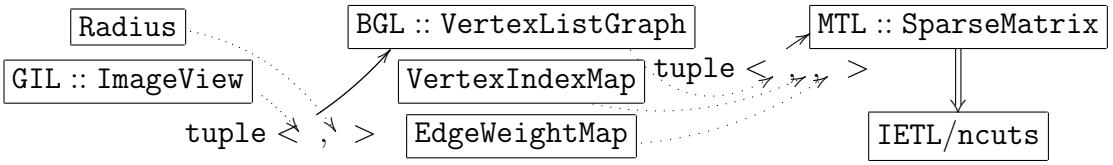



Fig. 22. A schematic of the implementation of Shi and Malik's image analysis architecture. On the left is GIL's `ImageView` concept and a run-time value representing the radius of the neighborhood, chosen by the user; a tuple of these two types (shown with dotted arrows) model BGL's `VertexListGraph` concept (shown with a solid arrow). A tuple of BGL's `VertexListGraph` with property maps storing the values of edge-weights and integral indexes of the vertices of the graph (shown with dotted-arrows) satisfy the requirements of MTL's `SparseMatrix` concept (shown with a solid arrow). The double arrow represents the use of the `SparseMatrix` concept as a requirement for the `ncuts` algorithm; the `ncuts` algorithm is implemented using functionality provided by IETL.

solid arrow); the first element of the pair is a pointer to a type that satisfies the requirements of the GIL `ImageView` concept, the second element of the pair is the type of a value for the `Radius` — this type must satisfy the requirements of the associated type `difference_type` of the `ImageView` concept. The composition from a graph to a matrix is shown on the right side. The composition is defined through a modeling relationship from a tuple of a graph and two property maps (shown with dotted arrows) to the MTL concept `SparseMatrix` (shown with a solid arrow). This is shown by the boxes in the cartoon `BGL::VertexListGraph` for the graph, and the boxes `EdgeWeightMap` and `VertexIndexMap` for the edge-weight and vertex-index property maps, respectively. The `SparseMatrix` concept is used within the `ncuts` algorithm, the requirement for the function shown by the double-lined arrow, which performs a set of matrix-matrix operations, such as multiplication, subtraction etc. The result of these matrix-matrix operations is a new, dense, ATLAS and Lapack-compatible matrix which is passed to the Lanczos iterative eigensolver through the

IETL interface.

All matrix concepts in the MTL require the capability to iterate over all of the rows (columns) of a matrix and all the elements within those rows (columns). This requirement on matrices translates to a requirement on graphs; in particular, since a vertex will be used to get an index into a row or column of a matrix, the graph must support efficient—but not necessarily random-access—iteration over the graph's vertices. The BGL graph concept `VertexListGraph` provides the capability of iteration over all the vertices in the graph [89]. However, a graph that satisfies the `VertexListGraph` does not necessarily contain every possible pair of vertices (a graph with such a capability is called an `AdjacencyMatrixGraph`). Since the `VertexListGraph` represents a matrix with mostly 0 values for the entries of the matrix we define a vertex list graph to be a model of the `SparseMatrix` concept. Thus, we adapt the BGL's `VertexListGraph` concept to the MTL's `SparseMatrix` concept. The adaptation (the concept map) is parametrized over additional types, i.e., the necessary property maps for the modeling relationship from vertex-descriptor to integral scalar, and out-edge to label-weight.

The concept for the MTL's `SparseMatrix` is shown in Figure 23. The definition of the `SparseMatrix` concept is somewhat awkward because ConceptG++ does not support concept maps which retroactively add member functions for the type being modeled: updates to this compiler and proposed extensions would allow a more elegant implementation. Briefly, lines 2–4 declare the associated types for iterators for rows, columns, and elements, respectively. Access to the iterator ranges of rows, columns, and elements are through the functions `rows`, `columns`, and `elements`. (Access to an iterator range for a sequence of elements requires either a row or column iterator, see lines 10 or 12.) The matrix operations that the MTL supports require that the element type of the matrix support the arithmetic operations like addition,

```
auto concept SparseMatrix<typename M> {
  std::ForwardIterator row_iterator = M::row_iterator;
  std::ForwardIterator column_iterator = M::row_iterator;
  std::ForwardIterator elt_iterator = M::elt_iterator;
  std::ArithmeticLike value_type = M::value_type;
  std::pair<row_iterator, row_iterator> rows (const M&);
  std::pair<column_iterator, column_iterator> columns (const M&);
  std::size_t num_rows (const M&); std::size_t num_columns (const M&);
  std::pair<elt_iterator, elt_iterator>
    elements (const row_iterator&, const M&);
  std::pair<elt_iterator, elt_iterator>
    elements (const column_iterator&, const M&);
  std::size_t get_index (const row_iterator&, const M&);
  std::size_t get_index (const column_iterator&, const M&);
  std::size_t get_index (const elt_iterator&, const M&);
  value_type get_value (const elt_iterator&, const M&);
}
```

Fig. 23. The concept for the MTL's `SparseMatrix`.

subtraction, division; we enforce this capability by adding a requirement for the concept `ArithmeticLike` for the element type of the matrix [90]. The various `get_index` functions return the index of the given row, column, or some element within a row or column. The function `get_value` returns the value of a particular element.

The concept map from the BGL `VertexListGraph` and the two property maps to the MTL's `SparseMatrix` is both long and trivial. The tuple needed for the modeling relationship is defined using references (pointers) to the graph and the two property maps. We show an example of the definition of a function required by the MTL `SparseMatrix` concept, in Figure 24. The implementation of the function `num_columns` is a single line: somewhat dwarfed by the many `SameType` constraints on the function.

The signature for the function `ncuts` that implements the *ncuts* algorithm is shown in Figure 25, along with pseudo-code describing the implementation of the

```
... // other signature definitions
template <BGL::VertexListGraph G,
          utl::ReadPropertyMap EWM,
          utl::ReadPropertyMap IDX>
requires std::SameType<G::vertex_descriptor, IDX::key_type>
  && std::SameType<IDX::value_type, std::size_t>
  && std::SameType<G::edge_descriptor, EWM::key_type>
  && std::SameType<EWM::value_type, double>
std::size_t num_columns (const std::tuple<G*, EWM*, IDX*>& gm) {
  return num_vertices(std::get<1>(*gm));
}
```

Fig. 24. Definition of the function `num_columns`. One of the definitions of the signatures that allow the BGL's `VertexListGraph` concept to satisfy the MTL's `SparseMatrix` concept. This figure shows the implementation of the signatures for `num_columns` function. Because the MTL `SparseMatrix` concept is `auto`, no explicit `concept_map` needs to be defined — the only requirement is that the functions exist in the correct scope.

algorithm. We note that the `ncuts` function is implemented entirely in terms of functions and algorithms defined by the MTL for sparse matrices. The final statement of the `ncuts` function is a call to the function `ncutsW` which computes the second eigenvector (the graph-cut, see [88]) of the matrix `System`, which represents the system of equations being solved.

The code fragment in Figure 26 shows a PNG file being opened and the *ncuts* algorithm used on the image. It is important to note that no translation of the PNG is performed — the image is passed to the function `ncuts` which is expecting an MTL `SparseMatrix`.

Note in the code in Figure 26 the construction of the tuple on line 3 which binds the image to the window radius value, which satisfies the more general graph modeling relationship required by `ncuts`. similarly the construction of the tuple on line 5 binds the `neighborhood` proxy to an edge-weight map `ewm` and a vertex-index map `idx` for

```
template <MTL::Matrix Mtx>
std::vector<Vector> ncuts (Mtx const& mtx) {
  // acquire associated types
  // find the size of the matrix, and build the problem
  const std::size_t N = num_rows(mtx);
  Matrix D(N, N), W(N, N), D_isqrt(N, N), D_m_W, System(N, N);
  // build the weighting matrix W
  // build the matrices D, and the inverse−square−root of D
  // build D − W
  // solve and put solution into System
  return ncutsW(System); // return 2nd eigenvector
}
```

Fig. 25. The `ncuts` function. The signature of the `ncuts` function, and an outline of its body, defined using the MTL. The requirement for this function is that the parameter `Mtx` satisfy the concept `MTL::Matrix`. The helper-function `ncutsW` is defined entirely in terms of C-Lapack data-structures so that the C-Lapack back-end of the Lanczos iterative eigensolver can be used.

```
S2DImg::image img(argv[1]);
// define edge−weight property map ewm and the vertex−index property map idx
auto graph = make_tuple(&img, 2);
std::vector<...> solutions = ncuts(
    make_tuple(&graph, &ewm, &idx));
```

Fig. 26. Code fragment showing a call to the `ncuts` function.

the modeling relationship from graphs to matrices. We note that the two uses of the auxiliary types at two positions within the adaptation stack are optional, and that the binding of values only occurs at the entry point; and, in particular, the rest of the composition occurs transparently.

## 5. Results of the BGL to GIL to MTL adaptation

Figure 27 shows the result of running the *ncuts* algorithm. The table shows that the algorithm correctly partitions the data.

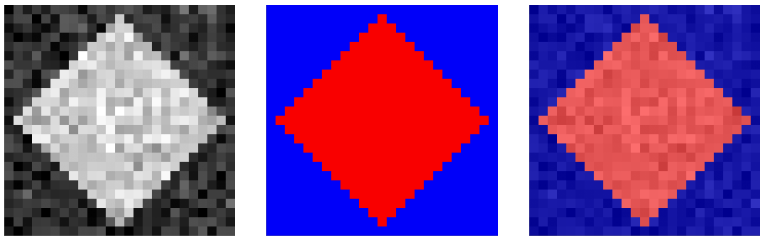


Figure (a) Figure (b) Figure (c)

Fig. 27. Segmentation results for `ncuts`. A $25 \times 25$ PNG image of a white diamond in a black field is shown in Figure (a). The image allows 8-bits of gray-scale, i.e., white is defined as the value 255 and black as the value 0. Gaussian noise was added to the whole image with $\mu = 50$ channels for the mean and $\sigma = 25$ channels for the standard deviation. The *ncuts* algorithm was run and the $2^{nd}$ eigenvector's values were split into vertices that are 'positive' and vertices that are 'negative' to create the graph-cut. All positive vertices went into one partition and all negative vertices into the other partition. This is shown graphically in Figure (b). A composite of Figures (a) and (b) is shown in Figure (c), demonstrating that the *ncuts* algorithm segmented the image into the diamond and the surrounding field.

## E. Concept maps and other adaptation mechanisms

This section relates the adaptation capabilities offered by ConceptC++'s concepts and concept maps to those of several other mainstream languages, including C++ 2003.[6]

Concept maps are a non-intrusive mechanism for adapting operations on a type (or collection of types) to model a concept. That is, a given collection of operations and associated types might offer the essential functionality required by an interface (concept), but these operations might not have the required names or signatures, and some associated types might not exist with the correct names. Such a collection can be made conform to the new interface using a concept map. No changes to the

[6]This analysis is adapted, with permission, from the paper by Järvi, Marcus, et al. [56].

original operations or types are needed.

Concept maps adapt types rather than individual objects. They do not offer direct facilities to store state. All state is maintained in the objects whose types concept maps adapt. New types and operations may need to be introduced before the concept map adaptation facilities come into play, if a collection of types does not provide the essential functionality required to model a concept. We saw this in Section C, where it was necessary to introduce a new class for traversing sets of pixels in an image, in order to satisfy the requirement for an associated type modeling edge iterators. Once all of the necessary functionality has been implemented with appropriate types and functions, concept maps can be applied in their intended role as non-intrusive identity-preserving adapters.

The template system of C++ 2003, and ConceptC++, is based on instantiating templates with full type information at compile time, allowing all functions defined in concept maps to be statically resolved, possibly inlined, and further optimized. Several concept map adapters can be layered without the adaptation mechanism causing performance degradation. The downside is that all template instantiations to be used in a program must be known at compile time. While this may be acceptable in domains like graph algorithms or linear algebra — indeed, generic C++ 2003 template libraries have found widespread use in these domains — more "dynamic" domains, such as GUIs, necessitate run-time polymorphism. The combination of concepts and run-time polymorphism is explored in the papers by Marcus, et al. [70, 55, 56].

In object-oriented languages, libraries publish their interfaces as abstract classes. (Here, this term covers the "interface" language construct found in some languages.) To satisfy the requirements of an interface then means to define a class that inherits from a particular abstract base class. This achieves run-time polymorphism but, in mainstream object-oriented languages, the subclass relation is established at the

time of defining a class, which makes inheritance a relatively rigid mechanism for library composition. A class cannot be made a subclass of another class retroactively without altering its definition. Variations of structural subtyping have been proposed as cures for problems of rigid class hierarchies [6, 62] but have not found wide use. Outside of mainstream object-oriented languages, Cecil [67] lets subtyping be defined externally to class definition. This feature was found beneficial for adapting existing types for use with generic libraries in a comparative study of programming languages' suitability for generic programming [31]. Aspect-oriented programming systems can be used to modify classes retroactively, independently of their original definitions, e.g., to implement new interfaces using "static crosscutting" [60].

The *adapter pattern* [30] is widely used to work around problems of rigid class hierarchies when composing libraries. Adapters can be divided into object and class adapters. Both kinds of adapters inherit an abstract base class that defines the desired interface. Object adapters store the adaptee as a member (as a reference to a distinct object), whereas class adapters inherit from the adaptee, storing both the adapter and adaptee as a single object. The problems of library composition and adaptation in object-oriented programming are widely studied and recognized. For example, if there is a need to adapt a class with new functionality, but neither the definition of that class nor code that is hard-wired to use that class can be changed, an adapter is not an adequate solution (see, e.g., [71, 103]). Class adapters suffer from *hierarchy hardening* and object adapters from inconsistency problems caused by breaking the state of a single entity into multiple objects [50, 20].

We demonstrate the techniques that allow us to express large, sophisticated architectures, as language constructs in Section C. Essential in our idioms is that we avoid the use of an abstract base class to describe the library interface. Instead, the library interface is specified in terms of concepts. As concept maps are entirely

external to both the types they adapt from and the concepts they adapt to, the problems of object-oriented adapters are avoided. It is possible to use concept maps to adapt client code to and from the abstract base class interface, which is provided for the sole purpose of run-time polymorphism [56]. The constructions to achieve this are somewhat involved, see [70], but can be hidden behind simple abstractions. The benefit is that the choice of whether to use run-time polymorphism is deferred to the time when the components are composed, rather than dictated by the library. Run-time dispatching may incur a performance penalty, which is thus avoided in the cases where run-time polymorphism is not needed. If we ignore, for the moment, concept refinement relationships, some ability to test whether operations are present (via down-casting), and performance optimizations (see [56, §4.3]), the central design feature of the `poly<>` template presented in the paper by Järvi, et al. [56] is that it behaves as a type constructor for defining existential types [63], where the hidden type is constrained to be a model of a particular concept. In this regard, `poly<>` is similar to Haskell's "`forall`" construct, which allows the definition of types with a hidden part constrained to be a type belonging to a given type class, or classes.

Concepts are in many ways similar to Haskell type classes [112], and concept maps to Haskell's instance declarations. A Haskell type class defines the signatures of the functions that instances (models) of the type class must implement. *Instance declarations* establish that a type, or a sequence of types in the case of multi-parameter type classes, belong to a particular type class. Analogous to concept maps, instance declarations are non-intrusive: external to both the definitions of the types and the definition of the type class. Lämmel and Ostermann collect formulations of problems reported in the object-oriented integration mechanisms [61], and demonstrate how type classes are effective solutions to many of them. Essential in evading the problems is the non-intrusive adaptation with instance declarations. Our experiences with non-

intrusiveness of concept maps support this view.

In their standard form (in Haskell98 [112, 80]), type classes have a few obvious restrictions, which have largely been remedied in non-standard but common extensions. First, standard type classes only accept one parameter. Multi-parameter type classes, however, are widely supported by Haskell compilers and interpreters. Second, standard type classes do not support associated types. They can, however, be emulated to an extent with functional dependencies [59], a well-established extension, or expressed directly using more recent extensions [15, 14].

There are also less obvious differences between concepts and type classes, some of which affect adaptation and library composition. We explain those differences, but refrain from a comparative evaluation, as we have not produced Haskell implementations of any of the library composition and adaptation scenarios described in this chapter. Garcia et al. compare the suitability of different mainstream languages for generic programming [31].

Haskell can infer the type class constraints of polymorphic functions automatically, while ConceptC++ does not support the analogous "concept inference." To ensure that the constraints of a generic function can be uniquely determined, Haskell requires that an overloaded function name (when called without module qualification) is declared in exactly one type class. When composing independently developed libraries, it is possible that the same function name is accidentally used in two type classes in different modules. Figure 28 translates the classic example of accidental conformance [69] to ConceptC++ and to Haskell. The Haskell version is erroneous and is fixed by qualifying the calls to `draw` and `shoot` with the module prefix as `Cowboy.draw` and `Cowboy.shoot`; the ConceptC++ version is inevitably valid because ConceptC++ requires a disambiguating annotation, the "`Cowboy C`" constraint, even if there are no conflicting concepts.

```
data Canvas = ...

class Rectangle r where
  draw :: r → Canvas → Canvas
  move :: r → Int → Int → r

class Cowboy c where
  draw::c → c
  move::c → Int → Int → c
  shoot::c → c

drawShoot = shoot . draw
```

```
concept Rectangle<typename R> {
  void draw(R r, Canvas& w);
  void move(R& r, int x, int y);
}
concept Cowboy<typename C> {
  void draw(C& c);
  void move(C& c, int x, int y);
  void shoot(C& c);
}
template <Cowboy C> void draw_shoot(C& c) {
  draw(c); shoot(c);
}
```

Fig. 28. Accidental use of the same function name in two different type classes (left column) and in two different concepts (right column).

An instance declaration in Haskell is in effect in all functions in which the declaration is visible. A concept map, however, is only in effect in a context where a type is constrained with the corresponding concept. The example in Figure 29 illustrates. The multiplication operator (∗) for integers is given different semantics in the two concept maps. The first concept map retains the multiplication operator's original meaning, the second maps the operator to perform addition. Neither mapping has an effect outside generic functions. One or the other of the mappings, neither of them, or both can be in effect within a particular generic function, depending on the func-

tions constraints. In our slightly contrived example function, both meanings apply. The fact that concept maps define views that are only active when requested is a desirable trait for adaptation and library composition. However, this may prove to be confusing as well, as it creates a rift between generic and non-generic functions.

As an incremental addition to an evolving C++ language, concept maps and concept-based overloading must co-exist, and sometimes compete, with the existing overloading mechanism of C++ 2003. This results in tension between the needs of traditional ad-hoc function template overloading and the desire to treat concrete types and operations as implementation details, and only expose required functionality through concept maps. In Section F we illustrate some insidious failure cases that can arise from this competition.

The Scala programming language [79] provides external adaptation with rather different mechanics, *implicit parameters*, but with an outcome that is close to adaptation using type classes or concepts. An implicit parameter to a method can be left out in a call to the method. The Scala compiler attempts to find a unique best matching value for that parameter in the call's context. A fairly faithful emulation of type classes is possible with implicit parameters that represent dictionaries of functions [78]. Furthermore, Scala *views* utilize implicit parameters to non-intrusively define implicit conversions between types, which seems promising for implementing cross-domain compositions like we discussed in Section C.

C++ 2003 allows the definition of efficient non-intrusive adaptation layers. As an example, we mentioned BGL's transparent adapters for LEDA graphs in Section B. Breuer et al. [11] report on a cross-domain library composition between the domains of linear algebra and graph theory. They adapt several concepts from the Parallel Boost Graph Library [41] to concepts found in the Iterative Eigensolver Template Library [104]. Their implementation is in C++ 2003, and uses overloading and template

```
concept Monoid <typename C, typename Tag> {
  C operator*(C, C);
  C identity(); }
class additive {}; class multiplicative {};
concept_map Monoid<int, multiplicative> {
  int identity() { return 1; } }
concept_map Monoid<int, additive> {
  int operator*(int a, int b) { return a + b; }
  int identity() { return 0; } }

template <InputIterator It, InputIterator It2, typename U>
  requires SameType<It::value_type, It2::value_type>,
  requires Monoid<It::value_type, multiplicative>,
  requires Monoid<U, additive>, Assignable<U>,
  requires Convertible<It::value_type, U>
U inner_product (It i1, It i1e, It2 i2, U init) {
  for (; i1 != i1e; ++i1, ++i2)
    init = init * ((*i1) * (*i2));
  return init; }
int main() {
  vector<int> v; v.push_back(3); v.push_back(5);
  cout << inner_product(v.begin(), v.end(), v.begin(), 100); }
```

Fig. 29. Comparison of concept-maps and type-classes. Concept maps are only in effect in contexts constrained by the corresponding concept. In the `inner_product` function, the multiplication between `*i1` and `*i2` comes from `Monoid<U, additive>`, and is therefore integer addition as defined by the concept map `Monoid<int, additive>`. The multiplication between `init` and the result of the "additive" element-wise multiplication comes from `Monoid<It::value_type, multiplicative>`, and is thus integer multiplications as defined by the concept map `Monoid<int, multiplicative>`. The `Assignable`, `Convertible`, and `InputIterator` concepts come from ConceptGCC's implementation of the proposal by Gregor et al. [42]. The `inner_product` function computes the inner product of two sequences, accumulating to an initial seed value `init`. When executed, the program outputs `134`.

specialization to achieve the necessary adaptation, not `concept` and `concept_map` constructs of ConceptC++. Non-intrusive adaptation in C++ 2003 relies on a host of tricky template techniques, such as *traits classes* [77] and conditional overloading using the *enable_if* template [57]. Though C++ 2003 can support complex non-intrusive adaptation, the resulting code is brittle; ConceptC++ offers improved support for non-intrusive adaptation.

Retroactive adaptation mechanisms certainly predate both Haskell's type-traits and ConceptC++'s concepts feature. Of particular interest to this dissertation is the *category* construct of OpenAxiom [25] variant Spad programming language [58] (part of the family of AXIOM computer algebra systems). Categories provide many of the features of concepts and type-classes; they are closer to concepts, i.e., OpenAxiom does not support "concept inference," and types are unambiguously annotated with the appropriate category. More powerful mechanisms are currently being added to OpenAxiom to support retroactive adaptation, i.e., the *assume* facility for categories.[7] The facility effectively retroactively binds a *category* to a domain (the specification of a type) separately from specification of the domain (or category). The assume facility also provides reflection capabilities; i.e., given that the operator `*` is *assumed* to be the signature called `BinaryOperator` it is possible to request the `NeutralElement`, which presumably returns the distinguished element constructor `1`. This means that categories using the assume feature differ from both type-traits and concepts by allowing the user to directly operate on the signatures of categories.

[7]The information here comes from personal correspondence with Dr. Gabriel Dos Reis and Yue Li, who are developing this facility

## F. Adaptation and overloading

We have demonstrated many benefits that mechanisms like concepts and concept maps provide when composing software components. There are, however, some stumbling blocks. In this section we illustrate some of the difficulties that arise when using these language features.

Non-intrusive adaptation is possible in C++ 2003 as well. For example, the BGL provides a `vector_as_graph` adapter, which adapts all instances of `vector<list<T>>` to satisfy the requirements in their `IncidenceGraph` table. Adaptation is accomplished by overloading the function templates required by `IncidenceGraph` for its model `vector<list<T>>`. In ConceptC++, the analogous adaptation is accomplished with a concept map

```
template <class T>
concept_map IncidenceGraph<vector<list<T>>>;
```

C++'s ad-hoc polymorphism defines a partial ordering amongst a set of overloaded functions based on specialization ordering of type patterns. In a nutshell, a function is selected from a set of overloads when it is "at least as specialized as" all other candidates and no other candidate is at least as specialized as it. This relationship considers a type pattern A to be at least as specialized as another type pattern B when A is substitutable for B. ConceptC++ defines an analogous specialization order between concept constraints [54]. For example, one can expect the `size` function below to match and be applied to all `IncidenceGraph`s, calculating the total number of edges in a graph, including calls with arguments of type `vector<list<T>>`, as shown:

```
template <IncidenceGraph G> int size(const G&);
...
vector<list<int>> g;
```

```
int total_num_edges = size(g);
```

When executed, the above code will use graph operations to calculate and return the total number of edges in the graph.

For the purposes of determining the partial ordering of constrained function templates, concept constraints, however, are subordinate to type patterns—constraints are only considered in case a "tie-breaker" is needed. For example, the type pattern `vector<T>` is considered strictly more specialized than, say, the `IncidenceGraph<T>` constraint.

Later in the software's life-cycle, an other overloaded `size` function may be introduced, for example, to return the size of an arbitrary vector:

```
template <typename A>
int size(const vector<A>& a) { return a.size(); }
```

Now, when we invoke `size` on a `vector<list<T>>` this new overload is considered to be the unique best-matching candidate since the type pattern in the overload defined for `IncidenceGraph<T>` is no more specialized than a plain type variable. That is, the `size(vector<T>)` overload has silently hijacked the function call to `size(IncidenceGraph<T>)`. The code will compile but will erroneously only return the number of `lists` in the `vector` representing the graph.

Type patterns are the primary overloading criteria in several other mainstream languages. For example, in C# and Java, overloading is based exclusively on type patterns: constraints on type parameters must be satisfied but they do not affect specialization ordering. Overloading rules in various languages support the notion that the most specific knowledge prevails. ConceptC++ offers two mechanisms for specialization, type patterns and constraints, and by subordinating one mechanism to the other we compromise this notion. A change to the function template partial

ordering rules could reinstate the principle that the most specific type knowledge is used for dispatch. In particular, programs with silent failures like the one in the example above could be rejected, if for the purpose of function overload partial ordering, concept map specialization patterns were considered at the same time as function type pattern specializations, rather than as tie-breakers. These issues are currently under consideration in the C++ standards committee.

The unrestricted non-intrusive adaptation allowed by current languages leads to conflicting adaptation layers, and even with the above modification to function overloading rules, surprises and ambiguities seem still possible. A situation akin to the one we demonstrated in C++ arises with Haskell's specialization ordering amongst instance declarations, with language extensions that allow definition of "overlapping instance declarations" [80, §3.7].

Non-intrusive adaptation helps to avoid much of the pre-planning and coordination that is necessary in the use of software libraries when component adaptation is intrusive. For this flexibility, one needs not give up efficiency: our work demonstrates that non-intrusive adaptation and efficiency are not mutually exclusive. The possibility of accidentally adapting the same component to a given interface in multiple ways exists, but can be controlled with language designs that support detecting conflicts and offer means for resolving them; further programming language research in this area is required.

## G. Conclusions

This chapter reports on programming with "concepts", a forthcoming set of potential new features of C++, and explains their use, benefits, and costs. In particular, concepts offer powerful mechanisms for adapting data types to specific library interfaces—we

provide an analysis of this aspect of C++ concepts, and a description of their use in complex cases of non-intrusive library composition. We demonstrate that transparent adaptation of data structures to multiple library interfaces is possible and straightforward. We conclude from our performance evaluations that such adaptations impose minimal penalties. As we show in Section C, such adaptations enable cross-domain composition of libraries. These adaptations allow us to reduce one domain to another domain, with a number of benefits that accrue. For example, algorithms and data-structures can be implemented in the domain that best (and most naturally) fits their expression and, even if those domains are different, the algorithms and data-structures can still be composed transparently to the user.

The main benefits of ConceptC++'s adaptation mechanisms are non-intrusiveness (a type can be adapted to one or more interfaces without altering the definition of the type), flexibility (instead of single data types, a generic family of data types, or classes of data types described using concepts, can be adapted with a single adapter), and performance (adaptation is implemented using small functions whose addresses are statically resolved, and are thus inlineable and optimizeable).

The generic programming paradigm, introduced into C++ via the STL, supports the non-intrusive design of efficient families of algorithms specified in terms of common abstractions. There is little language support, however, for rigorously specifying these abstractions. C++ is now evolving to raise such specifications from the level of naming conventions to compiler checkable artifacts. C++0x comes with language support for concepts, and the standard library has been re-specified to take advantage of this, while retaining backward compatibility. The backward compatibility requirement, considered essential for the evolutionary acceptance of C++0x, has had an impact on the concepts in the library, but the effort is nevertheless a significant step forward. The guidance for programming with concepts, as well as the libraries

and adaptation idioms presented in this chapter, unburdened by legacy compatibility concerns, are further advances towards leveraging the generic programming approach in constructing reusable software libraries. The utility of these idioms will further increase once the concepts language feature becomes widely available: we expect to see algorithm families designed from the ground up with this support in mind, and to see generic programming emerge as a central tool in the design of modern reusable libraries.

# CHAPTER IV

## ALGORITHMIC DIFFERENTIATION IN AXIOM

This chapter describes the design and implementation of a framework for *algorithmic differentiation* in the Axiom (OpenAxiom) computer algebra system [25]. Algorithmic differentiation (AD) is a method for computing the "derivative of a program" — augmenting a program to also compute its derivative [45, 94]. This AD framework is implemented as a sequence of transformations on the typed abstract syntax tree of a Spad program. (Spad is the library extension language for Axiom [58].) The framework illustrates an algebraic theory of algorithmic differentiation: if it is possible to define a compositional semantics for programs, we can define the requirements for when those programs can be algorithmically differentiated. By leveraging the generic programming paradigm our AD framework is not limited to programs that compute with built-in types like floating point numbers or integers, but can operate on any program that computes over, for instance, polynomials, or any type that satisfies certain algebraic requirements (algorithmically differentiable ring). Our algorithmic differentiation framework is a library-directed transformation tool — it does not rely on any modification of OpenAxiom's Spad compiler.

This chapter includes a specification for the core of the Spad language. This specification includes a formal grammar, an operational semantics of relevant parts of that grammar, and a sketch of the denotational semantics for the composition of programs in core Spad with respect to algorithmic differentiation. The grammar specifies the legal constructs that our framework can translate. The operational semantics specifies the meaning of the Spad language using standard tools of the theory of programming languages. The denotational semantics allows us to the discuss the algebraic properties of the composition of statements, and thus properties of

programs. We show several examples drawn from the literature of the theory of algorithmic differentiation. These examples show that our framework supports the basic features of algorithmic differentiation found in the literature, in addition to allowing the algorithmic differentiation of programs outside the scope of other tools.

## A. Introduction

*Algorithmic differentiation* (AD) is a technique for computing derivatives of a computer program avoiding both symbolic differentiation and divided difference approximations [45]. AD tools usually consist of set of transformations which augment the code with instructions for computing the value of a derivative. Compared to computing divided differences or symbolic differentiation, AD has several benefits, detailed below.

Divided-differences suffer from numerical accuracy problems when using fixed-precision numbers. The computation of the difference quotient has a form such as:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2 \times h}.$$

In this formula as $h$ becomes smaller the accuracy of the approximation increases; of course, this assumes an unlimited number of bits of accuracy. With fixed-precision arithmetic the accuracy of the division and subtraction operations decrease the accuracy of the computation when $h$ becomes small enough. Thus, counterintuitively, after a certain point, the approximation of the derivative becomes worse as $h$ becomes smaller. Even more frustrating is that the point where the accuracy no longer improves is actually a function of both the value of $h$ and the value of $f(x)$. (Sensitivity and accuracy analysis can be performed using symbolic differentiation or an AD tool to determine an ideal $h$ to alleviate this problem.) AD tools do not have any inher-

ent effect on numerical accuracy; in fact, the numerical accuracy of the derivative is proportional to the numerical accuracy of the unaugmented code [45].

Symbolic differentiation can suffer from the 'expression explosion' problem [45], where the size of the symbolic expression grows very large. Lifting directly from Griewank's "Evaluating Derivatives", we show Speelspenning's example [45, 96]:

$$f(x) = \prod_{i=1}^{n} x_i := x_1 \times x_2 \times \cdots \times x_n$$

whose gradient has the form of:

$$\nabla f(x) = \begin{pmatrix} x_2 \times x_3 \times \cdots \times x_i \times x_{i+1} \times \cdots \times x_{n-1} \times x_n \\ x_1 \times x_3 \times \cdots \times x_i \times x_{i+1} \times \cdots \times x_{n-1} \times x_n \\ \vdots \\ x_1 \times x_2 \times \cdots \times x_{i-1} \times x_{i+1} \times \cdots \times x_{n-1} \times x_n \\ \vdots \\ x_1 \times x_2 \times \cdots \times x_i \times x_{i+1} \times \cdots \times x_{n-2} \times x_n \\ x_1 \times x_2 \times \cdots \times x_i \times x_{i+1} \times \cdots \times x_{n-2} \times x_{n-1} \end{pmatrix}$$

This resultant symbolic derivative is neither informative, particularly memory efficient, nor fits nicely onto the printed page — a particularly keen observation by Griewank. AD tools have only a linear increase in the number of instructions, where the increase is sensitive either to the number of arguments of the function (forward-mode) or the number of returned values of the function (reverse-mode) [45].

We now demonstrate an example of AD with the Tchebychev polynomial of the first kind:

$$T_n(x) = \cosh(n \operatorname{arccosh}(x)).$$

This polynomial can be defined in the OpenAxiom computer algebra system, discussed in Section B, in (at least) two ways: directly entered into OpenAxiom from the command line as

```
Tchebychev(x,n) == cosh(n*acosh(x))
```

or by defining a recursive program, using the recurrence relation:

```
Tchebychev(x: Integer, n: Integer): Integer ==
  if n=0 then return 1
  if n=1 then return x
  2*x*Tchebychev(x,n-1) - Tchebychev(x,n-2) -- return value
```

Clearly, this function is outside the range of conventional symbolic differentiation tools. An AD tool, however, can handle this function without trouble.

Several software packages exist for algorithmic differentiation [76, 111, 46, 9, 8], some extending a compiler [76], some implemented as a pre-processor [8], and some purely as a software library [46]. Typically these packages operate on computations on concrete data types, such as the built-in floating point types. In particular, such software are difficult to use with generic programming techniques — techniques that typically do not know actual data representations in advance — but instead state abstract requirements on the types they work with.

The chapter discusses an algebraic differentiation system which is not limited to concrete data types. Instead, the tool can compute the algorithmic derivative of a program as long as the types of the program satisfy suitable requirements, outlined later. We have developed the beginnings of a theory for algorithmic differentiation to specify these requirements. We present the theory in the context of the OpenAxiom's library extension language, but suggest that the ideas are more general. Our algorithmic differentiation framework, based on the presented theory, is a library-directed transformation tool of Spad programs.

In this chapter we use tools from the theory of programming languages' semantics to formalize the meaning of algorithmic differentiation in a generic programming context. This is in the spirit of the work on generic programming by Davenport et al. [22], where languages features should be harnessed to expose the elegance of the underlying mathematical theory, not simply implement a computation. Finally we observe that our framework is designed as a library *without* modification of the OpenAxiom system.

## B. The OpenAxiom system

OpenAxiom is a strongly-typed general-purpose computational platform, supporting both numeric and symbolic computations. It uses type information, at compile time, to guide selection and application of operations — a process which supports generic programming. For example, given the recursive definition of the function `Tchebychev` from the previous section, the Axiom compiler generates direct calls to the multiplication and subtraction operations over `Integer` values for the fragment

```
2*x*Tchebychev(x,n-1) - Tchebychev(x,n-2)
```

as opposed to inspecting the values held by the variables `x`, `n`, and the values returned by the function `Tchebychev` at run-time and resolving the function to call. If the type specified for the variable `x` were unknown, or if `x` had a type for which no suitable operation named `*` could be found, then the expression would be rejected by the type checker and no code would be generated. This design aspect of OpenAxiom contrasts remarkably with most general purpose computer algebra systems which rely on dynamic, run-time type-checking. This aspect allows the use of techniques from active libraries and generic programming.

OpenAxiom users can extend the system with libraries implemented in Spad, a

general purpose programming language with affinity towards mathematical software construction and scientific computations. More information about the OpenAxiom system and its use can be found in "the Axiom Book" [58, 21], the seminal papers of J. Davenport and collaborators [23, 22], and on the OpenAxiom web site [25].

## C. Spad programming language

This section describes the (abstract) syntax and the semantics of the Spad programming language. For the purposes of this chapter, the intuitive behavior of a Spad program is described by a structural operational semantics, i.e., a semantics described in terms of the syntactic forms of Spad. In this chapter, we relate the behavior of the structural operational semantics to the function a Spad program computes through standard denotational semantics, and the meaning of the "derived program" to that of the input program. This methodology shapes the final form of the library-directed transformation AD tool.

### 1. Syntax

This section presents the syntax and semantics of a reasonably large subset of Spad, sufficient to demonstrate the concepts and capabilities of our framework. The language is an extension of the explicitly typed lambda calculus that contains elements of dependent types. Spad also contains mechanisms which make it a sutiable language for supporting generic programming and active libraries, notably: the category mechanism (similar to C++0x's *concepts*), and conditional categories and domains — which provide support for algorithm specialization. Figure 30 summarizes the abstract syntax of the Spad subset presented here.

*Module* A Spad module is a sequence of toplevel definitions.

| | | | |
|---|---|---|---|
| *Module* | M | ::= | $\Delta^+$ |
| *ToplevelDef* | $\Delta$ | ::= | C \| D \| P \| $d$ |
| *CategoryDef* | C | ::= | $\phi$ `:` `Category` `==` E |
| *Exports* | E | ::= | W \| X $\mathrm{W}^?$ |
| *Extension* | X | ::= | $\tau$ \| `Join`$(\tau^+)$ |
| *WithExpr* | W | ::= | `with` $\mathrm{S}^+$ |
| *Signature* | S | ::= | $x$ `:` $\tau$ |
| *Type* | $\tau$ | ::= | `Boolean` \| `Integer` \| `Float` \| `Record`$(\mathrm{S}^+)$ \| `Union`$(\mathrm{S}^+)$ |
| | | \| | $\tau_1 \times \cdots \times \tau_n \to \tau_0$ \| $\gamma$ \| $\gamma\,([\tau\|e]^+)$ |
| *DomainDef* | D | ::= | $\phi$ `:` E `==` K |
| *PackageDef* | P | ::= | $\phi$ `:` E `==` K |
| *Capsule* | K | ::= | `add` $\mathrm{R}^?$ $d^+$ |
| *Representation* | R | ::= | `Rep` `:=` $\tau$ |
| *Definition* | $d$ | ::= | [`n` \| $\phi$] `:` $\tau$ `==` $s$ |
| *CallForm* | $\phi$ | ::= | $x(\mathrm{S}^+)$ |
| *Statement* | $s$ | ::= | $e$ \| `if` $e$ `then` $s$ \| `if` $e$ `then` $s$ `else` s \| $i^+$ `repeat` $s$ |
| | | \| | S `==` $e$ \| S `:=` $e$ |
| *Iterator* | $i$ | ::= | `for` $n$ `in` $e$ [`suchthat` $p$]$^?$ \| `while` $p$ |
| *Expression* | $e, p$ | ::= | $c$ \| $x^\tau$ \| $e(e^+)$ \| $e.x^\tau$ \| $e$ `case` $\tau$ \| $e$ `:=` $e$ \| $e$ `where` $d^+$ |
| *Variable* | $x^\tau$ | | |
| *Constant* | $c^\tau$ | | |
| *Identifier* | $x$ | | |
| *CDPName* | $\gamma$ | | name of a Spad category, domain, or package |

Fig. 30. Abstract syntax of the Spad language. The notation $Z^?$ represents an optional $Z$, $Z^+$ a non-empty finite sequence of $Z$; the square brackets are used for grouping.

*ToplevelDef* A toplevel definition is either a Spad category definition, or a Spad domain definition, or a Spad package definition, or a delayed definition.

*CategoryDef* A Spad category definition specifies a class of algebras, by declaring the signatures of the required operations. A Spad category definition may extend existing Spad categories with new signatures. For example, the fragment

Monoid(): **Category** == **with**
  $*$: (%, %) $\rightarrow$ %
  1: %

declares `Monoid` as a Spad category with two signatures:

1. the symbol `*` is a binary operation on the domain belong to this category;
2. the identifier `1` denotes a constant object of domain belonging to the category being defined.

The following category definition

Group(): **Category** == Monoid **with**
  inverse: % $\rightarrow$ %

extends `Monoid` with the `inverse` operation, to capture the mathematical notion of group structure. One can think of Spad categories as *specifying* views on objects.

*Exports* Definitions for Spad categories, Spad domains, and packages specify exported operations, i.e., the "public interface" in programming languages jargon, through either a *WithExpr*, or an *Extension*, or combination of both.

*WithExpr* A *WithExpr* is essentially an unnamed Spad category consisting of a list of operation signatures (*Signature*).

*Extension* Definitions for Spad categories and Spad domains may extend existing Spad categories or domains. An extension may specify either a type, or multiple categories through the `Join` operator. The latter form corresponds to multiple inheritance in object-oriented programming languages.

*Signature* The specification of a type for an identifier can appear in a *WithExpr*, as a parameter declaration in *CallForm*, as the field of a record or union, or in a local variable definition.

*Type* A type is a built-in type (`Boolean`, `Integer`, `Float`), a record or union, a function type, the name of a Spad category or Spad domain, or an instantiation of a Spad category or Spad domain. All field names specified by signatures in a record must be distinct. Similarly, all field names specified in a union must be distinct and unique in the enclosing scope; this applies recursively to any other union types directly referenced in the signature list of the union.

*DomainDef* A Spad domain definition provides implementations for views specified by categories. A domain definition has an interface specification part (*Exports*) stating the categories and possible additional signatures it implements, and an implementation part called *capsule*. The implementation part may define the representation of the object belonging to the domain, and provide definitions for operations declared in its *Exports*. For example, the program fragment

```
IntMonoid(): Monoid == add
  Rep == Integer
  (x:%) * (y:%) == (rep x + rep y)$Integer
  1:% == 0::Integer
```

provides an implementation `IntMonoid` for the `Monoid` specification as follows:

- the object representation domain is `Integer`;
- "multiplication" of two objects in `IntMonoid` is the value obtained by adding their respective underlying values (returned by the `+` operator);
- the `Integer` constant `0` is the underlying value of the unit of `IntMonoid`.

Note that a Spad domain almost always references the "current domain" using the symbol `%`.

*PackageDef* A package definition provides implementations for functions that operate on a Spad domain. Unlike a Spad domain, a package does not define a *Representation* and does not reference the symbol `%`. Like a Spad domain, it has an *Exports* part and an implementation part.

*Capsule* The implementation part of a Spad domain or package is its capsule. A capsule may specify the representation of a domain (if it is the implementation of a domain), and specifies a sequence of toplevel definitions for operations on the Spad domain objects, or the operators in a package.

*Representation* A Spad domain specifies the underlying representation of its objects by assigning a type expression to the identifier `Rep`. A *Representation* can occur only in a Spad domain definition.

*Definition* A (delayed) definition is the binding of an identifier or a function call expression to a Spad category, Spad domain, or an ordinary function. The body of the definition is evaluated when needed. That evaluation may happen only once for a given argument list. Even though the evaluation is delayed, the body is still fully type checked at the definition point. The Spad language, as understood by the Spad compiler, does not allow ordinary function definitions at toplevel. However, they are the core of the language understood by the interpreter. For uniformity, we include toplevel function definitions in the Spad subset we describe.

*CallForm* A call form consists of an identifier and a parenthesized sequence of signatures declaring formal parameters. A call form is needed in the definitions of a Spad category, Spad domain, and function.

*Statement* Statements appear in the body of function definitions. A statement is either an expression, a one or two-arm if-statement, an iteration where the body of the iteration (a statement) is controlled by a list of iterators, a local variable definition, or a an assignment.

*Iterator* An iterator is either a sequence of items $x$ drawn from a sequence $e$, possibly filtered by a predicate $p$, or a repeated evaluation of a predicate.

*Expression* An expression is either a constant, variable, function call, member selection, type-case expression, an assignment, or a qualified expression. A qualified expression is an expression that contains free variables and is immediately followed by their definitions in a `where`-clause. We assimilate expressions built with built-in operations — such as addition on integers, etc. — as function calls.

*Variable* A variable is the use of a name declared with a given type.

*Constant* A constant is a built-in value, such as $+^{\text{Integer}\times\text{Integer}\to\text{Integer}}$, $342^{\text{Integer}}$, $\text{true}^{\text{Boolean}}$, etc.

*Identifier* An identifier is a finite sequence of characters. The set of identifiers in Spad is countably infinite.

## 2. Language features

The Spad programming language supports elements of dependent types, a result of the functorial nature of the data-structuring mechanisms available in Spad. That is, the Spad type system allows types to be function-like objects with arguments that depend on types and values. Dependent types enables an unusually direct style of implementation of mathematical structures. Aspects of this will be explored later in this dissertation V.

The Spad programming language also supports general function overloading; in particular, a function can be overloaded on its argument and return types. The overload resolution algorithm exploits all context information, including arguments and target types, to select the best matching function. Implicit conversion is supported through the `coerce` operator.

## 3. Semantics

The computational rules used to evaluate Spad programs are those of *eager* semantics (and call-by-value), and the arguments of functions are passed by reference. We sketch the semantics of Spad programs in two ways: small-step operational semantics, and denotational semantics. The small-step operational semantics gives an intuitive idea of the behavior of Spad programs, whereas the denotational semantics lets us associate mathematical functions to Spad programs. The latter allows us to formally talk about the notion of a derivative of a Spad program.

### a. Operational semantics

The Spad language is imperative in the sense that its programs operate on stores by explicit modification. Values of Spad programs can be booleans, integers, floating

point numbers, aggregates thereof, or function codes. We denote the collection of values by **Value**, inductively defined as:

| | |
|---|---|
| Location values: | object locations are in **Value** |
| Boolean values: | `true` $\in$ **Value** and `false` $\in$ **Value** |
| Integer values: | integer constants $n^{\text{Integer}}$ are in **Value** |
| Float values: | float constants $f^{\text{Float}}$ are in **Value** |
| Functions: | If $f$ is a defined function of type $\tau_1 \rightarrow \tau_2$ then the constant $f^{\tau_1 \rightarrow \tau_2}$ is in **Value** |
| Aggregates: | if $c_i^{\tau_i}$ are values of type $\tau_i$ in **Value**, then the tuple $(c_1^{\tau_1}, \ldots, c_n^{\tau_n})^{\tau_1 \times \cdots \times \tau_n}$ is in **Value**. Tuples represent record values. Similarly, if $c^{\tau_1}$ is in **Value**, so is $c^{\tau_2 \leftarrow \tau_1}$. It represents a value of a field of type $\tau_1$ in a union $\tau_2$. |

The behavior of a Spad program is a sequence of configurations $\langle p, \sigma, \Gamma \rangle$ where $p$ denotes fragments of Spad constructs, $\sigma$ the store of values, and $\Gamma$ the current environment of bindings of variables to types and expressions. The notation $\Gamma, x^{\tau} == e$ denotes an environment obtained by extending $\Gamma$ with the binding $x^{\tau} == e$. The $== e$ part may be missing. A store $\sigma$ is a mapping from memory locations to Spad values. We use the notation $\sigma[v/l]$ to designate an *updated function* defined by

$$\sigma[v/l]\,(x) = \begin{cases} v & \text{if} \quad x = l \\ \sigma\,(x) & \text{otherwise} \end{cases}.$$

Each configuration is defined by structural induction on the syntax of Spad, as specified in Figure 31.

b. Denotational semantics

The basic idea of algorithmic differentiation rests on the notion that a computer program computes a function whose range has a ring structure; and the collection of such functions can be endowed with a *differential algebra* structure. The theory of *denotational semantics* [98] is a useful tool in laying down the necessary theoretical

$$\textsc{Variable}\quad \langle x^\tau, \sigma, \Gamma\rangle \longrightarrow \langle \sigma\,(x^\tau), \sigma, \Gamma\rangle$$

$$\textsc{Call-Arguments}\quad \frac{\langle e_i, \sigma, \Gamma\rangle \longrightarrow \langle e_i', \sigma', \Gamma\rangle}{\langle e_0\,(v_1, \ldots, e_i, \ldots, e_n), \sigma, \Gamma\rangle \longrightarrow \langle e_0\,(v_1, \ldots, e_i', \ldots, e_n), \sigma', \Gamma\rangle}\ \text{where the } x_i \text{ are parameters of } v_0.$$

$$\textsc{Call-Operator}\quad \frac{\langle e_0, \sigma, \Gamma\rangle \longrightarrow \langle e_0, \sigma', \Gamma\rangle}{\langle e_0\,(v_1, \ldots, v_n), \sigma, \Gamma\rangle \longrightarrow \langle e_0'\,(v_1, \ldots, v_n), \sigma', \Gamma\rangle}\ \text{where the } x_i \text{ are parameters of } v_0.$$

$$\textsc{Call}\quad \langle v_0^{\tau_0}\,(v_1^{\tau_1}, \ldots, v_n^{\tau_n}), \sigma, \Gamma\rangle \longrightarrow \langle v_0^{\tau_0}[v_1^{\tau_1}/x_1, \ldots, v_n^{\tau_n}/x_n], \sigma, \Gamma\rangle\ \text{where the } x_i \text{ are parameters of } v_0.$$

$$\textsc{Qual. expr.}\quad \frac{\begin{array}{c}\langle \delta_1, \sigma, \Gamma_1\rangle \longrightarrow \langle \delta_1', \sigma, \Gamma_2\rangle \qquad \langle \delta_2, \sigma, \Gamma_2\rangle \longrightarrow \langle \delta_2', \sigma, \Gamma_3\rangle \\ \cdots \qquad \langle \delta_n, \sigma, \Gamma_n\rangle \longrightarrow \langle \delta_n', \sigma, \Gamma_{n+1}\rangle \qquad \langle e, \sigma, \Gamma_{n+1}\rangle \longrightarrow \langle e', \sigma', \Gamma_{n+2}\rangle\end{array}}{\langle e \text{ where } \delta_1 \cdots \delta_n, \sigma, \Gamma_1\rangle \longrightarrow \langle e', \sigma', \Gamma_{n+2}\rangle}$$

$$\textsc{Sequence-Head}\quad \frac{\langle s_1, \sigma_1, \Gamma_1\rangle \longrightarrow \langle s_1', \sigma_2, \Gamma_2\rangle}{\langle s_1; s_2, \sigma_1, \Gamma_1\rangle \longrightarrow \langle s_1'; s_2, \sigma_2, \Gamma_2\rangle} \qquad \textsc{Sequence-Tail}\quad \langle v_1; s_2, \sigma, \Gamma\rangle \longrightarrow \langle s_2, \sigma, \Gamma\rangle$$

$$\textsc{If}\quad \frac{\langle e, \sigma, \Gamma\rangle \longrightarrow \langle e', \sigma', \Gamma\rangle}{\langle \text{if } e \text{ then } s_1;\ s_2, \sigma, \Gamma\rangle \longrightarrow \langle \text{if } e' \text{ then } s_1;\ s_2, \sigma', \Gamma\rangle}$$

$$\textsc{If-true}\quad \frac{\langle s_1, \sigma, \Gamma\rangle \longrightarrow \langle s_1', \sigma', \Gamma'\rangle}{\langle \text{if true then } s_1;\ s_2, \sigma, \Gamma\rangle \longrightarrow \langle s_1', \sigma', \Gamma'\rangle}$$

$$\textsc{If-false}\quad \frac{\langle s_2, \sigma, \Gamma\rangle \longrightarrow \langle s_2', \sigma', \Gamma'\rangle}{\langle \text{if false then } s_1;\ s_2, \sigma, \Gamma\rangle \longrightarrow \langle s_2', \sigma', \Gamma'\rangle} \qquad \textsc{Assignment-Left}\quad \frac{\langle e_1, \sigma_0, \Gamma\rangle \longrightarrow \langle e_1', \sigma_1, \Gamma\rangle}{\langle e_1{:=}e_2, \sigma, \Gamma\rangle \longrightarrow \langle e_1'{:=}e_2, \sigma_1, \Gamma\rangle}$$

$$\textsc{Assignment-Right}\quad \frac{\langle e_2, \sigma, \Gamma\rangle \longrightarrow \langle e_2', \sigma', \Gamma\rangle}{\langle l{:=}e_2, \sigma, \Gamma\rangle \longrightarrow \langle l{:=}e_2', \sigma', \Gamma\rangle} \qquad \textsc{Assignment}\quad \frac{}{\langle l{:=}v^\tau, \sigma, \Gamma\rangle \longrightarrow \langle v^\tau, \sigma[v^\tau/l], \Gamma\rangle}$$

$$\textsc{Imm. def.}\quad \frac{\langle e, \sigma, \Gamma\rangle \longrightarrow \langle e', \sigma_1, \Gamma\rangle}{\langle x{:}\tau{:=}e, \sigma, \Gamma\rangle \longrightarrow \langle x{:}\tau{:=}e', \sigma_1, \Gamma\rangle} \qquad \textsc{Imm. def.}\quad \langle x{:}\tau{:=}v^\tau, \sigma, \Gamma\rangle \longrightarrow \langle v^\tau, \sigma, \Gamma, x^\tau == v^\tau\rangle$$

Fig. 31. Evaluation rules of Spad statements.

framework for meaningful discussion of computing the derivative of computer programs. We seek for a standard denotational semantics $[\![\bullet]\!]$ of the Spad programming language, that respects the operational semantics outlined in §a, i.e.,

$$t \to^* v^\tau \quad \Rightarrow \quad [\![t]\!] = [\![v^\tau]\!] \,.$$

D. Elements of algebraic theory of algorithmic differentiation

The meaning of transforming a program $P$ to another program $P'$ so that the function computed by the program $P'$ is the derivative of the function computed by the program $P$ requires a *semantics* function that computes the meaning of a program. Without such a function, there is no way to relate a sequence of *commands* given in Spad to our usual intuition of a function with a well-defined derivative. Given such a semantics function we can describe a sequence of transformations which act the same as the corresponding (mechanical) analytic derivative operations, i.e., the chain rule.

In our framework, we define a program written in the Spad language as a **SPAD**-algebra, where **SPAD** is the functor structuring the Spad language abstract syntax (Figure 30.) Let's call $\mathcal{S}$ the collection of all **SPAD**-algebras — which can be thought of as all the possible abstract syntax trees of programs of the Spad language. We are interested in meaning functions $[\![\bullet]\!] : \mathcal{S} \to \mathcal{D}$ where the semantics domain $\mathcal{D}$ is suitable for talking about derivatives.

1. Algorithmic differential rings

An *algorithmic differentiation* of Spad program is a transformation $\Phi : \mathcal{S} \to \mathcal{S}$ such that

- the function $[\![\Phi\,(-)]\!] : \mathcal{S} \to \mathcal{D}$ respects composition, i.e.,

$$[\![\Phi\,(\mathrm{P1};\mathrm{P2})]\!] = [\![\Phi\mathrm{P2}]\!] \circ [\![\Phi\mathrm{P1}]\!]$$

  where we have used `;` to indicate *sequencing*, i.e., the action of executing program `P1` first, followed by the execution of the program `P2`. This key functorial property embodies the usual *chain rule* from calculus.

- we let $U$ be the differential ring, an algebra. The differential ring adds an operator $\delta$ to a ring, with carrier set $U$. If two programs $P_1 \in \mathcal{S}$ and $P_2 \in \mathcal{S}$ have meanings

$$[\![P_1]\!] = f : T \to U_\perp \quad [\![P_2]\!] = g : T \to U_\perp$$

  where $U_\perp$ is a strict extension of a differential ring $(U, \delta)$, then the following identities hold

$$\delta\,([\![\Phi P_1]\!] + [\![\Phi P_2]\!]) = \delta\,([\![\Phi P_1]\!]) + \delta\,([\![\Phi P_2]\!])$$
$$\delta\,([\![\Phi P_1]\!] \cdot [\![\Phi P_2]\!]) = \delta\,([\![\Phi P_1]\!]) \cdot [\![P_2]\!] + [\![P_1]\!] \cdot \delta\,([\![\Phi P_2]\!])\,.$$

  These two identities relate the meaning of the transformed programs to the usual mathematical notion of derivation, namely additivity and Leibniz rule.

Note that if the domains of computation are that of polynomials, or power series, with usual derivation operation then the chain rule holds. In more general domains of computations, however, we add the chain rule as a requirement. That leads us to speak of *algorithmic differential ring*. Consequently, we require that all our domains of computations where we carry differentiation are actually algorithmic differential rings, and not just differential rings. This non-trivial subtlety is not currently expressable in OpenAxiom, and is left as an axiomatic requirement that the user must satisfy in writing a program.

## 2. Strategies of derivative evaluation

Based on the chain rule and associativity of function composition, one can develop various strategies for evaluating the derivative. For example, given the program

P1 ; .... ; Pn

the meaning of its algorithmic differentiation transform is

$$[\![\Phi\,(\mathrm{P1};...;\mathrm{Pn})]\!] = [\![\Phi\mathrm{Pn}]\!] \circ \cdots \circ [\![\Phi\mathrm{P1}]\!]$$

which can be evaluated using various computation strategies. One approach is a "naive" reading of the composition from right to left, leading to so-called *forward mode* where the first instruction is transformed, then the second, etc., and the derivatives are propagated forward. This approach is simple to comprehend and implement. However, it has the inconvenience that it generates computations with complexity expressed in terms of the number of (independent) input variables. Therefore it might not be very efficient for computing gradients of scalar functions of many variables.

Another approach is a reading of the compositions from left to right. That strategy requires the computation of derivatives of functions not yet executed. Consequently, its actual implementation requires running the original programs first, then 'reverting' the sequencing of computations to propagate the derivatives generated, thus leading to so-called *reverse mode*. This computation strategy has the property that its complexity is in terms of the number of (dependent) output variables. It is therefore a good candidate for computing the gradient of a scalar function of many variables. These two strategies of computation are "extreme" in some sense, and an AD tool may actually use a mix of association of compositions, a class of transformations which are called *hybrid mode*.

## 3. Control flow and differentiability

The practical realization of algorithmic differentiation of a program `P` builds on two concepts. First, the abstract evaluation [17] of the operational semantics (§a) of `P` yields a program `Q` that is in *simple form*. The simple form relies on the algebraic properties of the code guaranteed by the algorithmic differentiable ring. The simple form is a form of pseudo-SSA: each expression consists of an assignment (or a return statement) where the expression on the right is a single function call whose arguments are variables or constants, i.e., no sub-expressions. The simple form is computed using a recursive algorithm that acts on expressions by removing non-constant and non-variable sub-expressions out of the expression. The simple form more closely matches the intuitive notion of our theory of derivative — and also helps to mitigate potential problems from side-effects. Note that this stage is parameterized by the operational semantics of the programming language being used (Spad, in our case).

From this program `Q` in simple form one extracts the control flow and the data flow graphs. This data flow graph is usually called a *computational graph* [45] of the program `P`. The control flow graph extracted from `Q` is such that each node of the graph is a *basic block*, that is a maximal sequence of instructions without "jumps". Each basic block is therefore a straight line program and, through the denotational semantics §b, defines a differentiable function. However, at the joint points of the control flow graph, there is no guarantee that we obtain a differentiable function, e.g., the case of the absolute value function. Even when the mathematical function being computed is differentiable, it may be that its expression as an algorithm contains transfers of control that introduce artificial anomalies. Consider

```
funnyId(x: Integer): Integer ==
  if x=2 then 2 else x
```

which computes the identity function in a curious way. Its transform is

```
funnyId(x: Jet Integer): Jet Integer ==
  if x.value = 2 then jet(2,0) else x
```

with derivative 0 at $x = 2$, which is clearly unintuitive behavior.[1] It is impossible to detect such contrived constructs — the transformation acts on the program *as is* and can make no guesses as to any 'underlying' meaning. However, the theory of data flow analysis as abstract interpretation [17] provides a framework for studying certain classes of these issues, which we do not investigate further here. The programmer must be aware of these limitations.

## E. The Spad compiler

The OpenAxiom system can operate in interactive, "batch", and "compiled" modes. The interactive mode uses an interpreter, the batch and compiled modes a compiler. The compiler can be invoked from within the interpreter by issuing the system command:

`)compiler`.

The interpreter and the compiler understand slightly different dialects of the Spad language. This is due partly by design, and partly by a turbulent history. The compiler is intended for library development (large scale programming), whereas the interpreter is intended for convenient interactive conversation (small scale programming). Also, the interpreter supports type inference ("guessing"), as opposed to the compiler which, for the most part, restricts itself to type checking. Since our framework is primarily intended for library development, we focus on the compiler

[1] Although, according to the definitions of our operational and denotational semantics, correct. It is unintuitive to the user, only.

component of the OpenAxiom system. We emphasize that our framework works with both the compiled and the interpreted dialects of Spad.

An input Spad source file is decomposed into a stream of tokens by the Spad lexer. The token stream is transformed into a *parse tree* by the Spad parser. That parse tree undergoes further transformation by a post-parser transformer, resulting in a *parse form*. The parse form is still close to Spad source. For example, here are the parse forms of the definitions of `Monoid` and `IntMonoid` examples from §1:

```
(DEF (Monoid) ((Category)) (())
  (CATEGORY (SIGNATURE * (% % %))
    (SIGNATURE (One) (%))))

(DEF (IntMonoid) ((Monoid)) (())
  (CAPSULE (LET Rep (Integer))
    (DEF (* x y) (() % %) (() () ())
      ((elt (Integer) +) (rep x) (rep y)))
      (DEF (One) (%) (())
        (:: (Zero) (Integer)))))
```

The parse form is then transformed into another internal abstract syntax tree, which is used as input to the semantic analyzer. The job of the semantic analyzer is to type check the abstract syntax tree, to resolve dependencies on previously compiled programs and load them if necessary. The output of the semantic analyzer is a fully typed abstract syntax tree, which is translated by the code generator into Lisp code. A copy of the resulting Lisp code is saved on disk for future use, and another copy is given to the run time system (a Lisp system) for evaluation. The current OpenAxiom system uses the Steel Bank Common Lisp (SBCL) implementation. SBCL is capable of compiling Lisp code to native object code, which is subsequently loaded into the running Lisp image. As a result, the OpenAxiom compiler compiles input Spad programs to native object code for execution.

## F. Implementation

Our AD framework, shown schematically in Figure 32, is *entirely* implemented as an OpenAxiom library, meaning that no source code modification to the OpenAxiom compiler is required. The implementation consists of

- a small library interface to the Spad compiler, to retrieve the parse forms and the typed abstract syntax tree over objects of the `SExpression` domain;
- OpenAxiom domains and packages working on the output of the compiler interface.

The library interface for the transformation tool was built to support a large class of library-driven semantic enhancements to Spad. The interface to the compiler was developed as part of the engineering effort from a larger project to rationalize the OpenAxiom system. The interface, itself, is designed to allow sophisticated analysis and transformation of the internal representation of Spad codes, without having to modify the compiler. We have also written an unparser that pretty prints the internal representation as Spad code.

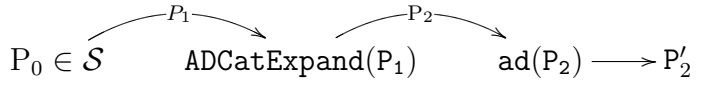



Fig. 32. The pipeline for performing AD. The process starts with a program $\mathrm{P}_0$ which is parsed by the compiler into the representation $\mathtt{P}_1$ and given to the function `expand` from the `ADCatExpand` package to convert $\mathtt{P}_1$ into simple form. The simple-form $\mathtt{P}_2$ is the input to the "`ad`" function which computes the algorithmic derivative $\mathtt{P}_2'$.

At the time of the writing, we support only the forward mode. We do not support direct computation of gradient yet. The gradient can be obtained from repeated calls to compute partial derivative in several directions; however, that is inefficient and

potentially incorrect, because if the function depends on global variables, there is no guarantee that successive calls yield the desired values.

### 1. Transformation to simple form

The main thrust of our transformer is that it requires the input program be in simple form (§3), which simulates the operational semantics outlined in §a. Since it is unrealistic to require users to write codes in that specific form, we have implemented a package called `ADCatExpand` that transforms an arbitrary Spad program into simple form. The expander "walks" the AST, identifying and expanding any expressions that are not in simple form. For example, all the arguments of function calls that are not constants or variables are replaced with temporary variables, such that every operation consists of the name of the operation and references to variables. This transformation does not change the meaning of the input program.

### 2. First order prolongation

Our AD framework uses the theory of jets [27]. A jet can be thought of as a value and sequence of derivatives (whichever derivatives are appropriate). We can construct a jet of a function by lifting a function that computes in values so that it computes in jets. If the input program computes a mathematical function $f : M \to N$ as determined by the standard semantics, then our tool generates a program that computes the first prolongation

$$\mathrm{jet}^1(f) : \mathrm{jet}^1 M \to \mathrm{jet}^1 N.$$

More precisely, the framework defines a domain `Jet` parameterized by a domain from the `Ring` category. For example, `Jet(Integer)` and `Jet(Float)` designate the first order jet of `Integer` and `Float`, respectively.

A subset of the code for implementing the domain `Jet` in our framework is shown below.

```
Jet(T: Ring): Public == Private where
  Public == with
    jet: (T, T) → %
      ++ construct a jet value from a pair
    elt: (%, "value") → T
      ++ retrieve the ''value'' field of a jet
    elt: (%, "delta") → T
      ++ retrieve the ''delta'' field of a jet
    setelt: (%, "value", T) → T
      ++ set the ''value'' field of a jet
    setelt: (%, "delta", T) → T
      ++ set the ''delta'' field of a jet
    -- ...
  Private == add
    Rep == Record(val: T, der: T)
    jet(v, d) == [v, d]
    elt(jet: %, x: "value") == jet.val
    elt(jet: %, x: "delta") == jet.der
    setelt(jet: %, x: "value", jetValue: T) ==
      jet.val := jetValue
    setelt(jet: %, x: "delta", jetDelta: T) ==
      jet.der := jetDelta
    -- ...
```

Note that here `Jet` takes types `T` of the `Ring` category; in reality, `Jet` should take types `T` from the category of algorithmic differential ring as explained in §1. That is a current limitation of our framework, partly due to lack of enough expressivity of the Spad language.

The generated program is defined as an overloaded function on the jets. In particular, we don't use a fancy symbol mirroring the mathematical "functorial notation" $\mathrm{jet}^1(f)$. A benefit from that approach is that the generated program is structurally similar to the input program. A fundamental drawback is that it prevents us from

defining functions on jet spaces which are not prolongations. For example, we would like `Jet(T)` to be a member of the `Ring` category but doing so requires defining operators `*` and `+`, which conflict with operators we may be prolonging when working with `Jet(Integer)` or `Jet(Float)` for example. This is an aspect that we plan to improve on in future work, as it makes perfect sense to define functions on jets (to represent differential equations) that are different from prolongations.

## 3. Initial environment

When our library is loaded into OpenAxiom, it starts with an environment that contains the prolongations of certain operators on built-in types. These operators are defined within the `Jet` domain:

```
Jet(T: Ring): Public == Private where
  Public == with
    -- ...
    _+: (%, %) → %
    _-: (%, %) → %
    _*: (%, %) → %
    _-: % → %
    if T has Field then
      _/: (%, %) → %
    if T has TranscendentalFunctionCategory
       and T has Field then
         log: % → %
    if T has TranscendentalFunctionCategory then
      exp: % → %
      sin: % → %
      cos: % → %
    -- ...

  Private == add
    _+(x: %, y: %): % ==
      jet(x.value + y.value, x.delta + y.delta)
    _-(x: %, y: %): % ==
```

```
    jet(x.value − y.value, x.delta − y.delta)
  _*(x: %, y: %): % ==
    jet(x.value ∗ y.value, _
        x.delta ∗ y.value + x.value ∗ y.delta)
  _−(x: %): % == jet(−x.value, −x.delta)

  if T has field then
    _/(x: %, y: %): % ==
      r : T := x.value / y.value
      jet(r, (x.delta − r ∗ y.delta) / y.value)

  if T has TranscendentalFunctionCategory
     and T has Field then
       log(x: %) ==
         jet(log(x.value), x.delta / x.value)
  −− ...
```

Note how the rules of computation of jets embody both the actual derivatives and the chain rule. The initial environment serves as a basis for the Spad compiler and interpreter for successfully completing the evaluation of derivatives.

## 4. Forward mode

The core of our framework's user interface consists of functions defined in the package `ADCatForward`. There are two versions: the first takes an s-expression (the internal data-structure used to hold the AST) and returns an s-expression that contains the differentiated code. The second takes a `String` (a file-name), and returns a list of s-expressions corresponding to the first prolongation of all domains, categories, packages, etc. found within that file.

The `AD` function is a simple layer on top of a routine that implements a Visitor Pattern [30] over the AST. The `AD` function walks the AST until all derivatives have been generated. We use the category membership assertion to reject invalid input functions, e.g., those with arguments and return types that are not from `Ring`. For

example, since the type `String` is not asserted to belong to `Ring`, we reject a request to compute the derivative of a function that returns a `String`. This is another place where we rely on type annotation for meaningful transformation.

## 5. Examples

### a. The GRADIENT paper

Our first example is the function "`f`" from Monagan and Neuenschwander [76] that we have translated to Spad as:

```
f (x:Float, n: Integer): Float ==
  if n = 0 then return 0
  else
    a : Float := 0
    b : Float := x
    for k in 1..(n−1) repeat
      h : Float := b
      b := log(a+b)
      a := h
    b
```

Monagan and Neuenschwander's example did not include explicit typing for the arguments. Here, we have chosen `Float` to emphasize the point that our framework (and Algorithmic Differentiation in general) does *not* require that function parameters be "symbols".

First, the tool runs `expand` (from `ADCatExpand`) on the program shown above to produce the output shown below. The code as shown is the output of our pretty-printer. The expander uses the Lisp function `GENSYM()` to generate fresh names for temporaries; therefore the names for the temporaries are not necessarily the same each time `expand` is run on the same function.

```
-- intermediate expanded code
f(x:Float,n:Integer):Float==
 if n = Zero() then
      G6327 := Zero()
      return(G6327)
   else
      a : Float := Zero()
      b : Float := x
      h : Float
      for k in (One())..(n − One()) repeat
        h := b
        G6328 := a + b
        b := log(G6328)
        a := h
      b
```

After which the actual AD transformation is performed and the final output program is shown below.

```
f(x: Jet(Float), n: Jet(Integer)): Jet(Float) ==
  if n.value = Zero() then
    G2668 := Zero()
    return(G2668)
  else
    a : Jet(Float) := Zero()
    b : Jet(Float) := x
    for k in (One())..(n.value − One()) repeat
      h : Jet(Float) := b
      G2669 := a + b
      b := log(G2669)
      a := h
    b
```

Note that all operator names have retained their original spelling. This is because the AD transformer has inserted the first prolongation of the operators as an overloaded function defined on the first prolongations of its source and target. This

means that the correct operation is selected based on its argument types in addition to its name.

The above program, despite the syntactic similarity with the original program `f`, does compute numerical values of the derivative of the function computed by `f`. To see that this is the case, we will use OpenAxiom's `Expressions`. Even with its strong emphasis on *numerical* algebraic computations, OpenAxiom also provides the parameterized domain `Expression` for symbolic manipulation. Replacing `Float` with `Expression Integer` in the original function `f` and transforming that function with our AD tool results in the function below. The result, again, is similar to the original function; only type annotations have changed and temporaries have been introduced to hold intermediate values:[2]

```
f(x: Jet(Expression(Integer)), _
  n: Jet(Integer)): Jet(Expression(Integer)) ==
  if n.value= Zero() then
    G2668 := Zero()
    return(G2668)
  else
    a : Jet(Expression(Integer)) := Zero()
    b : Jet(Expression(Integer)) := x
    for k in (One())..(n.value − One()) repeat
      h : Jet(Expression(Integer)) := b
      G2669 := a + b
      b := log(G2669)
      a := h
    b
```

Evaluation of `f(jet(x,1),jet(4,0)).delta` yields

[2]The coincidence of the fresh variable names is an artifact of running the framework from a new instance of the interpreter; the first 2500-or-so fresh variable names are consumed by the setup of the interpreter, itself.

```
                    log(x) + 2x + 1
-------------------------------------------------
             2                              2    2
(x log(x) + x )log(log(x) + x) + x log(x)  + x log(x)
```

which coincides with the derivative of `f(x,4)`:

```
log(log(log(x) + x) + log(x)).
```

b. Tchebychev polynomials

Next, we consider the Tchebychev polynomial defined by recurrence relation as shown in §A. Here, again we considered the original function taking a `Float`, and another taking `Expression Integer` to check the symbolic evaluation. The first prolongation of

```
Tchebychev(x: Expression Integer, n: Integer): Expression Integer ==
  if n=0 then return 1
  if n=1 then return x
  2*x*Tchebychev(x,n-1) - Tchebychev(x,n-2)
```

is computed as:

```
Tchebychev(x: Jet Expression Integer, n: Jet Integer):
    Jet Expression Integer ==
  if n.value = 0 then
    G2716 := 1
    return G2716
  if n.value = 1 then return x
  G2718 := 2 * x
  G2721 := 1
  G2720 := n - G2721
  G2719 := Tchebychev(x,G2720)
  G2717 := G2718 * G2719
  G2723 := n - 2
```

G2722 := Tchebychev(x,G2723)
G2717 − G2722

Evaluating `Tchebychev(jet(x,1), jet(5,0)).delta` yields

```
   4      2
80x  - 60x  + 5
```

which agrees with the derivative of $T_5 = 16x^5 - 20x^3 + 5x$.

## G. Employing generic algorithmic differentiation

In this chapter we have defined a generic library for AD and, also, demonstrated the use of that library. Our methodology requires a language with support for both generic programming and the ability to transform code, preferably at compile-time. This section summarizes what the task of creating an AD library entailed, and what it entails for an application programmer to use an AD library to obtain a code that algorithmically differentiates a function.

Our AD library consists of three main parts: (1) the concept *algorithmic differential ring*, which represents any type that carries the mathematical structure of a *ring*, and allows an operator to be defined that satisfies the algebraic definition of differentiation; (2) a *Jet* type, which is a type that carries the augmenting derivative information; and, (3) an algorithmic differentiation transform, which augments functions to carry derivative information. Our AD transform has three main duties: (1) replacing type annotations with the *Jet* of the type annotation; (2) converting literals to a *Jet*, e.g., the expression `10` becomes `jet(10)` (or the appropriate construct to lift from values to jets-of-values); and (3) projecting all variables used in control flow to be the *value* of that variable, e.g., if `v < 0` is the control-expression of a loop, then the `AD` function must emit an expression such as `v.value < jet(0).value`.

With a generic AD library and an AD transform available, obtaining the code to algorithmically differentiate a function becomes effortless. For example, assuming the programmer wants to compute the derivatives of the function call `f(a, 0)`, the programmer just rewrites the function call as `AD(f(a, 0))`. The result is that both the value and the derivative of the function are computed.

## H. Conclusion

Automatic differentiation is a well-known technique for computing derivatives, and matured software packages exist for applying it in practice. However, we found the theory of algorithmic differentiation incomplete. This chapter explores such a theory. We base our work on top of the OpenAxiom computer algebra system, which allows highly generic programs to be written in terms of "categories," or classes of algebras, instead of concrete data-types. We outline the formal semantics of Spad, the library extension language of OpenAxiom, and define the exact requirements for when Spad programs can be subjected to algorithmic differentiation.

We apply our theory in an implementation of an algorithmic differentiation tool for Spad. The implementation transforms Spad programs, at the level of typed abstract syntax trees, into programs that compute derivatives as well as their original values. Our prototype implementation illustrates the benefits of the algebraic approach to algorithmic differentiation.

### 1. Related work

M. Monagan and W. Neuenschwander [76] implemented the forward mode of AD — called GRADIENT — in the Maple computer algebra system. The GRADIENT package was latter extended by D. Villard and M. Monagan [111] to cover the reverse

mode. To the best of our knowledge, our work is the first documented attempt at implementing AD for the Axiom computer algebra system.

## 2. Future work

There are several directions we would like to extend our work. First, we are working on a way to voice algorithmic differentiation as a form of *abstract interpretation* as outlined by Cousot and Cousot [17]. The description of algorithmic differentiation in terms of abstract interpretation requires a rigorous description of the semantic meaning (and type checking); a first step is begun in Chapter V. We feel that such a description will allow us to rigorously describe the semantics of our forward mode implementation — even more, the reverse and hybrid modes take significant liberties with the meaning of programs, which we hope to capture. We naturally expect such an implementation to support higher order derivatives. Second, the "Axiom way" of implementing computational mathematics is to rely on strong typing to structure programs. Our initial work on interfacing with the Spad compiler suggests that we need Spad domains and packages for strongly typed representations of Spad programs themselves. We have found that the ability to integrate our framework into the Spad compiler as a library natural. Finally, and probably most importantly from the "Axiom way" of doing computational mathematics, we will continue the development of formal semantics of the Spad programming languages as well as the algebraic theory of algorithmic differentiation, to gain better understanding of algorithmic differentiation, and to stimulate the construction of generic AD libraries.

# CHAPTER V

## LOCAL SPECIALIZATION FOR OPENAXIOM

Support for active libraries is closely tied to support for the generic programming paradigm. In particular, the computations which allow an active library to observe and (more importantly) respond to the inputs of the library's components tend to be both static and typeful. By static, we mean that the generation of specialized codes based upon the inputs to the components occurs during compilation (or similar phases). By typeful we mean that the active library's activities are based upon type information and other static traits of the inputs to the components of the library.

The ability to select alternate variants of the implementations of algorithms and data structures based upon compile time parameters is called specialization. In the computer algebra system OpenAxiom, the library extension language Spad uses a variant of the specialization feature called *local specialization*. Local specialization is built upon the functorial nature of data-structures in Spad; functoriality in Spad is expressed through data-structures which act like functions that, given input values at compile-time, return a type. The local specialization feature allows multiple variants of a data-structure (and associated algorithms) at a single point of definition, dependent upon compile-time values such as types or values passed as parameters to the data-structure.

This chapter continues the rationalization of OpenAxiom computer algebra system; in particular, it provides a principled compilation strategy for local specialization informed by the theory of abstract interpretation.

## A. Introduction

Most algebraic structures in computational mathematics can be seen as instances of functors [23, 22]. Functors are naturally expressed in the typeful programming paradigm as parametrized datatypes. Certain properties of parametric data-structures may be common to all instances of a given data-structure. For instance, the integers $\mathbb{Z}/k\mathbb{Z}$ modulo a positive natural number $k$ always carry the structure of a ring. However, it can also happen that a specific instance of a parametrized structure has capabilities that are not shared by other instances [23]. For example, the ring $\mathbb{Z}/k\mathbb{Z}$ is also a field when $k$ is a prime number. In the generic programming setting, expressing such *conditional properties* for parametrized datatypes in a single definition is called *local specialization*. Any computer algebra system using a typeful programming language must include provisions for the support of local specialization.

The programming languages of the AXIOM system family [58] and the Aldor system [115, 114, 12] of computer algebra systems are categorial — they support abstract interfaces to concrete data-structures. Importantly, those systems' programming languages allow the direct expression of parametric data-structures and interfaces with conditional properties. The functorial data-structures (both abstract and concrete) of these systems are *parametrically polymorphic* [99]: the definition stays the same over a range of type and value parameters. The support for local specialization in such a categorial model presents interesting challenges.

In the OpenAxiom computer algebra system [25], and in all other variants of the AXIOM family, abstract algebraic structures are defined as *categories* and concrete instances of such structures as *domains*. One can think of categories as specifications, and of domains as implementations of specifications.[1] Each local specialization

[1]To relate the programming notions of categories and domains to the rest of this

is controlled by a conditional statement and each condition's predicate is a logical formula. Currently OpenAxiom only supports predicates over domains. General support for predicates involving user-defined functions or values does not exist, and this limitation has proven to be a hindrance in practice, and a source of several bugs. For instance, the current implementation of $\mathbb{Z}/k\mathbb{Z}$ in OpenAxiom always provides the operations of the structure field, particularly division and modular arithmeti — it is left to the user to test that the value $k$ is prime before using the field operations.

This chapter describes our extension to OpenAxiom that generalizes the conditions of conditional specifications to allow arbitrary user defined predicates. For example, the specification that the ring $\mathbb{Z}/k\mathbb{Z}$ is a field when the parameter $k$ is a prime can be written as:

```
IntegerModCategory(k: Integer()): Category == Type() with
  if prime? k then Field() else Ring()
    ...
```

We note that local specialization with generalized predicates is already handled in Aldor [115]. The algorithms underlying the implementation, however, have never been demonstrated or justified. A key contribution of this chapter is a rational and principled implementation of this feature. More specifically, our contributions include:

- Specification of the operational semantics of categories and `has`-predicates in the AXIOM system family.
- Generalization of the set of legal predicates in conditional specifications to allow user-defined predicates.
- Description and implementation of the conditional specification as an alternate

---

dissertation, we note that a category is (very roughly) analagous to a *concept* in C++0x, and a domain is analagous to a model of a concept.

evaluation of Spad code for deriving the static semantics of categories.

The rest of the chapter is structured as follows: Section B provides an overview of our general approach; Section C defines Spad language, and specifies the concrete semantics of Spad categories and predicates; we extend the set of legal predicates in Section D; we present our framework for the alernate evaluation of Spad in Section E, and demonstrate with practical examples that our framework allows in Section F; we discuss related work in Section G, and conclude in Section H.

## B. An overview

The family of AXIOM systems, OpenAxiom in particular, provide a very rich type system. OpenAxiom offers a strongly typed programming language where types are instantiated at run time. Yet, OpenAxiom supports an erasure semantics [19] for its core—except for (runtime) intensional representations of domains.

A standard formal presentation of OpenAxiom's programming language (Spad) would require a dynamic semantics for Spad, along with type checking rules (the static semantics), and a separate proof that the reduction rules are sound with respect to the type system. We feel that this approach makes a type system look like an after-thought, and sheds little light on the profound connection between the type system and the dynamic semantics. Our approach, instead, is inspired by the *types as abstractions* [16] philosophy that holds that a type system for a programming language is a *sound approximation* of the language's dynamic semantics.

The construction of the alternate evaluation for Spad is very general and proceeds in stages. First we define the dynamic semantics of the language. Next, we answer questions about or compute properties of programs based only on the programs' static structure. This leads to the first approximation: the *collecting semantics*. The

collecting semantics associates properties to each program point, for all conceivable executions, and is thus in general not computable. This allows us to seeks computable approximations, guided by the kind of information one is looking for. In our case, we are seeking computable characterizations of properties that hold in each branch of a conditional specification.

In our presentation we define a translation from Spad to an intermediate language; the intermediate language is a variant of the lambda calculus, see Section 2. The operational semantics are defined for this intermediate language, see Section 2; we then use these operational semantics to describe the collecting semantics, see Section 1, for the abstract interpretation presented in this chapter. This is a particularly heavyweight approach, however, our intention is to provide a framework onto which we can hang a number of analyses, and not just the analysis presented here.

## C. The Spad programming language

OpenAxiom is a library-centric and extensible system. Users extend OpenAxiom by writing libraries using the programming language called Spad. We describe enough of the Spad syntax to enable a discussion of Spad categories (and domains). We then specify the concrete semantics of Spad categories by defining a translation from Spad to an internal language, followed by the specification of the operational semantics of this internal language. We use the translation and the operational semantics of the internal language to inform the construction of the static semantics for categories in Spad.

The general philosophy of the Spad language is based on the abstract datatype methodology. In this methodology, a concrete algebraic structure is introduced by its specification (which we call its *category*) and its implementation (called its *domain*).

Specifications (abstract interfaces) will generally have many implementations, and it is typical for categories to be defined independently from their implementations. For example, we can capture the general notion of monoid structure by building a category that specifies a binary operation named `*` along with a constant named `1`. In Spad, we can express this as:

```
Monoid(): Category == Type with
  *: (%,%) → %
  1: %
```

The datatype `Integer` obviously satisfies the `Monoid` specification, i.e., with multiplication and the literal `1`. It so happens that the list data structure also satisfies this specification: the binary operator is the concatenation operation, and the constant `1` is the empty list. In the OpenAxiom system, datatypes satisfy (belong to) categories by explicit assertions. For instance, here is a definition for a domain that defines a list to carry the structure of a monoid:

```
ListMonoid(T: Type): Monoid() with
    coerce: List T → %
  == add
    import List T
    Rep == List T
    coerce(l: List T) == per l
    1:% == per empty()
    (x:%) * (y:%) == per concat(rep x, rep y)
```

The unusual indentation is due to the fact that Spad is a white-space sensitive language that defines code blocks by indentation level. This definition says that `ListMonoid` is a parametric data-structure that takes a single argument `T`, and returns a structure `ListMonoid` that carries the structure of a monoid (the first line of code). The specifications for `ListMonoid` are taken from the category `Monoid`, and are extended to include a function to convert a `List` to a `ListMonoid`. The

implementation of the specification comes after the statement `== add`; this section is called the *capsule*. The first line of the capsule imports the operations of the domain `List T` into the current scope. The next line of capsule `Rep == List T` states that the `ListMonoid`'s internal representation is a `List`. The remainder of the capsule provides definitions for the specifications defined by the category `Monoid`. As a brief note, the operation `per` changes the type of a `List` to a `ListMonoid`, thus converting a `List` value into a `ListMonoid` value. The operation `rep` does the reverse.

Below, we describe enough of the syntax of Spad to enable a discussion of Spad categories. We then specify the concrete semantics of Spad categories by defining a translation from Spad to an internal language, followed by the specification of the operational semantics of this internal language.

## 1. Syntax

A Spad program can be understood as a sequence of category and domain definitions, followed by an expression. Figure 33 shows the syntax of the parts of Spad needed for this chapter. Specifically, a category is a sequence of specifications, each of which is either a function declaration or a *category extension*. The latter is another category instance, and indicates that all of the specifications exported by that instance should be included in the current category. The category extension mechanism is transitive, reflexive, and antisymmetric. In the version of Spad used in this chapter every specification is conditional.

*Program* A Spad program can be understood as a sequence of category and domain definitions, followed by an expression.

| | | | |
|---|---|---|---|
| *Program* | P | ::= | $(C\|D)^*\ e$ |
| *CategoryDef* | C | ::= | $\chi$ `:` `Category ==` `Type() with` $\mathrm{W}^+$ |
| *DomainDef* | D | ::= | $\chi : \tau$ `==` `add` $\mathrm{A}^+$ |
| *CallForm* | $\chi$ | ::= | $x([x : \tau]^*)$ |
| *Cond. Spec.* | W | ::= | `if` $\pi$ `then` $\mathrm{E}_1$ `else` $\mathrm{E}_2$ |
| *Cond. Def.* | A | ::= | `if` $\pi$ `then` $\mathrm{d}_1$ `else` $\mathrm{d}_2$ |
| *Export* | E | ::= | $x : \tau \mid \tau$ |
| *Definition* | d | ::= | `Rep ==` $\tau \mid \chi : \tau == e \mid$ `Nil` |
| *Predicate* | $\pi$ | ::= | `true` $\mid \pi_1$ `and` $\pi_2 \mid \pi_1$ `or` $\pi_2 \mid$ `not` $\pi \mid \tau_1$ `has` $\tau_2$ |
| *Expression* | $e$ | ::= | `if` $e_1$ `then` $e_2$ `else` $e_3 \mid e(e^*) \mid c$ |
| *Type* | $\tau$ | ::= | $x(\tau^*) \mid x^\tau \mid c^\tau \mid (\tau^*) \to \tau \mid$ `%` |
| *Identifier* | $x$ | | |

Fig. 33. The syntax of the core parts of Spad.

*CategoryDef* A Spad category definition is a sequence of conditional specifications; the entire sequence is called the *Exports* section of the category. All Spad categories must extend the category `Type`, which is a special built-in root category with no specifications. In the example shown below, the left-hand category definition declares a Spad category with two signatures: a neutral element 1 and binary operator $*$ of a monoid structure. The right-hand definition extends `Monoid` with the `inverse` operation to capture the mathematical notion of group structure.[2]

Monoid(): **Category** == Type() **with**
  **if true then** 1: % **else** Type()
  **if true then** $*$: (%, %) $\to$ % **else** Type()

Group(): **Category** == Type() **with**
  **if true then** Monoid() **else** Type()
  **if true then** inverse: (%, %) $\to$ % **else** Type()

[2]The definition of Spad in this chapter does not allow specifications or definitions that are not conditional; we thus represent unconditional specifications and definitions through the use of conditionals with the predicate `true` for their condition.

*DomainDef* A Spad domain consists of a sequence of conditional definitions; the sequence of conditional definitions is referred to as the *capsule* of the domain. The definition of the domain provides implementations for the specifications defined by the categories the domain carries. The implementation part of the domain is called a *Capsule*; the capsule may define the representation of the object belonging to the domain and also provide definitions for operations declared in the exports section of the categories it implements.

*CallForm* A call form consists of an identifier and a parenthesized sequence of signatures declaring formal parameters. A call form is needed in the definitions of a Spad category, Spad domain, and function.

*Conditional Specification* A conditional specification is expressed in Spad as an if-statement. The condition of the if-statement can be any legal *Predicate* $\pi$. The then- and else- branches must contain a single statement, each. The statement that can go in these branches must be an *Export*.

*Conditional Definition* A conditional definition is written as an if-statement. The condition of the conditional definition can be any `Predicate` $\pi$. Each branch of the conditional definition must have one statement — a definition.

*Export* An export is either a definition — a signature of a function such as binary plus `+:(%,%)->%` or a distinguished value such as `1:%` — or, a category extension such as `Monoid()`.

*Definition* A definition is the binding of an identifier or a function call expression to a Spad category, Spad domain, or an ordinary function. The Spad language, as

understood by the Spad compiler, does not allow ordinary function definitions at toplevel. The special symbol `Nil` is introduced to allow a null statement.

*Predicate* A predicate is used as the condition of a *Conditional Specification*. It can be either a atomic predicate such as `true` or `has`-predicate, or a logical formulae of atomic predicates.

*Expression* An expression is either an if-statement, function call, or a constant. Looping and iteration are not possible but recursion is possible.

*Type* A type is an instantiation of a Spad category or Spad domain, a function type, the name of a Spad category or Spad domain, or a carrier set denoted by `%`. In a category definition, `%` denotes a placeholder for a domain. In a domain definition, `%` denotes the domain itself.

*Identifier* An identifier is a finite sequence of characters. The set of identifiers in Spad is countably infinite.

## 2. An internal language

We define the semantics of a Spad program by translation into an internal representation language which allows us to express categories and specifications within those categories. We provide a full description of the internal language, and a translation from Spad to this internal language. While the full framework is somewhat heavy-weight, the framework provides a foundation on which to build other transformations not considered in this chapter. We note that the current translation only covers the syntax and semantics of the Spad language needed to describe categories and the

conditional specifications within the categories. Figure 34 and Figure 35 show the syntax and operational semantics, respectively, for this language.

The internal language is the untyped lambda calculus with some extensions. We provide support for a list data structure, offering the primitive list operators *cons*[3], *car*, and *cdr*, as well as a predicate for detecting an empty list. The term $\mathbf{R}_{id}(c_1, e_2)$ represents a Spad domain or category, where the constant $c_1$ is the name of the domain or category. The variable $e_2$ is a list of representations of the specifications, i.e., the function declarations and category extensions, of the domain or category. Since an **R**-term represents a category (or domain), an **R**-term also represents a Spad type in the internal language. The **R**-term types are used by a number of (compile-time) functions, including the `has`-predicate. The term $\mathbf{R}_{\rightarrow}(e_1, e_2)$ represents a Spad function definition where $e_1$ is a list of the representations of the parameter types of the function, and $e_2$ is a lambda abstraction representing the body of the function. To access the elements of the above terms we define the functions $\pi_1$ and $\pi_2$ that return the first and second element, respectively, when applied to either of the "**R**-terms". The constants $c$ include OpenAxiom's set of primitive types (e.g., integers and floating-point numbers) and symbols (used, e.g., as names of categories and domains).

$$
\begin{array}{llll}
\textit{Terms} & e & ::= & \lambda x.e \mid e_1 e_2 \mid \mathit{cons}(e_1, e_2) \mid \mathit{car}(e) \mid \mathit{cdr}(e) \mid \mathit{isNil?}(e) \\
 & & \mid & \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid \mathbf{R}_{id}(c_1, e_2) \mid \mathbf{R}_{\rightarrow}(e_1, e_2) \\
 & & \mid & \pi_1(e) \mid \pi_2(e) \mid \mathit{is}{\rightarrow}?(e) \mid \mathit{isid?}(e) \mid v \\
\textit{Values} & v & ::= & c \mid x \mid \lambda x.e \mid \mathit{cons}(v_1, v_2) \mid \mathit{nil} \mid \mathit{true} \mid \mathit{false}
\end{array}
$$

Fig. 34. The syntax of the internal language.

The operational semantics for the intermediate language are shown in Figure 35.

[3]In the later sections, we use the notation "$[a_1, \ldots, a_n]$" as a shorthand for constructing a list from the elements $a_1, \ldots, a_n$.

The operational semantics are defined as small-step semantics. The rules include function application, the construction of lists, access to the head and tail of lists, and a test to determine if a list is empty. The rules also include the introduction of named functions (let-statements), and if-then-else statements. The final set of rules are for the projection onto the elements out of a $\mathbf{R}$-term and tests to determine if a value is either a $\mathbf{R}_{id}$-term or a $\mathbf{R}_{\rightarrow}$-term.

$$(\lambda x.e)v \mapsto e[v/x] \qquad \frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2} \qquad \frac{e \mapsto e'}{v\, e \mapsto v\ e'}$$

$$car(cons(v_1, v_2)) \mapsto v_1 \qquad cdr(cons(v_1, v_2)) \mapsto v_2$$

$$\frac{e_1 \mapsto e_1'}{cons(e_1, e_2) \mapsto cons(e_1', e_2)} \qquad \frac{e \mapsto e'}{cons(v, e) \mapsto cons(v, e')}$$

$$\frac{e \mapsto e'}{car(e) \mapsto car(e')} \qquad \frac{e \mapsto e'}{cdr(e) \mapsto cdr(e')}$$

$$\frac{e \mapsto e'}{isNil?(e) \mapsto isNil?(e')} \qquad \frac{e \mapsto nil}{isNil?(e) \mapsto true} \qquad \frac{e \mapsto v \quad v \neq nil}{isNil?(e) \mapsto false}$$

$$let\ x = v\ in\ e \mapsto e[v/x] \qquad \frac{e_1 \mapsto e_1'}{let\ x = e_1\ in\ e_2 \mapsto let\ x = e_1'\ in\ e_2}$$

$$\frac{e_2 \mapsto e_2'}{\begin{array}{l}if\ true\ then\ e_2\\ else\ e_3 \mapsto e_2'\end{array}} \qquad \frac{e_3 \mapsto e_3'}{\begin{array}{l}if\ false\ then\ e_2\\ else\ e_3 \mapsto e_3'\end{array}} \qquad \frac{e_1 \mapsto e_1'}{\begin{array}{l}if\ e_1\ then\ e_2\ else\ e_3\\ \mapsto if\ e_1'\ then\ e_2\ else\ e_3\end{array}}$$

$$\frac{e_i \mapsto e_i'}{\pi_i(\mathbf{R}_\bullet(e_1, e_2)) \mapsto e_i'} \qquad \frac{e_1 \mapsto e_1'}{\mathbf{R}_\bullet(e_1, e_2) \mapsto \mathbf{R}_\bullet(e_1', e_2)} \qquad \frac{e \mapsto e'}{\mathbf{R}_\bullet(v, e) \mapsto \mathbf{R}_\bullet(v, e')}$$

$$\frac{e \mapsto e'}{is\bullet?(e) \mapsto is\bullet?(e')} \qquad \frac{e = \mathbf{R}_\bullet(e_1, e_2)}{is\bullet?(e) \mapsto true} \qquad \frac{e \neq \mathbf{R}_\bullet(e_1, e_2)}{is\bullet?(e) \mapsto false}$$

Fig. 35. The small step operational semantics of the internal language. The $\bullet$ symbol stands for $id$ or $\rightarrow$. The index $i$ in the projection rule is either 1 or 2.

## 3. Translation of Spad to internal language

As the basis for the abstract interpretation discussed in Section E, we describe the translation of a Spad category into the internal language. We think of this translation as a specification for a Spad compiler that compiles Spad to a lower-level language such as assembly, or to a virtual machine. Formally, we define a syntax-directed translation function $\mathrm{T}_{\bullet}[\![\cdot]\!] : \mathrm{SPAD} \rightarrow \mathrm{IL}$, where SPAD is the set of Spad programs and IL the set of programs in the internal language. This function, defined by case for each syntactical form of Spad, is shown in Figure 36.

The translation of a Spad category into the internal language will result in a value (a function returning a list) that is the representation of a Spad category in the internal language. This construction allows us to understand the semantics of both instantiated and uninstantiated categories.

To translate a category definition to the internal language, we iteratively translate a conditional specification at each program point of the category definition, in addition to the callform of the category. (The callform is an identifier and a parenthesized list of formal parameters of a category.) Each iteration step computes the internal representation of a callform, a function declaration, or recursively translates a category extension following the transitivity of category extension. That is, when a category $C_1$ declares that it extends the category $C_2$, we replace this specification with the specifications of $C_2$, a recursive operation.

The translation of a category definition results in a lambda abstraction in IL introduced by a let-statement which names the lambda abstraction. The lambda abstraction takes an extra parameter $\hat{\%}$ for the *carrier set* `%` in SPAD. The body of the abstraction is a list of if-statements (whose branches contain "**R**-terms") representing a sequence of specifications. The if-statements of the conditional specifications in the

$$
\mathrm{T}_C \left[\!\!\left[ \begin{array}{l} x_0(x_1:\tau_1,\ldots,x_k:\tau_k): \\ \quad \mathsf{Category} == \ \mathsf{Type()\ with} \\ \quad\ \mathrm{W}_1 \cdots \mathrm{W}_n \end{array} \right]\!\!\right] \stackrel{\text{def}}{\Longrightarrow} \begin{array}{l} \mathit{let}\ \mathrm{T}_\tau [\![x_0]\!] \\ \quad = \lambda\, \mathrm{T}_\tau [\![\%]\!]\, . \lambda\, \mathrm{T}_\tau [\![x_1]\!] \ldots \lambda\, \mathrm{T}_\tau [\![x_k]\!]\, . \\ \quad \mathit{cons}\,(\mathit{if}\ \mathrm{true}\ \mathit{then}\ \mathrm{T}_X [\![x_0(x_1,\ldots,x_k)]\!] \\ \quad\ \mathit{else}\ \hat{Type}(\%), \\ \quad\ \mathrm{T}_W [\![W_1]\!] \oplus \cdots \oplus \mathrm{T}_W [\![W_n]\!] \\ \quad\ \oplus \hat{Type}(\%))\ \mathit{in} \end{array}
$$

$$
\begin{array}{rcl}
\mathrm{T}_W [\![\mathsf{if}\ \pi\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathit{if}\ \mathrm{T}_\pi [\![\pi]\!]\ \mathit{then}\ \mathrm{T}_E [\![E_1]\!]\ \mathit{else}\ \mathrm{T}_E [\![E_2]\!] \\
\mathrm{T}_\sigma [\![x]\!] & \stackrel{\text{def}}{\Longrightarrow} & `x \\
\mathrm{T}_X [\![x]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathrm{T}_\tau [\![x]\!] \\
\mathrm{T}_X [\![x(\tau_1,\ldots,\tau_k)]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathbf{R}_{id}(\mathrm{T}_\sigma [\![x]\!], [\mathrm{T}_X [\![\tau_1]\!], \ldots, \mathrm{T}_X [\![\tau_k]\!]]) \\
\mathrm{T}_X [\![(\tau_1,\ldots\tau_k) \rightarrow \tau_0]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathbf{R}_{\rightarrow}([\mathrm{T}_X [\![\tau_0]\!], \mathrm{T}_X [\![\tau_1]\!], \ldots, \mathrm{T}_X [\![\tau_k]\!]], \mathrm{nil}) \\
\mathrm{T}_E [\![x(\tau_1,\ldots,\tau_k)]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathrm{T}_\tau [\![x]\!]\, (\mathrm{T}_\tau [\![\%]\!], \mathrm{T}_\tau [\![\tau_1]\!], \ldots, \mathrm{T}_\tau [\![\tau_k]\!]) \\
\mathrm{T}_E [\![x]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathrm{T}_\tau [\![x]\!]\, (\mathrm{T}_\tau [\![\%]\!]) \\
\mathrm{T}_E [\![x:\tau]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathbf{R}_{id}(\mathrm{T}_\sigma [\![x]\!], [\mathrm{T}_X [\![\tau]\!]]) \\
\mathrm{T}_\pi [\![\mathsf{true}]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathrm{true} \\
\mathrm{T}_\pi [\![\tau_1\ \mathsf{has}\ \tau_2]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathit{car}(\mathrm{T}_X [\![\tau_2]\!]) \in \mathrm{T}_\tau [\![\tau_1]\!] \\
\mathrm{T}_\pi [\![\mathsf{not}\ \pi]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathit{not}(\mathrm{T}_\pi [\![\pi]\!]) \\
\mathrm{T}_\pi [\![\pi_1\ \mathsf{and}\ \pi_2]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathit{and}(\mathrm{T}_\pi [\![\pi_1]\!], \mathrm{T}_\pi [\![\pi_2]\!]) \\
\mathrm{T}_\pi [\![\pi_1\ \mathsf{or}\ \pi_2]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathit{or}(\mathrm{T}_\pi [\![\pi_1]\!], \mathrm{T}_\pi [\![\pi_2]\!]) \\
\mathrm{T}_\tau [\![x(\tau_1,\ldots,\tau_k)]\!] & \stackrel{\text{def}}{\Longrightarrow} & \mathrm{T}_\tau [\![x]\!]\, (\mathrm{T}_\tau [\![\tau_1]\!], \ldots, \mathrm{T}_\tau [\![\tau_k]\!]) \\
\mathrm{T}_\tau [\![x]\!] & \stackrel{\text{def}}{\Longrightarrow} & \hat{x}
\end{array}
$$

Fig. 36. The translation rules from Spad to the internal language. Functions $\oplus$ and $\in$ are primitive operators defined in Figure 37. The function $\oplus$ concatenates two lists together; the function $\in$ takes a value and a list as arguments, and checks the membership of the value to the list; the functions *and*, *or*, and *not* are the usual boolean operators. We use $`x$ and $\hat{x}$ to denote the symbol and identifier $x$, respectively, in the internal language.

```
let ⊕ = λleft.λright.
  if isnil? left then right else cons(car left, fix(concat)(cdr left, right)) in

let member_equal? = λleft.λright.
  if isnil? left and isnil? right then true else if isnil? left then false
  else if isnil? right then false else if isid? left and isid? right then
    if π1 left = π1 right then
      fix(member_equal?)(π2 left, π2 right)
    else false
      else if is→? left and is→? right then
        if fix(member_equal?)(π1 left, π2 right) then true else false
  else if member_equal?(car left, car right) then
    fix(member_equal?)(cdr left, cdr right) else false in

let ∈ = λvalue.λspeclist.
  if isnil? speclist then false else if member_equal?(car speclist, value) then true
  else fix(∈)(value, cdr speclist) in
```

Fig. 37. Definitions of the functions $\in$ and $\oplus$.

body of a category are translated to if-statements in IL; `has`-predicates are translated to the function $\in$; category extensions are translated to function calls to other translated categories in IL; the Spad boolean operators `and`, `or`, and `not` are translated to functions of the same name in the IL; function declarations are translated to "$\mathbf{R}_{\rightarrow}$-terms".

The translation in IL relies on two functions $\oplus$ and $\in$ being defined: the first function is the concatenation of two lists, and the second function determines if a value is within a list. These two functions are defined in Figure 37. We note that the function $\in$ is recursive, and is aware of the conventions used within the various **R**-terms.

As an example, we translate the category `ComplexCategory`, defined in Figure 38,

```
ComplexCategory(R: CommutativeRing()): Category == Type() with
  if true then CommutativeRing() else Type()
  if R has IntegralDomain() then exquo : (%, R) → % else Type()
  if true then FullyLinearlyExplicitRingOver(R) else Type()
    ...
```

Fig. 38. The definition of the category `ComplexCategory`. To simplify the translation, we use a preprocessor to transform an unconditional specification into a conditional form with condition `true` and the category `Type()` in the else branch—`Type()` is the empty root category that has no specifications.

which represents the extension of a ring structure by $\sqrt{-1}$. Applying the translation rules at the topmost level yields (showing only the parts of the category that are included in Figure 38):

$$
\begin{aligned}
&\mathit{let}\ \mathrm{T}_\tau[\![\mathsf{ComplexCategory}]\!] = \lambda\,\mathrm{T}_\tau[\![\%]\!].\lambda\,\mathrm{T}_\tau[\![\mathsf{R}]\!]\,.\\
&\quad \mathit{cons}(\mathit{if}\ \mathrm{true}\ \mathit{then}\ \mathrm{T}_X[\![\mathsf{ComplexCategory(R)}]\!]\ \mathit{else}\ \hat{Type}(\hat{\%}),\\
&\qquad \mathrm{T}_W[\![\mathsf{if\ true\ then\ CommutativeRing()\ else\ Type()}]\!] \oplus\\
&\qquad \mathrm{T}_W[\![\mathsf{if\ R\ has\ IntegralDomain()\ then\ exquo:(\%,R)} \to \mathsf{\%\ else\ Type()}]\!] \oplus\\
&\qquad \mathrm{T}_W[\![\mathsf{if\ true\ then\ FullyLinearlyExplicitRingOver(R)\ else\ Type()}]\!] \oplus \cdots \oplus \hat{Type}(\hat{\%}))\ \mathit{in}
\end{aligned}
$$

Where the function $\oplus$ is the concatenation of two lists, shown in Figure 37. We note that the callform of the category is used to generate both the name of the function representing the category, the names of the parameters of that function, and an **R**-term which will become the reflexive assertion that the `ComplexCategory` is the category `ComplexCategory`. Beginning with the translation of the callform to the name of function we get:

$$\mathrm{T}_\tau[\![\mathsf{ComplexCategory}]\!] \overset{\text{a.i.}}{\Longrightarrow} \hat{ComplexCategory}$$

that is, the identifier of the callform becomes an identifier in IL. The parameters of the callform are translated similarly into identifiers IL.

We note that we do not translate the types of the formal parameters: the type information is embedded within the specification lists that are passed as arguments

to the function. For example, an argument passed to parameter R would be a specification list, containing run-time evidence of all the types of the domain represented by R. Similarly, the callform is also translated into a specification (from the second line of the translation):

$$\begin{aligned} \mathrm{T}_X\,[\![\mathsf{ComplexCategory(R)}]\!] &\overset{\text{a.i.}}{\Longrightarrow} \mathbf{R}_{id}(\mathrm{T}_\sigma\,[\![\mathsf{ComplexCategory}]\!], [\mathrm{T}_X\,[\![\mathsf{R}]\!]]) \\ &\overset{\text{a.i.}}{\Longrightarrow} \mathbf{R}_{id}(\text{`ComplexCategory}, [\mathrm{T}_\tau\,[\![\mathsf{R}]\!]]) \\ &\overset{\text{a.i.}}{\Longrightarrow} \mathbf{R}_{id}(\text{`ComplexCategory}, [\hat{R}]). \end{aligned}$$

Conditional specifications in categories are translated into if-statements in IL; furthermore, the value true from Spad becomes the values *true* in IL. Category extension declarations are translated into function calls of the same name. The function $\hat{Type}$ returns the specification list $[\mathbf{R}_{id}(\text{`}Type, [])]$.

$$\begin{aligned} &\mathrm{T}_W\,[\![\mathsf{if\ true\ then\ CommutativeRing()\ else\ True}]\!] \\ &\quad\overset{\text{a.i.}}{\Longrightarrow} \mathit{if}\ \mathrm{T}_\pi\,[\![\mathsf{true}]\!]\ \mathit{then}\ \mathrm{T}_E\,[\![\mathsf{CommutativeRing()}]\!]\ \mathit{else}\ \mathrm{T}_E\,[\![\mathsf{Type()}]\!] \\ &\quad\overset{\text{a.i.}}{\Longrightarrow} \mathit{if\ true\ then}\ \mathrm{T}_\tau\,[\![\mathsf{CommutativeRing}]\!]\,(\mathrm{T}_X\,[\![\%]\!])\ \mathit{else}\ \mathrm{T}_\tau\,[\![\mathsf{Type}]\!]\,(\mathrm{T}_X\,[\![\%]\!]) \\ &\quad\overset{\text{a.i.}}{\Longrightarrow} \mathit{if\ true\ then}\ \hat{CommutativeRing}(\hat{\%})\ \mathit{else}\ \hat{Type}(\hat{\%}) \end{aligned}$$

We now turn our attention to the translation of a `has`-predicate. Intuitively, a `has`-predicate $X$ has $C$ is asking if a domain $X$ has the type $C$. We translate this into a question asking if the specification of $C$ is within the list of specifications of $X$:

$$\begin{aligned} \mathrm{T}_\pi\,[\![\mathsf{R\ has\ IntegralDomain()}]\!] &\overset{\text{a.i.}}{\Longrightarrow} \mathit{car}(\mathrm{T}_X\,[\![\mathsf{IntegralDomain()}]\!]) \in \mathrm{T}_\tau\,[\![\mathsf{R}]\!] \\ &\overset{\text{a.i.}}{\Longrightarrow} \mathbf{R}_{id}(\text{`IntegralDomain}[]) \in \hat{R}. \end{aligned}$$

D. User-defined predicates

Figure 39 shows the syntax we add to Spad to support user-defined predicates, as well as the translation rules for the new syntactic form. The rules below rely on a function called *lookup* for function resolution. The function *lookup* takes the name of the function being called, the representation of the types of the arguments of the

function, and domain the function is to be found in. For this chapter, we let Spad be parametrized by the implementation of *lookup*; however, a particular instance of *lookup* could be implemented by, for instance, name-lookup. Equipped with the new

$$\textit{Predicate} \quad \pi \quad ::= \quad x(\pi^*)\$\tau$$

$$\mathrm{T}_\pi [\![x_0(\pi_1, \ldots, \pi_k)\$\tau]\!] \stackrel{\text{def}}{\Longrightarrow} \begin{array}{l} \mathit{lookup}(\mathrm{T}_\sigma [\![x_0]\!], [\mathit{car}(\mathrm{T}_\pi [\![\pi_1]\!]), \ldots, \\ \quad \mathit{car}(\mathrm{T}_\pi [\![\pi_2]\!])], \mathrm{T}_\tau [\![\tau]\!])(\mathrm{T}_\pi [\![\pi_1]\!], \ldots, \mathrm{T}_\pi [\![\pi_k]\!]) \end{array}$$

Fig. 39. The extension of the Spad grammar and translation rules to support user-defined predicates. The function *lookup* returns the function called by the user, given the name of the function, and the domain the Spad function is defined in.

language construct we can show more of the definition of `ComplexCategory`. We add a specification that expresses that the category carries the structure of a `Field` when both its parameter `R` carries the structure of a `Field` and the polynomial $x^2 + 1$ is irreducible in `R`:

```
if R has Field() and irreducible?((monomial(1,2)$R+1)$R)$R
then Field()
else Type()
```

The syntax `$R` above specifies that `R` is the domain that provides the implementation of the function preceding the `$` symbol.

Using the extended translation rules, the above user-defined predicate is translated as follows:

$$\begin{array}{l} \mathrm{T}_\pi [\![\mathsf{irreducible?}((\mathsf{monomial}(1,2)\$\mathsf{R}+1)\$\mathsf{R})\$\mathsf{R}]\!] \implies \\ \quad \mathit{lookup}(\text{`irreducible?}, [\mathit{car}(\mathrm{T}_\pi [\![(\mathsf{monomial}(1,2)\$\mathsf{R}+1)\$\mathsf{R})]\!]], R)( \\ \quad\quad \mathrm{T}_\pi [\![(\mathsf{monomial}(1,2)\$\mathsf{R}+1)\$\mathsf{R}]\!]) \end{array}$$

## E. Static analysis of categories

Conditional specifications control the capabilities of categories, i.e., the set of exported function declarations of a category is determined by the run-time evaluation of the conditional specifications. However, we want to be able to reason about such capabilities at compile-time so that we can, for example, determine the best matching unambiguous function definition given a function call. The main thrust of our approach is to seek computable characterizations of specifications that hold in each branch of a conditional specification, and statically approximate a category's specifications and the specifications' conditions through an abstract interpretation framework.

The implementation of our framework is inspired by Cousot's notion of types as abstract evaluations. Our static analysis extracts a set of abstract specifications from each conditional specification of a Spad category, updating an abstract store with the set. Intuitively, the recursive translation and substitution of the specifications for category extension in the concrete translation is the recursive abstract evaluation of a category and join of its abstract store to the current store. The concrete evaluation of a conditional specification selects one branch according to the run-time evaluation of its condition, while, correspondingly, we abstractly evaluate both branches and join the resulting abstract stores.

The final step in the static analysis is to simplify the resulting abstract store. The abstract store can suffer from *expression explosion* due to the particular way we construct the store. At each step of the abstract evaluation we simplify the predicates of each abstract specification by applying the rules of classical logic, i.e., $\mathsf{true}\ \mathsf{and}\ \pi \rightarrow \pi$, etc. Furthermore, we define an operation called *reduction* which attempts to simplify the abstract store by using information found within the abstract store to simplify the abstract store (this operation is developed and explained in

Section 4).

## 1. The structure of the abstract domain

The abstract domain is defined as the set of all sets of abstract specifications. An abstract specification $\langle E, \pi \rangle$ is defined as an approximation of a conditional specification by attaching a category specification $E$ with a condition $\pi$, under which $E$ holds. We say that each set of abstract specifications is as an *abstract store* $\sigma$. We note that the smallest abstract store is $\{\langle$ % has Type(), true$\rangle\}$ for reason that all categories must extend `Type`. We define the operator join $\sqcup_{\mathcal{A}}$ as

$$\begin{array}{rcl} & & \{\langle E, \pi_1 \rangle \mid \forall \langle E, \pi_1 \rangle \in \sigma, \nexists \langle E, \pi_2 \rangle \in \sigma_\Delta\} \\ \sigma \sqcup_{\mathcal{A}} \sigma_\Delta = & \cup & \{\langle E, \pi_2 \rangle \mid \forall \langle E, \pi_2 \rangle \in \sigma_\Delta, \nexists \langle E, \pi_1 \rangle \in \sigma\} \\ & \cup & \{\langle E, \pi_1 \text{ or } \pi_2 \rangle \mid \langle E, \pi_1 \rangle \in \sigma \wedge \langle E, \pi_2 \rangle \in \sigma_\Delta\}. \end{array}$$

The join operator takes two abstract stores $\sigma$ and $\sigma_\Delta$, and constructs an updated store. The abstract specifications from both $\sigma$ and $\sigma_\Delta$ which have different category extensions will be directly added to the updated store; if two abstract specifications from different stores have a common extension, a combined abstract specification will be added. The combined abstract specification keeps the common extension, but combines the predicates using the logical connective *or*; the common extension holds when either of the conditions is true.

## 2. Abstract evaluation of syntactic forms

We define the syntax-directed rules for the static analysis of a Spad category in Figure 40. An interpretation function $[\![\cdot]\!]^{\Pi}_{\bullet} : \text{SPAD} \rightarrow \text{AS}$ takes a category definition in Spad syntactic form, then computes the abstract store containing the corresponding abstract specifications of the category. The superscript $\Pi$ is the *assumption* of the current interpretation, and is required when interpreting the branches of conditional specifications (its use is explained later).

$$
\begin{array}{rcl}
\left[\!\!\left[ \begin{array}{c} x_0(x_1:\tau_1,\ldots,x_k:\tau_k): \\ \mathsf{Category} == \mathsf{with} \\ \mathrm{W}_1 \cdots \mathrm{W}_n \end{array} \right]\!\!\right]_C^{\Pi} & \xRightarrow{\text{a.i.}} & \left( \begin{array}{c} [\![x_0(x_1:\tau_1,\ldots,x_k:\tau_k)]\!]_X^{\Pi} \\ \sqcup_{\mathcal{A}} [\![\mathrm{W}_1]\!]_W^{\Pi} \sqcup_{\mathcal{A}} \cdots \sqcup_{\mathcal{A}} [\![\mathrm{W}_n]\!]_W^{\Pi} \\ \sqcup_{\mathcal{A}} \{\langle \%\ \mathsf{has}\ \mathsf{Type}(),\ \mathsf{true}\rangle\} \end{array} \right) \\
[\![x_0(x_1:\tau_1,\ldots,x_k:\tau_k)]\!]_X^{\Pi} & \xRightarrow{\text{a.i.}} & \left( \begin{array}{c} \{\langle \%\ \mathsf{has}\ \chi, \Pi\rangle\} \\ \sqcup_{\mathcal{A}} [x_1/\%]\,[\![\tau_1]\!]_E^{\Pi} \sqcup_{\mathcal{A}} \ldots \sqcup_{\mathcal{A}} [x_k/\%]\,[\![\tau_k]\!]_E^{\Pi} \end{array} \right) \\
 & & \text{where } \chi = [\![x_0]\!]_\tau\,([\![x_1]\!]_\tau, \ldots, [\![x_k]\!]_\tau) \\
[\![\mathsf{if}\ \pi\ \mathsf{then}\ \mathrm{E}_1\ \mathsf{else}\ \mathrm{E}_2]\!]_W^{\Pi} & \xRightarrow{\text{a.i.}} & [\![\mathrm{E}_1]\!]_E^{\Pi\ \mathsf{and}\ \pi} \sqcup_{\mathcal{A}} [\![\mathrm{E}_2]\!]_E^{\Pi\ \mathsf{and}\ (\mathsf{not}\ \pi)} \\
[\![x_0(\tau_1,\ldots,\tau_k)]\!]_E^{\Pi} & \xRightarrow{\text{a.i.}} & \left( \left[\!\!\left[ \begin{array}{c} \tau_1/x_1,\ldots, \\ \tau_k/x_k \end{array} \right]\!\!\right] \begin{array}{c} x_0(x_1:\tau_1',\ldots,x_k:\tau_k'): \\ \mathsf{Category} == \mathsf{Type}()\ \mathsf{with} \\ \mathrm{W}_1 \cdots \mathrm{W}_n \end{array} \right]\!\!\right]_C^{\Pi} \Bigg) \\
[\![x:\tau]\!]_E^{\Pi} & \xRightarrow{\text{a.i.}} & \{\langle \%\ \mathsf{has}\ x:\tau, \Pi\rangle\} \\
[\![\mathsf{Type}()]\!]_E^{\Pi} & \xRightarrow{\text{a.i.}} & \{\langle \%\ \mathsf{has}\ \mathsf{Type}(), \Pi\rangle\} \\
[\![x]\!]_\tau & \xRightarrow{\text{a.i.}} & \text{`x}
\end{array}
$$

Fig. 40. Rules for abstract interpretation of Spad programs.

To abstractly evaluate a category definition we interpret the callform, followed by iteratively abstractly evaluating each conditional specification. We demonstrate the top-level rule by showing the abstract evaluation of `ComplexCategory` from Figure 38, along with the additional conditional specification enabled in Section D:

$$
\begin{array}{ll}
 & [\![\mathsf{ComplexCategory(R : CommutativeRing())}]\!]_X^{\mathsf{true}} \\
\sqcup_{\mathcal{A}} & [\![\mathsf{if\ true\ then\ CommutativeRing()\ else\ Type()}]\!]_W^{\mathsf{true}} \\
\sqcup_{\mathcal{A}} & [\![\mathsf{if\ R\ has\ IntegralDomain()\ then\ exquo\ :\ (\%,R) \to \%\ else\ Nil}]\!]_W^{\mathsf{true}} \\
\sqcup_{\mathcal{A}} & [\![\mathsf{if\ true\ then\ FullyLinearlyExplicitRingOver(R)\ else\ Type()}]\!]_W^{\mathsf{true}} \\
\sqcup_{\mathcal{A}} & [\![\mathsf{if\ R\ has\ Field()\ and\ irreducible?(monomial(1,2)\$R+1)\$R\ then\ Field()\ else\ Type()}]\!]_W^{\mathsf{true}} \\
\sqcup_{\mathcal{A}} & \cdots \sqcup_{\mathcal{A}} \{\langle \%\ \mathsf{has}\ \mathsf{Type}(),\ \mathsf{true}\rangle\}
\end{array}
$$

In this abstract evaluation, we assign $\Pi$ to the value true as the initial assumption, which means no assumptions (only a top-level abstract evaluation can have the assumption true). In the abstract evaluation we can think of the join operator $\sqcup_{\mathcal{A}}$ as the approximation of the concatenation operator $\oplus$ in the concrete semantics.

The callform introduces an abstract specification for the category itself (reflexivity of category extension), and abstract specifications of the formal parameters in the callform. The type of each formal parameter is abstractly evaluated and the resulting

abstract store for each parameter has all uses of the variable % renamed to the name of the formal parameter. For the callform of `ComplexCategory`, the abstract evaluation results in an abstract specification asserting the category is the `ComplexCategory`; this is joined to the abstract store of `CommutativeRing` — which has had all instances of the variable % renamed to `R`, i.e., specifications about `CommutativeRing` are now about `R`:

$$[\![\text{ComplexCategory(R:CommutativeRing())}]\!]_X^{\text{true}} \stackrel{\text{a.i.}}{\Longrightarrow} \{\langle \text{\% has `ComplexCategory(`R), true}\rangle\} \sqcup_{\mathcal{A}} [\text{R/\%}]\,[\![\text{CommutativeRing()}]\!]_E^{\text{true}}.$$

Each conditional specification of a category definition results in the join of abstract stores from evaluation of each branch's specification: function declarations are directly interpreted as abstract specifications; category extensions are recursively interpreted following the transitivity of category extensions. The assumption for interpreting each branch of the conditional specification is computed by the conjunction of the enclosing category's assumption $\Pi$ and either the predicate $\pi$ or not $\pi$ for the then- and else- branches, respectively. For instance, the last conditional specification in the example above is interpreted as

$$[\![\text{if } \pi \text{ then Field() else Type()}]\!]_W^{\text{true}} \stackrel{\text{a.i.}}{\Longrightarrow} [\![\text{Field()}]\!]_E^{\text{true and } \pi} \sqcup_{\mathcal{A}} [\![\text{Type()}]\!]_E^{\text{true and (not } \pi)}$$
$$\text{where } \pi \leftarrow \text{R has Field() and irreducible?(monomial(1, 2)\$R+1)\$R.}$$

We show part of the abstract store for the then-branch:

$$\{\langle \text{\% has `Field(), true and } \pi\rangle, \ldots, \langle \text{\% has `Type(), true and } \pi\rangle\}$$

Note that the condition of the abstract specifications for this branch are not just true: they are the conjunction of the assumption of `ComplexCategory` and the predicate of the conditional specification. This means that the abstract specifications in this branch only hold when both the assumption of `ComplexCategory` holds, and the predicate holds.

We briefly illustrate the interpretation of an extension to a parameterized category using the category extension `FullyLinearlyExplicitRingOver(R)` in the then-

branch of the third conditional specification. The abstract interpretation results in a recursive interpretation over the definition of `FullyLinearlyExplicitRingOver`:

$$[\![[\mathsf{R/X}]\ \ \mathsf{FullyLinearlyExplicitRingOver(X : Ring()) : Category == Type() with\ \ \ldots}\ ]\!]_C^{\mathsf{true}}$$

The formal parameter `X` is renamed to `R`, however, we do not rename the variable `%` in category extensions—this mimics the fact that in the concrete evaluation, the variable $\hat{\%}$ of the current category is passed as the formal parameter to instantiate the category which is to be extended.

We now consider the more interesting category `IntegerModCategory`, given in the introduction; we show the interpretation of the first conditional specification:

$$\begin{aligned}&[\![\mathsf{if\ prime?(k)\ then\ Field()\ else\ Ring()}]\!]_W^{\mathsf{true}}\\ &\quad\overset{\text{a.i.}}{\Longrightarrow} [\![\mathsf{Field()}]\!]_E^{\mathsf{true\ and\ prime?(k)}} \sqcup_{\mathcal{A}} [\![\mathsf{Ring()}]\!]_E^{\mathsf{true\ and\ not\ prime?(k)}}\end{aligned}$$

The abstract stores from each branch of this conditional specification are:

$$\begin{aligned}\sigma_{\mathsf{Field}} &= \{\langle \mathsf{\%\ has\ `Field(),\ true\ and\ prime?(k)}\rangle, \langle \mathsf{\%\ has\ `Ring(),\ true\ and\ prime?(k)}\rangle, \ldots\}\\ \sigma_{\mathsf{Ring}} &= \{\langle \mathsf{\%\ has\ `Ring(),\ true\ and\ not\ prime?(k)}\rangle, \ldots\}\end{aligned}$$

And the final result of the interpretation becomes

$$\begin{aligned}\sigma_{\mathsf{Field}} \sqcup_{\mathcal{A}} \sigma_{\mathsf{Ring}} = \{&\langle \mathsf{\%\ has\ `Field(),\ true\ and\ prime?(k)}\rangle,\\ &\langle \mathsf{\%\ has\ `Ring(),\ (true\ and\ prime?(k))\ or\ (true\ and\ not\ prime?(k))}\rangle, \ldots\}\end{aligned}$$

By applying a Boolean simplification function over the conditions of each abstract specification, we get

$$\sigma_{\mathsf{Field}} \sqcup_{\mathcal{A}} \sigma_{\mathsf{Ring}} = \{\langle \mathsf{\%\ has\ `Field(),\ prime?(k)}\rangle, \langle \mathsf{\%\ has\ `Ring(),\ true}\rangle, \ldots\}$$

which is the result we expect: the category `IntegerModCategory` unconditionally extends category `Ring`, and extends category `Field` when `k` is a prime.

## 3. Expression explosion

Due to the construction of the abstract evaluation the size of the expression of the predicate $\pi$ of an abstract specification $\langle E, \pi\rangle$ grows exponentially in the number of

times a specification $E$ is used within a category, and the depth of category extension. For instance, the category `Type` is involved in a predicate consisting of several hundred terms for the examples given above. Such large predicates can defeat the proposed purpose of the analysis simply by making compilation and static checking too expensive.

Our solution is to simplify the predicates of the abstract specifications at all points in the abstract evaluation. In general, we can use the rules of classical logic to the simplify the predicates — all of the predicates are tautologies. An example of such a rule is the elimination of a disjunction: false or $\pi \Rightarrow \pi$, etc. In general, the use of binary-decision diagrams or other mechanisms, along with boolean-expression simplification can prevent significant expression explosion.

### 4. Reduction of abstract stores

An abstract store can contain abstract specifications which provide information about other abstract specifications. This comes in the form of abstract specifications whose category specification $E$ is part of (or all of) the predicate $\pi$ of some other abstract specification. We use the fact that abstract specifications have information about each other to simplify the predicates of abstract specifications within an abstract store. This process emulates the process of concrete evaluation of predicates for categories. In addition, this mechanism allows the user to add abstract specifications to an abstract store; this can be done to perturb the abstract store. For instance, if the user adds extra abstract specifiations to the abstract store, the reduction mechanism can be used to find the abstract representation of a concrete instance of a category.

Consider, for instance, the following two abstract specifications from the abstract store for the complex commutative ring:

$$\left\{ \begin{array}{l} \langle \%\ \mathsf{has}\ \text{`}\mathsf{Field}(), \text{`}\mathsf{R}\ \mathsf{has}\ \text{`}\mathsf{Field}()\ \mathsf{and}\ \mathit{lookup}(\ldots)\rangle, \\ \langle \text{`}\mathsf{R}\ \mathsf{has}\ \text{`}\mathsf{Field}(), \mathsf{true}\rangle \end{array} \right\}$$

These two abstract specifications reduce to the following specifications:

$$\left\{ \begin{array}{l} \langle \%\ \mathsf{has}\ \text{`}\mathsf{Field}(), \mathit{lookup}(\ldots)\rangle, \\ \langle \text{`}\mathsf{R}\ \mathsf{has}\ \text{`}\mathsf{Ring}(), \mathsf{true}\rangle \end{array} \right\}.$$

In general the reduction function takes each abstract specification $a = \langle E_a, \pi_a \rangle$ in the abstract store and for all $b = \langle E_b, \pi_b \rangle$ if $\pi_b = E_a$ then the function replaces $\pi_b$ with $\pi_a$. The fixed point of this function is taken, which results in the abstract store being fully reduced.

We consider another example taken from the `IntegerModCategory`:

$$\{\langle \langle \%\ \mathsf{has}\ \text{`}\mathsf{IntegerModCategory}(\text{`}\mathsf{k}), \mathsf{true}\rangle, \ldots, \langle \%\ \mathsf{has}\ \text{`}\mathsf{Field}(), \mathsf{prime?}\ \mathsf{k}\rangle\}$$

where the abstract specification $\langle \mathsf{prime?}\ \mathsf{k}, \mathsf{true}\rangle$ is added to the abstract store. Explicitly adding this abstract specification can happen, for instance, in a domain implementing `IntegerModCategory`, under the then-branch of a conditional definition whose condition is prime? k. The fixed-point of the reduction results in the abstract store:

$$\{\langle \langle \%\ \mathsf{has}\ \text{`}\mathsf{IntegerModCategory}(\text{`}\mathsf{k}), \mathsf{true}\rangle, \ldots, \langle \%\ \mathsf{has}\ \text{`}\mathsf{Field}(), \mathsf{true}\rangle\}.$$

That is, all of the specifications of the structure field are added to the category `IntegerModCategory` at this point.

## F. Implementation

In OpenAxiom, Spad has a well-defined library interface to its internal representation. Furthermore, the compiler provides access to this interface during compilation. This

allows extensions to the compiler for static analysis to be written as a library, and not as a patch to the compiler. Our static analysis framework is implemented as a Spad library, and is available on our project website [95]; the provided implementation operates as a stand-alone analysis tool.

A non-trivial problem associated with the analysis is the potential exponential explosion in the size of the predicates associated with each abstract specification. However, the facts generated through our static analysis almost always have the form $(A \wedge B) \vee (A \wedge notB)$ which is equivalent to just $A$. In addition, a large number of the abstract specifications include the predicates true and false — expressions with these predicates are immediately simplified. The result is that most of the expression explosion is mitigated, and not a significant performance barrier.

Another potential problem is the recursive nature of the static analysis: every category will repeatedly analyze the same class of categories. Even more worrisome is the tree-like nature of most categories, where many categories extend the same set of categories, and the same analysis is performed multiple times. We could (but do not) build a cache to memoize the result of previous analyses, which would result in a significant speed up of the static analysis.

We tested our framework on the definition of `ComplexCategory`, and a proposed category for the type of the domain `IntegerMod`. `ComplexCategory` is taken from OpenAxiom's algebra library (the standard library of OpenAxiom). We based `IntegerModCategory` on the algebra's implementation of the domain `IntegerMod`. For both examples, we show only a relevant subset of each result. First, the results for `ComplexCategory`:

```
ComplexCategory := {
  <% has Name(ComplexCategory(R)), true>,
  <% has Name(Field()), R has Field() and_
```

```
        irreducible?(monomial(1,2)$R + 1)$R)>,
    <% has Name(FullyLinearlyExplicitRingOver(R)), true>,
    <% has Sig(exquo(Mapping(%,%,R),Nil), R has IntegralDomain()>,
    <R has Name(CommutativeRing()), true>, ... }
```

The conditions of the abstract specifications have been simplified, reduced, and outputted using a pretty-printer built into the analysis tool.

Next, we show the results for `IntegerModCategory`:

```
IntegerModCategory := {
  <% has Name(IntegerModCategory(k)), true>,
  <% has Name(Ring()), true>,
  <% has Name(Field()), prime?(k)$Integer()>,
  <k has Name(IntegerCategory()), true>, ... }
```

We point out that just as shown in Section 2, the abstract store shown above precisely states that `IntegerModCategory` unconditionally extends `Ring`, which matches with our expectation. The results for both examples presented above have had the conditions of their abstract specifications simplified by a Boolean simplification function.

## G. Related work

The static semantics of conditional categories for the AXIOM family is informally discussed in the "AXIOM book" (Section 11.8 and 12.11) [58], and further explored by Santas [85] where typing rules for conditional categories and domains are specified and implemented. However, the implementation details of conditional categories and domains are still not obvious to compiler writers because of the missing formalization of their dynamic semantics. This chapter specifies the dynamic semantics of conditional categories using a lambda calculus based internal language. Specification of the semantics of conditional domains is still in progress, which we leave for the future work.

Local specialization with generalized predicates can be handled in system Aldor [115], but does not exist in the AXIOM family. We stress that, although the local specialization has been implemented in Aldor before this chapter, the algorithms underlying the implementation have not been demonstrated or justified. Another key contribution of this chapter is a rational and principled implementation of local specialization as an application of abstract interpretation.

The theory of abstract interpretation was studied by Cousot and Cousot [17], first; the development of this novel theory allows the redefinition of the role of a compiler: type-checking and other static analyses become abstract interpretations written as portable library code, instead of patches to the compiler. To the best of our knowledge, our work is the first documented attempt at formal specification of the concrete semantics of conditional categories, and an implementation of an abstract interpretation framework for local specializations in a categorial computer algebra system.

## H. Conclusion

Local specialization has been a well-known technique for computer algebra systems for decades. However, we found its theoretical support to be under-explored. This chapter provides a principled construction of both the dynamic and static behaviors which support local specialization for computer algebra systems. In addition, it provides a formal description for general predicates, increasing the expressivity of algebraic structures written in OpenAxiom. Beginning with an operational semantics for the dynamic behavior of categories we apply the theory of Abstract Interpretation to derive the static semantics of categories. For example, the derived static semantics can be used to discover the visibility of functions for type-checking during compile-time.

We have implemented the abstract interpretation framework in the experimental branch of OpenAxiom. This prototype illustrates the benefits of a principled description of behaviors for the implementation of local specialization in computer algebra systems.

# CHAPTER VI

## CONCLUSION

In the half century of language development and design, as a community, we have discovered and promulgated a large number of language paradigms. Each of these paradigms is made up of numerous (mostly) orthogonal features. These features are re-composable because, since we language designers are naturally good programmers[1], we seek to break down our problems (the paradigm) into re-usable components. Re-usable components cry out for re-combination (in multi-paradigm languages). The result is a surplus of combinations of features with very little *guidance* to the use of those features to actually make software, nor confidence that these features work as advertised.

Since the writing of most of the (papers contributing to the) chapters of this dissertation both Spad and C++ have continued to evolve. Particularly notable to this dissertation is that Spad has been forked: this has led to the (in my opinion) far superior implementation of the AXIOM System and Spad language: Open-Axiom. This allowed me to re-implement Chapter IV in a fraction of the time and a fraction of the number of lines code. Furthermore, Chapter V was made possible both because of our desire to provide a more rational basis for the description of the features of Spad, and also because of certain new features (reflection) that have rapidly matured in Open-Axiom. C++ has also continued to evolve; the concepts proposal has been 'deferred' [101, 39] until a later date. (Other new features have been voted in: anonymous functions, variadic templates, an expanded standard library, etc.)

The main thesis of this dissertation is that, in many ways, the capabilities pro-

[1]Given a rather large grain of salt.

vided by modern languages have far outreached our understanding of how to deploy those capabilities, gainfully. We have a large set of language features already available to us, as programmers, and that set of features continues to expand. These language features were added to support certain mental frameworks for programming, for instance: classes, virtual functions (and inheritance), member functions for object-oriented programming; or, templates, associated types, and compile time values to support parametric polymorphism. In languages like C++ and Spad, these features can be used in one fashion to support the programming paradigm they are intended for, but like any well-written library, the features can be combined and recombined. The result is that we are constantly discovering new ways (paradigms) of programming.[2]

When we combine old features into new paradigms, we inevitably find places where it is inconvenient or inefficient to express our intention. Since programmers (and especially language designers) are an especially lazy breed, we inevitably propose yet another language feature to patch this difficulty.

Regardless of the status of individual language features, an important and central tenet of any kind of programming remains that abstraction allows more sophisticated software. applies to language design: we can have easy-to-build compilers (with easy-to-understand diagnostics), high-performance and efficient run-times, or abstraction and expressivity. Thus, an assembler is (relatively) easy to build compiler, it provides fairly high performance, but assembly lacks significantly in terms of abstraction capabilities. The typical trade-off made is to increase the abstraction capabilities and expressivity of a language at the cost of performance, i.e., Lisp or Python. Languages which support generic programming tend to take the "opposite" course: high-levels

[2]A terribly unscientific poll is to note that as of September 2009, Wikipedia lists a few dozen programming paradigms.

of abstraction and expressivity, with high levels of performance. The results, for example in C++, are spectacular: error messages (see the footnote in Section III.1) of legendary length; and compilers which literally take decades to implement.

In this dissertation I hope that I have shown ways of synthesizing and completing (rationalizing) features for efficient abstraction. Dr. Veldhuizen's discussion of *parsimony* [105] is a perfect framework to mount the discussion from Chapter II: it is important to provide as many (and varied) entry points into a library as possible. As library writers we should cater to our users, and not to the minimal level functionality.[3] Chapter III shows idioms and provides a mental framework for combining libraries, and how the act of library composition should be a programming construct that in and of itself should be a re-usable component. Chapters IV and V cast light on how to build robust library-directed extensions for languages and their compilers.

There are clear directions left for this line of research. Perhaps the most tangible direction is the full enumeration of a type- and deduction- system using the theory abstract interpretation for Open-Axiom's Spad. While most of the Spad language, itself, is not covered in the construction given in the dissertation (Chapter V), the actual difficulty lay merely in the dimensions of size and time, and not complexity. For C++ there are still clearly unresolved issues in the understanding of the basic features needed for generic programming. For instance, C++'s use of specialization with generic programming functions is not well-characterized. And, even if generic programming functions were well-understood, the guarantees they give are razor thin: monotonically non-decreasing improvements in "performance." Yet, C++ has shown (consider the MTL4 vs. GOTO BLAS in Chapter III) that with just those tools it is

[3] I am currently engaged in a project to provide a typeful enumeration of a complex instruction set computer — *each* assembly instruction is represented by several hundred high-level signatures.

possible to dramatically outperform even the most sophisticated optimizing compiler, or low-level manual assembler.

# REFERENCES

[1] D. Abrahams, A. Gurtovoy, C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond, Addison-Wesley, 2004.

[2] P.D. Adams, R.W. Grosse-Kunstleve, L.W. Hung, T.R. Ioerger, A.J. McCoy, N.W. Moriarty, R.J. Read, J.C. Sacchettini, N.K. Sauter, T.C. Terwilliger, PHENIX: building new software for automated crystallographic structure determination, Acta Crystallographica Section D 58 (2002) 1948–1954.

[3] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, L. Rauchwerger, STAPL: An adaptive, generic parallel C++ library, in: Languages and Compilers for Parallel Computing, volume 2624 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 193–208.

[4] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J.D. Croz, S. Hammarling, J. Demmel, C. Bischof, D. Sorensen, LAPACK: A portable linear algebra library for high-performance computers, Proceedings of Supercomputing '90 (1990) 2–11.

[5] M.H. Austern, Generic programming and the STL: Using and extending the C++ Standard Template Library, Professional Computing Series, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[6] G. Baumgartner, M. Jansche, K. Läufer, Half & Half: Multiple Dispatch and Retroactive Abstraction for Java, Technical Report OSU-CISRC-5/01-TR08, Ohio State University, 2002.

[7] R.E. Bellman, On a routing problem, Quart. Appl. Math. 16 (1958) 87–90.

[8] C. Bischof, A. Carle, G. Corliss, A. Griewank, ADIFOR: Automatic differentiation in a source translator environment, in: ISSAC '92: Papers from the International Symposium on Symbolic and Algebraic Computation, New York, NY, USA, 1992, ACM Press, pp. 294–302.

[9] C.H. Bischof, L. Roh, A.J. Mauer-Oats, ADIC: an extensible automatic differentiation tool for ANSI-C, Software—Practice and Experience 27 (1997) 1427–1456.

[10] L. Bourdev, H. Jin, Generic Image Library, 2006. opensource.adobe.com/gil.

[11] A. Breuer, P. Gottschling, D. Gregor, A. Lumsdaine, Effecting parallel graph eigensolvers through library composition, in: Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL), in Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006), Los Alamitos, CA, USA, 2006, IEEE Computer Society, p. 466. http://doi.ieeecomputersociety.org/10.1109/IPDPS.2006.1639723.

[12] P.A. Broadbery, T. Gómez-Díaz, S.M. Watt, On the implementation of dynamic evaluation, in: ISSAC '95: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation, New York, NY, USA, 1995, ACM, pp. 77–84.

[13] A.T. Brünger, X-Plor Version 3.1: A System for X-Ray Crystallography and NMR, Yale University Press, 1993.

[14] M.M.T. Chakravarty, G. Keller, S. Peyton Jones, Associated type synonyms, in: ICFP '05: Proceedings of the International Conference on Functional Programming, New York, NY, USA, 2005, ACM Press, pp. 241–253.

[15] M.M.T. Chakravarty, G. Keller, S. Peyton Jones, S. Marlow, Associated types with class, in: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, 2005, ACM Press, pp. 1–13.

[16] P. Cousot, Types as abstract interpretations, invited paper, in: Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, 1997, ACM Press, New York, NY, pp. 316–331.

[17] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of programming languages, New York, NY, USA, 1977, ACM Press, pp. 238–252.

[18] K. Cowtan, The clipper project, Joint CCP4 and ESF-EACBM Newsletter on Protein Crystallography 40 (2002).

[19] K. Crary, S. Weirich, G. Morrisett, Intensional polymorphism in type-erasure semantics, J. Funct. Program. 12 (2002) 567–600.

[20] K. Czarnecki, U.W. Eisenecker, Generative Programming, Addison-Wesley, 2000.

[21] T. Daly, Axiom Volume 1: Tutorial, Lulu.com, 2005. ISBN: 978-1-4116-6597-2.

[22] J.H. Davenport, P. Gianni, B.M. Trager, Scratchpad's view of algebra II: A categorical view of factorization, New York, NY, USA, 1991, ACM Press, pp. 32–38.

[23] J.H. Davenport, B.M. Trager, Scratchpad's view of algebra I: Basic commutative algebra, in: DISCO '90: Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems, London, UK, 1990, Springer-Verlag, pp. 40–54.

[24] S. Dickinson, M. Pelillo, R. Zabih, Introduction to the special section on graph algorithms in computer vision, IEEE Transactions on Pattern Analysis and Machine Intelligence 23 (2001) 1049–1052.

[25] G. Dos Reis, Open-Axiom: The open scientific computation platform, http://open-axiom.org, 2009.

[26] ECMA-334, C# language specification, 2006. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

[27] C. Ehresmann, Les prolongements d'une variété différentiable, C.R. Acad. Sc. Paris 233 (1951) 598–600.

[28] A. Fabri, G.J. Giezeman, L. Kettner, S. Schirra, S. Schönherr, On the design of CGAL, a computational geometry algorithms library, Software – Practice and Experience 30 (2000) 1167–1202. Special Issue on Discrete Algorithm Engineering.

[29] P.F. Felzenszwalb, D.P. Huttenlocher, Efficient graph-based image segmentation, Int. J. Comput. Vision 59 (2004) 167–181.

[30] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[31] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, J. Willcock, An extended comparative study of language support for generic programming, Journal of Functional Programming 17 (2007) 145–205.

[32] K. Gopal, R. Pai, T.R. Ioerger, T. Romo, J.C. Sacchettini, TEXTAL: Artificial intelligence techniques for automated protein structure determination, Proceedings of the 15th Conference on Innovative Applications in Artificial Intelligence (IAAI) (2003) 93–100.

[33] K. Gopal, T. Romo, E. Mckee, K. Childs, L. Kanbi, R. Pai, J. Smith, J. Sacchettini, T. Ioerger, TEXTAL: Automated crystallographic protein structure determination, Proceedings of the Seventeenth Conference on Innovative Applications of Artificial Intelligence (2005) 1483–1490.

[34] K. Goto, R.A. van de Geijn, Anatomy of high-performance matrix multiplication, ACM Trans. Math. Softw. 34 (2008) 1–25.

[35] P. Gottschling, A. Lumsdaine, The Matrix Template Library 4, http://www.osl.iu.edu/research/mtl/mtl4/, 2008.

[36] P. Gottschling, D.S. Wise, A. Joshi, Generic support of algorithmic and structural recursion for scientific computing, The International Journal of Parallel, Emergent and Distributed Systems 0 (2008) 1–23. www.informaworld.com.

[37] D. Gregor, High-level static analysis for generic libraries, PhD thesis, Rensselaer Polytechnic Institute, 2004. http://www.osl.iu.edu/publications/prints/2004/Gregor-Thesis.pdf.

[38] D. Gregor, ConceptGCC: Concept extensions for C++, http://www.generic-programming.org/software/ConceptGCC, 2005.

[39] D. Gregor, What happened in frankfurt?, 2009. http://cpp-next.com/archive/2009/08/what-happened-in-frankfurt/.

[40] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G.D. Reis, A. Lumsdaine, Concepts: Linguistic support for generic programming in C++, in: OOPSLA '06: Proceedings of the 2006 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2006, ACM Press, pp. 291–310.

[41] D. Gregor, A. Lumsdaine, Lifting sequential graph algorithms for distributed-memory parallel computation, in: OOPSLA '05: Proceedings of the 2005 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 423–437.

[42] D. Gregor, M. Marcus, T. Witt, A. Lumsdaine, Foundational Concepts for the C++0x Standard Library, Technical Report N2677=08-0187, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming language C++, 2008. www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2677.pdf.

[43] D. Gregor, J. Siek, Implementing Concepts, Technical Report N1848=05-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2005. www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1848.pdf.

[44] D. Gregor, B. Stroustrup, J. Widman, J. Siek, Proposed Wording for Concepts (Revision 8), Technical Report N2741=08-025, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2008.

[45] A. Griewank, Evaluating derivatives: principles and techniques of algorithmic differentiation, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[46] A. Griewank, D. Juedes, J. Utke, Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++, ACM Trans. Math. Softw. 22 (1996) 131–167.

[47] R.W. Grosse-Kunstleve, N.K. Sauter, N.W. Moriarty, P.D. Adams, The Computational Crystallography Toolbox: crystallographic algorithms in a reusable software framework, J. Appl. Cryst. 35 (2002) 126–136.

[48] T. Hahn, International Tables for Crystallography, Volume A: Space Group Symmetry, Spring, 2002.

[49] T.R. Holton, T.R. Ioerger, J.A. Christopher, J.C. Sacchettini, TEXTAL: A pattern recognition system for interpreting electron density maps, in: Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology (ISMB), pp. 130–137.

[50] IBM Research, Subject-oriented programming and the adapter pattern, IBM Research, 2008. www.research.ibm.com/sop/sopcadap.htm.

[51] International Organization for Standardization, ISO/IEC 14882:2003: Programming languages: C++, ISO, Geneva, Switzerland, 2nd edition, 2003. http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110.

[52] International Standards Organization, Technical Report on C++ performance, Technical Report N1487=03-0070, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2003.

[53] T.R. Ioerger, J.C. Sacchettini, TEXTAL system: Artificial intelligence tech-

niques for automated protein model building, Methods in Enzymology 374 (2003) 244–270.

[54] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek, Algorithm specialization in generic programming: challenges of constrained generics in C++, in: PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, 2006, ACM Press, pp. 272–282.

[55] J. Järvi, M.A. Marcus, J.N. Smith, Library composition and adaptation using c++ concepts, in: GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, New York, NY, USA, 2007, ACM, pp. 73–82.

[56] J. Järvi, M.A. Marcus, J.N. Smith, Programming with C++ concepts, Science of Computer Programming (2009).

[57] J. Järvi, J. Willcock, A. Lumsdaine, Concept-controlled polymorphism, in: F. Pfennig, Y. Smaragdakis (Eds.), GPCE '03: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering, volume 2830 of *LNCS*, Erfurt, Germany, 2003, Springer Verlag, pp. 228–244.

[58] R.D. Jenks, R.S. Sutor, AXIOM: The Scientific Computation System, Springer-Verlag, 1992.

[59] M.P. Jones, Type classes with functional dependencies, in: ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems, volume 1782 of *Lecture Notes in Computer Science*, New York, NY, 2000, Springer-Verlag, pp. 230–244.

[60] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, An overview of AspectJ, in: J.L. Knudsen (Ed.), Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01), volume 2072 of *Lecture Notes in Computer Science*, Springer-Verlag, London, UK, 2001, pp. 327–353.

[61] R. Lämmel, K. Ostermann, Software extension and integration with type classes, in: GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, New York, NY, USA, 2006, ACM Press, pp. 161–170.

[62] K. Läufer, G. Baumgartner, V.F. Russo, Safe structural conformance for Java, The Computer Journal 43 (2000) 469–481.

[63] K. Läufer, M. Odersky, Polymorphic type inference and abstract data types, ACM Transactions on Programming Languages and Systems 16 (1994) 1411–1430.

[64] C.L. Lawson, R.J. Hanson, R.J. Kincaid, F.T. Krogh, Basic linear algebra subprograms for FORTRAN usage, ACM Transactions on Mathematical Software 5 (1979) 308–323.

[65] D. Lea, Public conversation, 2006. Workshop of Library-Centric Software Design at OOPSLA'06, Portland Oregon.

[66] R. Ley-Wild, U.A. Acar, M. Fluet, A cost semantics for self-adjusting computation, in: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, 2009, ACM, pp. 186–199.

[67] V. Litvinov, Constraint-based polymorphism in Cecil: towards a practical and static type system, in: OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, New York, NY, USA, 1998, ACM Press, pp. 388–411.

[68] U. von Luxburg, A tutorial on spectral clustering, 2006. Technical Report No. TR-149; http://www.kyb.mpg.de/publications/attachments/Luxburg06_TR_\%5B0\%5D.pdf.

[69] B. Magnusson, Code reuse considered harmful, Journal of Object-Oriented Programming 4 (1991) 8.

[70] M. Marcus, J. Järvi, S. Parent, Runtime polymorphic generic programming—mixing objects and concepts in ConceptC++, in: K. Davis, J. Striegnitz (Eds.), Multiparadigm Programming 2007: Proceedings of the MPOOL Workshop at ECOOP '07, Berlin, Germany. homepages.fh-regensburg.de/~mpool/.

[71] M. Mattsson, J. Bosch, M.E. Fayad, Framework integration problems, causes, solutions, Commun. ACM 42 (1999) 80–87.

[72] D. Mcilroy, Mass-produced software components, in: Proceedings of the 1st International Conference on Software Engineering sponsored by the NATO Science Committee, Garmisch Pattenkirchen, Germany, Brussels, Belgium, 1969, Scientific Affairs Division, NATO, pp. 138–155.

[73] B. McNamara, Y. Smaragdakis, Static interfaces in C++, in: First Workshop on C++ Template Programming, Erfurt, Germany. oonumerics.org/tmpw00/.

[74] K. Mehlhorn, S. Näher, The LEDA Platform of Combinatorial and Geometric Computing, Cambridge University Press, 1999.

[75] N. Mitchell, G. Sevitsky, H. Srinivasan, The diary of a datum: An approach to modeling runtime complexity in framework-based applications, in: Proceedings of the First International Workshop of Library-Centric Software Design (LCSD '05). An OOPSLA '05 workshop, San Diego, CA, USA. As technical report 06-12 of Rensselaer Polytechnic Institute, Computer Science Department.

[76] M.B. Monagan, W.M. Neuenschwander, GRADIENT: algorithmic differentiation in Maple, in: ISSAC '93: Proceedings of the 1993 International Symposium on Symbolic and Algebraic Computation, New York, NY, USA, 1993, ACM Press, pp. 68–76.

[77] N. Myers, A New and Useful Template Technique: "Traits", volume 7, SIGS Publications, Inc., 1995.

[78] M. Odersky, Poor man's type classes, Presentation at the meeting of IFIP WG 2.8, Functional Programming, 2006. lamp.epfl.ch/~odersky/talks/wg2.8-boston06.pdf.

[79] M. Odersky, The Scala language specification: Version 2.0, draft march 17, 2006, http://scala.epfl.ch/docu/files/ScalaReference.pdf, 2006.

[80] S. Peyton Jones, M. Jones, E. Meijer, Type classes: an exploration of the design space, in: Proceedings of the Second Haskell Workshop of the ICPF '97, New York, NY, USA, 1997, ACM Press. citeseer.ist.psu.edu/peytonjones97type.html.

[81] W.R. Pitt, M.A. Williams, M. Steven, B. Sweeney, A.J. Bleasby, D.S. Moss, The Bioinformatics Template Library–generic components for biocomputing, Bioinformatics 17 (2001) 729–737. http://bioinformatics.oupjournals.org/cgi/content/abstract/17/8/729.

[82] Princeton Satellite Systems, Matrixlib: BLAS and LAPACK, Matrix library API overview, 2008. http://www.psatellite.com/matrixlib/api/lapack.html.

[83] G. Rhodes, Crystallography Made Crystal Clear, Third Edition : A Guide for Users of Macromolecular Models, Academic Press, 2006.

[84] M. Rosendahl, Automatic complexity analysis, in: FPCA '89: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, New York, NY, USA, 1989, ACM, pp. 144–156.

[85] P.S. Santas, Conditional categories and domains, in: Design and Implementation of Symbolic Computation Systems, volume 1128, 1996, Springer Berlin / Heidelberg, pp. 112–125.

[86] G.E. Schalnat, A. Dilger, G. Randers-Perhson, C. Truta, S.P. Cadieux, E.S. Raymond, G. Vollant, T. Lange, W.V. Schaik, J. Bowler, K. Bracey, S. Bushell, M. Holmgren, G. Roelofs, T. Tanner, D. Martindale, P. Schmidt, T. Wegner, libpng home page, Website, 2009. http://www.libpng.org/pub/png/libpng.html.

[87] U. Shani, Filling regions in binary raster images: A graph-theoretic approach, SIGGRAPH Computer Graphics 14 (1980) 321–327.

[88] J. Shi, J. Malik, Normalized cuts and image segmentation, Transactions on Pattern Analysis and Machine Intelligence 22 (2000) 888–905.

[89] J. Siek, L.Q. Lee, A. Lumsdaine, The Boost Graph Library: User Guide and Reference Manual, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[90] J. Siek, A. Lumsdaine, The Matrix Template Library: Generic components for high-performance scientific computing, Computing in Science and Engineering 1 (1999) 70–78. http://dx.doi.org/10.1109/5992.805137.

[91] J. Siek, A. Lumsdaine, Concept checking: Binding parametric polymorphism in C++, in: First Workshop on C++ Template Programming. oonumerics.org/tmpw00/.

[92] J. Siek, A. Lumsdaine, L.Q. Lee, Boost Graph Library, Boost, 2001. www.boost.org/libs/graph.

[93] Silicon Graphics, Inc., SGI Implementation of the Standard Template Library, 2004. http://www.sgi.com/tech/stl/.

[94] J. Smith, G. Dos Reis, J. Järvi, Algorithmic differentiation in Axiom, in: ISSAC '07: Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation, New York, NY, USA, 2007, ACM, pp. 347–354.

[95] J. Smith, Y. Li, Y. Solodkyy, G. Dos Reis, J. Järvi, Local Specialization in Open-Axiom, Technical Report, Texas A&M University, 2009. http://parasol.cs.tamu.edu/groups/pttlgroup/local-specialization/.

[96] B. Speelpenning, Compiling fast partial derivatives of functions given by algorithms, PhD thesis, The University of Illinois at Urbana-Champaigne, 1980.

[97] A. Stepanov, M. Lee, The Standard Template Library, Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, 1994. www.hpl.hp.com/techreports.

[98] J.E. Stoy, Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory, The MIT Press, 1977.

[99] C. Strachey, Fundamental concepts in programming languages, Higher Order Symbol. Comput. 13 (2000) 11–49.

[100] B. Stroustrup, The C++ Programming Language (Third Edition and Special Edition), Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[101] B. Stroustrup, The C++0x "remove concepts" decision, Dr. Dobb's Journal (2009).

[102] N. Su, Java 2 platform standard ed. 5.0 API, 2004. http://java.sun.com/j2se/1.5.0/docs/api/.

[103] C. Szyperski, Component Software: Beyond Object-Oriented Programming, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[104] M. Troyer, P. Dayal, R. Villiger, The Iterative Eigensolver Template Library, 2004. www.comp-phys.org/software/ietl/.

[105] T. Veldhuizen, Parsimony principles for software components and metalanguages, in: GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, New York, NY, USA, 2007, ACM, pp. 115–122.

[106] T. Veldhuizen, M. Jernigan, Will C++ be faster than FORTRAN, in: Volume 1343, Springer Verlag KG, 1997, pp. 49–56.

[107] T.L. Veldhuizen, Expression templates, j-C-PLUS-PLUS-REPORT 7 (1995) 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.

[108] T.L. Veldhuizen, Arrays in Blitz++, in: ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments, London, UK, 1998, Springer-Verlag, pp. 223–230.

[109] T.L. Veldhuizen, Active libraries and universal languages, PhD thesis, Indiana University Computer Science, 2004. http://osl.iu.edu/~tveldhui/papers/2004/dissertation.pdf.

[110] T.L. Veldhuizen, D. Gannon, Active Libraries: Rethinking the roles of compilers and libraries, Technical Report, University of Waterloo, 1998. http://ubiety.uwaterloo.ca/~tveldhui/papers/oo98.html.

[111] D. Villard, M.B. Monagan, ADrien: an implementation of automatic differentiation in Maple, in: ISSAC '99: Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, New York, NY, USA, 1999, ACM Press, pp. 221–228.

[112] P. Wadler, S. Blott, How to make ad-hoc polymorphism less ad-hoc, in: ACM Symposium on Principles of Programming Languages, 1989, ACM, pp. 60–76.

[113] J. Walter, M. Koch, Boost Basic Linear Algebra, C++ Boost, 2002. http://www.boost.org/doc/libs/1_38_0/libs/numeric/ublas/doc/index.htm.

[114] S.M. Watt, P.A. Broadbery, S.S. Dooley, P. Iglio, S.C. Morrison, J.M. Steinbach, R.S. Sutor, A first report on the A# compiler, in: ISSAC '94: Proceedings of the International Symposium on Symbolic and Algebraic Computation, New York, NY, USA, 1994, ACM, pp. 25–31.

[115] S.M. Watt, P.A. Broadbery, S.S. Dooley, P. Iglio, S.C. Morrison, J.M. Steinbach, R.S. Sutor, Aldor User Guide, Aldor, 2000. http://www.aldor.org.

[116] R.C. Whaley, J. Dongarra, Automatically Tuned Linear Algebra Software, in: SuperComputing 1998: High Performance and Network Computing.

[117] J. Willcock, J. Järvi, A. Lumsdaine, Active libraries vs. separate compilation, 2004. Lecture Slides.

# VITA


| | |
|---|---|
| Name: | Jacob Nyffeler Smith |
| Address: | Järvi-Labs |
| | Dept. of Computer Science and Engineering |
| | Texas A&M University |
| | College Station, TX 77843-3112 |
| Email Address: | jacob.nyffeler.smith@gmail.com |
| Education: | B.S., Mathematics, The University of Texas at Austin, 2001 |
| | B.A., Plan II, The University of Texas at Austin, 2001 |


This document was typeset in LaTeX by Jacob Nyffeler Smith.