A LANGUAGE FOR GENERIC PROGRAMMING

Jeremy G. Siek

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University
August, 2005

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Andrew Lumsdaine, Ph.D.

R. Kent Dybvig, Ph.D.

Daniel P. Friedman, Ph.D.

Steven D. Johnson, Ph.D.

Amr Sabry, Ph.D.

August, 2005

This dissertation is dedicated to my wonderful wife Katie who was next to me every step of this journey.

Acknowledgements

First and foremost I thank my parents for all their love and for teaching me to enjoy learning. I especially thank my wife Katie for her support and understanding through this long and sometimes stressful process. I also thank Katie for insisting on good error messages for $\mathcal{G}$! My advisor, Andrew Lumsdaine, deserves many thanks for his support and guidance and for keeping the faith as I undertook this long journey away from scientific computing and into the field of programming languages. I thank my thesis committee: R. Kent Dybvig, Daniel P. Friedman, Steven D. Johnson, and Amr Sabry for their advice and encouragement. A special thanks goes to Ronald Garcia, Christopher Mueller, and Douglas Gregor for carefully editing and catching the many many times when I accidentally skipped over the important stuff. Thanks to Jaakko and Jeremiah for hours of stimulating discussions and arguments concerning separate compilation and concept-based overloading. Thanks to David Abrahams for countless hours spent debating the merits of one design over another while jogging through the hinterlands of Norway. Thanks to Alexander Stepanov and David Musser for getting all this started, and thank you for the encouragement over the years. Thanks to Matthew Austern, his book *Generic Programming in the STL* was both an inspiration and an invaluable reference. Thanks to Beman Dawes and everyone involved with the Boost libraries. The collective experience from Boost was vital in the creation of this thesis. Thanks to Vincent Cremet and Martin Odersky for answering questions about Scala and virtual types.

Abstract

The past decade of software library construction has demonstrated that the discipline of generic programming is an effective approach to the design and implementation of large-scale software libraries. At the heart of generic programming is a semi-formal interface specification language for generic components. Many programming languages have features for describing interfaces, but none of them match the generic programming specification language, and none are as suitable for specifying generic components. This lack of language support impedes the current practice of generic programming. In this dissertation I present and evaluate the design of a new programming language, named $\mathcal{G}$ (for generic), that integrates the generic programming specification language with the type system and features of a full programming language. The design of $\mathcal{G}$ is based on my experiences, and those of colleagues, in the construction of generic libraries over the past decade. The design space for programming languages is large, thus this experience is vital in guiding choices among the many tradeoffs. The design of $\mathcal{G}$ emphasizes modularity because generic programming is inherently about composing separately developed components. In this dissertation I demonstrate that the design is implementable by constructing a compiler for $\mathcal{G}$ (translating to C++) and show the suitability of $\mathcal{G}$ for generic programming with prototypes of the Standard Template Library and the Boost Graph Library in $\mathcal{G}$. I formalize the essential features of $\mathcal{G}$ in a small language and prove type soundness.

Contents

Chapter 1. Introduction 1

1.1. Lowering the cost of developing generic components 5

1.2. Lowering the cost of reusing generic components 7

1.3. $\mathcal{G}$: a language for generic programming 9

1.4. Related work in programming language research 10

1.5. Claims and evaluation 11

1.6. Road map 12

Chapter 2. Generic programming and the STL 13

2.1. An example of generic programming 15

2.2. Survey of generic programming in the STL 21

2.3. Relation to other methodologies 49

2.4. Summary 54

Chapter 3. The language design space for generics 55

3.1. Preliminary design choices 55

3.2. Subtyping versus type parameterization 59

3.3. Parametric versus macro-like type parameterization 65

3.4. Concepts: organizing type requirements 73

3.5. Nominal versus structural conformance 83

3.6. Constrained polymorphism 85

3.7. Summary 89

Chapter 4. The design of $\mathcal{G}$ 90

4.1. Generic functions 92

4.2. Concepts 95

4.3. Models 96

4.4. Modules 98

4.5. Type equality 98

4.6. Function application and implicit instantiation 101

4.7. Function overloading and concept-based overloading 108

4.8. Generic user-defined types 110

4.9. Function expressions 112

4.10. Summary 114

Chapter 5. The definition and compilation of $\mathcal{G}$ 118

5.1. Overview of the translation to C++ 120

5.2. A definitional compiler for $\mathcal{G}$ 133

5.3. Compiler implementation details 160

5.4. Summary 162

Chapter 6. Case studies: generic libraries in $\mathcal{G}$ 163

6.1. The Standard Template Library 164

6.2. The Boost Graph Library 174

6.3. Summary 180

Chapter 7. Type Safety of $F^{\mathcal{G}}$ 184

7.1. $F^{\mathcal{G}}$ = System F + concepts, models, and constraints 185

7.2. Translation of $F^{\mathcal{G}}$ to System F 192

7.3. Isabelle/Isar formalization 201

7.4. Associated types and same-type constraints 204

7.5. Summary 213

Chapter 8. Conclusion 215

Appendix A. Grammar of $\mathcal{G}$ 219

A.1. Type expressions 220

A.2. Declarations 221

A.3. Statements and expressions 221

A.4. Derived forms 223

Appendix B. Definition of $\mathrm{F}^{\mathcal{G}}$ 224

Appendix. Bibliography 228

Appendix. Index 247

Software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors...
One could not stock 300 sine routines unless they were all in some sense instances of just a few models, highly parameterized, in which all but a few parameters were intended to be permanently bound before run time. One might call these early-bound parameters 'sale time' parameters...
Choice of Data structures... this delicate matter requires careful planning so that algorithms be as insensitive to changes of data structure as possible. When radically different structures are useful for similar problems (e.g., incidence matrix and list representations for graphs), several algorithms may be required.

M. Douglas McIlroy, 1969 [126]

1 Introduction

A decade or two ago computers were primarily the tools of specialists and the toys of hobbyists. Now they are a part of everyday life: they are used to create the family photo albums, make travel reservations, communicate with friends, and get directions for a trip. Despite the advances in computer science and software engineering, computers still must be told what to do in excruciating detail. Thus, the production of software to control our computers is an important endeavor, one that affects more and more aspects of our lives.

Producing software is hard: massive amounts of time and money go into creating the software applications we use today. This cost affects the prices we pay for shrink wrapped software and factors into the prices of many other goods and services. Further, software

quality affects our lives: buggy software is a constant annoyance and software bugs sometimes cause or contribute to more serious harm

The 1968 NATO Conference on Software Engineering popularized the terms "software crisis" and "software engineering". The crisis they faced was widespread difficulties in the construction of large software systems such as IBM's OS/360 and the SABRE airline reservation system [64, 154]. The conference attendees felt it was time for programmers and managers to get more serious about the process of producing software. McIlroy gave an invited talk entitled *Mass-produced Software Components* [126]. In this talk he proposed the systematic creation of reusable software components as a solution to the software crisis. The idea was that most software products are created from building blocks that are quite similar, so software productivity would be increased if a standard set of blocks could be shared among many software products.

Barnes and Bollinger define a simple equation that summarizes the savings that can be achieved through software reuse [15]. Let D stand for the cost of developing a reusable component and n be the number of uses of the component. The savings is calculated by:

$$\left(\sum_{i=1}^{n}(C_i - R_i)\right) - D \tag{1}$$

where C_i is the cost of writing code from scratch to solve a problem and R_i is the cost of reusing the component. A particularly interesting aspect of this equation is that if $C_i > R_i$, then as n tends to infinity so does the savings from reuse. On the other hand, if n is small, then the benefits of reuse may be outweighed by the initial investment D of developing the reusable component. Studies by Margono and Rhoads have shown that a typical value for D is twice the cost of building a non-reusable version of the component [125].

In addition to the savings in software production, reuse can increase software quality. One of the reasons given by Lim [116] is that the more a piece of software is used, the faster the bugs in the software are found and fixed. Further, the bugs need only be fixed in one place, in the reusable component, and then all uses of the component benefit from the increase in quality.

Today we are starting to see the benefits of software reuse: Douglas McIlroy's vision is gradually becoming a reality. The number of commercial and open source software component libraries has steadily grown and it is commonplace for application builders to turn to libraries for user-interface components, database access, report creation, numerical routines, and network communication, to name a few. In addition, many software companies have benefited from the creation of in-house domain-specific libraries which they use to support entire software product lines. The software product lines approach is described by Clements and Northrop in [46]. One of the strengths of the Java language is its large suite of standard libraries developed by Sun Microsystems. Software libraries have also seen particularly heavy use in scripting languages such as Visual Basic, Perl, Python, and PHP, and for a long time there has been considerable library building activity in C, C++, and Fortran for systems-level and performance-oriented domains. There is also a growing number of libraries available for research languages such as Objective Caml and Haskell.

As the field of software engineering progresses, we learn better techniques for building reusable software. In 1994, Stepanov and Lee [181] presented a library of sequential algorithms and data structures to the C++ standards committee that was immediately recognized as a leap forward in library design. The Standard Template Library (STL), as it was called, was the product of a methodology called *Generic Programming* developed during the 1980's by Stepanov, Musser, and colleagues [103–105, 137–139, 179]. The term "generic programming" is often used to mean any use of "generics", i.e., any use of parametric polymorphism or templates. The term is also used in the functional programming community for function generation based on algebraic datatypes, i.e., "polytypic programming". This thesis uses the term "generic programming" solely in the sense of Stepanov and Musser.

The main idea behind generic programming is the separation of algorithms from data-structures via abstractions that respect efficiency. For example, instead of writing functions on arrays we write generic functions implemented in terms of abstract iterators. The iterator abstraction can be implemented in terms of arrays, linked-lists, and many other data-structures that represent sequences of elements. The advantage of generic programming is that it greatly increases the number of situations in which a component may be used,

thereby increasing n in Equation 1. Generic programming accomplishes this by making components more general while retaining the efficiency of specialized components. Chapter 2 describes how this is done.

The STL was accepted as part of the C++ Standard Library [86] thereby introducing generic programming to mainstream programmers. Since 1994 generic programming has been successfully applied in domains such as computer vision [108], computational geometry [21], bioinformatics [152], geostatistics [156], physics [190], text processing [55, 122], numerical linear algebra [174, 198], graph theory [113, 169], and operations research [12].

My interest in generic programming began in 1998, with work on the Matrix Template Library [166, 174] with Andrew Lumsdaine and Lie-Quan Lee, building on earlier work by Andrew Lumsdaine and Brian McCandless [120, 121]. We were successful in producing numerical routines that could compete with Fortran codes in terms of performance and that offered greater functionality and flexibility. In 1999, motivated by the need for sparse matrix reordering algorithms, we turned our attention to graph theory and developed a library of generic graph algorithms and data structures [113]. With this library we exceeded the expectations expressed by McIlroy in the quote at the beginning of this chapter: we implemented algorithms that were insensitive to whether an incidence matrix or list representation is used to represent graphs.

In 2000 we began collaborating with the Boost open source community [22] and our graph library evolved into the Boost Graph Library (BGL) [169]. Boost is an on-line community founded by members of the C++ standards committee to foster the development of modern C++ libraries with an emphasis on generic programming. The Boost library collection currently contains 65 peer reviewed libraries (it is continuously growing) and there were over 90,000 downloads of the latest release. The C++ Standards Committee is expanding the C++ standard library with the publication of a technical report on C++ library extensions [10]. Most of the libraries in that report started as Boost libraries.

I found the construction and maintenance of generic libraries in C++ to be both rewarding and frustrating. It was rewarding because we were able to deliver highly reusable and

efficient software and received positive feedback from users. On the other hand, it was frustrating because constructing libraries in C++ was difficult and the resulting libraries were not as easy to use or as robust in the face of user error as we would have hoped. The methodology of generic programming is effective, and while C++ provides good support for generic programming, it is not the ideal language for this purpose. In terms of Equation 1, *both the cost of developing reusable components and the cost of reusing a component were higher than they should be, thereby reducing the savings from reuse.*

Our frustration with C++ motivated several of us at the Open Systems Lab to study to what extent other programming languages support generic programming. In 2003 we analyzed six programming languages: C++, Standard ML, Haskell, Eiffel, Java, and C#. We implemented a subset of the BGL in each of these languages and then evaluated them with respect to how straightforward it was to express and use the BGL algorithms and abstractions [69]. Since then we have evaluated several more languages, including Cecil and Objective Caml [70]. All of these languages provide some support for generic programming but none is ideal.

Given the state of the art in programming languages, it is time to incorporate what we have learned from the past decade of generic library construction back into the design of programming languages. In this dissertation, I present and evaluate the design of a language named $\mathcal{G}$ that provides improved support for generic programming with the goal of lowering the cost of developing reusable components and lowering the cost of reusing components. The next section summarizes the problems we encountered with generic programming in C++ and the proposed solutions for $\mathcal{G}$.

1.1. Lowering the cost of developing generic components

Generic programming in C++ is considered an advanced technique because the construction of generic libraries requires the use of many advanced idioms. There is a cost associated with translating the intent of the programmer to the appropriate idiom. Further, the idioms require an in-depth knowledge of language features such as partial template specialization, partial ordering of function templates, and argument dependent lookup. The acquisition

and maintenance of this knowledge is expensive. Nonetheless, generic libraries created using these idioms have proved exceeding useful despite the extra cost. The language $\mathcal{G}$ instead provides direct and simple language mechanisms that fulfill the same purposes.

Testing and debugging generic functions in C++ is difficult. C++ does not perform type checking on definitions of templates. Thus, a generic library developer does not enjoy the usual benefits of a static type system. Type checking is performed on the result of instantiating a template with particular type parameters. A library developer can test the generic function on particular types, but this does not guarantee that the generic function will work for other types and, in general, a generic function is supposed to work for an infinite number of types. The language $\mathcal{G}$ type checks the definition of a generic function independently of any of its instantiations. A generic function that passes type checking is guaranteed to be free of type errors when instantiated with type arguments that satisfy the requirements of the generic function.

Most generic functions make some assumptions about their type parameters, such as the assumption that an `operator==` is defined for the type. From the user's point of view, these assumptions are requirements. Since type requirements are not directly expressible in C++, library authors instead state the type requirements in the documentation for the generic function. It is important that the documented assumptions be complete, otherwise a user may attempt to apply a generic function in a situation it is not equipped to handle. The author of a C++ generic library must manually compare the documented assumptions to the implementation of the generic function. This process is time consuming and error prone. The language $\mathcal{G}$ provides the means to express type requirements as part of the interface of a generic function, and the type checker ensures that the assumptions are complete with respect to the implementation.

Another problem that plagues generic library developers in C++ is that namespaces do not provide complete protection from name pollution, so library developers must go out of their way to ensure that their calls to internal helper functions do not accidentally resolve to functions in other libraries. The language $\mathcal{G}$ provides complete name protection.

Developing high-quality generic libraries in C++ is costly, much more so than it should be, thereby reducing the savings from reuse (Equation 1). The design of $\mathcal{G}$ reduces the cost of generic library development by simplifying the language mechanisms for generic programming, by introducing static error detection for generic functions, and by making generic libraries more robust.

The next section discusses costs associated with using generic components. Many generic components use other generic components, so reductions in the cost of using generic components also reduces the cost of producing generic components.

1.2. Lowering the cost of reusing generic components

The productivity gains due to reuse are highly sensitive to the cost of using a generic component because this cost is multiplied by n in Equation 1. This section discusses factors that affect the cost of using generic components.

A strength of the C++ template system is that calling a generic function is syntactically identical to calling a normal function. Many alternative approaches to generics require the user to explicitly provide the type arguments for the generic function or explicitly provide the type-specific operations needed by the generic function. The C++ compiler, in contrast, deduces the type arguments for a function template from the types of the normal arguments. I refer to this as ***implicit instantiation***. C++ also provides an implicit mechanism for resolving type-specific operations within a template. The language $\mathcal{G}$ retains these strengths of C++, although the mechanism for resolving type-specific operations is much different.

The most visible disadvantage of generic programming in C++ is the infamous error messages that a user experiences after making mistakes. The error messages are long, hard to understand, and do not point to the source of the problem. Instead the error messages point deep inside the implementation of the generic library. The problem is that the C++ type system does not know the type requirements for the generic function (they are written in English in the documentation) and therefore cannot warn the user when the requirements are violated. As mentioned above, in the language $\mathcal{G}$, the interface of a generic function includes its type requirements. The type checker uses this information to verify whether

the requirements are satisfied at a particular use of the generic function. In this thesis I use the term ***separate type checking*** to mean that type checking the use of a generic function is independent of the generic function's implementation, and conversely, type checking the implementation of a generic function is independent of its uses.

Another disadvantage of C++ is that the time to compile a program is a function of the size of the program plus the size of all generic components used by the program (and all the generic components used by those generic components, etc.). This has proven to be a serious problem in practice: compile times become prohibitive when several large generic libraries are used in the same program. This problem is especially acute during development and debugging, when the compilation time becomes the bottleneck in the compile-run-debug cycle. In C++, the size of non-generic components used in a program does not factor into the compile time because the non-generic components can be ***separately compiled*** to object code. The addition of the `export` facility of C++ [86] does not provide true separate compilation for templates because the compile time of a program remains a function of all the generic components it uses. The language $\mathcal{G}$ provides separate compilation for both generic and non-generic components. As we shall see, there is a run-time cost associated with separate compilation so $\mathcal{G}$ provides the programmer with the choice of whether to compile modules together or separately.

As described in the previous section, $\mathcal{G}$ aids in the discovery of bugs and inconsistencies in generic functions. This improvement in quality translates into saving for users of generic libraries because bugs in libraries are extremely costly to users.

Many generic functions are ***higher-order functions***: they take functions as parameters. The function arguments are typically task-specific and only used in a single place in a program. Thus it is convenient to define the function in place with an anonymous function expression. C++, however, does not have a facility for creating function expressions: instead, function objects are used. A ***function object*** is an instance of a class with an `operator()` member function. Creating a class is more work than writing a function expression so this adds to the syntactic cost of calling a generic higher-order function. The language $\mathcal{G}$ provides function expressions (as is common in functional languages).

1.3. $\mathcal{G}$: a language for generic programming

The primary challenge in the design of $\mathcal{G}$ is resolving the tension between modularity and interaction. A component is trivially modular if it has no inputs or outputs and operates only on private data. Of course, such a component is useless. On the other hand, a system with unrestrained interaction between components is difficult to debug and maintain. Thus the challenge is to allow for rich interactions between components so that they may accomplish useful work while at the same time protecting the components from one another.

$\mathcal{G}$ ensures modularity for generic components by basing its design on parametric polymorphism, which by default severely restricts interaction. $\mathcal{G}$ makes rich interactions possible by providing an expressive language for describing contracts between generic components. The contracts, or interface specifications, are used by the type system to govern the interactions between components. For the generic components of $\mathcal{G}$, contracts mainly consist of requirements on their type parameters. I refer to language mechanisms that provide type parameterization and requirements on type parameters as ***generics***.

The primary influence on the design for generics in this dissertation is the semi-formal specification language currently used to document C++ libraries [11, 86, 169, 176]. I performed a thorough survey of the documentation of the STL (Chapter 2), recording what kinds of requirements were expressed, and then incorporated each kind of requirement into the design of $\mathcal{G}$. Another influence on $\mathcal{G}$ is the Tecton specification language by Kapur, Stepanov, and Musser [101, 102] and related work [164, 200] that formalizes the generic programming specification language.

The non-generic language features of $\mathcal{G}$ are borrowed from C++, though the design for generics mandated modifications to non-generic parts of the language. The design for generics in $\mathcal{G}$ could be applied to other programming languages, such as Java or C#. We chose C++ because it would facilitate the evaluation of $\mathcal{G}$, easing the translation of the STL and BGL from C++.

A secondary challenge faced in the design of $\mathcal{G}$ is the tension between run-time efficiency and fast compile times. To achieve fast compile times, separate compilation of components is needed. However, to produce the most optimized code, the compiler must have access to the whole program. For example, the C++ compilation model for function templates stamps out a specialized version of the function for each set of type arguments, producing highly efficient code but forcing templates to be compiled with their uses. If a C++ programmer wants separate compilation, then a generic function must be expressed using classes and subtype polymorphism instead of using templates. Providing both versions of a generic function is a costly endeavor and is seldom done in practice.

Compilers for languages such as Java and Standard ML typically produce a single set of instructions for a given generic function, thereby achieving fast compile times but sacrificing efficiency. However, this second approach leaves open the door to allowing the programmer or compiler to choose when run-time efficiency is favored over compile-time efficiency. A compiler (or just-in-time compiler) may perform function specialization and inlining as an optimization (without changing the semantics of the program) and gain the efficiency of C++ templates. The design of $\mathcal{G}$ is similar to Java and Standard ML: a generic function may be separately compiled to single set of instructions or it may be compiled to a specialized function. The compiler for $\mathcal{G}$ described in Chapter 5 does not perform function specialization or inlining but these optimizations are well-known and the relevant literature is discussed in Section 3.3.

1.4. Related work in programming language research

The design for generics in $\mathcal{G}$ is most closely related to type classes in Haskell: there is an analogy between the `concept` and `model` features of $\mathcal{G}$ and the `class` and `instance` features of Haskell [196], respectively. However, many of the design goals and details differ. There are also some similarities between ML signatures and $\mathcal{G}$ concepts and we have applied several compilation techniques developed for ML to the compilation of $\mathcal{G}$. Both Haskell and ML are based on the Hindley-Milner type system whereas $\mathcal{G}$ is based on the polymorphic lambda calculus of Girard and Reynolds [71, 157].

Chapter 3 gives an in-depth discussion of language mechanisms for generics and surveys the various forms of polymorphism in programming languages.

1.5. Claims and evaluation

The following points list the concrete claims of this thesis and the methods used to substantiate the claims.

(1) The type system of $\mathcal{G}$ separately type checks definitions and uses of generic components. This is verified in Chapter 5 by inspection of the type rules for $\mathcal{G}$.

(2) $\mathcal{G}$ is not type safe because it inherits type safety holes from C++, such as the potential to dereference dangling pointers. However, the design for generics does not contain type holes. Chapter 7 verifies this claim with a type safety proof for the language $\mathrm{F}^{\mathcal{G}}$ which captures the essence of the generics of $\mathcal{G}$ in a small formal language.

(3) $\mathcal{G}$ provides implicit instantiation of generic functions. Chapter 5 defines the static semantics of $\mathcal{G}$ including how implicit instantiation is performed. The algorithm is based on the variant of unification used in ML^{F}, which was proved effective and sound [24].

(4) $\mathcal{G}$ provides a mechanism for implicitly satisfying the type requirements of a generic function at its point of instantiation. The static semantics of $\mathcal{G}$ described in Chapter 5 demonstrates how this is accomplished by translating `model` definitions into function dictionaries and by explicitly passing dictionaries to generic functions with type requirements.

(5) $\mathcal{G}$ provides separate compilation of both generic and non-generic functions. This is demonstrated with the construction of a compiler for $\mathcal{G}$ that in fact compiles generic functions to object code. Chapter 5 describes the compilation of $\mathcal{G}$ to C++.

(6) $\mathcal{G}$ provides complete namespace protection. That is, the author of a module has complete control over which names and model definitions are visible to the module and which names are exported from the module. The module author can determine the bindings of all variable references in the module by static inspection of the module code.

(7) $\mathcal{G}$ supports the common idioms [11] of generic programming and formalizes the specification language used to document generic libraries. We substantiate this claim by implementing prototypes of the Standard Template Library and the Boost Graph Library and verifying that $\mathcal{G}$ provides all the necessary language facilities for their expression, which is described in Chapter 6.

1.6. Road map

Chapter 2 is an introduction to generic programming and to the Standard Template Library of C++, which is representative of current practice in generic C++ libraries. The current practice of generic programming is directly supported and formalized in the design of $\mathcal{G}$. Chapter 3 is a survey and evaluation of programming language mechanisms that support generic programming, describing various forms of polymorphism and ways to constrain it. This evaluation establishes the foundation for the design of $\mathcal{G}$ and explains the inherent tradeoffs in the solution space. Chapter 5 describes the design and implementation of $\mathcal{G}$. This includes an introduction to generics in $\mathcal{G}$ and the rationale for the design. Chapter 5 then covers the type system of $\mathcal{G}$ in detail and the translation of $\mathcal{G}$ to C++, which serves to define the semantics of $\mathcal{G}$ and shows how to compile $\mathcal{G}$. Chapter 6 evaluates the suitability of $\mathcal{G}$ for generic programming with two case studies: prototype implementations of the STL and the BGL. Chapter 7 formalizes the essential features of $\mathcal{G}$, defining a core calculus named $F^{\mathcal{G}}$ and proves type safety for $F^{\mathcal{G}}$. Chapter 8 concludes this dissertation.

To become a generally usable programming product, a program must be written in a generalized fashion. In particular the range and form of inputs must be generalized as much as the basic algorithm will reasonably allow.

Frederick P. Brooks, Jr., [64]

That is the fundamental point: algorithms are defined on algebraic structures.

Alexander Stepanov [160]

2

Generic programming and the STL

This chapter reviews the generic programming methodology of Stepanov and Musser and how this methodology is applied in modern C++ libraries, with the Standard Template Library (STL) as the prime example. This chapter starts with a short history of generic programming and a description of the methodology. The description is made concrete with a small example: the development of an algorithm for accumulating elements of a sequence. The design and implementation of the STL is then discussed, with emphasis placed on how the STL components are specified and on which C++ features are used in the implementation. The generic programming facilities of C++ are analyzed so that the design of $\mathcal{G}$ may build on the strengths and improve on the weaknesses. This chapter concludes with a comparison of generic programming to other programming methodologies.

Generic programming has roots in mathematics, especially abstract algebra. Abstraction plays an important role in mathematics: it helps mathematicians capture the essence of the entities they study and makes theorems more general and therefore more widely applicable. In the 1800's Richard Dedekind and Emmy Noether began to distill algebra into fundamental abstract concepts such as Group, Ring, and Field. These concepts were generalizations of the mathematical entities they were studying; they captured the essential properties needed to prove their theorems. Noether's student van der Waerden popularized these ideas in his book *Modern Algebra* [192].

In the early 1980's, Alexander Stepanov and David Musser, with several colleagues, discovered how to use algebraic structures, and similar abstractions, to organize programs to enable a high degree of reuse [104]. (There were similar developments around the same time in a language for computer algebra by Jenks and Trager [93].) Stepanov and Musser drew ideas from research on abstract data types [32, 77, 118, 185, 203], functional programming languages [13, 61, 87], and mathematics [25]. Their initial idea was to use "operators" (higher-order functions) to express generic algorithms, and to organize function parameters along the lines of algebraic structures.

In the late 1980's Stepanov and Musser applied their ideas to the creation of libraries for processing sequences and graphs in the Scheme programming language [105, 180] and also in Ada [138]. Their work came to fruition in the early 1990's with the C++ Standard Template Library [181], when generic programming began to see widespread use.

Figure 1 reproduces the standard definition of generic programming from Jazayeri, Musser, and Loos [92]. In the next section we show how this methodology can be applied to implement a generic algorithm in Scheme [3, 56, 65]. The generic programming methodology always consists of the following steps: 1) identify a family of useful and efficient concrete algorithms with some commonality, 2) resolve the differences by forming higher-level abstractions, and 3) lift the concrete algorithms so they operate on these new abstractions. When applicable, there is a fourth step to implement automatic selection of the best algorithm, as described in Figure 1.

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Key ideas include:

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.
- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.
- When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.
- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.

FIGURE 1. Definition of Generic Programming from Jazayeri, Musser, and Loos[92]

2.1. An example of generic programming

Figure 2 presents a family of concrete functions that operate on lists and vectors, computing the sum or product of the elements or concatenating the elements (which in this case are lists). These functions share a common control-flow; at some level of abstraction these functions are essentially the same. Each of these functions is recursive, with a base case that returns an object and a recursion step that combines the current element of the sequence with the result of applying the function to the rest of the sequence.

There is a special relation between the base case object and the combining function used in each algorithm. The following equations express the relationship: an application of the combining function to the base object and an arbitrary value `a` yields `a`. Thus the base object is the *identity element*.

```
(+ 0 a) = a
(* 1 a) = a
```

FIGURE 2. A family of related algorithms implemented in Scheme.

```
(define sum-list
  (λ (ls)
    (cond [(null? ls) 0]
          [else (+ (car ls) (sum-list (cdr ls)))])))
(define product-list
  (λ (ls)
    (cond [(null? ls) 1]
          [else (* (car ls) (product-list (cdr ls)))])))
(define concat-list
  (λ (ls)
    (cond [(null? ls) '()]
          [else (append (car ls) (concat-list (cdr ls)))])))
(define sum-vector
  (λ (vs)
    (letrec ([loop (λ (i)
                     (cond [(eq? i (vector-length vs))
                            0]
                           [else (+ (vector-ref vs i)
                                    (loop (+ i 1)))]))])
      (loop 0))))
(define product-vector
  (λ (vs)
    (letrec ([loop (λ (i)
                     (cond [(eq? i (vector-length vs))
                            1]
                           [else (* (vector-ref vs i)
                                    (loop (+ i 1)))]))])
      (loop 0))))
(define concat-vector
  (λ (vs)
    (letrec ([loop (λ (i)
                     (cond [(eq? i (vector-length vs))
                            '()]
                           [else (append (vector-ref vs i)
                                         (loop (+ i 1)))]))])
      (loop 0))))
```

```
(append '() a) = a
```

This grouping of an identity element, binary operator, and a set of values (e.g., integers or strings), is traditionally called a Monoid. The first step of lifting the algorithms in Figure 2

is to recognize that they operate on Monoids. Thus, we can reduce the six algorithms to just two by writing them in terms of an arbitrary `id-elt` and `binop`. We use the more generic name `accumulate` for these algorithms and pass the `id-elt` and `binop` in as parameters.

```
(define accumulate-list
  (λ (ls binop id-elt)
    (cond [(null? ls) id-elt]
          [else (binop (car ls) (accumulate-list (cdr ls) binop id-elt))])))

(define accumulate-vector
  (λ (vs binop id-elt)
    (letrec ([loop (λ (i)
                     (cond [(eq? i (vector-length vs))
                            id-elt]
                           [else (binop (vector-ref vs i)
                                        (loop (+ i 1)))]))])
      (loop 0))))
```

In the generic programming literature, abstractions such as Monoid [11, 103] are called ***concepts***. There are two equivalent ways to think about concepts. First, a concept can be thought of as a list of requirements. The requirements include things like function signatures and equalities. The following table shows the requirements for the Monoid concept. We use `X` as a place-holder for a type that satisfies the Monoid concept and `a` is an arbitrary value of type `X`.

Monoid concept
`binop : X × X → X`
`id-elt : X`
`(binop id-elt a) = id-elt = (binop a id-elt)`
`(binop a (binop b c)) = (binop (binop a b) c)`

The Monoid concept includes the requirement that the binary operator be associative. While this is not strictly necessary for the `accumulate` function it is a useful requirement because it would allow us to change the implementation later on to process portions of the sequence in parallel.

The second way to think about a concept is as a set of types. This is equivalent to thinking of a concept as a list of requirements because a type is in a concept (a set of types) if and only if it satisfies the list of requirements. When a type satisfies a concept,

we say that the type ***models*** the concept. Sometimes it is useful to generalize the notion of a concept from a set of types to a relation on types, functions, and objects. For example, with the Monoid concept, there are multiple ways in which the type `integer` can satisfy the requirements, for example, with `+` and `0` or with `*` and `1`.

Getting back to the accumulate example, the two new algorithms still differ in the data structures they process: a linked list and a vector. However, both data structures represent a *sequence*. When viewed at this higher level of abstraction the algorithms can be seen to perform the same operations:

- Access the element at the current position (`car` for lists and `vector-ref` for arrays).
- Move the position to the next element (`cdr` for lists and `(+ i 1)` for arrays).
- Check if the position is past the end of the sequence (`null?` for lists and `eq?` for arrays).

There is a concept named Input Iterator in the STL that groups together these operations. The following table describes the requirements for the Input Iterator concept (loosely translated into Scheme).

Input Iterator concept
type `value`
type `difference`
`difference` models the Signed Integral concept
`next` : `X` $\rightarrow$ `X`
`curr` : `X` $\rightarrow$ `value`
`equal?` : `X` $\times$ `X` $\rightarrow$ `bool`
`(equal? i j)` implies `(eq? (curr i) (curr j))`
`next`, `curr`, and `equal?` must be constant time

The `value` and `difference` types that appear in the requirements for Input Iterator are helper types. A `value` type is needed for the return type of `curr` and a `difference` type is needed for measuring distances between iterators of type `X`. The `difference` type is required to be an appropriate integer type. The helper types may vary from iterator to iterator and are determined by the iterator type. We refer to such helper types as ***associated types***.

FIGURE 3. A generic accumulate function in Scheme.

```
(define accumulate
  (λ (binop id-elt next curr equal?)
    (λ (first last)
      (letrec ([loop (λ (first)
                       (cond [(equal? first last)
                              id-elt]
                             [else (binop (curr first)
                                          (loop (next first)))]))])
        (loop first)))))
```

The Input Iterator concept also includes ***complexity guarantees*** about the required operations: they must have constant time complexity. Such complexity guarantees are important for describing the time complexity of algorithms. For example, our accumulate algorithms are linear time provided that the iterator and monoid operations are constant time.

Lifting the accumulate algorithms with respect to Input Iterator produces the generic `accumulate` function in Figure 3. The `accumulate` function is curried according to the two different times at which the inputs are available. The client of `accumulate` first supplies the Monoid and Input Iterator operations and in return gets a concrete function, where the meaning of `id-elt`, `binop`, `next`, etc. is fixed. This corresponds to McIlroy's notion of "sale time" parameters in the quotation from Chapter 1. Iterators can be fed into the concrete function to compute a result.

The original concrete functions can be recovered by applying the generic `accumulate` to the appropriate type-specific operations. The following code implements the list processing algorithms using cons-lists directly as iterators.

```
(define sum-list (λ (ls) ((accumulate + 0 cdr car eq?) ls '())))
(define product-list (λ (ls) ((accumulate * 1 cdr car eq?) ls '())))
(define concat-list (λ (ls) ((accumulate append '() cdr car eq?) ls '())))
```

The iterators for vectors are pairs consisting of the vector and the index of the current position. The following functions implement the iterator operations in terms of these pairs.

```
(define vnext (λ (v-i) (cons (car v-i) (+ 1 (cdr v-i)))))
```

```
(define vcurr (λ (v-i) (vector-ref (car v-i) (cdr v-i))))
(define veq? (λ (v-i v-j) (eq? (cdr v-i) (cdr v-j))))
```

The vector processing algorithm can then be implemented using the generic `accumulate` and the vector iterator functions.

```
(define sum-vector (λ (vs) ((accumulate + 0 vnext vcurr veq?)
                            (cons vs 0) (cons vs (vector-length vs)))))
(define product-vector (λ (vs) ((accumulate * 1 vnext vcurr veq?)
                                (cons vs 0) (cons vs (vector-length vs)))))
(define concat-vector (λ (vs) ((accumulate append '() vnext vcurr veq?)
                               (cons vs 0) (cons vs (vector-length vs)))))
```

With the generic `accumulate` we can implement a potentially infinite number of concrete algorithms with very little effort. Granted, because `accumulate` is only a few lines of code, this is not a huge gain, but many of the STL and BGL algorithms are hundreds of lines long, encapsulating large amounts of domain knowledge and expertise. Reusing that code results in a significant savings. In general, if we wish to implement M algorithms for N data structures we would need M times N concrete algorithms. With generic programming we write M generic functions plus N data structure implementations. Thus we get a multiplicative amount of functionality for an additive amount of work. M and N do not have to grow very large before the generic programming approach realizes significant savings.

The approach used in this section to implement generic functions, passing concept operations as parameters, was one of the first language mechanisms used by Stepanov, Musser, and Kershenbaum to implement generic algorithms [105, 180] and it remains an important tool for building modern generic libraries. However, we do not use function parameters as the primary mechanism for providing access to concept operations. The reason is that generic functions can become difficult to use due to the large number of parameters. In some cases, a library author can supply specific versions of the algorithms, as we did above. However, a user may wish to apply the generic algorithm to some new data type.

Ideally, we would like calls to generic algorithms to be uncluttered by concept operation parameters. Instead, if the author of a data type registers which concepts the data type models and then the programming language can take care of passing the concept operations

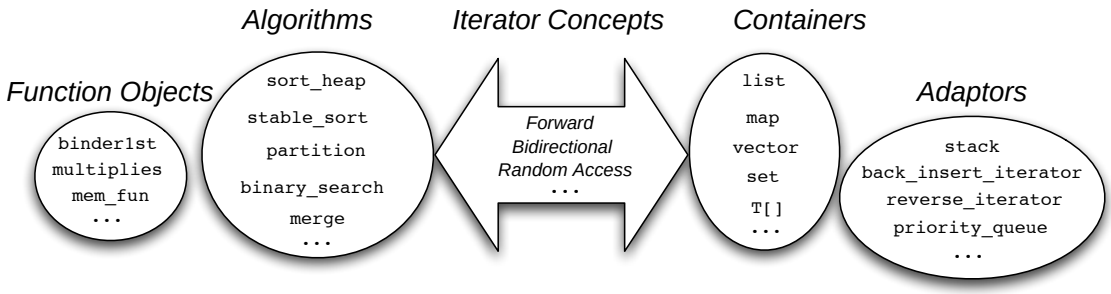

FIGURE 4. High-level structure of the STL.

into a generic function. The language Haskell has a type class feature that provides this capability as does the language $\mathcal{G}$ of this thesis.

2.2. Survey of generic programming in the STL

The high-level structure of the STL is shown in Figure 4. There are five categories of components in the STL, but of primary importance are the algorithms. The STL contains over fifty classic algorithms on sequences including sorting, searching, binary heaps, permutations, etc. The STL also includes a handful of common data structures such as doubly-linked lists, resizeable arrays, and red-black trees.

Many of the STL algorithms are higher-order: they take functions as parameters, allowing users to customize the algorithm to their own needs. The STL includes function objects for creating and composing functions. For example, the `plus` function object adds two numbers and the `unary_compose` function object combines two function objects, f and g, to create a function that performs the computation $f(g(x))$.

The STL also contains a collection of adaptor classes. An adaptor class is parameterized on the type being adapted. The adaptor class then implements some functionality using the adapted type. The adapted type must satisfy the requirements of some concept and the adaptor class typically implements the requirements of another concept. For example, the `back_insert_iterator` adaptor is parameterized on a Back Insertion Sequence and implements Output Iterator. Adaptors play an important role in the plug-and-play nature of the

STL and enable a high degree of reuse. One example is the `find_end` algorithm which is implemented using the `search` algorithm and the `reverse_iterator` adaptor.

The rest of this section takes a closer look at the STL components, reviewing how they are implemented in C++ and highlighting the interface specification elements used in the STL documentation. The goal is to come up with the list of language features that are needed in $\mathcal{G}$ to allow for a straightforward implementation of the STL.

2.2.1. Generic algorithms and STL concepts. The algorithms of the STL are organized into the following categories:

(1) Iterator functions
(2) Non-mutating algorithms
(3) Mutating algorithms
(4) Sorting and searching
(5) Generalized numeric algorithms

In this context, "mutating" means that the elements of an input sequence are modified in-place by the algorithm. Most of the algorithms operate on sequences of elements, but a few basic algorithms operate on a couple of elements.

We look in detail at a selection of algorithms from the STL, at least one from each of the above categories. Algorithms were selected to demonstrate all the C++ techniques and specification elements that are used in the STL.

`min`: (sorting) This simple function makes for a good starting point to talk about C++ function templates, type requirements, and the Less Than Comparable concept.

`count`: (non-mutating) This algorithm operates on iterators, so we introduce the Input Iterator concept and describe the STL's iterator hierarchy. An important but unusual aspect of concepts (for those unfamiliar to generic programming) is the notion of associated types. We introduce the C++ traits idiom that is used to access associated types.

`unique`: (mutating) A generic algorithm usually places constraints on its type parameters. In this algorithm (and many others) constraints are also placed on associated types.

`stable_sort`: (sorting) We show a misuse of this algorithm and the resulting C++ error message. This leads to a short discussion of C++ techniques for improving error messages and for checking whether an algorithm's implementation is consistent with its documented interface.

`merge`: (sorting) This algorithm demonstrates the need for another kind of constraint which we call same-type constraints. The `merge` algorithm also shows the need to generalize concepts so that they can place requirements on multiple types instead of just a single type (not counting associated types).

`accumulate`: (generalized numeric) Like most STL algorithms, there are two versions of `accumulate`. One of the versions takes an extra function parameter and is therefore an example of a higher order function. We discuss function objects, function concepts like Binary Function, and conversion requirements.

`advance`: (iterator functions) This function uses a C++ idiom called tag dispatching to dispatch to different implementations depending on the capabilities of the iterator.

2.2.1.1. *min, function templates, and type requirements.* Perhaps the simplest of STL algorithms is `min`, which returns the smaller of two values. The STL algorithms are implemented in C++ with function templates. The `min` template is parameterized on type `T`.

```
template<typename T>
const T& min(const T& a, const T& b) {
  if (b < a) return b; else return a;
}
```

The `min` function template does not work on an arbitrary type `T`; the STL SGI documentation lists the following restriction:

- `T` is a model of Less Than Comparable.

The C++ Standard defines concepts with requirements tables. The table below defines when a type `T` is a model of Less Than Comparable. The values `a` and `b` have type `T`.

expression	return type	semantics
`a < b`	convertible to `bool`	< is a strict weak ordering relation

FIGURE 5. Requirements for Less Than Comparable

In C++ documentation, ***valid expressions*** are used to express requirements instead of function signatures. The reason is that in C++ a function signature would be more restrictive, ruling out other function signatures that could also be used for the same expression. For example, the signature

```
bool operator<(const T&, const T&);
```

would require a free function, ruling out less-than operators implemented as member functions.

A function template is ***instantiated*** by binding concrete types to the type parameters, thereby creating a concrete function, The following program shows two instantiations of the `min` template. The first instantiation is explicit, with `min` applied to the type argument `foo::bar`. The second instantiation is implicit: the type argument is deduced by the C++ compiler from the types of `a` and `b`.

```
namespace foo {
  struct bar { int n; };
  bool operator<(bar a, bar b) { return a.n < b.n; }
}

#include <algorithm>
using std::min;
int main() {
  foo::bar a = { 0 }, b = { 1 };
  foo::bar c = min<foo::bar>(a, b); // explicit instantiation, T=foo::bar
  foo::bar d = min(a, b); // implicit instantiation, T=foo::bar
  assert(c.n == 0 && d.n == 0);
}
```

Name lookup in C++. In C++, uses of names inside of a template definition, such as the use of `operator<` inside of `std::min`, are resolved after instantiation. For the instantiation

`std::min<foo::bar>`, overload resolution looks for an `operator<` defined for `foo::bar`. There is no such function defined in the scope of `std::min`, but the C++ compiler also searches the namespace where the arguments' types are defined, so it finds the `operator<` in namespace `foo`. This rule is known as ***argument dependent lookup*** (ADL).

The combination of implicit instantiation and ADL makes it convenient to call generic functions. This is a nice improvement over passing concept operations as arguments to a generic function, as in the `accumulate` example from Section 2.1. However, ADL has two flaws. The first problem is that the programmer calling the generic algorithm no longer has control over which functions are used to satisfy the concept operations. Suppose that namespace `foo` is a third party library and the application programmer writing the `main` function has defined his own `operator<` for `foo::bar`. ADL does not find this new `operator<`.

The second and more severe problem with ADL is that it opens a hole in the protection that namespaces are suppose to provide. ADL is applied uniformly to all name lookup, whether or not the name is associated with a concept in the type requirements of the template. Thus, it is possible for calls to helper functions to get hijacked by functions with the same name in other namespaces. Figure 6 shows an example of how this can happen. The function template `lib::generic_fun` calls `load` with the intention of invoking `lib::load`. In `main` we call `generic_fun` with an object of type `foo::bar`, so in the call to `load`, `x` also has type `foo::bar`. Thus, argument dependent lookup also consider namespace `foo` when searching for `load`. There happens to be a function named `load` in namespace `foo`, and it is a slightly better match than `lib::foo`, so it is called instead, thereby hijacking the call to `load`.

2.2.1.2. *`count`, iterator concepts, and associated types.* Most STL algorithms operate on sequences of elements and access to the sequence is expressed in terms of iterator concepts. The `count` algorithm is a simple example but touches many aspects of generic programming in C++. This algorithm counts hows many elements in the sequence are equal to the `value` parameter. The sequence is delimited by the pair of iterators `first` and `last`, where `first` points to the first element of the sequence and `last` points "one past the end" of the sequence.

FIGURE 6. Example problem caused by ADL.

```
namespace lib {
  template<class T> void load(T x, string)
      { std::cout << "Proceeding as normal!\n"; }

  template<class T> void generic_fun(T x)
      { load(x, "file"); }
}
namespace foo {
  struct bar { int n; };
  template<class T> void load(T x, const char*)
      { std::cout << "Hijacked!\n"; }
}
int main() {
  foo::bar a;
  lib::generic_fun(a);
}
// Output: Hijacked!
```

```
template<typename Iter, typename T>
typename iterator_traits<Iter>::difference_type
count(Iter first, Iter last, const T& value) {
  typename iterator_traits<Iter>::difference_type n = 0;
  for ( ; first != last; ++first)
    if (*first == value)
      ++n;
  return n;
}
```

The following are the type requirements for this function template.

- `Iter` is a model of Input Iterator.
- An object of `Iter`'s value type can be compared for equality with an object of type `T`.

Figure 7 shows the definition of the Input Iterator concept following the presentation style used in the SGI STL documentation [9, 176]. The definition says that Input Iterator is a ***refinement*** the Trivial Iterator concept.[1] Thus, all of the requirements of Trivial Iterator are

[1]The specification of Input Iterator in the SGI STL documentation differs somewhat from the C++ Standard.

included in the requirements for Input Iterator and any type that models Input Iterator must also model Trivial Iterator.

The Input Iterator concept also includes requirements for several associated types: the `value_type`, `difference_type`, and the `iterator_category`. The Input Iterator concept requires that these associated types be accessible via the `iterator_traits` class. The return type of the `count` function is an example of using `iterator_traits` to map from the `Iter` type to its `difference_type`:

```
iterator_traits<Iter>::difference_type
```

The reason the `count` uses the iterator specific `difference_type` instead of `int` is to accommodate iterators that traverse sequences that may be too long to be measured with an `int`.

Traits classes and template specialization. A ***traits class*** [140] maps from a type to other types or functions. Traits classes rely on C++ template specialization to perform this mapping. The following is the main template definition for `iterator_traits`.

```
template<typename Iterator>
struct iterator_traits { ... };
```

A ***template specialization*** is defined by specifying particular type arguments for the template parameter and by specifying an alternate body for the template. When a programmer creates a new iterator class, such as the `my_iter` class below, the `iterator_traits` template can be specialized to specify the `value_type`, etc., for the new iterator. In this case the `value_type` of `my_iter` should be `float` to match the return type of `operator*`.

```
class my_iter {
  float operator*() { ... }
  ...
};

template<>
struct iterator_traits<my_iter> {
  typedef float value_type;
  typedef int difference_type;
  typedef input_iterator_tag iterator_category;
};
```

Input Iterator

Description
An Input Iterator is an iterator that may be dereferenced to refer to some object, and that may be incremented to obtain the next iterator in a sequence. Input Iterators are not required to be mutable. The underlying sequence elements is not required to be persistent. For example, an Input Iterator could be reading input from the terminal. Thus, an algorithm may not make multiple passes through a sequence using an Input Iterator.

Refinement of
Trivial Iterator.

Notation

`X`	A type that is a model of Input Iterator
`T`	The value type of `X`
`i, j`	Objects of type `X`
`t`	Object of type `T`

Associated types

`iterator_traits<X>::value_type` The type of the value obtained by dereferencing an Input Iterator
`iterator_traits<X>::difference_type` A signed integral type used to represent the distance from one iterator to another, or the number of elements in a range.
`iterator_traits<X>::iterator_category` A type convertible to `input_iterator_tag`.

Definitions
An iterator is *past-the-end* if it points beyond the last element of a container. Past-the-end values are nonsingular and nondereferenceable. An iterator is *valid* if it is dereferenceable or past-the-end. An iterator `i` is *incrementable* if there is a "next" iterator, that is, if `++i` is well-defined. Past-the-end iterators are not incrementable. An Input Iterator `j` is *reachable* from an Input Iterator `i` if, after applying `operator++` to `i` a finite number of times, `i == j`. The notation `[i,j)` refers to a range of iterators beginning with `i` and up to but not including `j`. The range `[i,j)` is a *valid range* if both `i` and `j` are valid iterators, and `j` is reachable from `i`.

Valid expressions
In addition to the expressions in Trivial Iterator, the following expressions must be valid.

expression	return type	semantics, pre/post-conditions
`*i`	Convertible to `T`	pre: `i` is incrementable
`++i`	`X&`	pre: `i` is dereferenceable, post: `i` is dereferenceable or past the end
`i++`		Equivalent to `(void)++i`.
`*i++`		Equivalent to `{T t = *i; ++i; return t;}`

Complexity guarantees
All operations are amortized constant time.

Models
`istream_iterator`

FIGURE 7. Input Iterator requirements

When iterator_traits<my_iter> is used in other parts of the program it refers to the above specialization, whereas iterator_traits< list_iterator<T> > refers to the main template definition.

For parameterized types, ***partial template specialization*** can be used to define a trait. A partial specialization still has template parameters but restricts which types it applies to. For example, the following is the partial specialization of iterator_traits for all pointer types, as specified by the <T*>. The rules for template instantiation ensure that the specialization that best matches the type arguments is used.

```
template<typename T>
struct iterator_traits<T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T                          value_type;
  typedef ptrdiff_t                  difference_type;
  typedef T*                         pointer;
  typedef T&                         reference;
};
```

Template specialization is a form of dispatching on types and therefore the use of specialized class templates inside of function templates relies on the C++ compilation model: template definitions are type checked, etc., after instantiation, when all type arguments are known. For the return type of count, the type iterator_trait<Iter> is analyzed after instantiation, when it is known that Iter=my_iter.

Requirements on associated types in concept definitions. The Input Iterator concept requires that the associated difference_type be a signed integral type. This requirement is needed in count, for example, because it applies the increment operator to n. Placing requirement on associated types is fairly common in concept definitions. Another example is the Container concept, with its associated iterator type that is required to be a model of Input Iterator.

The iterator concept hierarchy. The Input Iterator concept provides limited functionality: the ability to read elements from a sequence in a single pass. More iterator concepts are needed to fulfill the needs of other sequence algorithms. For example, the copy algorithm copies one sequence into another and therefore needs an Output Iterator to accommodate

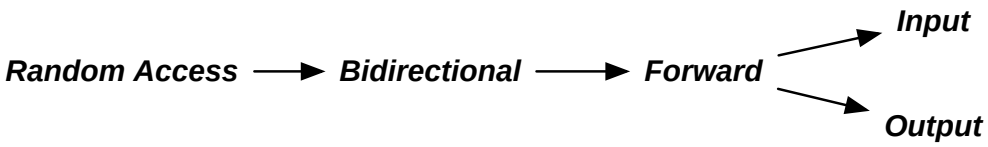

FIGURE 8. The refinement hierarchy of iterator concepts.

writing values. Another example is the `search` algorithm which finds occurrences of a particular subsequence within a larger sequence. To accomplish this, `search` must make multiple passes through the sequence. This capability is captured in Forward Iterator. The `inplace_merge` algorithm needs to move backwards and forwards through the sequence, so it requires Bidirectional Iterator. And finally, the `sort` algorithm needs to jump arbitrary distances within the sequence, so it requires Random Access Iterator. (The `sort` function uses the introsort algorithm [133] which is based on quicksort [83].)

The Forward Iterator concept is a refinement of (includes the requirements of) Input Iterator and Output Iterator. Likewise, Bidirectional Iterator refines Forward Iterator and Random Access Iterator refines Bidirectional Iterator. The refinement hierarchy for this family of iterator concepts is shown in Figure 8.

2.2.1.3. *`unique` and requirements on associated types.* The `unique` algorithm removes subsequences of duplicate elements, replacing them with a single occurrence of the element. The following is the signature and type requirements for `unique`.

```
template<class Iter>
Iter unique(Iter first, Iter last);
```

- `Iter` is a model of Forward Iterator.
- `Iter`'s value type is a model of Equality Comparable.

Here we see a requirement placed on the associated type of a type parameter. The above language is a slightly informal way of saying that the type

```
iterator_traits<Iter>::value_type
```

must model the Equality Comparable concept.

2.2.1.4. *stable_sort, error messages, and concept checking.* The stable_sort algorithm sorts a sequence in place into ascending order according to the value type's operator<. Also, stable_sort preserves the original ordering of equivalent elements, that is, an x and y are equivalent if neither x < y nor y < x.

```
template<typename Iter>
void stable_sort(Iter first, Iter last);
```

The type requirements for stable_sort are:

- Iter is a model of Random Access Iterator.
- Iter is mutable.
- Iter's value type is Less Than Comparable.

C++ template libraries have become infamous for their hard to understand error messages. When the user of an algorithm makes a mistake, such as accidentally attempting to use the algorithm with the wrong kind of iterator. The resulting compiler error is usually quite long and points into the internals of the template library. The following code tries to use stable_sort with the iterators from std::list.

```
#include <algorithm>
#include <list>
int main() {
  std::list<int> l;
  std::stable_sort(l.begin(), l.end());
}
```

Figure 9 shows the error message from GNU C++. The error message mentions lots of functions and types that the user should not know about such as __inplace_stable_sort and _List_iterator. Further, it is not clear from the error message who is responsible for the error. The error message is pointing inside the STL so the user might conclude that there is an error in the STL.

Concept checking. We developed a C++ idiom to combat the error message problem. The basic idea was to exercise all of the requirements of a function template at the beginning of the function using concept checking classes [170]. Thus, if a user sees errors coming from a concept checking class, then the user knows he made a mistake and sees the name of

```
stl_algo.h: In function ‘void std::__inplace_stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’:
stl_algo.h:2565:   instantiated from ‘void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’
stable_sort_error.cpp:5:   instantiated from here
stl_algo.h:2345: error: no match for ‘std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&’ operator
stl_algo.h:2565:   instantiated from ‘void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’
stable_sort_error.cpp:5:   instantiated from here
stl_algo.h:2349: error: no match for ‘std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&’ operator
stl_algo.h:2352: error: no match for ‘std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&’ operator
stl_algo.h:2352: error: no match for ‘std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&’ operator
stl_algo.h: In function ‘void std::__stable_sort_adaptive(_RandomAccessIter, _RandomAccessIter, _Pointer, _Distance)
    [with _RandomAccessIter = std::_List_iterator<int, int&, int*>, _Pointer = int*, _Distance = int]’:
stl_algo.h:2567:   instantiated from ‘void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’
stable_sort_error.cpp:5:   instantiated from here
stl_algo.h:2497: error: no match for ‘std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&’ operator
stl_algo.h:2498: error: no match for ‘std::_List_iterator<int, int&, int*>& + int&’ operator
stl_algo.h:2567:   instantiated from ‘void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’
stable_sort_error.cpp:5:   instantiated from here
stl_algo.h:2507: error: no match for ‘std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&’ operator
stl_algo.h:2507: error: no match for ‘std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&’ operator
stl_algo.h: In function ‘void std::__insertion_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’:
stl_algo.h:2346:   instantiated from ‘void std::__inplace_stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’
stl_algo.h:2565:   instantiated from ‘void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’
stable_sort_error.cpp:5:   instantiated from here
stl_algo.h:2095: error: no match for ‘std::_List_iterator<int, int&, int*>& + int’ operator
stl_algo.h:2346:   instantiated from ‘void std::__inplace_stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’
stl_algo.h:2565:   instantiated from ‘void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’
stable_sort_error.cpp:5:   instantiated from here
stl_algo.h:2099: error: no match for ‘std::_List_iterator<int, int&, int*>& + int’ operator
stl_algo.h: In function ‘void std::__merge_sort_with_buffer(_RandomAccessIter, _RandomAccessIter, _Pointer)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>, _Pointer = int*]’:
stl_algo.h:2504:   instantiated from ‘void std::__stable_sort_adaptive(_RandomAccessIter, _RandomAccessIter, _Pointer, _Distance)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>, _Pointer = int*, _Distance = int]’
stl_algo.h:2567:   instantiated from ‘void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’
stable_sort_error.cpp:5:   instantiated from here
stl_algo.h:2457: error: no match for ‘std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&’ operator
stl_algo.h: In function ‘void std::__chunk_insertion_sort(_RandomAccessIter, _RandomAccessIter, _Distance)
  [with _RandomAccessIter = std::_List_iterator<int, int&, int*>, _Distance = int]’:
stl_algo.h:2461:   instantiated from ‘void std::__merge_sort_with_buffer(_RandomAccessIter, _RandomAccessIter, _Pointer)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>, _Pointer = int*]’
stl_algo.h:2504:   instantiated from ‘void std::__stable_sort_adaptive(_RandomAccessIter, _RandomAccessIter, _Pointer, _Distance)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>, _Pointer = int*, _Distance = int]’
stl_algo.h:2567:   instantiated from ‘void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’
stable_sort_error.cpp:5:   instantiated from here
stl_algo.h:2431: error: no match for ‘std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&’ operator
stl_algo.h:2432: error: no match for ‘std::_List_iterator<int, int&, int*>& + int&’ operator
stl_algo.h:2433: error: no match for ‘std::_List_iterator<int, int&, int*>& += int&’ operator
stl_algo.h: In function ‘void std::__merge_sort_loop(_RandomAccessIter1, _RandomAccessIter1, _RandomAccessIter2, _Distance)
   [with _RandomAccessIter1 = std::_List_iterator<int, int&, int*>, _RandomAccessIter2 = int*, _Distance = int]’:
stl_algo.h:2464:   instantiated from ‘void std::__merge_sort_with_buffer(_RandomAccessIter, _RandomAccessIter, _Pointer)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>, _Pointer = int*]’
stl_algo.h:2504:   instantiated from ‘void std::__stable_sort_adaptive(_RandomAccessIter, _RandomAccessIter, _Pointer, _Distance)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>, _Pointer = int*, _Distance = int]’
stl_algo.h:2567:   instantiated from ‘void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]’
stable_sort_error.cpp:5:   instantiated from here
stl_algo.h:2388: error: no match for ‘std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&’ operator
stl_algo.h:2389: error: no match for ‘std::_List_iterator<int, int&, int*>& + int&’ operator
stl_algo.h:2389: error: no match for ‘std::_List_iterator<int, int&, int*>& + int&’ operator
stl_algo.h:2389: error: no match for ‘std::_List_iterator<int, int&, int*>& + int&’ operator
stl_algo.h:2392: error: no match for ‘std::_List_iterator<int, int&, int*>& += int&’ operator
stl_algo.h:2395: error: no match for ‘std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&’ operator
stl_algo.h:2396: error: no match for ‘std::_List_iterator<int, int&, int*>& + int&’ operator
stl_algo.h:2396: error: no match for ‘std::_List_iterator<int, int&, int*>& + int&’ operator
```

FIGURE 9. Error message from a misuse of `stable_sort`.

the concept that was not satisfied. The main trick is to get the compiler to exercise all the requirements without creating run-time overhead. This is achieved by writing expressions for the requirements in a separate function and creating a pointer to this function inside the generic algorithm (but the function is not called).

The following is a concept checking class for the Less Than Comparable concept. It contains a `constraints` method that uses `operator<` on two variables of type `T` and checks that the return type is convertible to `bool`.

```
template <class T>
struct LessThanComparableConcept {
  void constraints() {
    bool x = a < b;
  }
  T a, b;
};
```

The `stable_sort` function listed below is annotated with concept checks. The Boost Concept Checking library [167] provides pre-defined concept checking classes for the STL concepts and the `function_requires` utility that triggers the concept checking.

```
template<typename Iter>
void stable_sort(Iter first, Iter last)
{
  typedef typename iterator_traits<Iter>::value_type ValueType;
  function_requires(Mutable_RandomAccessIteratorConcept<Iter>)
  function_requires(LessThanComparableConcept<ValueType>)
  ...
}
```

The concept checking idiom has been applied in the SGI STL implementation and the GNU C++ standard library. With concept checking, error messages are much more informative but they are still confusing and hard to read. The following is the error message with concept checking.

```
concept_check.h:48:   instantiated from 'void boost::function_requires(boost::mpl::identity<T>*)
  [with Concept = boost::Mutable_RandomAccessIteratorConcept<std::_List_iterator<int, int&, int*> >]'
stable_sort_error.cpp:16:   instantiated from 'void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
  [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]'
stable_sort_error.cpp:5:   instantiated from here
concept_check.h:665: error: no match for 'std::_List_iterator<int, int&, int*>& [ptrdiff_t&]' operator
...
```

Completeness of type requirements and archetype classes. Concept checking helps to find and understand errors in the use of generic functions, but it does not check whether the documented type requirements of a generic function are enough to cover the needs of the implementation. If the documented type requirements are not enough, a user may experience compiler errors despite having satisfied the requirements.

The common practice for ensuring that all operations used in a generic function are covered by the type requirements is to manually inspect the implementation. This is tedious and error prone. A more automated approach is to create classes that minimally satisfy concepts, we call them ***archetype classes***, and then see if the function template compiles when used with the archetype class [170]. The generic library author no longer needs to manually inspect each algorithm, but creating correct archetype classes is non-trivial and error prone.

The following is an archetype class for the Less Than Comparable concept.

```
template <class Base = null_archetype<> >
class less_than_comparable_archetype : public Base {
public:
  less_than_comparable_archetype(detail::dummy_constructor x) : Base(x) { }
};
template <class Base>
boolean_archetype
operator<(const less_than_comparable_archetype<Base>&,
          const less_than_comparable_archetype<Base>&)
{
  return boolean_archetype(static_object<detail::dummy_constructor>::get());
}
```

The only function required by Less Than Comparable is the `operator<`. One minor complication in this archetype is that the return type is declared to be `boolean_archetype` instead of `bool`. The reason is that Less Than Comparable does not require that `operator<` return `bool`, but only that the return type be convertible to `bool`, which `boolean_archetype` satisfies. The `Base` type parameter is to allow for the composition of archetype classes.

2.2.1.5. *merge, same-type constraints, and multi-parameter concepts.* The `merge` algorithm combines two sorted ranges into a single sorted range.

```
template<class InIter1, class InIter2, class OutIter>
OutIter merge(InIter1 first1, InIter1 last1,
              InIter2 first2, InIter2 last2,
              OutIter result);
```

The following are the type requirements for `merge`. An interesting aspect of these requirements is the use of a ***same-type constraints*** that requires two type expressions to refer to the same type.

- `InIter1` is a model of Input Iterator.
- `InIter2` is a model of Input Iterator.
- `InIter1`'s value type is the same type as `InIter2`'s value type.
- `InIter1`'s value type is a model of Less Than Comparable.
- `OutIter` is a model of Output Iterator and `InIter1`'s value type is a type in `OutIter`'s set of value types.

With the same type constraint, an expression of the form `*first1 < *first2` is valid since both sides of the equality have the same type, and that type is a model of Less Than Comparable.

Another interesting aspect of the type requirements for `merge` is the requirement that `OutIter` be a model of Output Iterator. The Output Iterator concept has an associated *set of types* that are writable to the iterator. Instead of using the notion of a set of types, Output Iterator can be formulated as a ***multi-parameter concept***. The following is the concept checking class for this two-parameter version of Output Iterator. The parameter `X` is for the iterator and `ValueType` is for the value type.

```
template <class X, class ValueType>
struct OutputIteratorConcept {
  void constraints() {
    function_requires< AssignableConcept<X> >();
    ++i; i++; *i++ = t;
  }
  X i;
  ValueType t;
};
```

The requirements on the `OutIter` parameter of `merge` can then be written as follows.

- `OutIter` and `InIter1`'s value type together model Output Iterator.

The `merge` algorithm only uses the output iterator with a single value type, so one might wonder why the Output Iterator concept is not formulated with a single value type. There are other algorithms, such as `replace_copy`, that use an output iterator with multiple value types. The following is the interface and type requirements for `replace_copy`. This algorithm uses another multi-parameter concept: Equality Comparable2 which requires an `operator==` that works on arguments of different types.

```
template <class InIter, class OutIter, class T>
OutIter replace_copy(InIter first, InIter last, OutIter result,
                     const T& old_value, const T& new_value);
```

- `InIter` is a model of Input Iterator.
- `OutIter` and `T` together model Output Iterator.
- `OutIter` and `InIter`'s value type together model Output Iterator.
- `InIter`'s value type and `T` together model Equality Comparable2.

The multiple value types formulation of Output Iterator would be useless if there were no concrete iterator classes that could output multiple value types. However, there are examples of such iterator classes. The example in Figure 10 shows an output iterator that writes to an output stream. It has an `operator=` member template that may be used with any value type accepted by `std::ostream`.

2.2.1.6. *`accumulate`, function objects, and conversion requirements.* The `accumulate` algorithm of the STL combines the elements of a sequence with the `binop` function, starting with `id_elt`. Like many STL algorithms, there are two versions of `accumulate`, one that relies on an operator (in this case +) and the other that has an extra function parameter.

```
template <class Iter, class T>
T accumulate(Iter first, Iter last, T id_elt);

template <class Iter, class T, class Fun>
T accumulate(Iter first, Iter last, T id_elt, Fun binop);
```

The type requirements for the second version of `accumulate` are:

- `Iter` is a model of Input Iterator.

FIGURE 10. An output stream iterator that can output values of multiple types.

```
class outstream_iterator
   : public std::iterator<std::output_iterator_tag, void, void, void, void> {
public:
  explicit outstream_iterator(std::ostream& out) : out(out) {}
  struct output_proxy {
    output_proxy(std::ostream& out) : out(out) { }
    template<class T> output_proxy& operator=(const T& value)
      { out << value;  return *this; }
    std::ostream& out;
  };
  output_proxy operator*() { return output_proxy(out); }
  outstream_iterator& operator++() { return *this; }
  outstream_iterator& operator++(int) { return *this; }
private:
  std::ostream& out;
};
```

- `T` is a model of Assignable.
- `Fun` is a model of Binary Function.
- `T` is convertible to `Fun`'s first argument type.
- The value type of `Iter` is convertible to `Fun`'s second argument type.
- `Fun`'s return type is convertible to `T`.

Function objects. The second version of `accumulate` can be adapted to solve more problems than the first version, but the user must do a little more work by supplying the binary operator. Typically, the binary operator is a ***function object***: an instance of a class with a call operator (which is written `operator()`). The STL includes many predefined function classes. There is a function class for each built-in operator, such as the `multiplies` class for `operator*`, and there are function combinators, such as `unary_compose` and `bind2nd`. The following example computes the product of an array of integers. The standard `multiplies` function object is used for the binary operation and `1` for the identity element.

```
int a[] = { 1, 1, 2, 3, 5, 8 };
std::multiplies<int> binop;
int prod = std::accumulate(a, a + 6, 1, binop);
```

```
assert(prod == 240);
```

If there is no combination of predefined function objects that meet the user's need, or if the combination is too complex, then the user may instead write a custom function object. The following example sums the elements of an array, modulo 10.

```
struct add_modulo {
  add_modulo(int m) : m(m) { }
  int operator()(int a, int b) { return (a + b) % m; }
  int m;
};
int main() {
  int a[] = { 1, 1, 2, 3, 5, 8 };
  add_modulo binop(10);
  int sum_mod10 = std::accumulate(a, a + 6, 0, binop);
  assert(sum_mod10 == 0);
}
```

The type requirements for `accumulate` state that the `Fun` type parameter must model the Binary Function concept. Figure 11 shows the requirements for this concept. Basically, any function pointer or class with an `operator()` that takes two parameters is a model of Binary Function.

Function objects have a few disadvantages. The class typically needs to be defined at global scope, which fragments the code. Also, when the function object needs to refer to a variable from the surrounding scope, such as `a2` in the above example, the variable must be passed to the function object. Finally, the syntactic overhead associated with function objects is significant.

Conversion requirements. The type requirements for `accumulate` require that the type `T` of the identity element be convertible to the first argument type of `Fun`. A conversion requirement says there must be an implicit conversion from one type to another. Many STL algorithms use conversion requirements instead of same-type constraints to provide more flexibility.

2.2.1.7. *`advance` and tag dispatching.* One of the main points in the definition of generic programming in Figure 1 was that it is sometimes necessary to provide more than one generic algorithm for the same purpose. When this happens, the standard approach in C++

Binary Function

Description
A Binary Function is a kind of function object: an object that is called as if it were an ordinary C++ function. A Binary Function is called with two arguments.

Refinement of
Assignable, Copy Constructible.

Notation

`F`	A type that is a model of Binary Function
`X`	The first argument type of `F`
`Y`	The second argument type of `F`
`Result`	The result type of `F`
`f`	Object of type `F`
`x`	Object of type `X`
`y`	Object of type `Y`

Associated types

First argument type The type of the Binary Function's first argument.
Second argument type The type of the Binary Function's second argument.
Result type The type returned when the Binary Function is called

Definitions
The *domain* of a Binary Function is the set of all ordered pairs `(x, y)` that are permissible values for its arguments. The *range* of a Binary Function is the set of all possible value that it may return.

Valid expressions
In addition to the expressions in Trivial Iterator, the following expressions must be valid.

expression	return type	semantics, pre/post-conditions
`f(x,y)`	`Result`	Calls `f` with arguments `(x,y)`. pre: `(x,y)` is in the domain of `f`, post: The return value is in `f`'s range.

Complexity guarantees
None.

Models

- `Result (*)(X,Y)`
- `plus`
- `project1st`

FIGURE 11. Binary Function requirements

FIGURE 12. The advance algorithm and the tag dispatching idiom.

```
template<typename InIter, typename Distance>
void advance_dispatch(InIter& i, Distance n, input_iterator_tag) {
  while (n--) ++i;
}
template<typename BidirIter, typename Distance>
void advance_dispatch(BidirIter& i, Distance n, bidirectional_iterator_tag) {
  if (n > 0) while (n--) ++i;
  else while (n++) --i;
}
template<typename RandIter, typename Distance>
void advance_dispatch(RandIter& i, Distance n, random_access_iterator_tag) {
  i += n;
}
template<typename InIter, typename Distance>
void advance(InIter& i, Distance n) {
  typename iterator_traits<InIter>::iterator_category cat;
  advance_dispatch(i, n, cat);
}
```

libraries is to provide automatic dispatching to the appropriate algorithm using the ***tag dispatching idiom*** or `enable_if` [90]. Figure 12 shows the `advance` algorithm of the STL as it is typically implemented using the tag dispatching idiom. The `advance` algorithm moves an iterator forward (or backward) `n` positions. There are three overloads of `advance_dispatch`, each with an extra iterator tag parameter. The C++ Standard Library defines the following iterator tag classes, with their inheritance hierarchy mimicking the refinement hierarchy of the corresponding concepts.

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

The main `advance` function obtains the tag for the particular iterator from `iterator_traits` and then calls `advance_dispatch`. Normal static overload resolution then chooses the appropriate overload of `advance_dispatch`.

2.2.2. Generic containers. The STL container classes include some of the most common data structures for representing sequences of elements, though they are not a comprehensive collection.

`vector`: is a resizeable array.

`list`: is a doubly-linked list.

`deque`: is a container with fast insertion and removal at the beginning and end of the sequence in addition to fast access to arbitrary elements.

`set`: is a container of sorted elements.

`multiset`: is a container of sorted elements that permits multiple equivalent elements.

`map`: is an associative container that maps keys to values.

`multimap`: is an associative container that allows multiple values with the same key.

The C++ committee's official Technical Report on C++ Library Extensions [10] adds hash tables to the above selection. Also, many STL implementations include a singly-linked `slist` class.

The `list` class is typical of the STL containers. The following is an outline of `list` class template. `list` is parameterized on two types: `T` is the element type, and `Alloc` is a policy class that determines how elements of the list are allocated.

```
template<typename T, typename Alloc = allocator<T> >
class list { ... };
```

Just as the function templates of the STL have type requirements, so do the container classes. The type requirements for the `list` template are:

- `T` must model Copy Constructible and Assignable.
- `Alloc` must model Allocator.

The member functions of the `list` class rely on these type requirements. For example, the copy constructor of the `list` class uses the copy constructor for `T`.

There are many helper types that play a role in the functionality of the `list` class. The `list` class contains type definitions for each of the helper types, so for example, a programmer can access the iterator type for a list with a type expression such as `list<int>::iterator`. The identities of many of the helper types are intended to be hidden implementation details. The C++ `typedef` does not make them truly hidden, but it is good programming style to treat them as if they were. The following are the nested type definitions within `list`:

```
template<typename T, typename Alloc = allocator<T> >
class list {
public:
  typedef T                                    value_type;
  typedef value_type*                          pointer;
  typedef const value_type*                    const_pointer;
  typedef _List_iterator<T,T&,T*>              iterator;
  typedef _List_iterator<T,const T&,const T*> const_iterator;
  typedef reverse_iterator<const_iterator>     const_reverse_iterator;
  typedef reverse_iterator<iterator>           reverse_iterator;
  typedef value_type&                          reference;
  typedef const value_type&                    const_reference;
  typedef size_t                               size_type;
  typedef ptrdiff_t                            difference_type;
  typedef Alloc                                allocator_type;
  ...
};
```

As is usual for C++ classes, `list` has several constructors, a destructor, and an assignment operator. These member functions are vital for allowing a user to treat an object of type `list` as if it were a built-in type, giving the object ***value semantics***. For example, just like an object of type `int`, a list object may be declared as a local variable (allocated on the stack), it may be assigned and copied, and when the variable goes out of scope, the lifetime of the object ends.

```
{
  std::list<int> x;      // default constructor is called
  x.push_back(1);
  std::list<int> y = x;  // copy x into y
  x.push_back(2);
  assert(y.size() == 1);
  assert(x.size() == 2);
}                        // the destructors for x and y are called
```

The list object contains a pointer to heap allocated nodes, so the implicit call to the destructor is needed to allow for the manual deletion of this memory. The following are the constructors, destructors, and related member functions for `list`.

```
template<typename T, typename Alloc = allocator<T> >
class list {
public:
  ...
  explicit list(const allocator_type& a = allocator_type());
  list(size_type n, const value_type& value,
       const allocator_type& a = allocator_type());
  explicit list(size_type n);
  list(const list& x);
  template<typename InIter>
  list(InIter first, InIter last, const allocator_type& a = allocator_type());
  ~list();

  list& operator=(const list& x);
  void assign(size_type n, const value_type& val);
  template<typename InIter> void assign(InIter first, InIter last);
  void swap(list& x);
  allocator_type get_allocator() const;
  ...
};
```

Two of the above `list` members are of interest because they are member function templates parameterized on an `InIter` type. The type requirements for `InIter` are:

- `InIter` must be a model of Input Iterator.
- `InIter`'s value type must be convertible to the value type of the list.

The `list` class provides iterators so that it may be used with the STL sequence algorithms. The `list` iterators are models of the Bidirectional Iterator concept; the nodes are doubly-linked so they enable both forward and backward traversal. The `begin` and `end` member functions return iterators pointing to the first elements and pointing just after the last element, respectively. There are constant iterators for read-only access and mutable iterator for read-write access. The `list` class also provides reverse iterators that flip the direction of traversal. These iterators are implemented using the `reverse_iterator` adaptor which is described in Section 2.2.3.

```
template<typename T, typename Alloc = allocator<T> >
class list {
public:
  ...
  iterator begin();
  const_iterator begin() const;

  iterator end();
  const_iterator end() const;

  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;

  reverse_iterator rend();
  const_reverse_iterator rend() const;
  ...
};
```

The rest of the member functions of `list` are typical of a container class. We list them below for completeness.

```
template<typename T, typename Alloc = allocator<T> >
class list {
public:
  ...
  bool empty() const;
  size_type size() const;
  size_type max_size() const;
  void resize(size_type new_size, const value_type& x);
  void resize(size_type new_size);
  void clear();

  reference front();
  const_reference front() const;

  reference back();
  const_reference back() const;

  void push_front(const value_type& x);
  void pop_front();
  void push_back(const value_type& x);
  void pop_back();

  iterator insert(iterator position, const value_type& x);
  void insert(iterator pos, size_type n, const value_type& x);
  template<typename InIter>
```

```
  void insert(iterator pos, InIter first, InIter last);

  iterator erase(iterator position);
  iterator erase(iterator first, iterator last);

  void remove(const T& value);
  template<typename Predicate>
  void remove_if(Predicate);

  void splice(iterator position, list& x);
  void splice(iterator position, list&, iterator i);
  void splice(iterator position, list&, iterator first, iterator last);

  void unique();
  template<typename BinaryPredicate>
  void unique(BinaryPredicate);
  void merge(list& x);
  template<typename StrictWeakOrdering>
  void merge(list&, StrictWeakOrdering);
  void reverse();
  void sort();
  template<typename StrictWeakOrdering>
  void sort(StrictWeakOrdering);
};
```

2.2.3. Adaptors and container concepts. An adaptor class transforms the interface or behavior of other classes. The adapted class is typically assumed to satisfy the requirements of some concept and the adaptor class usually implements the requirements of another concept. The STL has several adaptors, listed below, that implement iterator concepts on top of containers. Thus, a family of container concepts are needed to express the requirements of these adaptors. Figure 13 shows the refinement hierarchy for the container concepts.

`back_insert_iterator`: adapts a Back Insertion Sequence and implements Output Iterator.

`front_insert_iterator`: adapts a Front Insertion Sequence and implements Output Iterator.

`insert_iterator`: adapts a Container and implements Output Iterator.

The following example computes the set difference of two arrays of integers using the generic `set_difference` algorithm. The output is stored in a vector and the vector is

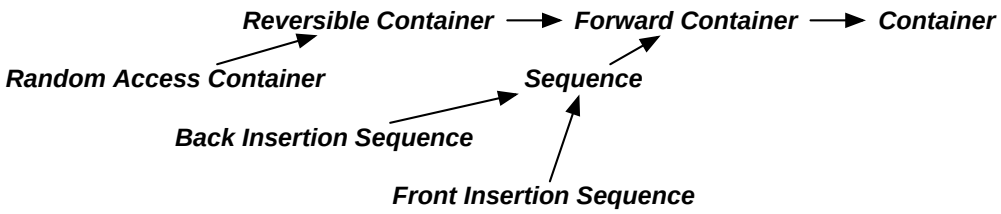

FIGURE 13. The refinement hierarchy of container concepts.

adapted to the expected Output Iterator interface using `back_insert_iterator`. The function `back_inserter` provides a convenient way to create a `back_insert_iterator`.

```
int a1[] = { 0, 1, 2, 3, 4, 5, 6 };
int a2[] = { 1, 4, 5 };
std::vector<int> a3;
int a4[] = { 0, 2, 3, 6 };
std::set_difference(a1, a1 + 5, a2, a2 + 3, std::back_inserter(a3));
assert(std::equal(a3.begin(), a3.end(), a4));
```

The Container concept and same-type constraints. The Container concept requires a modeling type to provide several associated types, including a `value_type` and an `iterator` type that models Input Iterator. The `iterator` type has its own associated `value_type`. The Container concept requires that the `iterator`'s `value_type` be the same type as the container's `value_type`.

The Sequence concept and parameterized function requirements. The Sequence concept includes the requirement for an `insert` function that inserts a range of elements from another sequence. The range is specified as a pair of arbitrary Input Iterators. Thus, a class that models Sequence must implement `insert` as a template member function. The `list` class, for example, includes an `insert` member function template.

Reverse iterators and conditional models. The `reverse_iterator` class template adapts a Bidirectional Iterator and implements Bidirectional Iterator, flipping the direction of traversal, so `operator++` goes backwards and `operator--` goes forwards. An excerpt from the `reverse_iterator` class template is shown below.

```
template<typename Iter>
class reverse_iterator {
```

```
protected:
  Iter current;
public:
  explicit reverse_iterator(Iter x) : current(x) { }
  Iter base() const { return current; }
  reference operator*() const { Iter tmp = current; return *--tmp; }
  reverse_iterator& operator++() { --current; return *this; }
  reverse_iterator& operator--() { ++current; return *this; }
  reverse_iterator operator+(difference_type n) const
    { return reverse_iterator(current - n); }
  ...
};
```

The `reverse_iterator` class template is an example of a type that models a concept conditionally: if the `Iter` type models Random Access Iterator, then so does `reverse_iterator<Iter>`. The definition of `reverse_iterator` defines all the operations, such as `operator+`, required of a Random Access Iterator. The implementations of these operations rely on the Random Access Iterator operations of the underlying `Iter`. One might wonder why `reverse_iterator` can be used on iterators such as `list<int>::iterator` that are bidirectional but not random access. The reason this works is that a member function such as `operator+` is type checked and compiled only if it is used.

The `find_end` algorithm is a nice example of the reuse enabled by adaptors such as `reverse_iterator`. The `find_end` algorithm searches within the first sequence (`first1` and `last1`) for the last subsequence that matches the second sequence (`first2` and `last2`). Figure 14 shows the version of the `find_end` algorithm for Bidirectional Iterators. This version is implemented with `search` and `reverse_iterator`. The `search` algorithm finds the first matching subsequence, so applying this algorithm in reverse finds the last matching subsequence.

Container adaptors. The STL contains the following three container adaptors.

`stack`: adapts a Back Insertion Sequence and implements a last-in-first-out interface.

`queue`: adapts type that models both Back Insertion Sequence and Front Insertion Sequence and implements a first-in-first-out interface.

FIGURE 14. The `find_end` algorithm implemented with `search` and `reverse_iterator`.

```
template<typename BidirIter1, typename BidirIter2>
BidirIter1
find_end_dispatch(BidirIter1 first1, BidirIter1 last1,
                  BidirIter2 first2, BidirIter2 last2,
                  bidirectional_iterator_tag, bidirectional_iterator_tag)
{
  typedef reverse_iterator<BidirIter1> RevIter1;
  typedef reverse_iterator<BidirIter2> RevIter2;
  RevIter1 rlast1(first1);
  RevIter2 rlast2(first2);
  RevIter1 rresult = search(RevIter1(last1), rlast1,
                            RevIter2(last2), rlast2);
  if (rresult == rlast1)
    return last1;
  else {
    BidirIter1 result = rresult.base();
    advance(result, -distance(first2, last2));
    return result;
  }
}
```

`priority_queue`: adapts a Random Access Container and a comparison function and implements a queue interface where the element with the highest priority is first to leave the queue.

2.2.4. Summary of language requirements. In this chapter we surveyed how generic programming is accomplished in C++, taking note of the variety of language features and idioms that are used in current practice. In this section we summarize the findings as a list of requirements for a language to support generic programming.

(1) The language provides type parameterized functions with the ability to express constraints on the type parameters. The definitions of parameterized functions are type checked independently of how they are instantiated.

(2) The language provides a mechanism, such as “concepts”, for naming and grouping requirements on types, and a mechanism for composing concepts (refinement). Concepts should be allowed to have multiple parameters.

(3) Type requirements include:
   - requirements for functions and parameterized functions
   - associated types
   - requirements on associated types
   - same-type constraints
   - conversion requirements

(4) The language provides an implicit mechanism for providing type-specific operations to a generic function, but this mechanism should maintain modularity (in contrast to argument dependent lookup in C++).

(5) The language implicitly instantiates generic functions when they are used.

(6) The language provides a mechanism for concept-based dispatching between algorithms.

(7) The language provides function expressions and function parameters.

(8) The language supports conditional modeling.

(9) The language provides a mechanism for creating abstract data types, such as `list`, that manage some private resources. It should be possible to implement ADT's that behave like built-in types in that they have value semantics.

2.3. Relation to other methodologies

This section describes the place of generic programming within the larger realm of software engineering. The relationship between generic programming and other software construction techniques and methodologies is discussed.

Object-Oriented Programming. Definitions of object-oriented programming vary, but the equation

$$\text{object} = \text{data} + \text{functions}$$

is central to all of them: an object consists of data fields and pointers to functions. In generic programming there is also a connection between data and operations on the data, but the operations are not physically attached to the data. Instead, generic programming adheres to the view that data types and operations on those types are grouped (usually in

a module) to form an abstract data type. Instead of physically attaching functions to data, they are merely logically attached.

$$\text{abstract data type} = \text{representation types} + \text{functions}$$

This subtle difference has many repercussions, both in program design and in programming language design. When designing a program using object-oriented techniques, there is a strong motivation to assign each function to a particular class. This is difficult to do when there are multiple classes in tight collaboration. With generic programming, such classes are simply placed in the same module together with free-standing functions.

Another repercussion on language design concerns data-encapsulation and information hiding. In object-oriented languages, protection is associated with classes, whereas with abstract data types the information hiding occurs at the module level. Again, in the situation where multiple classes are in tight collaboration, module level protection is a better fit; with class level protection one is forced to bypass the protection by granting friendship. It makes sense to ask the question, *who* do we need to deny access to? It is certainly important to protect the internals of a module from users of the module, but why protect two different parts of a module from each other, especially when the two parts are either implemented by the same programmer or by programmers working in close collaboration?

Another idea central to object-oriented programming is late binding (dynamic dispatch). This allows object-oriented programs to be extremely flexible. Dynamic dispatch provides a mechanism for data-directed programming [4]: dispatching to different routines based on the run-time type of the data.

Generic programming, with its emphasis on libraries, more often relies on static binding: fixing generic parameters and operations at "sale time" (when the application program is compiled and the library linked in). One interesting question is whether polymorphism based on concepts can be extended to allow for run-time dispatch. Indeed, this is the case, through the use of existential types [111, 130]. For a long time existential types were a research language novelty but they are starting to see more widespread use [42, 107, 111].

It is also interesting to ask whether generic programming can be accomplished in object-oriented languages. The answer is yes. For example, the Template Method design pattern [68] can be used to define generic algorithms. However, object-oriented languages are not a particularly good fit for generic programming, which is discussed in depth in Section 3.2.

Perhaps the most important difference between generic programming and object-oriented programming is the emphasis that generic programming places on algorithms. Generic algorithms are separated from objects and classes; this separation allows for the high degree of reuse in generic libraries. Object-oriented libraries often attach algorithms to particular classes and in doing so miss opportunities for reuse.

Functional Programming. There are two characteristics that are often associated with functional programming: higher-order functions and a lack of side-effects. As mentioned at the beginning of this chapter, generic programming was inspired by ideas from functional programming languages. In particular, the appearance of higher-order functions (then called operators) in early functional languages such as FP [13] and APL [61, 87] inspired the higher-order approach to generic programming used by Stepanov and Musser in their generic Scheme libraries [180].

At first, Stepanov was fond of the absence of side-effects in some functional languages. However, he was soon convinced by Aaron Kershenbaum that side effects were necessary because many efficient algorithms and data-structures are deeply imperative [105, 169]. For example, the best purely functional implementation of Dijkstra's single-source shortest paths is $O((V + E) \log V)$ [82] while the best imperative implementation is $O(V \log V + E)$ [49]. Generic programming methodology places a high priority on efficiency, so the most efficient algorithm (including constant factors) is always chosen, regardless of whether it is imperative or pure-functional.

Generative Programming. is an approach to the automatic generation of families of software components developed by Czarnecki and Eisenecker [51]. Generative programming concerns the construction of generators whose inputs are specifications expressed in a domain specific language (DSL) and whose outputs are software components assembled from

many pre-built components based on some configuration knowledge. Generic programming is one of the key implementation technologies used to build the components within a generative system (Czarnecki and Eisenecker dedicate a chapter to generic programming in their book). Metaprogramming techniques can be used to automatically select and compose generic components. Both generic and generative programming place a strong emphasis on domain engineering and on the analysis of the common and variable properties of elements in a domain. Closely related to generative programming is the Intentional Programming of Simonyi, also described by Czarnecki and Eisenecker in [51]. The main idea behind Intentional Programming is the use of structured editors to lower the cost of creating new DSLs and to make it possible to use several DSLs together (similar to the use of several software libraries in an application). The underlying representation for DSLs is abstract syntax trees which are manipulated and transformed within the Intentional Programming framework.

Software Product Lines. The Software Engineering Institute has developed a framework, called Software Product Lines [46], for managing software product lines that target specific market segments or problem domains. The key to this approach is improving the time to market and quality of the product lines by building on a shared set of software libraries and frameworks. As in generic programming, there is an emphasis on domain engineering and the systematic organization of the libraries. Thus, generic programming can be seen as an enabling technology for software product lines.

Aspect-Oriented Programming. AOP [106] consists of methodologies and tools for separating the cross-cutting concerns of a program into aspects. Each aspect is programmed separately and then combined to form the program using an aspect weaver. Generic programming traditionally deals with the separation of concerns through function parameterization, separating data structure concerns from algorithm concerns. However, function parameterization is not ideal for other cross-cutting concerns. Thus, generic libraries could benefit from aspect-oriented programming. At the same time, aspects can benefit from the abstraction and parameterization afforded by generic programming. Some work in this vein has already begun [119, 175].

Parameterized Programming. This is a programming methodology developed by Goguen and colleagues [72, 74]. It is very similar to generic programming: it emphasizes abstraction and the construction of parameterized components. For example, the analogue of a concept in parameterized programming is a ***theory***. Parameterized programming has not been applied to the same extent as generic programming to the construction of libraries of algorithms, such as the STL, nor does parameterized programming have the same emphasis on efficiency that is characteristic of generic programming. Also, parameterized programming traditionally uses parameterized modules whereas generic programming more often relies on parameterized functions.

Metaprogramming. There is a close relationship between metaprogramming and generic programming, and they are often confused, especially with regards to C++ template metaprogramming [5]. Metaprogramming, in general, deals with various programming techniques and language features for code generation and for compile-time (or more generally, staged) computation [75, 165, 183]. Template metaprogramming is used in generic C++ libraries to generate customized implementations of data-structures [6, 51, 169, 174] and sometimes to specialize parts of algorithms. Generic programming and template metaprogramming, however, are distinct in that generic programming is primarily concerned with constructing type-independent components whereas template metaprogramming usually consists of type-dependent computations.

Model Driven Architecture. The Object Management Group has coined the phrase Model Driven Architecture (MDA) [177] to refer to a style of software development that emphasizes the modeling of both problem domain abstractions, with high-level platform independent models (PIMs), and solution domain abstractions, with mid-level platform specific models (PSMs), using the Unified Modeling Language [145] and similar standards. Associated with MDA are tools for generating executable specifications, or even programs, from UML models. There are links between MDA and generative programming: UML can be viewed as a particular framework for defining domain specific languages which can then be used within a generative system to construct components.

In generic programming, the results of domain engineering are usually captured with concept definitions, using the semi-formal specification language of generic programming. It is possible to embed concept definitions in UML. For example, Eichelberger modeled the STL in UML [57] but found it is necessary to extend the basic UML constructs because the built-in features of UML are biased towards object-oriented designs.

2.4. Summary

This chapter introduced the generic programming methodology of Stepanov and Musser by stepping through the process of creating a generic `accumulate` function. We then analyzed the language features needed to implement the Standard Template Library (STL). Our goal is to meet these needs in the design of $\mathcal{G}$. The last section of the chapter described the relationship between generic programming and other programming methodologies.

polymorphism: the quality or state of existing in or assuming different forms

Webster's Dictionary

Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure. Ad-hoc polymorphism is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type.

Christopher Strachey [182]

3

The language design space for generics

This chapter surveys and evaluates the design space of ***generics***, that is, language features relating to the parameterization of components on types. I evaluate the points in this design space with respect to how well they support generic programming, and in particular, whether they meet the requirements discussed in Chapter 1 and in Section 2.2.4. This evaluation serves as the rationale for the fundamental design decisions of $\mathcal{G}$. This chapter includes material from our earlier study of language support for generic programming [69].

3.1. Preliminary design choices

The focus on generics presupposes several design decisions, such as whether to use static typing or dynamic typing and whether to use type parameters or subtype polymorphism.

Thus, this chapter begins with a brief discussion of higher-level design decisions before analyzing the design space for generics.

Types as contracts. A large part of this chapter is dedicated to discussing type systems. Type systems are typically used to detect errors and to aid the compiler in producing efficient executables. My interest in types is primarily as a lightweight language for expressing contracts between generic components and their users.

With the types-as-contracts view, the type system plays the role of the contract enforcer: it identifies errors (contract violations) and determines which piece of code is responsible, either the user's code or the library code. This role is vital, for without it, when something goes wrong during the use of a component, the user may find it difficult to determine the problem. For example, to understand the problem, the user may need knowledge of the component's implementation. This is knowledge the user should not be expected to have. Thus I view a type system as a tool that facilitates black-box reuse.

There are many semantic aspects of a component's interface that are not easily captured with types. Nevertheless, types are a convenient way to express some basic assumptions about the input and output of a component.

Semantic and behavioral contracts. There is a considerable body of research on expressing semantic contracts and putting them to use. There are numerous specification languages that employ some form of logic, such as Larch [76], Z [178], CASL [47], Tecton [101], OBJ [73], and ANNA [109, 194]. There are also many languages and tools for annotating programs with assertions such as Eiffel's support for Design by Contract [127].

Semantic specifications can be put to use in a number of ways. One straightforward use of specifications is to generate run-time checks [62, 63, 159, 162]. Such checks can help identify who is to blame when a contract violation occurs during program execution.

Another use of specifications is in formal program verification. There is ongoing work to develop formal methods for verifying generic algorithms by Musser [134, 135] using the Athena theorem prover [7]. Several other theorem provers also show promise for formal generic programming with their support for concept-like abstractions: axiomatic type classes in the Isabelle-Isar system [143, 144, 199] and theory parameters in PVS [161].

Despite the advances in tool support there are a number of hurdles to overcome before formal program verification can be directly applied to generic algorithms. One of the hurdles is dealing with pointers and arrays in a way that scales to complex algorithms. There has been recent progress in this area, for example, the work by Bornat [23] and the development of separation logic by Reynolds and colleagues [158]. However, much more research is needed to scale these ideas up to production languages.

Another use of semantic specifications is in optimizing compilers. Many compilers apply simplifications and rewrites based on properties they know to be true of scalar types such as `int` and `float`. With the appropriate semantic specifications, these compiler optimizations can also be applied to user-defined types, as shown by Schupp, Gregor, Musser, and Liu [163]. Extending $\mathcal{G}$ with semantic contracts is a promising area of future research, but it is beyond the scope of this thesis.

Static vs. dynamic type checking. Type checking can be performed during compilation, in which case a type is attached to each program point. Type checking can also happen during execution, by examining type tags that are attached to an object. Either of these approaches is compatible with the use of types as contracts. In the dynamic setting, a type annotation in the interface of a function would correspond to a run-time guard that would be checked during calls to the function.

Static type checking has benefits and costs: it aids with the early detection of bugs but forces the programmer to type check the entire program before running and testing parts of the program. It would be nice to have a single language that provides both static and dynamic type checking, allowing the programmer to gradually add type annotations and type check more of the program as development progresses. There has been some work on integrating static and dynamic type systems. For example, the soft types of Cartwright and Fagan [36] and dynamic types [2]. However, there has been little work on implicitly mixing static and dynamic type checking within a single program.

It would be interesting to integrate both static and dynamic checking in the design for $\mathcal{G}$. However, this integration would be non-trivial to implement because $\mathcal{G}$ relies on many forms of complex type-based dispatching. With a static type system, the dispatching

is resolved during compilation and therefore no run-time support is needed. However, if type checking were delayed until runtime, a complex run-time system would be needed to perform type-based dispatching. Because the integration of static and dynamic typing is non-trivial and not central to the goals of this thesis, I leave this for future research. In this thesis I restrict my attention to static type systems.

Explicit vs. implicit type annotations. Many languages allow the programmer to omit type annotations for function parameters, local variables, etc. while retaining static type safety. The type of every declaration and expression is instead inferred by the type system. This provides some of the convenience of dynamically typed languages.

However, type inferencing has some disadvantages:

- Type inferencers often produce error messages that are difficult to understand. In particular, the messages sometimes point to a line that is not the real cause of the type error. There has been work on improving error reporting [19, 43, 80, 201], but these approaches add yet more complication to the type inferencing system.
- The programming language is often constrained to make type inferencing tractable. For example, function overloading (not type classes but the conventional overloading of a function name) and first-class polymorphism are two features typically not included in languages with type inferencing. Adding support for first-class polymorphism is an active area of research [24, 97, 112, 148, 150, 155].

In the design of $\mathcal{G}$ we balance the complexity and usability of the type system with the convenience of omitting type annotations. We allow programmers to omit annotations in frequently occurring locations: we infer the types of local variables from their initializing expression and we perform implicit instantiation of polymorphic functions. Neither of these forms of inference require Hindley-Milner style inferencing and can be incorporated into a conventional type system (similar to that of System F) that produces easy to understand error messages. We do require type annotations on function parameters. Function parameters occur less frequently in code and are a valuable form of specification. A programmer

can write down the type signature of a function and then check whether the implementation matches the intended specification. Also, the type annotations are a valuable source of documentation and making it part of the type system allows for automatic consistency checking.

3.2. Subtyping versus type parameterization

The decision to focus on static type systems poses a challenge because simple static type systems, such as those of Pascal and Fortran, inhibit the expression of generic functions. Each parameter of a function has a particular concrete type, preventing a function from being used with any other data types. There are two approaches to enabling the use of a function with different types: subtype polymorphism and type parameterization. In this section I present reasons for preferring type parameterization to subtyping for the purposes of generic programming.

With subtype polymorphism, a function's parameter types are base classes (interfaces). A call to the function is valid so long as the argument types are subtypes of the parameter types, respectively. To enable this, the type system includes a rule known as the ***subsumption principle***: an expression of type σ may be implicitly coerced to type τ if σ is a subtype of τ. The following small Java program demonstrates subtype polymorphism. The function `identity` is used with two different types, `A` and `B`, both of which are subtypes of `I`.

```
interface I { }
class A implements I { }
class B implements I { }

class Main {
  public static I identity(I i) { return i; }
  public static void main(String args[]) {
    A a = new A();
    I a2 = identity(a);
    B b = new B();
    I b2 = identity(b);
  }
}
```

With type parameterization, the function's parameter types are themselves parameters. The following Java 1.5 program shows the `identity` function parameterized on type `T`. When `identity` is called with `c` (which is an instance of `C`), then `C` is substituted for `T`, whereas when `identity` is called with `d`, `D` is substituted for `T`. Thus, instead of the argument changing its type to match the function as with subtyping, the function is instantiated to match the type of the argument.

```
class C { }
class D { }

class Main2 {
  public static <T> T identity(T i) { return i; }
  public static void main(String args[]) {
    C c = new C();
    C c2 = identity(c);
    D d = new D();
    D d2 = identity(d);
  }
}
```

The return type of the parameterized `identity` function is more accurate than the return type of the `identity` that uses subtyping. The lack of accuracy with subtyping leads to significant difficulties in using subtyping to implement generic algorithms.

3.2.1. The binary method problem. The well known ***binary method problem*** [29] is particularly problematic for generic algorithms. For example, the accumulate example from Section 2.1 includes two instances of the binary method problem, one of them concerning the Monoid concept. The following is a Java interface definition for this concept.

```
interface Monoid {
  Monoid binop(Monoid other);
  Monoid id_elt();
};
```

The problem arises when we try to define a class, such as the following `IntAddMonoid`, that is a subtype of `Monoid`.

```
class IntAddMonoid implements Monoid {
  public IntAddMonoid(int x) { n = x; }
```

```
  public IntAddMonoid binop(IntAddMonoid other)
    { return new IntAddMonoid(n + other.n); }
  public IntAddMonoid id_elt() { return new IntAddMonoid(0); }
  public int n;
};
```

The Java type system rejects the above definition because `IntAddMonoid` fails to override the `binop` method of `Monoid`. The `binop` method in `IntAddMonoid` is not an override because the parameter type differs from the parameter type in `Monoid`.

The language rules can not be changed to allow covariant parameters (parameter types that change with the method's class) without either making the type system unsound or relying on whole program checks [28, 48, 85]. To see how allowing covariance makes a type system unsound, consider the following program. We define another derived class of `Monoid` and then assign instances of `IntAddMonoid` and `DoubleAddMonoid` to variables of type `Monoid` and invoke the `binop` method.

```
class DoubleAddMonoid implements Monoid {
  public DoubleAddMonoid(double x) { n = x; }
  public DoubleAddMonoid binop(DoubleAddMonoid other)
    { return new DoubleAddMonoid(n + other.n); }
  public DoubleAddMonoid id_elt() { return new DoubleAddMonoid(0.0); }
  public double n;
};

class Main {
  public static void main(String args[]) {
    Monoid a = new IntAddMonoid(1);
    Monoid b = new DoubleAddMonoid(1.0);
    b = a.binop(b);
  }
}
```

The method call will resolve to the `binop` in `IntAddMonoid` which tries to use member `n` of object `b` as an `int`, but this member is a `double`, so there is a type error. A sound type system must catch such problems during type checking.

Type soundness can be regained with whole program checks, but this is incompatible with the goal of providing separate type checking for individual functions and modules within a program.

The workaround for the binary method problem is to use `Monoid` as the parameter for `binop` and then cast to `IntAddMonoid`.

```
class IntAddMonoid implements Monoid {
  public IntAddMonoid(int x) { n = x; }
  public IntAddMonoid binop(Monoid other)
    { return new IntAddMonoid(n + ((IntAddMonoid)other).n); }
  public IntAddMonoid id_elt() { return new IntAddMonoid(0); }
  int n;
};
```

However, this introduces clutter and opens the door to run time errors. Methods with covariant parameters, such as `binop`, are pervasive so this is a serious problem for implementing and using generic algorithms based on subtype polymorphism.

In contrast to covariant argument types, covariant return types cause no type soundness problems and are allowed in Java 1.5.

3.2.2. Associated types. Many object-oriented languages lack a facility for dealing with associated types, so for example, the `Iterator` interface would have to use the `Object` type for the return type of the `curr()` method instead of the precise type of its elements. This forces the user of the iterator to insert casts, again cluttering the code and opening the door to run-time errors.

```
interface Iterator {
  boolean equal(Iterator other);
  Object curr();
  Iterator next();
};
```

The following is the definition of the generic accumulate algorithm using subtyping.

```
class Accumulate {
  public static Monoid run(Iterator first, Iterator last, Monoid factory) {
    if (first.equal(last)) return factory.id_elt();
    else return ((Monoid)first.curr()).binop(run(first.next(), last, factory));
  }
};
```

One minor irritation in the above definition is handling the case for an empty sequence. We need to invoke the `id_elt()` method but have no elements on which to invoke it, hence the extra `factory` parameter.

3.2.3. Virtual types. One of the proposed solutions for dealing with binary methods and associated types in object-oriented languages is ***virtual types***, that is, the nesting of abstract types in interfaces and type definitions within classes or objects. The beginning of this line of research was the ***virtual patterns*** feature of the BETA language [110]. Patterns are a generalization of classes, objects, and procedures. An adaptation of virtual patterns to object-oriented classes, called ***virtual classes***, was created by Madsen and Moller-Pedersen [123] and an adaptation for Java was created by Thorup [186]. These early designs for virtual types were not statically type safe, but relied on dynamic type checking. However, a statically type safe version was created by Torgersen [189]. A statically type safe version of BETA's virtual patterns was developed for the gbeta language of Ernst [58, 59]; the Scala programming language also includes type safe virtual types [146, 147].

It turns out that virtual types can be viewed as a kind of type parameterization. To give an intuition for this, we show an implementation of the accumulate example in Scala using virtual types. The following are Scala traits (interfaces) for the Iterator and Monoid concepts.

```
trait Iterator {
  type iter;
  type elt;
  def curr(x: iter): elt;
  def next(x: iter): iter;
  def equal(x: iter, y: iter): Boolean;
}
```

```
trait Monoid {
  type a;
  def id_elt: a;
  def binop(x: a, y: a): a;
}
```

The accumulate algorithm can now be written as follows. We encapsulate the generic accumulate function in an abstract class with virtual types `i` and `t`, which effectively serve as type parameters. The `monoid` and `iterator` value parameters are used here analogously to structures in ML with subtyping playing the role that signature matching plays in ML.

The type of the parameters `first` and `last` is `i` and the return type is `t`, so here we are relying on type parameterization instead of subtype polymorphism.

```
abstract class accumulate {
  type i;
  type t;
  def run(monoid: Monoid{type a = t},
          iterator: Iterator{type iter = i; type elt = t})
  (first: i, last: i): t =
    if (iterator.equal(first, last)) monoid.id_elt
    else monoid.binop(iterator.curr(first),
                      run(monoid, iterator)(iterator.next(first), last));
}
```

The notation

```
Monoid{type a = t}
```

creates an anonymous class derived from `Monoid` with type `t` bound to the abstract type `a`. To use accumulate, we must first provide bindings for the abstract types `i` and `t` to obtain a concrete class. We then create an instance object and invoke the `run` method.

```
val s: Int = new accumulate{type i=List[Int]; type t = Int}
    .run(intMonoid, makeIterator(intEq))(lsi, List());
```

The virtual types `i` and `t` act just like type parameters and the process of creating an anonymous derived class with bindings for the virtual types is just like instantiating a parameterized class. So virtual types can be seen as one approach to adding type parameterization to object-oriented languages.

3.2.4. Evaluation. Some form of polymorphism is necessary to enable the expression of generic algorithms in a statically typed language. The two main forms of polymorphism are subtype polymorphism and type parameterization. This section showed that subtype polymorphism is not suitable for expressing generic algorithm due to its imprecision. In particular, it suffers from the binary method problem and does not provide a way to accurately track relationships between types. Given the problems with subtype polymorphism, we focus on type parameterization as the mechanism for polymorphism in $\mathcal{G}$.

3.3. Parametric versus macro-like type parameterization

There are subtle and important differences from language to language in the meaning of type parameterization. It is useful to distinguish between parameterization that relies on ***parameteric polymorphism*** versus ***macro-like parameterization*** mechanisms.

Parametric polymorphism. With parametric polymorphism, a parameterized function is a single object that can be viewed as having many types. For example, the following function (written in ML) can have the type `fn : int -> int` or `fn : real -> real` or any type of the form `fn : 'a -> 'a` where `'a` is a type variable.

```
- fun id x = x;
val id = fn : 'a -> 'a
- id 1;
val it = 1 : int
- id 1.0;
val it = 1.0 : real
```

A polymorphic function such as `id` is like a chameleon, it can change it's color (type) at will. The reason that `id` may take on different types is that parametric polymorphism requires the body of the function to take a "hands off" approach with any object whose type is a type parameter. Type-specific operations may not be applied to such objects. Therefore, the computation of the function is independent of the type parameters.

This "hands off" restriction seems rather stringent at first, but it is not as bad as it seems. A parametric function may be passed function parameters that perform type-specific operations. For example, the following `map` function applies the `int`-specific `dub` function to every element of a list.

```
- fun map f [] = []
-   | map f (x::ls) = (f x)::map f ls;
val map = fn : ('a -> 'b) -> 'a list -> 'b list

- fun dub x = x + x;
val dub = fn : int -> int

- map dub [1,2,3];
val it = [2, 4, 6] : int list
```

Macro-like type parameterization. The macro-like approach to type parameterization is exemplified by C++ templates. An early precursor to templates can be seen in the definition facility in ALGOL-D [66, 67] of Galler and Perlis. In C++ a function template is not itself a function, nor does it have a type. Instead, a function template is a generator of functions. In the following program, the compiler generates two functions from the original template, one for `int` and one for `double`.

```
template <typename T>
T id(T x) { return x; }

int main() {
  id(1);       // id<int> is generated during compilation
  id(1.0);     // id<double> is generated during compilation
}
```

Whereas a parametric polymorphic function is like a chameleon that can change its color, a function template is like a lizard farm, producing lizards of many different but fixed colors.

A function template produces different functions for different type arguments. This behavior is often used to produce more optimized versions of a function for specific types. For example, the `converter` class in the Boost libraries [22] converts between two numbers of different type. Normally, the `converter` checks to make sure the input number is representable in the output type. However, if the range of the output type encloses the range of the input type, no check is needed and so the check is omitted for the sake of efficiency. The following program shows a simple use of the `converter` class to flip a pair of numbers inside a function template.

```
template<typename T, typename U>
pair<T,U> flip(pair<T,U> p) {
  U a = converter<U,T>::convert(p.first);
  T b = converter<T,U>::convert(p.second);
  return make_pair(b,a);
}
int main() {
  pair<double,float> p1 = make_pair(3.1415, 1.6180f);
  pair<float,double> p2 = flip(p1);
  cout << "(" << p2.first << "," << p2.second << ")\n";
}
```

The converter template produces functions that performs different actions given different type arguments. The function converter<float,double>::convert (double to float) performs a check whereas the function converter<double,float>::convert does not. In contrast, this kind of compile-time type-dispatching can not be expressed using parametric polymorphism because a polymorphic function has uniform behavior for all type arguments.

C++ templates are often categorized as a form of parametric polymorphism. Indeed, a common use of templates is to write type independent code. However, C++ templates are closer to ad-hoc polymorphism as defined by Strachey [182]: they may behave in different ways for different types.

3.3.0.1. *First-class polymorphism.* A strength of parametric polymorphism over macro-like parameterization is that polymorphic functions can be treated as first-class: they can be passed to functions and stored in data structures. The following program, written in System F [71, 157], passes the polymorphic id function as a parameter to the function f, which then views its polymorphic argument at several different types. In System F, a λ expression creates a function and Λ parameterizes an expression on a type. A parameterized object can be viewed at a particular type by providing type arguments in square brackets.

```
f ≡ λ g : ∀ t. t→t. (g[int] 1, g[real] 1.0)
id ≡ Λ t. λ x: t. x

f id
```

Many interesting uses of first-class polymorphism (also referred to as higher-rank polymorphism) may be found in an annotated bibliography by Shan [42].

Under restricted circumstances, the above can be achieved with the macro-like approach. However, if f is compiled separately from the application f id, then the macro-like approach does not work. When compiling f id, all that is known about f is its type (∀ t. t→t) → (int * real). It is not known how parameter g is used inside of f, so the instantiations of id for int and real cannot be generated when compiling f id. On the other hand, when compiling f, it is not known that id will be bound to g, so the instantiations of id can not be produced while compiling f. Of course, the instantiation of id could

be done at run-time, using just-in-time compiler technology, but this would incur significant run-time overhead and complicates the run-time system.

3.3.1. Separate type checking. As described in Chapter 1, separate type checking is vital because it lowers the cost of using generic components and improves the quality of generic components by catching errors in the implementation and by catching inconsistencies in the interface.

Separate type checking is straightforward with parametric polymorphism. To type check a polymorphic function, the type system treats type parameters as abstract types, different from any other type. Another way to say this is that a polymorphic function is type checked under the conservative assumption that the type parameter could be instantiated with any type. Once a polymorphic function has passed type checking, it is guaranteed to be well-typed for *any* type argument.

With C++ templates, type checking is performed after instantiation, once the type arguments are known, so templates are not type checked separately from their use. However, it is possible for a macro-like system to type check a template prior to instantiation. We took this approach in our proposal for extending C++ with support for concepts [168]. To ensure type soundness, some restrictions must be placed on type-dispatching. This is analogous to object-oriented method dispatch where the type system must ensure that overriding methods in derived classes conform to the method signature in the base class. Such restrictions block most kinds of metaprogramming but allows for dispatching between different versions of an algorithm according to differing type requirements.

Our proposal for C++ retains the unrestricted templates of C++ for purposes of template metaprogramming and adds a new kind of template with opaque type parameters for purposes of generic programming. This distinction between generic programming (with its focus on generic algorithms) and metaprogramming is important. Generic algorithms typically need very little type dispatching and metaprogramming, whereas *generative* components, such as Blitz arrays [193] and matrix types in MTL [174] and GMCL [50], require

highly sophisticated metaprogramming that uses compile-time type dispatching. It therefore makes sense to use two different language mechanisms to fulfill the differing needs of generic programming and generative programming.

3.3.2. Compilation and run-time efficiency. Separate compilation is important for ensuring that the time to compile a component is just a function of the component's size and not a function of the size of all components it uses (transitively). However, there are tradeoffs between compile time and run time.

With the macro-like approach to type parameterization, the compiler in general must produce a distinct sequence of machine code instructions for each instantiation of a parameterized function. Each of these sequences is *specialized* for the particular type arguments given in the instantiation. All the type-dependent dispatching is resolved at compile time and the results are hard coded into the instruction sequences. Parameter passing conventions and run-time representations of data structures are exactly the same as for non-parameterized code. The end result is highly efficient: there is no run-time overhead associated with the parameterization. However, this approach gives up separate compilation.

With parametric polymorphism, the compiler (or programmer) can choose between two different approaches. For each use of a generic function, the compiler may perform ***function specialization*** (sometimes called monomorphization), generating a type-specific sequence of instructions, or the compiler may use a uniform, or generic, code sequence for the function.

For the compiler to perform function specialization, two conditions must be satisfied: it must have access to the implementation of the polymorphic function and it must know on which types to instantiate the function. In the macro-like approach, both of these conditions are always guaranteed: separate compilation is disallowed and function templates are second-class citizens. With the polymorphic model, the story is more complicated. In languages like ML and Haskell 98, polymorphism is second-class, so it is straightforward to determine which polymorphic functions are instantiated on which types [20, 37, 98].

However, in languages with first-class polymorphism—such as System F and Quest [34]—a control flow analysis is needed to determine which polymorphic functions are instantiated on which types [191]. (The use of control flow analysis and function specialization has been studied in the setting of dynamically typed languages [39, 41, 88, 195].) Of course, the flow analysis must be conservative, so in some situations unnecessary specializations may be produced or alternatively the compiler must fall back and use non-specialized code. There are several other compiler analysis and optimizations that help enable function specialization; most of them are components of the more general technique of ***partial evaluation*** [99].

The alternative to function specialization is to compile a polymorphic function to a single sequence of machine-code instructions that works for any type arguments. There are several challenges to overcome with this approach. The first is that objects of different types may have different sizes: some fit in general registers, some fit in floating point registers, and others must be placed on the stack or heap. Thus, different instructions are needed to access function parameters and local variables depending on their type. One solution, called "boxing", is to pass pointers to objects instead of the objects themselves, since pointers have a uniform size and fit into general registers. However, this approach forces all objects to be stored in memory (even small objects). On modern computer architectures CPU speed has out-paced memory access speed, so increased memory traffic can be a significant source of overhead.

Higher-order polymorphic functions also present a challenge to uniform compilation. Consider again the polymorphic `map` function. It is parameterized on types `'a` and `'b`, and has a function parameter `f` from `'a` to `'b`.

```
val map = fn : ('a → 'b) → 'a list → 'b list
val dub = fn : int → int

- map dub [1,2,3];
val it = [2, 4, 6] : int list
```

The `map` function is applied to the function `dub : int → int`. The difficulty is that `map` passes a boxed `int` to function parameter `f`, but `dub` is expecting an unboxed `int`. Similarly, `dub` returns an unboxed `int`, but `map` is expecting a boxed `int`. The solution proposed by

Leroy is to coerce `dub` as it is passed to `map` by wrapping `dub` in a function that unboxes the input, calls `dub`, and then boxes the result [114].

A related challenge is how to layout memory for parameterized data types. With the macro-like approach, different instantiations of a parameterized data type may have a different layout. Consider the following `pair` template. When instantiated with `int`, the `second` field is typically placed at an offset of 4 bytes from the start of the struct (on a 32 bit architecture). When instantiated with `double`, the `second` field is placed at an offset of 8 bytes.

```
template<class T>
struct pair {
  T first;
  T second;
};

// pair<int> is equivalent to:
struct int_pair {
  int first;
  int second; // at offset of 4 bytes
};
// pair<double> is equivalent to:
struct double_pair {
  double first;
  double second; // at offset of 8 bytes
};
```

This non-uniformity in field layout poses a problem for the compilation of polymorphic functions because different instructions are needed to access the fields depending on the type parameters. A common solution to this problem is to box the polymorphic fields of a struct, thereby ensuring a uniform field layout. The following struct shows how this representation would look in C.

```
// a uniform representation for pair<T>
struct pair_T {
  void* first; // first and second point to objects of type T
  void* second;
};
```

The problem with this solution is that the overhead from the indirection affects both normal (monomorphic) functions and polymorphic functions. One alternative is to use non-uniform representations in monomorphic functions and uniform representations in polymorphic functions and to coerce objects as they pass between monomorphic and polymorphic code [114]. Of course, such coercions introduce run-time overhead, but at least there is no overhead in purely monomorphic code.

Another alternative, called intensional type analysis of Harper and Morrisett [79], is to pass a run-time representation of the type parameters to the polymorphic function and use this information for dispatching inside primitive operations. This idea could be applied to accessing a field of a struct in the following way. The run-time type information could include the size of the type. The instructions for field access would then use this information to compute the offset of the field within the struct. Thus, the same flattened, or unboxed, representation could be used within polymorphic code. With the approach, in non-polymorphic code, parameterized data structures are as efficient as non-parameterized data structures. In separately compiled polymorphic code, there a small amount of overhead when accessing fields of a parameterized data structure since the offset is not a constant but computed.

To summarize, parametric polymorphism can be compiled using either function specialization or uniform compilation. This choice can be made at each call site and it can be under the control of the compiler (based on a static analysis) or under the control of the programmer. With specialization, highly-efficient code is produced, but separate compilation is lost. With uniform compilation, separate compilation is achieved, but there is a constant factor of run-time overhead. Regardless of the compilation model, data-structures may be represented in their normal unboxed form by using intensional type analysis to manipulate this data within polymorphic functions.

3.3.3. Evaluation. The most difficult tradeoff in the design of $\mathcal{G}$ is between parametric polymorphism and the macro-like approach. The following points summarize the issues, with the second two points being the distinguishing factors.

separate type checking: can be achieved with both approaches.

low run-time overhead: is possible with both approaches via function specialization.

separate compilation: can be achieved with parametric polymorphism, but not with the macro-like approach.

convenient dispatching on types: is provided by the macro-like approach, but not by parametric polymorphism. In Section 6.1 I discuss how dispatching can be performed in a language based on parametric polymorphism, but at the cost of some inconvenience to the library author.

3.4. Concepts: organizing type requirements

Section 3.3 discussed how type-specific operations can be used in a polymorphic function by adding function parameters to the polymorphic function. This approach can be used to express generic algorithms: each concept operation is passed as a function parameter. This is the same approach we used to implement the `accumulate` example in Scheme in Section 2.1. The following code shows the definition and use of a generic `accumulate` written in ML. The types for the declarations are also listed.

```
fun accumulate binop id_elt next curr equal =
  fn (first, last) =>
    let fun loop first =
      if equal(first, last) then id_elt
      else binop(curr first, loop (next first))
    in loop first end;

fun sum_list ls = (accumulate (+) 0 tl hd (=))(ls,[]);
sum_list [1,2,3,4,5];

val accumulate = fn
:  ('a * 'b -> 'b) -> 'b -> ('c -> 'c) -> ('c -> 'a) -> ('c * 'd -> bool)
    -> 'c * 'd -> 'b
val sum_list = fn : int list -> int
val it = 15 : int
```

The use of function parameters to pass concept operations becomes unmanageable as the concepts become more complex and the number of parameters grow. The `accumulate`

function is rather simple and already has 5 concept operation parameters. Many of the STL and BGL algorithms would require dozens of function parameters. This section describes language mechanisms that solve this problem.

Type-specific operations are just one kind of requirement on the type parameters of a generic function, there are also associated types, same-type constraints, and conversion requirements. For large libraries of generic algorithms, the task of writing requirements for the type parameters of algorithms is a huge task that can be much simplified by reusing requirements. In fact, many algorithms share requirements: for example, the Input Iterator concept appears in the specification of 28 STL algorithms. Also, several other iterator concepts build on the Input Iterator concept, so Input Iterator is either directly or indirectly used in most STL algorithms. Thus, it is important to be able to group a set of requirements, give the grouping a name, and then compose groups of requirements to form new groups. There are a wide variety of programming language features that fulfill this role. In this section, I discuss the major alternatives in the design of concepts and evaluate them with respect to the needs of generic programming. The following list recalls from Section 2.2.4 the kinds of requirements that a concept should be able to express:

- requirements for functions and parameterized functions,
- associated types,
- requirements on associated types,
- same-type constraints,
- convertability constraints.

3.4.1. Parameteric versus object-oriented interfaces. The facilities for representing concepts in object-oriented languages differ from the facilities in languages with parametric polymorphism, such as Haskell, Objective Caml, and ML. At first glance, the differences may seem trivial but they have significant implications. To make the discussion concrete, I contrast Java interfaces with Haskell type classes. The first difference is that in the definition of a Java interface, there is no direct way to name the exact type of the modeling type. On the other hand, with Haskell type classes, a type parameter serves as a place-holder for the

modeling type. The following shows how a `Clonable` concept can be represented using interfaces and type classes.

```
interface Clonable {
  Clonable clone();
}
```

```
class Clonable a where
  clone :: a -> a
```

The return type of `clone` can not be expressed precisely in the Java interface; instead the return type is `Clonable`, which says that `clone()` may return an instance of any class derived from `Clonable`. (This inability to refer to the modeling type was also the reason for the binary method problem.) On the other hand, the return type of `clone` in the type class is precise: it is the same type as its input parameter.

The following code shows generic functions, written in Java and Haskell. In Java, the type parameter `a` is constrained to be a subtype of the `Clonable` interface. In Haskell, the type parameter `a` is constrained to be an instance of the `Clonable` type class.

```
import java.util.LinkedList;
class clone_list {
  public static <a extends Clonable>
  LinkedList<a> run(LinkedList<a> ls) {
    LinkedList<a> newls = new LinkedList<a>();
    for (a x : ls) newls.add(x.clone());
    return newls;
  }
}
```

```
clone_list :: Clonable a => [a] -> a
clone_list [] = []
clone_list (x#ls) = (clone x)#(clone_list ls)
```

The idea of using subtyping to constrain type parameters was first introduced by Cardelli [35] and later refined into F-bounded polymorphism by Canning and colleagues [33]. F-bounded polymorphism is used in Eiffel, Java, and C#.

Subtype and instance relations are quite different. For example, subtyping typically drives a subsumption rule, allowing implicit conversions, whereas the instance relation does not. Also, type substitution plays an important role in the instance relation, but not in

subtyping. For example, the following instance declaration is valid because substituting `Int` for `a` in `Clonable` gives the signature `clone :: Int -> Int` which matches the type of the `clone` function in the instance declaration.

```
instance Clonable Int where
  clone i = i
```

3.4.1.1. *Parameterized object-oriented interfaces.* With Java generics, interfaces may be parameterized on types. This provides an indirect way to refer to the modeling type. A type parameter is added to the interface and used as a place-holder for the modeling type. Then, when defining a class that inherits from the interface, the programmer follows the convention of passing the derived class as a parameter to the interface.

```
interface Clonable<Derived> {
  Derived clone();
}
class Foo implements Clonable<Foo> {
  Foo clone();
}
```

Parameterized interfaces provide a solution to the binary method problem. The following shows a definition of the `Monoid` interface and a derived class. With this version there is no need for a dynamic cast in the `binop` method. The type of parameter `other` can be `IntAddMonoid`, which exactly matches the parameter type of `Monoid<IntAddMonoid>.binop`.

```
interface Monoid<Derived> {
  Derived binop(Derived other);
  Derived id_elt();
}


class IntAddMonoid implements Monoid<IntAddMonoid> {
  public IntAddMonoid(int x) { n = x; }
  public IntAddMonoid binop(IntAddMonoid other)
    { return new IntAddMonoid(n + other.n); }
  public IntAddMonoid id_elt() { return new IntAddMonoid(0); }
  int n;
};
```

In addition to solving the binary method problem, type parameters can be used to represent associated types. For example, the Iterator concept has an associated element type,

so an `Iterator` interface can represent this with an extra `elt` parameter (the `Derived` parameter is necessary because of the `equal` binary method).

```
interface Iterator<Derived,elt> {
  elt curr();
  Derived next();
  boolean equal(Derived other);
}
```

3.4.1.2. *Concept refinement.* Composition of requirements via concept refinement is straightforward to express with both parametric and object-oriented interfaces. With object-oriented interfaces, inheritance provides the composition mechanism. The following code shows Semigroup and Monoid concepts represented as Java interfaces.

```
interface Semigroup<T> {
  T binop(T other);
};

interface Monoid<T> extends Semigroup<T> {
  T id_elt();
};
```

With parametric interfaces, such as with Haskell's type classes, subclassing is used to express refinement. The `Semigroup t =>` syntax says that an instance of `Monoid` must also be an instance of `Semigroup`.

```
class Semigroup t where
  binop :: t -> t -> t

class Semigroup t => Monoid t where
  id_elt :: t
```

3.4.1.3. *Composing requirements on associated types.* An important form of concept composition is the inclusion of constraints on associated types within a larger concept. For example, in the Boost Graph Library, there is an Incidence Graph concept with three associated types: vertex, edge, and out-edge iterator. The Incidence Graph concept includes the requirement that the edge type model the Graph Edge concept (which requires a source and target function) and that the out-edge iterator type model the Iterator concept.

In Haskell, this composition can be expressed as follows by referring to the GraphEdge and Iterator type classes in the definition of the IncidenceGraph type class.

```
class GraphEdge e v | e -> v where
  source :: e -> v
  target :: e -> v

class (GraphEdge e v, Iterator iter e) =>
  IncidenceGraph g e v iter | g -> iter where
  out_edges :: v -> g -> iter
  out_degree :: v -> g  -> Int
```

We can use IncidenceGraph to constrain type parameters of a generic function. The requirement for IncidenceGraph g e v iter implies GraphEdge e v, so it is valid to use source and target in the body of breadth_first_search.

```
breadth_first_search ::
  (IncidenceGraph g e v iter, VertexListGraph g v, BFSVisitor vis a g e v) =>
  g -> v -> vis -> a -> a
```

With Java interfaces it is not possible to express this kind of concept composition. The following shows a failed attempt to group the constraints. We define two new interfaces: GraphEdge and IncidenceGraph and use type parameters for the associated types. Also, we put bounds on the type parameters in an attempt to compose the requirement for Graph and Iterator in the requirements for IncidenceGraph. The goal is to use IncidenceGraph as a bound in a generic method and have that imply that its edge type extends GraphEdge and its out-edge iterator extends Iterator.

```
public interface GraphEdge<Vertex> {
  Vertex source();
  Vertex target();
}

interface IncidenceGraph<Vertex,
                         Edge extends GraphEdge<Vertex>,
                         OutEdgeIter extends Iterator<OutEdgeIter,Edge>> {
  OutEdgeIter out_edges(Vertex v);
  int out_degree(Vertex v);
}
```

The reason this approach fails is subtle. The following shows an attempt at writing a generic method for the breadth-first search algorithm that fails to type check.

```
class breadth_first_search_bad {
  public static <
    GraphT extends IncidenceGraph<Vertex, Edge, OutEdgeIter>
                   & VertexListGraph<Vertex, VertexIter>,
    Vertex, Edge, OutEdgeIter, VertexIter,
    Visitor extends BFSVisitor<GraphT,Vertex,Edge>>
  void run(GraphT g, Vertex s, Visitor vis) { ... }
}
```

The bounds on a type parameter must be well-formed types. So, for example, the type

```
IncidenceGraph<Vertex, Edge, OutEdgeIter>
```

must be well-formed. This type is well-formed if the constraints of `IncidenceGraph` are satisfied:

```
Edge extends GraphEdge<Vertex>
OutEdgeIter extends Iterator<OutEdgeIter,Edge>
```

However, these constraints are not satisfied in the context of the `run` method. The `run` method can be made to type check by adding the following bounds to its type parameters:

```
class breadth_first_search {
  public static <
    GraphT extends IncidenceGraph<Vertex, Edge, OutEdgeIter>
                   & VertexListGraph<Vertex, VertexIter>,
    Vertex,
    Edge extends GraphEdge<Vertex>,
    OutEdgeIter extends Iterator<OutEdgeIter,Edge>,
    VertexIter extends Iterator<VertexIter,Vertex>,
    Visitor extends BFSVisitor<GraphT,Vertex,Edge>>
  void run(GraphT g, Vertex s, Visitor vis) { ... }
}
```

Unfortunately, this defeats our original goal of grouping constraints to allow for succinct expression of algorithms. The constraints on the associated types must be duplicated in every generic algorithm that uses the `IncidenceGraph` interface.

This problem with Java's interfaces can be remedied. For example, the duplication of constraints in not necessary in the language Cecil [40]. In Cecil, bounds on type parameters

of interfaces are treated differently in the context of a generic method: they need not be satisfied and instead are added as assumptions. Going further, Järvi, Willcock, and Lumsdaine [91] propose an extension to Generic C# to add associated types and constraints on associated types to object-oriented interfaces. Virtual types, for example in Scala [146] and gbeta [58], is another approach to solving this problem.

3.4.1.4. *MyType and matching.* The programming language $\mathcal{LOOM}$ [30] provides a direct way to refer to the modeling type. The keyword `MyType` is introduced within the context of an interface to refer to the exact type of `this`. Here is what the `Monoid` interface would look like with `MyType`'s.

```
interface Monoid {
  MyType binop(MyType other);
  MyType id_elt();
}
```

It would not be type sound for $\mathcal{LOOM}$ to use inheritance in the presence of `MyType`'s to establish subtyping (and hence subsumption) since that would introduce a form of covariance. Instead, $\mathcal{LOOM}$ introduces a ***matching*** relationship and a weaker form of subsumption that does not allow coercion during assignment but does allow coercion when passing arguments to a function with a hash parameter type. This is very similar to the object types of Objective Caml, where polymorphism is provided by implicit row variables, which are a kind of parametric polymorphism. In fact, the Msg and $Msg\#$ type rules of $\mathcal{LOOM}$ (which handle sending a message to an object) perform type substitution on the type of the method, replacing `MyType`'s with the type of the receiver. Thus, interfaces with `MyType` and matching are parametric in flavor and quite different from traditional object-oriented interfaces with subtyping.

3.4.2. Type parameters versus abstract types. Among the parametric approaches to concepts there are two different ways to introduce types: type parameters and abstract types. The following shows the Iterator concept represented with a Haskell type class and an ML signature. The type class has type parameters for the iterator and element types whereas the signature has abstract types declared for the iterator and element types.

```
class Iterator iter elt | iter -> elt
  where
  next :: iter -> iter
  curr :: iter -> elt
  equal :: iter -> iter -> Bool
```

```
signature Iterator =
sig
  type iter
  type elt
  val next : iter -> iter
  val curr : iter -> elt
  val equal : iter * iter -> bool
end
```

Each of these approaches has its strengths and weaknesses. The following paragraphs argue that in fact both approaches are needed and that they are complementary to one another.

Type parameter clutter. With the type parameter approach to concepts, the main modeling type and all associated types of a concept are represented with type parameters. Each algorithm that uses the concept must have type parameters for each of the concept's parameters. This causes considerable clutter when the number of associated types grows large, as it does in real-world concepts. Part of the reason for this is that if a concept refines other concepts, it must have type parameters for each of the parameters in the concepts being refined. For example, the Reversible Container concept of the STL has 2 associated types and also inherits another 8 from Container for a total of 10 associated types. Now, if a generic function were to have two type parameters that are required to model Reversible Container, then the function would need to have an additional 20 type parameters for all the associated types. In contrast, with abstract types, a concept can be used without explicitly mentioning any of its associated types.

Implicit model passing. The strength of the type parameter approach is that it facilitates the implicit passing of models to a generic function. When a generic function is instantiated, model declarations (instances in Haskell) can be found because they are indexed by the type arguments of the concept. For example, consider the Haskell Prelude function `elem` (which indicates whether an element is in a list):

```
elem : Eq a => a -> [a] -> Bool
```

and the following function call:

```
elem 2 [1,2,3]
```

The type `Int` is deduced for the type parameter `a` and then the type requirement `Eq a` is satisfied by finding an instance declaration for `Eq Int` (this instance declaration is also in the Prelude). So the type parameters of the concept enable implicit model passing, which is an extremely important feature for making generic functions easy to use.

Best of both worlds. For the design of $\mathcal{G}$ we would like to have the best of both worlds: implicit model passing without type parameter clutter. The approach taken in $\mathcal{G}$ is to provide both type parameters and abstract types in concepts. When writing concepts, we use type parameters for the modeling type and abstract types for the associated types. So, for example, the Iterator concept could be written as follows in $\mathcal{G}$, using both a type parameter and an abstract type:

```
concept Iterator<iter> {
  type elt;
  fun next(iter c) -> iter@;
  fun curr(iter b) -> elt@;
  fun equal(iter a, iter b) -> bool@;
};
```

3.4.3. Same-type constraints. In Section 2.2.3 we discussed the Container concept and the need for same-type constraints in concepts. We needed to express that the element type of the Container is the same type as the element type of the Container's iterator. ML signatures provide support for this in the form of ***type sharing***. The following `Container` signature shows the use of type sharing to equate the elements types for the container, iterator, and reverse iterator.

```
signature Container =
sig
  type container
  type iter
  type rev_iter
  type elt

  structure Iter : Iterator
```

```
  structure RevIter : Iterator
  sharing type iter = Iter.iter
  sharing type rev_iter = RevIter.iter
  sharing type elt = Iter.elt = RevIter.elt

  val start : container -> iter
  val finish : container -> iter
  val rstart : container -> rev_iter
  val rfinish : container -> rev_iter
end
```

3.5. Nominal versus structural conformance

The fundamental design choice regarding the modeling relation is whether it should depend on the name of the concept or just on the requirements inside the concept. For example, do the below concepts create two ways to refer to the same concept or are they different concepts that happen to have the same constraints?

```
concept A<T> {
  fun foo(T x) -> T;
};
```

```
concept B<T> {
  fun foo(T x) -> T;
};
```

With ***nominal conformance***, the above are two different concepts, whereas with ***structural conformance***, `A` and `B` are two names for the same concept. Examples of language mechanisms providing nominal conformance include Java interfaces and Haskell type classes. Examples of language mechanisms providing structural conformance include ML signatures [128], Objective Caml object types [115], CLU type sets [117], and Cforall specifications [53].

Choosing between nominal and structural conformance is difficult because both options have good arguments in their favor.

Structural conformance is more convenient than nominal conformance. With nominal conformance, the modeling relationship is established by an explicit declaration. For example, a Java class declares that it `implements` an interface. In Haskell, an `instance` declaration establishes the conformance between a particular type and a type class. When

the compiler sees the explicit declaration, it checks whether the modeling type satisfies the requirements of the concept and, if so, adds the type and concept to the modeling relation.

Structural conformance, on the other hand, requires no explicit declarations. Instead, the compiler determines on a need-to-know basis whether a type models a concept. The advantage is that programmers need not spend time writing explicit declarations.

Nominal conformance is safer than structural conformance. The usual argument against structural conformance is that it is prone to ***accidental conformance***. The classic example of this is a cowboy object being passed to something expecting a `Window` [124]. The `Window` interface includes a `draw()` method, which the cowboy has, so the type system does not complain even though something wrong has happened. This is not a particularly strong argument because the programmer has to make a big mistake for this kind accidental conformance to occur.

However, the situation changes for languages that support concept-based overloading. For example, in Section 2.2.1.7 we discussed the tag-dispatching idiom used in C++ to select the best `advance` algorithm depending on whether the iterator type models Random Access Iterator or only Input Iterator. With concept-based overloading, it becomes possible for accidental conformance to occur without the programmer making a mistake. The following C++ code is an example where an error would occur if structural conformance were used instead of nominal.

```
std::vector<int> v;
std::istream_iterator<int> in(std::cin), in_end;
v.insert(v.begin(), in, in_end);
```

The `vector` class has two versions of `insert`, one for models of Input Iterator and one for models of Forward Iterator. An Input Iterator may be used to traverse a range only a single time, whereas a Forward Iterator may traverse through its range multiple times. Thus, the version of `insert` for Input Iterator must resize the vector multiple times as it progresses through the input range. In contrast, the version of `insert` for Forward Iterator is more efficient because it first discovers the length of the range (by calling `std::distance`, which traverses the

input range), resizes the vector to the correct length, and then initializes the vector from the range.

The problem with the above code is that `istream_iterator` fulfills the syntactic requirements for a Forward Iterator but not the semantic requirements: it does not support multiple passes. That is, with structural conformance, there is a false positive and `insert` dispatches to the version for Forward Iterators. The program resizes the vector to the appropriate size for all the input but it does not initialize the vector because all of the input has already been read.

Why not both? It is conceivable to provide both nominal and structural conformance on a concept-by-concept basis. Thus, concepts that are intended to be used for dispatching could be nominal and other concepts could be structural. This would match the current C++ practice: some concepts come with traits classes that provide nominal conformance whereas other concepts do not (the default situation with C++ templates is structural conformance). However, providing both nominal conformance and structural conformance complicates the language, especially for programmers new to the language, and degrades its uniformity. Therefore, with $\mathcal{G}$ we provide only nominal conformance, giving priority to safety and simplicity over convenience.

3.6. Constrained polymorphism

In this section we discuss some design choices regarding parametric polymorphism and type constraints. First we discuss at what granularity polymorphism should appear in the language and then we discuss how constraints are satisfied by the users of a generic component.

3.6.1. Granularity. Polymorphism can be provided at several different levels of granularity in a programming language: at the expression level (as in System F), at the function level (Haskell, Ada), at the class level (Java, C# Eiffel), and at the module level (ML, Ada). For libraries of generic algorithms, it is vital to have polymorphism at the function level because type requirements for an algorithm are typically unique to that algorithm. We strive to minimize the requirements for each algorithm, and the result of this minimization results

in different requirements for different algorithms. There is often commonality between requirements, which is why we group requirements into concepts, but two algorithms rarely have exactly the same requirements. Also, polymorphism should be provided at the function level to enable implicit model passing, the topic of the next subsection.

Polymorphism at the module level is sometimes useful for generic libraries, but is less important than function-level polymorphism. Polymorphism at the class level is important for defining generic containers, and polymorphism at the expression level is useful for defining polymorphic function expressions.

3.6.2. Explicit versus implicit model passing. We use the term ***model passing*** to refers to the language mechanisms and syntax by which all the type-specific operations and associated types of a model are communicated to a generic component. We say a language has explicit model passing if the programmer must explicitly pass a representation of the model to the generic component. We say a language has implicit model passing if the compiler finds and passes in the appropriate model when a generic component is instantiated.

Many languages with sophisticated module systems have support for module-parameterized modules: Standard ML [128], Objective Caml [115], Ada 95 [1], Modula-3 [141], OBJ [73], Maude [45], and Pebble [31] to name a few. With these languages, a model can be represented by a module, and a generic algorithm can be represented by a module-parameterized module. The programmer explicitly instantiates a generic module by passing in the modules that provide the type-specific operations required by the generic algorithm.

We illustrate this approach by implementing an accumulate algorithm with modules in Standard ML. In ML, a module is called a ***structure*** and a module-parameterized module is called a ***functor***. In Section 2.1 we found that accumulate operates on two abstractions: Monoid and Iterator. So here we implement accumulate as a functor with two parameters: parameter `M` for the monoid structure and parameter `I` the iterator structure.

```
functor MakeAccumulate(structure M : Monoid
                       structure I : Iterator
                       sharing type M.t = I.elt) =
```

```
struct
  fun run first last =
    if I.equal(first, last) then M.id_elt
    else M.binop(I.curr first, run (I.next first) last)
end
```

The type of a structure in ML is given by a ***signature***. The following are signature definitions for `Monoid`. (The signature `Iterator` was defined in Section 3.4.2.)

```
signature Monoid =
sig
  type t
  val id_elt : t
  val binop : t * t -> t
end
```

Abstract types such as `t` in `Monoid` and `elt` in `Iterator` are assumed to be different from each other unless otherwise specified. So the type sharing constraint in `MakeAccumulate` is necessary to allow the result of `I.curr` (which has type `I.elt`) to be passed to `M.binop` (which is expecting type `M.t`).

Applying the `MakeAccumulate` functor to two structures produces a concrete accumulate algorithm. The following produces a structure that sums an array of integers.

```
structure SumIntArray = MakeAccumulate(structure M = IntAddMonoid
                                       structure I = ArrayIter)
```

The `IntAddMonoid` and `ArrayIter` structures are defined as follows:

```
structure IntAddMonoid =
struct
  type t = int
  val id_elt = 0
  fun binop(a,b) = a + b
end
```

```
structure ArrayIter =
struct
  datatype iter = Iter of int * int Array.array
  type elt = int
  fun equal (Iter(n1,a1), Iter(n2,a2)) = (n1 = n2)
  fun curr (Iter(n,a)) = Array.sub(a,n)
  fun next (Iter(n,a)) = Iter(n+1,a)
end
```

The disadvantage of the explicit model passing is that the user must do extra work to use a generic component. We want to keep the cost of using generic components as low

as possible, so we turn our attention to various implicit language mechanisms for passing type-specific operations to a generic algorithm.

Haskell provides implicit model passing. For example, below is a generic `accumulate` function in Haskell.

```
accumulate :: (Monoid t, Iterator i t) => i -> i -> t
accumulate first last =
  if (equal first last) then id_elt
  else binop (curr first) (accumulate (next first) last)
```

The following are instance declarations establishing `Float` as a `Monoid` and `ArrayIter` as an `Iterator`.

```
instance Semigroup Float where
  binop a b = a * b
instance Monoid Float where
  id_elt = 1.0


data ArrayIter t = AIter (Int, Array Int t)

instance Iterator (ArrayIter t) t where
  curr (AIter (i,a)) = a!i
  next (AIter (i,a)) = AIter (i + 1, a)
  equal (AIter (i,a)) (AIter (j,b)) = (i == j)
```

The call to `accumulate` shown below does not need to mention the instances `Monoid Float` and `Iterator (ArrayIter Float) Float` which are needed satisfy the type requirements of `accumulate`. Instead, the compiler finds the appropriate instances, looking them up by pattern matching against the type patterns in the instance declarations.

```
a = listArray (0,4::Int) [1.0,2.0,3.0,4.0,5.0::Float]
start = (AIter (0,a))
end = (AIter (5,a))
p = accumulate start end
```

In more detail, first the compiler deduces the type arguments for `accumulate` from the type of the arguments `start` and `end` obtaining `i=ArrayIter Float` and `t=Float`. The compiler then tries to satisfy the constraints for `accumulate`, so it needs `Monoid Float` and `Iterator (ArrayIter Float) Float`. The instance declaration for `Monoid Float` satisfies

the first requirement and the instance declaration `Iterator (ArrayIter t) t` satisfies the second requirement: the pattern matching succeeds with the pattern variable `t` matching with `Float`.

3.7. Summary

This chapter surveyed the design space for programming language support for generic programming. The chapter began with the motivation for focusing on statically typed languages and explicitly typed languages. It then compared two forms of polymorphism, subtyping and parametric, and argued that type parameters are a better choice because they offer better accuracy. Type parameterization comes in two flavors, the type-independent parametric polymorphism, as found in ML, and the type-dependent generational parameterization as found in C++ templates. Parametric polymorphism was favored because it is compatible with separate compilation whereas generational parameterization is not.

The chapter then evaluated language mechanisms for representing concepts such as object-oriented interfaces, Haskell type classes, and ML signatures. It concluded that a mixture of features from type classes and signatures would provide the best design. In particular, the type parameters of Haskell's type classes are needed to support implicit model passing and the abstract types and type sharing of ML signatures are needed to support associated types. Moving on to models, we compared the nominal and structural approaches to conformance, and decided that nominal conformance is the better choice because it is safer in the presence of concept-based overloading. The chapter then addressed the question of at what granularity type parameterization should occur, arguing that it should occur at least at the function level. Finally, the chapter discussed the need for implicit model passing and showed an example of how this works in Haskell.

I wish someone would construct a language more suitable to generic programming than C++. After all, one gets by in C++ by the skin of one's teeth. Fundamental concepts of STL, things like iterators and containers, are not describable in C++ since STL depends on rigorous sets of requirements that do not have any linguistic representation in C++. (They are, of course, defined in the standard, but they are defined in English.)

Alexander Stepanov [136]

4

The design of $\mathcal{G}$

The goal of this thesis is to design language features to support generic programming, and Chapter 7 describes a core calculus, named $F^{\mathcal{G}}$, that captures the essence of this design. However, the design must be field tested; it must be used to implement generic libraries. Any nontrivial library requires many language features unrelated to generics so a complete programming language is needed. Therefore the design for generics in this thesis is embedded in a language, named $\mathcal{G}$, that is modeled after C++ but with redesigned generics. $\mathcal{G}$ is an imperative language with declarations, statements, and expressions. $\mathcal{G}$ shares the same built-in types as C++, and has classes, structs, and unions, though in simplified forms. Objects may be allocated on the stack, with lifetimes that match a procedure's activation and objects may be allocated on the heap with programmer controlled lifetimes.

While $\mathcal{G}$ is modeled after C++ it is not strictly an extension to C++. Several details of C++ are incompatible with the design for generics developed in this thesis. Further, C++ is a large and complex language so implementing a compiler for C++ or even modifying an existing C++ compiler would be a large and difficult task. In contrast, $\mathcal{G}$ is a simpler language that is straightforward to parse and compile.

Modeling $\mathcal{G}$ on C++ allows for a straightforward translation of generic libraries from C++ to $\mathcal{G}$, thereby facilitating the field tests of $\mathcal{G}$. Furthermore, the compiler for $\mathcal{G}$ translates $\mathcal{G}$ to C++. The bulk of the compiler implementation is concerned with translating the generic features of $\mathcal{G}$, since those differ from C++, but the rest of the features in $\mathcal{G}$ are straightforward to translate to C++.

The language support for generics in $\mathcal{G}$ is based on parametric polymorphism, System F in particular. As discussed in Chapter 3, this has advantages for modularity: it allows for separate type checking and separate compilation. $\mathcal{G}$ augments parametric polymorphism with a language for describing interfaces of generic components, a language inspired by the semi-formal specification language used to document C++ libraries. The support for generic programming in $\mathcal{G}$ is provided by the following language features:

(1) Polymorphic functions enable the expression of generic algorithms. They include a `where` clause that states type requirements. To use a polymorphic function with certain types, the `where` clause of the polymorphic function must be satisfied in the lexical scope of the instantiation. Polymorphic functions may be called just like normal functions; the polymorphic function is implicitly instantiated with type arguments deduced from the types of the actual arguments.

(2) The `concept` feature directly supports the notion of "concept" in generic programming. This feature is used to define and organize requirements on types.

(3) A `model` definition verifies that a particular type τ satisfies the requirements of a concept c and adds the pair (c, τ) to the modeling *relation* associated with the current scope. This modeling relation is consulted when a generic function (or class) is instantiated and its `where` clause must be satisfied.

FIGURE 1. Syntax for generic functions

fundef	::=	`fun` *id polyhdr* `(`*type mode* [*id*]`,` ...`)` `->` *type mode* `{`*stmt* ...`}`	Function definition
funsig	::=	`fun` *id polyhdr* `(`*type mode* [*id*]`,` ...`)` `->` *type mode*`;`	Function signature
decl	::=	*fundef* | *funsig*	
mode	::=	*mut* [`&`]	pass by reference
		`@`	pass by value
mut	::=	[`const`]	constant
		`!`	mutable
polyhdr	::=	[`<`*tyvar*`,` ...`>`][`where` `{`*constraint*`,` ...`}`]	polymorphic header
constraint	::=	*cid*`<`*type*`,` ...`>`	model constraint
		type `==` *type*	same-type constraint
		funsig	function constraint
id			identifier
tyvar			type variable
cid			concept name

(4) Polymorphic classes, structs, and unions allow for the definition of generic data structures. As with polymorphic functions, constraints on type parameters are expressed with a `where` clause.

(5) Function expressions (anonymous functions) enable the convenient customization of generic algorithms with user-specified actions.

The following sections give a detailed description of these language features and show how this design meets the goals described in Chapter 1 and the criteria set forth in Section 2.2.4.

4.1. Generic functions

The syntax for generic function definitions and signatures is shown in Figure 1. The function name is given by the identifier following `fun`. Generic functions are parameterized on a list of types enclosed in angle brackets. The type parameters are constrained by requirements in the `where` clause. The body of a generic function is type checked under the conservative assumption that the type parameters could be any type that satisfies the constraints. Non-generic functions are taken as a special case of generic functions where the type parameter list is empty.

The default parameter passing mode in $\mathcal{G}$ is read-only pass-by-reference, which can also be specified with `&`. Read-write pass-by-reference is indicated by `!` and pass-by-value is indicated by `@`. Pass-by-value is not the default calling convention in $\mathcal{G}$, as it is in C++, because it adds requirements on the parameter type: the type must be copy constructible. Unlike C++, $\mathcal{G}$ does not have reference types because they allow the calling convention to change based on whether a generic function is instantiated with a reference type or non-reference type, such as instantiating a parameter `T` with `int&` versus `int`. Such a dependency on instantiation would complicate separate compilation and allow the semantics of a generic function to change.

4.1.0.1. *Constraints.* Three kinds of constraints may appear in a `where` clause: model constraints, same-type constraints, and function signatures. Constraints are treated as assumptions when type checking the body of a generic function. Also, constraints must be satisfied when a generic function is instantiated.

Model constraints: such as c<τ> indicate that type τ must be a model of concept c. At the point of instantiation, there must be a best-match model definition in the lexical scope for c<$[\bar{t}/\bar{\rho}]\tau$>, where $\bar{t}$ are the type parameters of the generic function and $\bar{\rho}$ are the type arguments. Section 4.6.2 discusses model lookup in more detail. Inside the generic function, the constraint c<τ> is treated as a surrogate model definition. All of the refinements and requirements of concept c are added as surrogate model definitions. Finally, all the function signatures from these concepts are introduced into the scope of the function.

Same-type constraints: such as τ_1 == τ_2 say that two type expressions must denote the same type. False constraints such as `int == float` are not allowed. Inside a generic function, the constraint τ_1 == τ_2 is treated as an assumption that plays a role in deciding when two type expressions are equal.

Function constraints: such as `fun foo(T) -> T@` say that a function definition must be in the scope of the instantiation of the generic function that has the given name

FIGURE 2. Generic accumulate function in $\mathcal{G}$.

```
fun accumulate<Iter>
where { InputIterator<Iter>,
        Monoid<InputIterator<Iter>.value> }
(Iter@ first, Iter last) -> InputIterator<Iter>.value@ {
  let t = identity_elt();
  for (; first != last; ++first)
    t = binary_op(t, *first);
  return t;
}
```

FIGURE 3. Syntax for accessing associated types.

type	::=	*scope*.*tyvar*	scope-qualified type
scope	::=	*scopeid*	
		scope.*scopeid*	scope member
scopeid	::=	*mid*	module identifier
		cid<*type*, ...>	model identifier

and with a type coercible to the specified type. Also, this constraint introduces the specified function signature into the scope of the generic function.

Figure 2 shows the generic `accumulate` function from Section 2.1 written in $\mathcal{G}$. The `accumulate` function is parameterized on the iterator type `Iter`. The `where` clause includes the requirements that the `Iter` type must model `InputIterator` and the value type of the iterator must model `Monoid`. The definitions of `InputIterator` and `Monoid` appear in Section 4.2 and 4.3.

The dot notation is used to refer to the associated value type of the iterator. Figure 3 shows the syntax for referring to associated types. The recursion in the *scope* production is necessary for handling nested requirements in concepts. For example, consider the following excerpt from the `Container` concept.

```
concept Container<C> {
  type iterator;
  type const_iterator;
  require InputIterator<iterator>;
  require InputIterator<const_iterator>;
```

FIGURE 4. Syntax for concepts.

decl	::=	`concept` *cid*`<`*tyvar*, ...`> {` *cmem* ... `};`	concept definition
cmem	::=	*funsig*	Function requirement
		fundef	" with default implementation
		`type` *tyvar*`;`	Associated type
		type `==` *type*`;`	Same-type requirement
		`refines` *cid*`<`*type*, ...`>;`	Refinement
		`require` *cid*`<`*type*, ...`>;`	Nested requirement

```
  ...
};
```

The following type expressions show how to refer to the `value` type of the container's `iterator` and `const_iterator`.

```
type iter = Container<X>.iterator;
type const_iter = Container<X>.const_iterator;
Container<X>.InputIterator<iter>.value
Container<X>.InputIterator<const_iter>.value
```

4.2. Concepts

The syntax for concepts is presented in Figure 4. A concept definition consists of a name for the concept and a type parameter, enclosed in angle brackets, that serves as a placeholder for the modeling type (or a list of type parameters for a list of modeling types). The type parameters are in scope for the body of the concept. Concepts contain the following kinds of members.

Function signatures: A function signature in a concept expresses the requirement that a function definition with a matching name and type must be provided by a model of the concept.

Function definition: A model of the concept may provide a function with the matching name and type, but if not, the default implementation provided by the function definition in the concept is used.

Associated types: An associated type in a concept requires that a model provide a type definition for the specified type name.

FIGURE 5. Syntax for models.

decl	::=	`model` *polyhdr* `<`*type*, . . . `>` `{` *decl* . . . `};`	model definition

Same-type constraints: A same-type constraint states the requirement that two type expressions must denote the same type in the context of a model definition.

Refinements: Requirements for the refined concept are included as requirements for this concept. A model definition for the concept being refined must precede a model definition for this concept.

Requirements: Nested requirements are similar to refinement in that they compose concepts. However, in this case the associated types of the required concept are not directly included but can be accessed indirectly. For example, the Container concept has a requirement that the associated `iterator` type model Iterator. The difference type of the `iterator` is accessed as follows:

```
type iter = Container<X>.iterator;
Container<X>.ForwardIterator<iter>.difference
```

The following example is the definition of the `InputIterator` concept in $\mathcal{G}$:

```
concept InputIterator<X> {
  type value;
  type difference;
  refines EqualityComparable<X>;
  refines Regular<X>; // this includes Assignable and CopyConstructible
  require SignedIntegral<difference>;
  fun operator*(X b) -> value@;
  fun operator++(X! c) -> X!;
};
```

4.3. Models

The modeling relation between a type and a concept is established with a model definition using the syntax shown in Figure 5. A model definition must satisfy all requirements of the concept. Requirements for associated types are satisfied by type definitions. Requirements for operations may be satisfied by function definitions in the model, by the `where`

clause, or by functions in the lexical scope preceding the model definition. The functions do not have to be an exact match, but they must be coercible to the required function signature. Refinements and nested requirements are satisfied by preceding model definitions.

The following simple example shows concept definitions for `Semigroup` and `Monoid` as well as model definitions for `int`.

```
concept Semigroup<T> {
  refines Regular<T>;
  fun binary_op(T,T) -> T@;
};
concept Monoid<T> {
  refines Semigroup<T>;
  fun identity_elt() -> T@;
};

use "basic_models.g"; // for Regular<int>
model Semigroup<int> {
  fun binary_op(int x, int y) -> int@ { return x + y; }
};
model Monoid<int> {
  fun identity_elt() -> int@ { return 0; }
};
```

Model definitions, like all other kinds of definitions in $\mathcal{G}$, may be enclosed in a module thereby controlling the scope in which the model is visible. Model definitions may be imported from another module with an `import` declaration or statement. Modules are described in Section 4.4.

4.3.0.2. *Parameterized models.* A model may be parameterized: the identifiers in the angle brackets are type parameters and the `where` clause introduces constraints. The following statement establishes that all pointer types are models of `InputIterator`:

```
model <T> InputIterator<T*> {
  type value = T;
  type difference = ptrdiff_t;
};
```

FIGURE 6. Syntax for modules.

decl	::=	`module` *mid* `{` *decl* ... `}`	module
		`scope` *mid* = *scope*;	scope alias
		`import` *scope*.*c*<$\overline{\tau}$>;	import model
		`public:` *decl* ...	public region
		`private:` *decl* ...	private region

Like generic functions, generic model definitions are type checked independently of any instantiation, so no type dependent operations are allowed on objects of type `T`, except as specified in the `where` clause.

The following is another example of a parameterized model, this time with a `where` clause. This model definition says that the `reverse_iterator` adaptor is a model of `InputIterator` if the underlying `Iter` type is a model of `BidirectionalIterator`. We discuss `reverse_iterator` is more detail in Section 6.1.5.

```
model <Iter> where { BidirectionalIterator<Iter> }
InputIterator< reverse_iterator<Iter> > {
  type value = BidirectionalIterator<Iter>.value;
  type difference = BidirectionalIterator<Iter>.difference;
};
```

4.4. Modules

The syntax for modules is shown in Figure 6. The important features of modules in $\mathcal{G}$ are import declarations for models and access control (`public` and `private`). An interesting extension would be parameterized modules, but we leave that for future work.

4.5. Type equality

There are several language constructions in $\mathcal{G}$ that make it difficult to decide when two types are equal. Generic functions complicate type equality because the names of the type parameters do not matter. So, for example, the following two function types are equal:

```
fun<T>(T)->T = fun<U>(U)->U
```

The order of the type parameters does matter (because a generic function may be explicitly instantiated) so the following two types are not equal.

```
fun<S,T>(S,T)->T ≠ fun<T,S>(S,T)->T
```

Inside the scope of a generic function, type parameters with different names are assumed to be different types (this is a conservative assumption). So, for example, the following program is ill formed because variable `a` has type `S` whereas function `f` is expecting an argument of type `T`.

```
fun foo<S, T>(S a, fun(T)->T f) -> T { return f(a); }
```

Associated types and same-type constraints also affect type equality. First, if there is a model definition in the current scope such as:

```
model C<int> { type bar = bool; };
```

then we have the equality `C<int>.bar` = `bool`.

Inside the scope of a generic function, same-type constraints help determine when two types are equal. For example, the following version of `foo` is well formed:

```
fun foo_1<T, S> where { T == S } (fun(T)->T f, S a) -> T { return f(a); }
```

There is a subtle difference between the above version of `foo` and the following one. The reason for the difference is that same-type constraints are checked after type argument deduction.

```
fun foo_2<T>(fun(T)->T f, T a) -> T { return f(a); }

fun id(double x) -> double { return x; }

fun main() -> int@ {
  foo_1(id, 1.0); // ok
  foo_1(id, 1); // error: Same type requirement violated, double != int
  foo_2(id, 1.0); // ok
  foo_2(id, 1); // ok
}
```

In the first call to `foo_1` the compiler deduces `T=double` and `S=double` from the arguments `id` and `1.0`. The compiler then checks the same-type constraint `T == S`, which in this case is

satisfied. For the second call to `foo_1`, the compiler deduces `T=double` and `S=int` and then the same-type constraint `T == S` is not satisfied. The first call to `foo_2` is straightforward. For the second call to `foo_2`, the compiler deduces `T=double` from the type of `id` and the argument `1` is implicitly coerced to `double`.

Type equality is a ***congruence relation***, which means several things. First it means type equality is an ***equivalence relation***, so it is reflexive, transitive, and symmetric. Thus, for any types ρ, σ, and τ we have

- $\tau = \tau$
- $\sigma = \tau$ implies $\tau = \sigma$
- $\rho = \sigma$ and $\sigma = \tau$ implies $\rho = \tau$

For example, the following function is well formed:

```
fun foo<R,S,T> where { R == S, S == T}
(fun(T)->S f, R a) -> T { return f(a); }
```

The type expression `R` (the type of `a`) and the type expression `T` (the parameter type of `f`) both denote the same type.

The second aspect of type equality being a congruence is that it propagates in certain ways with respect to type constructors. For example, if we know that `S` = `T` then we also know that `fun(S)->S` = `fun(T)->T`. Similarly, if we have defined a generic struct such as:

```
struct bar<U> { };
```

then `S` = `T` implies `bar<S>` = `bar<T>`. The propagation of equality also goes in the other direction. For example, `bar<S>` = `bar<T>` implies that `S` = `T`. The congruence extends to associated types. So `S` = `T` implies `C<S>.bar` = `C<T>.bar`. However, for associated types, the propagation does not go in the reverse direction. So `C<S>.bar` = `C<T>.bar` does not imply that `S` = `T`. For example, given the model definitions

```
model C<int> { type bar = bool; };
model C<float> { type bar = bool; };
```

we have `C<int>.bar` = `C<float>.bar` but this does not imply that `int` = `float`.

Like type parameters, associated types are in general assumed to be different from one another. So the following program is ill-formed:

```
concept C<U> { type bar; };
fun foo<S, T> where { C<S>, C<T> } (C<S>.bar a, fun(C<T>.bar)->T f) -> T
{ return f(a); }
```

The next program is also ill formed.

```
concept D<U> { type bar; type zow; };
fun foo<T> where { D<T> } (D<T>.bar a, fun(D<T>.zow)->T f) -> T
{ return f(a); }
```

In the compiler for $\mathcal{G}$ we use the congruence closure algorithm by Nelson and Oppen [142] to keep track of which types are equal. The algorithm is efficient: $O(n \log n)$ time complexity on average, where n is the number of types. It has $O(n^2)$ time complexity in the worst case. This can be improved by instead using the Downey-Sethi-Tarjan algorithm which is $O(n \log n)$ in the worst case [54].

4.6. Function application and implicit instantiation

The syntax for calling functions (or polymorphic functions) is the C-style notation:

expr	::=	*expr*`(`*expr*`,...)`	function application

Type arguments for the type parameters of a polymorphic function need not be supplied at the call site: $\mathcal{G}$ deduces the type arguments by unifying the types of the arguments with the types of the parameters. The type arguments are substituted into the `where` clause and then each of the constraints must be satisfied in the current lexical scope. The following is a program that calls the `accumulate` function, applying it to iterators of type `int*`.

```
fun main() -> int@ {
  let a = new int[8];
  a[0] = 1; a[1] = 2; a[2] = 3; a[3] = 4; a[4] = 5;
  let s = accumulate(a, a + 5);
  if (s == 15) return 0;
  else return -1;
}
```

Type arguments of a polymorphic function may be specified explicitly with the following syntax.

expr	::=	*expr*`<|` *type* `, ... |>`	explicit instantiation

Following Mitchell [129] we view ***implicit instantiation*** as a kind of coercion that transforms an expression of one type to another type. In the example above, the `accumulate` function was coerced from

```
fun <Iter> where { InputIterator<Iter>, Monoid<InputIterator<Iter>.value> }
  (Iter@, Iter) -> InputIterator<Iter>.value@
```

to

```
fun (int*@, int*) -> InputIterator<int*>.value@
```

There are several kinds of implicit coercions in $\mathcal{G}$, and together they form a subtyping relation $\leq$. The subtyping relation is reflexive and transitive. Like C++, $\mathcal{G}$ contains some bidirectional implicit coercions, such as `float` $\leq$ `double` and `double` $\leq$ `float`, so $\leq$ is not anti-symmetric. The subtyping relation for $\mathcal{G}$ is defined by a set of subtyping rules. The following is the subtyping rule for generic function instantiation.

$$\text{(INST)}\ \frac{\Gamma \text{ satisfies } \overline{c}}{\Gamma \vdash \texttt{fun<}\overline{\alpha}\texttt{>where\{}\overline{c}\texttt{\}(}\overline{\sigma}\texttt{)->}\tau \leq [\overline{\rho}/\overline{\alpha}](\texttt{fun(}\overline{\sigma}\texttt{)->}\tau)}$$

The type parameters $\overline{\alpha}$ are substituted for type arguments $\overline{\rho}$ and the constraints in the `where` clause must be satisfied in the current environment. To apply this rule, the compiler must choose the type arguments. We call this ***type argument deduction*** and discuss it in more detail momentarily. Constraint satisfaction is discussed in Section 4.6.2.

The subtyping relation allows for coercions during type checking according to the subsumption rule:

$$\text{(SUB)}\ \frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \sigma \leq \tau}{\Gamma \vdash e : \tau}$$

The (SUB) rule is not syntax-directed so its addition to the type system would result in a non-deterministic type checking algorithm. The standard workaround is to omit the above rule and instead allow coercions in other rules of the type system such as the rule for

function application. The following is a rule for function application that allows coercions in both the function type and in the argument types.

$$\text{(APP)}\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash \overline{e_2} : \overline{\sigma_2} \quad \Gamma \vdash \tau_1 \leq \texttt{fun}(\overline{\sigma_3})\texttt{->}\tau_2 \quad \Gamma \vdash \overline{\sigma_2} \leq \overline{\sigma_3}}{\Gamma \vdash e_1(\overline{e_2}) : \tau_2}$$

4.6.1. Type argument deduction. As mentioned above, the type checker must guess the type arguments $\overline{\rho}$ to apply the (INST) rule. In addition, the (APP) rule includes several types that appear from nowhere: $\overline{\sigma_3}$ and $\overline{\tau_2}$. The problem of deducing these types is equivalent to trying to find solutions to a system of inequalities. Consider the following example program.

```
fun apply<T>(fun(T)->T f, T x) -> T { return f(x); }
fun id<U>(U a) -> U { return a; }
fun main() -> int@ { return apply(id, 0); }
```

The application `apply(id, 0)` type checks if there is a solution to the following system:

$$\begin{aligned}
&\texttt{fun<T>(fun(T)->T, T) -> T} \leq \texttt{fun}(\alpha, \beta)\texttt{ -> } \gamma \\
&\texttt{fun<U>(U)->U} \leq \alpha \\
&\texttt{int} \leq \beta
\end{aligned}$$

The following type assignment is a solution to the above system.

$$\begin{aligned}
\alpha &= \texttt{fun(int)->int} \\
\beta &= \texttt{int} \\
\gamma &= \texttt{int}
\end{aligned}$$

Unfortunately, not all systems of inequalities are as easy to solve. In fact, with Mitchell's original set of subtyping rules, the problem of solving systems of inequalities was proved undecidable by Tiuryn and Urzyczyn [187]. There are several approaches to dealing with this undecidability.

4.6.1.1. *Remove the (*ARROW*) rule.* Mitchell's subtyping relation included the usual co/-contravariant rule for functions.

$$\text{(ARROW)}\frac{\overline{\sigma_2} \leq \overline{\sigma_1} \quad \tau_1 \leq \tau_2}{\texttt{fun}(\overline{\sigma_1})\texttt{->}\tau_1 \leq \texttt{fun}(\overline{\sigma_2})\texttt{->}\tau_2}$$

The (ARROW) rule is nice to have because it allows a function to be coerced to a different type so long as the parameter and return types are coercible in the appropriate way. In

the following example the standard `ilogb` function is passed to `foo` even though it does not match the expected type. The (ARROW) rule allows for this coercion because `int` is coercible to `double`.

```
include "math.h"; // fun ilogb(double x) -> int;
fun foo(fun(int)->int@ f) -> int@ { return f(1); }
fun main() -> int@ { return foo(ilogb); }
```

However, the (ARROW) rule is one of the culprits in the undecidability of the subtyping problem; removing it makes the problem decidable [187]. The language ML^{F} of Le Botlan and Remy [24] takes this approach, and for the time being, so does $\mathcal{G}$. With this restriction, type argument deduction is reduced to the variation on unification used in ML^{F}. Instead of working on a set of variable assignments, this unification algorithm keeps track of either a type assignment or the tightest lower bound seen so far for each variable. The (APP) rule is reformulated as follows to use this `unify` algorithm.

$$\text{(APP)}\ \frac{\begin{array}{cc}\Gamma \vdash e_1 : \tau_1 & \Gamma \vdash \overline{e_2} : \overline{\sigma_2} \\ Q = \{\tau_1 \leq \alpha, \overline{\sigma_2} \leq \overline{\beta}\} & Q' = \texttt{unify}(\alpha, \texttt{fun}(\overline{\beta})\texttt{->}\gamma, Q)\end{array}}{\Gamma \vdash e_1(\overline{e_2}) : Q'(\gamma)}$$

In languages where functions are often written in curried form, it is important to provide even more flexibility than in the above (APP) rule by postponing instantiation, as is done in ML^{F}. Consider the `apply` example again, but this time written in curried form.

```
fun apply<T>(fun(T)->T f) -> (fun(T)->T)@ {
  return fun(T x) { return f(x); };
}
fun id<U>(U a) -> U { return a; }
fun main() -> int@ { return apply(id)(0); }
```

In the first application `apply(id)` we do not yet know that `T` should be bound to `int`. The instantiation needs to be delayed until the second application `apply(id)(0)`. In general, each application contributes to the system of inequalities that needs to be solved to instantiate the generic function. In ML^{F}, the return type of each application encodes a partial system of inequalities. The inequalities are recorded in the types as lower bounds on type parameters. The following is an example of such a type.

```
fun<U> where { fun<T>(T)->T ≤ U } (U) -> U
```

Postponing instantiation is not as important in $\mathcal{G}$ because functions take multiple parameters and currying is seldom used.

Removal of the arrow rule means that, in some circumstances, the programmer would have to wrap a function inside another function before passing the function as an argument.

4.6.1.2. *Restrict the language to predicative polymorphism.* Another alternative is to restrict the language so that only monotypes (non-generic types) may be used as the type arguments in an instantiation. This approach is used in by Odersky and Läufer [148] and also by Peyton Jones and Shields [100]. However, this approach reduces the expressiveness of the language for the sake of the convenience of implicit instantiation.

4.6.1.3. *Restrict the language to second-class polymorphism.* Restricting the language of types to disallow polymorphic types nested inside other types is another way to make the subtyping problem decidable. With this restriction the subtyping problem is solved by normal unification. Languages such as SML and Haskell 98 use this approach. Like the restriction to predicative polymorphism, this approach reduces the expressiveness of the language for the sake of implicit instantiation (and type inference). However, there are many motivating use cases for first-class polymorphism [42], so throwing out first-class polymorphism is not our preferred alternative.

4.6.1.4. *Use a semi-decision procedure.* Yet another alternative is to use a semi-decision procedure for the subtyping problem. The advantage of this approach is that it allows implicit instantiation to work in more situations, though it is not clear whether this extra flexibility is needed in practice. The down side is that there are instances of the subtyping problem where the procedure diverges and never returns with a solution.

4.6.2. Model lookup (constraint satisfaction). The basic idea behind model lookup is simple though some of the details are a bit complicated. Consider the following program containing a generic function `foo` with a requirement for `C<T>`.

```
concept C<T> { };
model C<int> { };
```

```
fun foo<T> where { C<T> } (T x) -> T { return x; }

fun main() -> int@ {
  return foo(0);// lookup model C<int>
}
```

At the call `foo(0)`, the compiler deduces the binding `T=int` and then seeks to satisfy the `where` clause, with `int` substituted for `T`. In this case the constraint `C<int>` must be satisfied. In the scope of the call `foo(0)` there is a model declaration for `C<int>`, so the constraint is satisfied. We call `C<int>` the ***model head***.

In $\mathcal{G}$, a model definition may itself be parameterized and the type parameters constrained by a `where` clause. Figure 7 shows a typical example of a parameterized model. The model definition in the example says that for any type `T`, `list<T>` is a model of `Comparable` if `T` is a model of `Comparable`. Thus, a model definition is an inference rule, much like a Horn clause [84] in logic programming. For example, a model definition of the form

```
model <T1,...,Tn> where { P1, ..., Pn }
Q { ... };
```

corresponds to the Horn clause:

$$(P_1 \text{ and } \ldots \text{ and } P_n) \text{ implies } Q$$

The model definitions from the example in Figure 7 could be represented in Prolog with the following two rules:

```
comparable(int).
comparable(list(T)) :- comparable(T).
```

The algorithm for model lookup is essentially a logic programming engine: it performs unification and backward chaining (similar to how instance lookup is performed in Haskell). ***Unification*** is used to determine when the head of a model definition matches. For example, in Figure 7, in the call to `generic_foo` the constraint `Comparable< list<int> >` needs to be satisfied. There is a model definition for `Comparable< list<T> >` and unification of `list<int>` and `list<T>` succeeds with the type assignment `T` $=$ `int`. However, we have not yet satisfied `Comparable< list<int> >` because the `where` clause of the parameterized

FIGURE 7. Example of parameterized model definition.

```
concept Comparable<T> {
  fun operator==(T,T)->bool@;
};
model Comparable<int> { };

struct list<T> { /*...*/ };

model <T> where { Comparable<T> }
Comparable< list<T> > {
  fun operator==(list<T> x, list<T> y) -> bool@ { /*...*/ }
};

fun generic_foo<C> where { Comparable<C> } (C a, C b) -> bool@
  { return a == b; }

fun main() -> int@ {
  let l1 = @list<int>(); let l2 = @list<int>();
  generic_foo(l1,l2);
  return 0;
}
```

model must also be satisfied. The model lookup algorithm therefore proceeds recursively and tries to satisfy `Comparable<int>`, which in this case is trivial. This process is called ***backward chaining***: it starts with a goal (a constraint to be satisfied) and then applies matching rules (model definitions) to reduce the goal into subgoals. Eventually the subgoals are reduced to facts (model definitions without a `where` clause) and the process is complete. As is typical of Prolog implementations, $\mathcal{G}$ processes subgoals in a depth-first manner.

It is possible for multiple model definitions to match a constraint. When this happens the most specific model definition is used, if one exists. Otherwise the program is ill-formed. We say that definition A is a ***more specific model*** than definition B if the head of A is a substitution instance of the head of B and if the `where` clause of B implies the `where` clause of A. In this context, implication means that for every constraint c in the `where` clause of A, c is satisfied in the current environment augmented with the assumptions from the `where` clause of B.

$\mathcal{G}$ places very few restrictions on the form of a model definition. The only restriction is that all type parameters of a model must appear in the head of the model. That is, they must appear in the type arguments to the concept being modeled. For example, the following model definition is ill formed because of this restriction.

```
concept C<T> { };
model <T,U> C<T> { }; // ill formed, U is not in an argument to C
```

This restriction ensures that unifying a constraint with the model head always produces assignments for all the type parameters.

Horn clause logic is by nature powerful enough to be Turning-complete. For example, it is possible to express general recursive functions. The program in Figure 8 computes the Ackermann function at compile time by encoding it in model definitions. This power comes at a price: determining whether a constraint is satisfied by a set of model definitions is in general undecidable. Thus, model lookup is not guaranteed to terminate and programmers must take some care in writing model definitions. We could restrict the form of model definitions to achieve decidability however there are two reasons not to do so. First, restrictions would complicate the specification of $\mathcal{G}$ and make it harder to learn. Second, there is the danger of ruling out useful model definitions.

4.7. Function overloading and concept-based overloading

Multiple functions with the same name may be defined and static overload resolution is performed to decide which function to invoke at a particular call site. The resolution depends on the argument types and on the model definitions in scope. When more than one overload may be called, the most specific overload is called if one exists. The basic overload resolution rules are based on those of C++.

In the following simple example, the second `foo` is called.

```
fun foo() -> int@ { return -1; }
fun foo(int x) -> int@ { return 0; }
fun foo(double x) -> int@ { return -1; }
fun foo<T>(T x) -> int@ { return -1; }

fun main() -> int@ { return foo(3); }
```

FIGURE 8. The Ackermann function encoded in model definitions.

```
struct zero { };
struct succ<n> { };
concept Ack<x,y> { type result; };

model <y> Ack<zero,y> { type result = succ<y>; };

model <x> where { Ack<x, succ<zero> > }
Ack<succ<x>, zero> { type result = Ack<x, succ<zero> >.result; };

model <x,y> where { Ack<succ<x>,y>, Ack<x, Ack<succ<x>,y>.result > }
Ack< succ<x>,succ<y> > {
  type result = Ack<x, Ack<succ<x>,y>.result >.result;
};

fun foo(int) { }
fun main() -> int@ {
  type two = succ< succ<zero> >; type three = succ<two>;
  foo(@Ack<two,three>.result());
  // error: Type (succ<succ<succ<succ<succ<succ<succ<succ<succ<zero>>>>>>>>>)
  // does not match type (int)
}
```

The first `foo` has the wrong number of arguments, so it is immediately dropped from consideration. The second and fourth are given priority over the third because they can exactly match the argument type `int` (for the fourth, type argument deduction results in `T=int`), whereas the third `foo` requires an implicit coercion from `int` to `double`. The second `foo` is favored over the fourth because it is more specific.

A function f is a ***more specific overload*** than function g if g is callable from f but not vice versa. A function g is ***callable from*** function f if you could call g from inside f, forwarding all the parameters of f as arguments to g, without causing a type error. More formally, if f has type `fun<`$\overline{t_f}$`>where`$C_f(\overline{\sigma_f})$`->`τ_f and g has type `fun<`$\overline{t_g}$`>where`$C_g(\overline{\sigma_g})$`->`τ_g then g is callable from f if

$$\overline{\sigma_f} \leq [\overline{t_g}/\overline{\rho}]\overline{\sigma_g} \text{ and } C_f \text{ implies } [\overline{t_g}/\overline{\rho}]C_g$$

for some $\overline{\rho}$.

In general there may not be a most specific overload in which case the program is ill-formed. In the following example, both `foo`'s are callable from each other and therefore neither is more specific.

```
fun foo(double x) -> int@ { return 1; }
fun foo(float x) -> int@ { return -1; }
fun main() -> int@ { return foo(3); }
```

In the next example, neither `foo` is callable from the other so neither is more specific.

```
fun foo<T>(T x, int y) -> int@ { return 1; }
fun foo<T>(int x, T y) -> int@ { return -1; }
fun main() -> int@ { return foo(3, 4); }
```

4.7.0.1. *Concept-based overloading.* In Section 2.2.1.7 we showed how to accomplish concept-based overloading of several versions of `advance` using the tag dispatching idiom in C++. Figure 9 shows three overloads of `advance` implemented in $\mathcal{G}$. The signatures for these overloads are the same except for their `where` clauses. The concept `BidirectionalIterator` is a refinement of `InputIterator`, so the second version of `advance` is more specific than the first. The concept `RandomAccessIterator` is a refinement of `BidirectionalIterator`, so the third `advance` is more specific than the second.

The code in Figure 10 shows two calls to `advance`. The first call is with an iterator for a singly-linked list. This iterator is a model of `InputIterator` but not `RandomAccessIterator`; the overload resolution chooses the first version of `advance`. The second call to `advance` is with a pointer which is a `RandomAccessIterator` so the second version of `advance` is called.

Concept-based overloading in $\mathcal{G}$ is entirely based on static information available during the type checking and compilation of the call site. This presents some difficulties when trying to resolve to optimized versions of an algorithm from within another generic function. Section 6.1.3 discusses the issues that arise and presents an idiom that ameliorates the problem.

4.8. Generic user-defined types

The syntax for polymorphic classes, structs, and unions is defined below.

FIGURE 9. The advance algorithms using concept-based overloading.

```
fun advance<Iter> where { InputIterator<Iter> }
(Iter! i, InputIterator<Iter>.difference@ n) {
  for (; n != zero(); --n)
    ++i;
}
fun advance<Iter> where { BidirectionalIterator<Iter> }
(Iter! i, InputIterator<Iter>.difference@ n) {
  if (zero() < n)
    for (; n != zero(); --n)
      ++i;
  else
    for (; n != zero(); ++n)
      --i;
}
fun advance<Iter> where { RandomAccessIterator<Iter> }
(Iter! i, InputIterator<Iter>.difference@ n) {
  i = i + n;
}
```

FIGURE 10. Example calls to advance and overload resolution.

```
use "slist.g";
use "basic_algorithms.g"; // for copy
use "iterator_functions.g"; // for advance
use "iterator_models.g"; // for iterator models for int*

fun main() -> int@ {
  let sl = @slist<int>();
  push_front(1, sl); push_front(2, sl); push_front(3, sl); push_front(4, sl);
  let in_iter = begin(sl);
  advance(in_iter, 2); // calls version 1, linear time

  let rand_iter = new int[4];
  copy(begin(sl), end(sl), rand_iter);
  advance(rand_iter, 2);  // calls version 3, constant time

  if (*in_iter == *rand_iter) return 0;
  else return -1;
}
```

decl	::=	`class` *clid polyhdr* `{`*clmem* ...`};`	class
		`struct` *clid polyhdr* `{`*type id*`;` ...`};`	struct
		`union` *clid polyhdr* `{`*type id*`;` ...`};`	union
clmem	::=	*type id*`;`	data member
		polyhdr clid`(`*type mode* [*id*]`,` ...`){`*stmt* ...`}`	constructor
		`~`*clid*`(){`*stmt* ...`}`	destructor
clid			class name

In $\mathcal{G}$, as in C++, classes enable the definition of abstract data types. Classes consist of data members, constructors, and a destructor. There are no member functions; normal functions are used instead. Data encapsulation (`public/private`) is specified at the module level instead of inside the class.

In $\mathcal{G}$, structs are distinct from classes, and merely provide a mechanism for composing data, i.e., structs are like Pascal records. Unions are provided for situations where the type of data may vary at run-time and data-directed programming is necessary.

The type of a class, struct, or union is referred to using the syntax below. Such a type is well-formed if the type arguments are well-formed and if the requirements in its `where` clause are satisfied in the current scope.

type ::= *clid*[`<`*type*`,` ...`>`]

4.9. Function expressions

The following is the syntax for function expressions and function types.

expr	::=	`fun` *polyhdr* `(`*type mode* [*id*]`,` ...`)` *id*`=`*expr*`,` ...`(`{*stmt* ...}|`:`*expr*`)`
type	::=	`fun` *polyhdr* `(`*type mode*`,` ...`)`[`->` *type mode*]

The body of a function expression may be either a sequence of statements enclosed in braces or a single expression following a colon. The return type of a function expression is deduced from the return statements in the body, or from the single expression.

The following example computes the sum of an array using `for_each` and a function expression. [1]

```
fun main() -> int@ {
  let n = 8;
  let a = new int[n];
  for (let i = 0; i != n; ++i)
    a[i] = i;
  let sum = 0;
  for_each(a, a + n, fun(int x) p=&sum { *p = *p + x; });
  return sum - (n * (n-1))/2;
}
```

The expression

```
fun(int x) p=&sum { *p = *p + x; }
```

creates a function object. The body of a function expression is not lexically scoped, so a direct use of `sum` in the body would be an error. The initialization `p=&sum` both declares a data member inside the function object with type `int*` and copy constructs the data member with the address `&sum`.

The primary motivation for non-lexically scoped function expressions is to keep the design close to C++ so that function expressions can be directly compiled to function objects in C++. However, this design has some drawbacks as we discovered during our implementation of the STL. Section 6.1.6 discusses the problem we encountered.

4.9.0.2. *First-class polymorphism.* At the beginning of this chapter we mentioned that $\mathcal{G}$ is based on System F. One of the hallmarks of System F is that it provides first class polymorphism. That is, polymorphic objects may be passed to and returned from functions. This is in contrast to the ML family of languages, where polymorphism is second class. In Section 4.6 we discussed how the restriction to second-class polymorphism simplifies type

[1] Of course, the `accumulate` function is the appropriate algorithm for this computation, but then the example would not demonstrate the use of function expressions.

argument deduction, reducing it to normal unification. However, we prefer to retain first-class polymorphism and use the somewhat more complicated variant of unification from ML^{F}.

One of the reasons to retain first-class polymorphism is to retain the expressiveness of function objects in C++. A function object may have member function templates and may therefore by used polymorphically. The following program is a simple use of first-class polymorphism in $\mathcal{G}$. Note that `f` is applied to arguments of different types.

```
fun foo(fun<T>(T)->T f) -> int@ { return f(1) + d2i(f(-1.0)); }
fun id<T>(T x) -> T { return x; }
fun main() -> int@ { return foo(id); }
```

4.10. Summary

This section reviews how the design of $\mathcal{G}$ fulfills the goals from Chapter 1 and the criteria set forth in Section 2.2.4. In Chapter 1 we discussed the importance of separate type checking and separate compilation for the production and use of generic libraries. The design for $\mathcal{G}$ provides both separate type checking and separate compilation by basing its generics on parametric polymorphism. The essential property for separate type checking is that generic functions are checked under the conservative assumption that the type parameters could be any type that satisfies the type requirement. Also, to enable separate compilation, the only type-dependent operations that are allowed are those specified by the `where` clause.

In Section 2.2.4 we listed nine specific language requirements for generic programming. Each of those requirements is satisfied by the design for $\mathcal{G}$.

(1) $\mathcal{G}$ provides generic functions with `where` clauses to express constraints on how the generic functions may be instantiated, and dually to express assumptions that may be used inside the generic functions. Type checking is performed independently of any instantiation.

(2) $\mathcal{G}$ includes `concept` definitions for grouping and organizing requirements. Concepts are composable via refinements and via nested requirements.

(3) Concepts contain requirements for function signatures, associated types, and same-type constraints. This chapter did not discuss conversion requirements, but that is because they are trivial to express in $\mathcal{G}$. A user-defined implicit conversion may be created by defining a function named `coerce`. Thus a conversion requirement is expressed with a function signature in a concept.

(4) The design for $\mathcal{G}$ provides implicit model passing via model definitions, `where` clauses, and a model lookup algorithm similar to a logic programming engine.

(5) Type argument deduction is provided in $\mathcal{G}$ by borrowing the approach of ML^{F} which is compatible with the presence of first class polymorphism.

(6) Concept-based dispatching is provided through the function overloading rules that take the `where` clause into consideration when determining the most specific overload.

(7) Conditional modeling is needed for generic adaptors such as `reverse_iterator` (Section 2.2.3). Conditional modeling is provided in $\mathcal{G}$ by parameterized model definitions with `where` clauses.

(8) $\mathcal{G}$ includes a simple `class` feature with constructors and a destructor that enables the creation of abstract data types.

It is also instructive to evaluate the design of $\mathcal{G}$ with respect to the criteria from our previous study comparing support for generic programming in several languages [69]. Table 1 shows the results of that study but with a new column for $\mathcal{G}$. The table also includes a new row for concept-based dispatching. The following describes the criteria and explains how it is fulfilled in the design of $\mathcal{G}$.

Multi-type concepts: are concepts with multiple type parameters. The syntax for concepts in $\mathcal{G}$, as shown in Figure 4, provides for multiple type parameters.

Multiple constraints: refers to the ability to place multiple constraints on a type parameter. This is supported in $\mathcal{G}$ in that a `where` clause may include any number of requirements each each requirement may constrain one or more of the type parameters. See Figure 1 for the syntax of `where` clauses.

	C++	SML	Haskell	Java	C#	$\mathcal{G}$
Multi-type concepts	-	●	●*	○	○	●
Multiple constraints	-	◒	●	●	●	●
Associated type access	●	●	◒	◒	◒	●
Constraints on assoc. types	-	●	●	◒	◒	●
Retroactive modeling	-	●	●	○	○	●
Type aliases	●	●	●	○	○	●
Separate compilation	○	●	●	●	●	●
Implicit instantiation	●	○	●	●	●	●
Concept dispatching	●	○	○	○	○	◒

*Using the multi-parameter type class extension to Haskell 98 [149].

TABLE 1. The level of support for generic programming in several languages. The rating of "-" in the C++ column indicates that while C++ does not explicitly support the feature, one can still program as if the feature were supported due to the flexibility of C++ templates.

Associated type access: refers to the ease in which types are mapped to other types within the context of a generic function. In $\mathcal{G}$ this is accomplished with the dot notation, as shown in Figure 3.

Retroactive modeling: indicates the ability to add new modeling relationships after a type has been defined. This is supported in $\mathcal{G}$ because `model` definitions (see Figure 5) are separate from class definitions.

Type aliases: indicates whether a mechanism for creating shorter names for types is provided. $\mathcal{G}$ provides type aliases, though we have not yet discussed them. The syntax for type aliases is shown in Appendix A and the compilation of type aliases is given in Section 5.2.4 and 5.2.5.

Separate compilation: indicates whether generic functions are type-checked and compiled independently from their use. $\mathcal{G}$ provides both separate type checking and separate compilation.

Implicit instantiation: indicates that type arguments are deduced without requiring explicit syntax for instantiation. How implicit instantiation is performed in $\mathcal{G}$ is explained in Section 4.6.

Concept-based dispatching: indicates whether the language provides facilities for dispatching between different versions of an algorithm based on which concepts are modeled by the input.

5

The definition and compilation of $\mathcal{G}$

There are many approaches to defining the meaning of phrases in a programming language. The denotational approach maps a phrase to an object in some pre-defined formal domain, such as mathematical sets or functions. The operational approach describes how a phrase causes an abstract machine to change states, or describes what value will result from evaluating the phrase. The translational approach maps phrases to phrases in another (hopefully well-defined) language. The axiomatic approach to defining programming languages assigns a predicate to each point between statements in a program and describes how each kind of phrase transforms these predicates. Each of the approaches is good for particular

purposes. For example, an axiomatic semantics is good for proving the correctness of programs, whereas an operational semantics is good for giving programmers a mental model of program execution.

In this chapter we use the translational approach: we describe a translation from $\mathcal{G}$ to C++. There are several reasons for this choice. The first is a matter of economy of expression: $\mathcal{G}$ is a full-featured language so a denotational or operation semantics for $\mathcal{G}$ would be rather large. On the other hand, $\mathcal{G}$ is quite similar to C++, so defining $\mathcal{G}$ in terms of C++ reuses much of the effort that went into defining C++. Another reason to use the translational approach is that the semantics of Haskell type classes is defined by translation, either translating to an ML-like language [196] or to System F [78], and it is easier to compare $\mathcal{G}$ with Haskell if the semantics are in the same style. The primary reason for choosing the translational approach is that it also provides an implementation of a prototype compiler for the language. This compiler was useful in testing the design of $\mathcal{G}$ with the implementation of the STL and BGL, which is described in Chapter 6.

There are several disadvantages to defining $\mathcal{G}$ by translation to C++. First, the C++ standard is a rather informal description of the language. Second, the translation overspecifies the language $\mathcal{G}$, after all, an implementation of $\mathcal{G}$ does not have to translate to C++, it could instead be written as an interpreter, or could translate to some other language such as C or even directly to assembly. Of course, what is intended is that an implementation of $\mathcal{G}$ should be observationally equivalent to the translation described in this chapter, for some suitably loose definition of observational equivalence.

The first section gives an overview of the translation, describing the C++ output from translating each of the major language features of $\mathcal{G}$: generic functions, concept, models, and generic classes. The second section describes the translation to C++ in more detail.

The full grammar for $\mathcal{G}$ is defined in Appendix A.

5.1. Overview of the translation to C++

This section gives an informal description of the translation from $\mathcal{G}$ to C++. The focus is on *what* is output from the translation. The *how* is described in Section 5.2. The basic idea of the translation is the same as for Haskell type classes [78, 196]. The implicit passing of models to generic functions is translated into explicit dictionary passing, where a "dictionary" is a data structure holding the functions that implement the requirements of a concept for a particular type. Thus a dictionary is a run-time representation of a model. Mark Jones introduces a nice way to think about dictionaries in his Ph.D. thesis [96]. A concept can be thought of as a predicate on types, so `Comparable<int>` is a proposition which states that `Comparable` is true for the type `int`. In constructive logic, a proposition is accompanied by ***evidence*** that demonstrates that the proposition is true. Analogously, we can think of a dictionary as the evidence that a type models a concept.

While the basic idea is the same, the translation described here differs from that of Haskell in the following respects.

- Concepts and models in $\mathcal{G}$ differ in several respects from type classes, especially with regard to scoping rules and the presence of associated types in $\mathcal{G}$.
- The target language is C++ instead of ML [196] or System F [78]. This impacts the translation because C++ has neither parametric polymorphism nor closures, both of which are used extensively in the translations for Haskell. C++ has templates, but we do not use them in the translation of generic functions because that would not provide separate compilation.
- The translation does not perform type inference.

Instead of using parametric polymorphism and closures in the target language, we use a combination of dynamic types and object-oriented features such as abstract base classes (interfaces) and derived classes. In some sense, this translation can be seen as establishing a relationship between generic programming and object-oriented programming. The translation also shows that it is possible to do generic programming in an object-oriented

language. However, the compilation is non-trivial so without it the programmer would have to do considerable work and would be giving up the static type safety of $\mathcal{G}$.

The translator mangles identifiers to prevent name clashes and to assign different names to function overloads. However, for the sake of readability, identifiers are not mangled in the excerpts shown in this section.

5.1.1. Generic functions. To achieve separate compilation, a generic function must be compiled to a single function that can work on many different types of input. This presents a small challenge for compiling to C++ because C++ is a statically typed language. In particular, we need to pass objects of different types as arguments to the same parameter. For example, we need to pass objects of type `int` and `double` to parameter `x` of the following `id` function.

```
fun id<T>(T x) -> T { return x; }

fun main() -> int@ {
  let xi = 1; let yd = 1.0;
  let x = id(xi); let y = id(yd);
  return 0;
}
```

We use dynamic types to allow arguments of different types to be passed to the same parameter. In particular, we use a family of classes based on the Boost `any` class. (This class is similar to the `any` type of CLU [117].) The `any` class is used for pass-by-value, `any_ref` for mutable pass-by-reference, and `any_const_ref` for constant pass-by-reference. Figure 1 shows the implementation of the `any` class; the implementation of the other members of the `any` family is similar. The following is the C++ translation of the above program.

```
any_const_ref id(any_const_ref x) { return x; }

int main() {
  int xi = 1; double yd = 1.0;
  int const& x = any_cast<int const&>(id(xi));
  double const& y = any_cast<double const&>(id(yd));
  return 0;
}
```

FIGURE 1. The C++ any Class

```
struct any_placeholder {
  virtual ~placeholder() { }
  virtual const std::type_info& type() const = 0;
  virtual placeholder* clone() const = 0;
};

template<typename T>
struct any_holder : public any_placeholder {
  any_holder(const T& value) : held(value) { }
  virtual const std::type_info& type() const { return typeid(T); }
  virtual any_placeholder* clone() const { return new any_holder(held); }
  T held;
};

struct any {
  template<typename T>
  any(const T& value) : content(new any_holder<T>(value)) { }
  any(const any& x) : content(x.content ? x.content->clone() : 0) { }
  ~any() { delete content; }
  placeholder* content;
};

template<typename ValueType>
ValueType any_cast(to_type<ValueType>, const any& operand) {
  if (operand->type() == typeid(ValueType))
    return static_cast<any_holder<ValueType> *>(operand.content)->held;
  else
    throw bad_any_cast();
}
```

The `id` function is translated to a normal (non-template) function with type `T` replaced by `any_const_ref` (because pass by const reference is the default passing mode). The coercion from `int` to `any_const_ref` is handled implicitly by a constructor in the `any_const_ref` class and the coercion in the other direction is accomplished by a cast that throws an exception if the actual type does not match the target type.

Alternatively, we could use `void*` instead of `any` and a C-style cast instead of `any_cast`. However, that approach would complicate the translation, requiring code to be produced for managing the lifetime of temporary objects.

5.1.1.1. *Function expressions.* Anonymous functions expression in $\mathcal{G}$ are compiled to function objects in C++. Consider the following program that creates a function and applies it to -1.

```
fun main() -> int@ {
  let x = 1;
  return fun(int y) mem=x { return mem + y; } (-1);
}
```

The C++ translation is

```
struct __functor_384 {
  __functor_384(int mem) : mem(mem) { }
  int operator()(int const& y) { return mem + y; }
  int mem;
};
int main() {
  int x = 1;
  return (__functor_384(x))(-1);
}
```

A struct is defined with a function call operator containing the body of the function expression. The function expression itself is replaced by a call to the constructor of the struct. The data member initialization `mem=x` in the $\mathcal{G}$ program translates to the data member `mem` in the struct and its initialization in the constructor.

5.1.1.2. *Function parameters, function types.* A function may take another function as a parameter, such as parameter `f` in the following `apply` function.

```
fun apply<S,T>(S x, fun(S)->T f) -> T { return f(x); }
```

Function pointers are a natural choice for translating $\mathcal{G}$ function types. However, function objects like `__functor_384` can not be passed as function pointers. We need a C++ type that can be used for either function objects or built-in function pointers. The Boost Function Library [22] provides a solution with its `function` class template. The following example shows the use of `function` to declare a variable `f` that can hold a function pointer, such as `add`, and later can hold a function object, such as an instance of `sub`.

```
int add(int x, int y) { return x + y; }
```

```
struct sub { int operator()(int x, int y) { return x - y; } };

int main() {
  function<int(int,int)> f = add;
  std::cout << f(1,2) << " ";
  f = sub();
  std::cout << f(1,2) << "\n";
}
// output is 3 -1
```

The following is the C++ translation of `apply`, using the `function` class template for the type of parameter `f`.

```
any_const_ref apply(any_const_ref x,
                    function<any_const_ref (any_const_ref)> const& f)
  { return f(x); }
```

We call the `apply` function with the following `deref` function as the argument for parameter `f`.

```
fun deref<U>(U* x) -> U { return *x; }
```

Compiling pointers in generic functions, such as `U*` above, is somewhat challenging because many pointer operations are *type dependent*. For example, to increment a pointer we must know the size of the object pointed to, and the size depends on the type of the object. However, the information is not available when the generic function is compiled. The solution we currently use is to extend the family of `any` classes to include `any_ptr`, `any_ptr_ref`, and `any_ptr_const_ref`. (There are also classes for pointers to constant objects.) These classes implement all of the usual pointer operations by dispatching through virtual functions to the real type-specific pointer. The following is the translation of `deref`.

```
any_const_ref deref(any_ptr_const_ref x) { return *x; }
```

One major drawback of this approach is that we cannot define a constructor for `any_ptr` with zero arguments (a so-called default constructor), for `any_ptr` must be initialized with

a real pointer (such as `int*`). Thus, to default construct a pointer inside a generic function, the `where` clause must include the requirement `DefaultConstructible<T*>`. In Section 3.3.2 we discussed using intensional type analysis (run-time type passing) to access information about types. Instead of the `any_ptr` class we could instead use `void*` along with the type information provided by the intensional type analysis to implement the pointer operations. Such an approach has the potential to be more efficient in time (both compile time and run time) and space (in code size) and also avoids the default construction problem (it is trivial to default construct a `void*`). We plan to experiment with this approach in the future.

The following code shows a call to `apply`, to which we pass a pointer and the `deref` function.

```
fun main() -> int@ {
  let p = new int(0);
  return apply(p, deref);
}
```

The `deref` function does not exactly match the expected type but it can implicitly instantiate to the expected type: `deref` is coerced from the type `fun<U>(U*)->U` to `fun(S)->T`. To accomplish this coercion, a function with type `fun(S)->T` is created that dispatches to `deref` and applies the appropriate coercions to the arguments and return value. In general, the inner function may be from an expression or a lexically bound variable, so the wrapper function must hold onto it. C++ lacks real closures but function objects can be used instead. The following is the function object wrapper that coerces `deref`.

```
struct __functor_412 {
  function<any_const_ref (any_ptr_const_ref)> f;
  __functor_412(function<any_const_ref (any_ptr_const_ref)> f): f(f) { }
  ~__functor_412() { }
  any_const_ref operator()(any_const_ref x) {
    return f(any_cast<any_ptr_const_ref>(x));
  }
};
```

The translation for the `main` function is shown below.

```
int main() {
  int* p = new int(0);
  return any_cast<int const&>(apply(p, __functor_412(deref)));
}
```

5.1.2. Concepts and models. As mentioned above, the translation associates a dictionary with each model and passes these dictionaries into generic functions. A convenient representation for dictionaries in C++ is objects with virtual function tables. We translate each concept to an abstract base class, and each model to a derived class with a singleton instance that will act as the dictionary. The `LessThanComparable` concept serves as a simple example.

```
concept LessThanComparable<X> {
  fun operator<(X, X) -> bool@;
  fun operator<=(X a, X b) -> bool@ { return not (b < a); }
  fun operator>(X a, X b) -> bool@ { return b < a; }
  fun operator>=(X a , X b) -> bool@ { return not (a < b); }
};
```

The following is the corresponding C++ abstract base class. Function signatures in the concept are translated to pure virtual functions and function definitions are translated to virtual functions (that may be overridden in derived classes.)

```
struct LessThanComparable {
  virtual bool __less_than(any_const_ref p, any_const_ref p) = 0;
  virtual bool __less_equal(any_const_ref a, any_const_ref b)
      { return ! __less_than(b, a); }
  virtual bool __greater_than(any_const_ref a, any_const_ref b)
      { return __less_than(b, a); }
  virtual bool __greater_equal(any_const_ref a, any_const_ref b)
      { return ! __less_than(a, b)); }
};
```

A model definition translates to a derived class with a singleton instance. The following definition establishes that `int` is a model of `LessThanComparable`.

```
model LessThanComparable<int> { };
```

For this model, all of the operations are implemented by the built-in comparisons for `int`. Thus, the implementation of the each virtual function coerces the arguments to `int` and then applies the built-in operator.

```
struct model_LessThanComparable_int : public LessThanComparable {
  virtual bool __less_than(any_const_ref a, any_const_ref b)
      { return any_cast<int const&>(a) < any_cast<int const&>(b); }
  virtual bool __less_equal(any_const_ref a, any_const_ref b)
      { return any_cast<int const&>(a) <= any_cast<int const&>(b); }
  virtual bool __greater_than(any_const_ref a, any_const_ref b)
      { return any_cast<int const&>(a) > any_cast<int const&>(b); }
  virtual bool __greater_equal(any_const_ref a, any_const_ref b)
      { return any_cast<int const&>(a) >= any_cast<int const&>(b); }
};
```

The following is a singleton instance of the model class that is passed to generic functions, such as `minimum`, to satisfy its requirement for the model `LessThanComparable<int>`.

```
LessThanComparable* __LessThanComparable_int = new model_LessThanComparable_int();
```

5.1.3. Generic functions with constraints. A generic function in $\mathcal{G}$ is translated to a normal C++ function with parameters for dictionaries corresponding to the models required by the `where` clause. Calling this C++ function corresponds to instantiating the generic function. The result of the call is a specialized function that can then be applied to the normal arguments. The generic `minimum` function below has a `where` clause that requires `T` to model `LessThanComparable`. Inside the generic function this capability is used to compare parameters `a` and `b`.

```
fun minimum<T> where { LessThanComparable<T> }
(T a, T b) -> T {
  if (b < a) return b;
  else return a;
}
```

The following code shows an explicit instantiation of `minimum` followed by a function application. These two steps are combined when implicit instantiation is used but it is easier to understand them as separate steps.

```
fun main() -> int@ {
```

```
  let m = minimum<|int|>;
  return m(0,1);
}
```

The translated `minimum` function is shown below.

```
function<any_const_ref (any_const_ref, any_const_ref)>
minimum(LessThanComparable* __LessThanComparable_T) {
  return __functor_550(minimum, __LessThanComparable_T);
}
```

The body of the `minimum` function is placed in the `operator()` of `__functor_550` and the use of `operator<` inside `minimum` is translated to `__LessThanComparable_T->__less_than`. `__functor_550` includes the function `minimum` as a data member to allow for recursion in the body of `minimum`, though in this case there is no recursion.

```
struct __functor_550 {
  typedef function<function<any_const_ref (any_const_ref, any_const_ref)>
                   (LessThanComparable*)> fun_type;
  fun_type minimum;
  LessThanComparable* __LessThanComparable_T;

  __functor_550(fun_type minimum, LessThanComparable* __LessThanComparable_T)
      : minimum(minimum), __LessThanComparable_T(__LessThanComparable_T) { }

  ~__functor_550() { }

  any_const_ref operator()(any_const_ref a, any_const_ref b) {
    if (__LessThanComparable_T->__less_than(b, a))
      return b;
    else
      return a;
  }
};
```

The instantiation (`minimum<|int|>` is translated to an application of the `minimum` function to the dictionary corresponding to the model required by its `where` clause, in this case `__LessThanComparable_int`, followed by an application of `__functor_551` to handle the coercions from `int const&` to `any_const_ref` and back.

```
__functor_551(minimum(__LessThanComparable_int))
```

The translation of the `main` function contains the instantiation of `minimum` and a call to `m`.

```
int main() {
  function<int const& (int const&, int const&)>
    m = __functor_551(minimum(__LessThanComparable_int));
  return m(0, 1);
}
```

5.1.4. Concept refinement. The `InputIterator` concept is an example of a concept that refines other concepts and includes nested requirements.

```
concept InputIterator<Iter> {
  type value;
  type difference;
  refines EqualityComparable<Iter>;
  refines Regular<Iter>;
  require SignedIntegral<difference>;
  fun operator*(Iter b) -> value@;
  fun operator++(Iter! c) -> Iter!;
};
```

Refinements and nested requirements are treated in a similar fashion in the translation. Both are added as data members to the abstract base class. One might expect refinements to instead translate to inheritance, but treating refinements and requirements uniformly results in a simpler implementation. The following shows the translation for `InputIterator`, with three data members for the refinements and requirement. A constructor is defined to initialize these data members.

```
struct InputIterator {
  InputIterator(EqualityComparable* EqualityComparable_Iter,
                Regular* Regular_Iter,
                SignedIntegral* SignedIntegral_difference)
      : EqualityComparable_Iter(EqualityComparable_Iter),
        Regular_Iter(Regular_Iter),
        SignedIntegral_difference(SignedIntegral_difference) { }

  virtual any __star(any_const_ref b) = 0;
  virtual any_ref __increment(any_ref c) = 0;

  EqualityComparable* EqualityComparable_Iter;
```

```
  Regular* Regular_Iter;
  SignedIntegral* SignedIntegral_difference;
};
```

The data members are used inside generic functions when a model for a refined concept is needed. For example, the function g requires `InputIterator` and calls `f`, which requires `EqualityComparable`.

```
fun f<X> where { EqualityComparable<X> }
(X x) { x == x; }

fun g<Iter> where { InputIterator<Iter> }
(Iter i) { f(i); }
```

In the translation of g we pass the `EqualityComparable_Iter` member from the input iterator dictionary to `f`. The following is the translation of g.

```
struct __functor_1262 {
  function<function<void (any_const_ref)> (InputIterator*)> g;
  InputIterator* __InputIterator_T;

  __functor_1262(function<function<void (any_const_ref)> (InputIterator*)> g,
                 InputIterator* __InputIterator_T,)
      : g(g), __InputIterator_T(__InputIterator_T) { }

  void operator()(any_const_ref i) {
    (f(__InputIterator_T->.EqualityComparable_Iter))(i);
  }
};
function<void (any_const_ref)> g(InputIterator* __InputIterator_T) {
  return __functor_1262(g,__InputIterator_T);
}
```

5.1.5. Parameterized models. Parameterized models, such as the following model of Input Iterator for `reverse_iterator`, introduce some challenges to compilation, and is one of the reasons concepts are translated to abstract base classes.

```
model <Iter> where { BidirectionalIterator<Iter> }
InputIterator< reverse_iterator<Iter> > {
  type value = BidirectionalIterator<Iter>.value;
  type difference = BidirectionalIterator<Iter>.difference;
};
```

When an instance of this model is created, it must be supplied a model of Bidirectional Iterator for the underlying `Iter` type. The parameterized model needs to store away this model for later use, so it needs some associated state. This motivated our approach of using derived classes for model definitions. Each derived class can define different data members corresponding to the requirement in its where clause. The following shows the translation for the above model definition.

```
struct model_InputIterator_reverse_iterator : public InputIterator {
  model_InputIterator_reverse_iterator(...,
              BidirectionalIterator* __BidirectionalIterator)
      : InputIterator(...),
        __BidirectionalIterator_Iter(__BidirectionalIterator_Iter) { }
  virtual any __star(any_const_ref i) {
    return (__star_reverse_iterator(__BidirectionalIterator_Iter))
             (any_cast<reverse_iterator const&>(i));
  }
  any_ref __increment(any_ref i) {
    return (__increment_reverse_iterator(__BidirectionalIterator_Iter))
             (any_cast<reverse_iterator&>(i));
  }
  BidirectionalIterator* __BidirectionalIterator_Iter;
};
```

For parameterized model definitions we do not create a singleton object but instead create the objects on-demand.

5.1.6. Model member access. Model members may be accessed explicitly with the dot notation, as in the following.

```
let plus = model Monoid<int>.binary_op;
let z = plus(0, 0);
```

A model member access translates to an access of a member in the corresponding dictionary. In this case, `binary_op` is a member of the `Semigroup` concept, which `Monoid` refines. So the C++ output must access the sub-dictionary for `Semigroup` and then access the `binary_op` member. However, there are two small complications handled by the two functors in the translation:

```
int main() {
```

```
  function<int (int const&, int const&)> plus
    = __functor_522(__functor_521(__Monoid_i->Semigroup_T));
  return plus_517(0, 0);
}
```

The first complication is that in C++ there is no direct representation for a member function bound to its receiver object. (There is a representation for an unbound member function.) Thus, we must bundle the `binary_op` together with the dictionary in the following functor to obtain a first class function.

```
struct __functor_521 {
  __functor_521(Semigroup* dict) : dict(dict) { }
  any operator()(any_const_ref param_1, any_const_ref param_2)
    { return dict->binary_op(param_1, param_2); }
  Semigroup* dict;
};
```

The second complication is that the parameter and return types of `binary_op` are dynamic types:

```
struct Semigroup {
  Semigroup(Regular* const& Regular_T) : Regular_T(Regular_T) { }
  virtual any binary_op(any_const_ref, any_const_ref) = 0;
  Regular* Regular_T;
};
```

To obtain a function with the correct parameter and return types we wrap the `binary_op` in the following function object which coerces the arguments and return value. (The arguments are implicitly coerced.)

```
struct __functor_522 {
  __functor_522(function<any (any_const_ref, any_const_ref)> f) : f(f) { }
  int operator()(int const& __1, int const& __2)
    { return any_cast<int>(f(__1, __2)); }
  function<any (any_const_ref, any_const_ref)> f;
};
```

5.1.7. Generic classes. In Section 3.3.2 we discussed the problem of how to layout the memory for parameterized classes and how to access fields in a uniform way inside a generic function. Using intensional type analysis we could use the same flattened layout for generic

classes and for non-generic classes. However, there is some challenge to implementing this portably: we need to mimic the layout of the underlying C++ compiler, which is not completely specified by the C++ standard. This is feasible but tricky. For now the compiler uses the simpler approach of boxing the data members of a class.

Consider the following simple class in $\mathcal{G}$. It is parameterized on type `T` and there is a constraint that `T` model `Regular`, which is needed for the copy construction of the `data` member.

```
class cell<T> where { Regular<T> }
{
  cell(T x) : data(x) { }
  T data;
};
```

The translation to C++ is shown below.

```
struct cell {
  cell(any_const_ref x, Regular* __Regular_T)
    : __Regular_T(__Regular_T), data(__Regular_T->new_on_stack(x)) { }
  any data;
  Regular* __Regular_T;
};
```

The type of the `data` member is `any` and the dictionary for `Regular<T>` is stored as an extra member of the class. The reason the dictionary is stored as a member is that in general the destructor for a class may need to use the dictionary.

5.2. A definitional compiler for $\mathcal{G}$

The compiler from $\mathcal{G}$ to C++ is a set of mutually recursive functions that recur on the structure of the abstract syntax tree (AST) of a $\mathcal{G}$ program. There are three categories of syntactic entities in $\mathcal{G}$: declarations, statements, and expressions, and so there is a recursive function for each of these categories. These functions are mutually recursive because, for example, some statements contain expressions and some expressions contain statements.

The compiler is type-directed, which means that many of the decisions made by the compiler are dependent on the type of an expression. Furthermore, the process of translating from implicit model passing to explicit dictionary passing is closely tied to the model lookup aspect of the type system of $\mathcal{G}$. Thus, the compiler and type checker are implemented together as the same functions.

Each of the recursive functions takes an ***environment*** parameter. The environment data structure includes information such as the type for each variable and function that is in scope. We describe the environment in detail in Section 5.2.2.

In addition to determining the type of an expression, the compiler also keeps track of whether an expression refers to an lvalue or rvalue and whether it is constant or mutable. We use the term ***annotated type*** to refer to a type together with this extra information.

The following describes the input and output of the compiler's main functions.

Compile declaration: The input is a declaration, an environment, and whether the current access context is public or private. The output is a list of C++ declarations and an updated environment. The reason that the output is a *list* of C++ declarations is that for some $\mathcal{G}$ declarations the compiler produces several C++ declarations. For example, a model definition translates to two C++ declarations: a class definition and a variable declaration for the singleton instance of the class.)

Compile statement: The input is a statement, an environment, and the declared return type of the enclosing function (if there is one). The return type is used to check the type of expressions in return statements. The output is a list of C++ statements, a list of annotated types, and an updated environment. The list of annotated types are the types from any return statements within the statement, which is used in the context of a function expression to deduce its return type.

Compile expression: The input is an expression, an environment, and the lvalue/rvalue context. For example, an expression on the left-hand side of an assignment is in an lvalue context. The compiler needs to know this context to make sure that

an rvalue expression does not appear in an lvalue context. The output is a C++ expression and an annotated type.

5.2.1. Types and type equality. One of the main operations performed on types during compilation is checking whether two types are equal. As discussed in Section 4.5, checking for type equality is somewhat complicated in $\mathcal{G}$ because of type parameters and same-type constraints. In $\mathcal{G}$, type equality is a congruence relation, and we use a congruence closure algorithm [142] to maintain equivalence classes of type expressions that refer to the same type.

The congruence closure algorithm requires that types be represented by a directed acyclic graph (DAG), with one node for each type. Figure 2 shows the DAG for the following types.

```
fun(cell<int>)->int
pair<cell<int>,fun(int)->float>
fun<T>(fun(T)->T, T)->T
```

Common parts of types are represented with a single subgraph. For example, there is a single `int` node which is used in three larger types. Each node is labeled with its type, except the sub-types are replaced with dots. The out-edges of the nodes are ordered, and the notation $u[i]$ denotes the target of the ith out-edge. We say that u is a ***predecessor*** of $u[i]$.

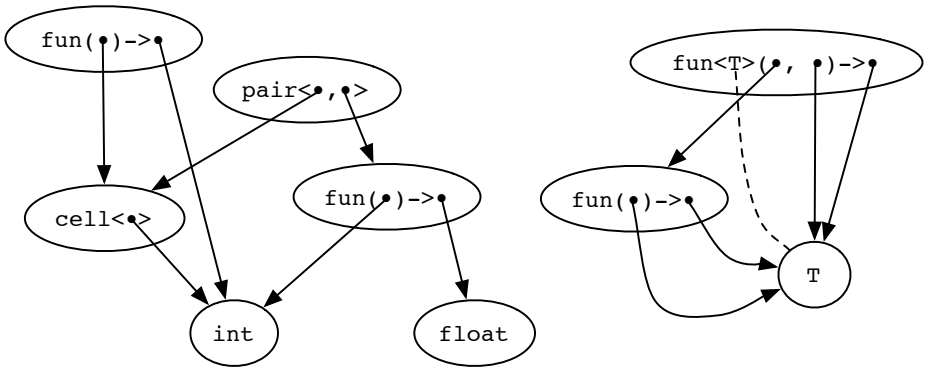

FIGURE 2. Types represented as a directed acyclic graph.

During compilation we may discover that two type expressions should denote the same type, so we need to merge two nodes into a single node. However, merging nodes is somewhat expensive because all the in-edges and out-edges must be rewired. Instead of merging the nodes we record that the two nodes are equivalent using a union-find data structure [49, 184] (also known as disjoint sets). For each equivalence class of nodes, the union-find data structure δ chooses a representative node and provides a find operation that maps a node to the representative for its equivalence class. Therefore, two nodes u and v are equivalent iff $\mathrm{find}(u, \delta) = \mathrm{find}(v, \delta)$. The union-find data structure also provides the $\mathrm{union}(u, v, \delta)$ operation which merges the equivalence classes of u and v, updated δ in place.

The merging of two nodes is complicated by the need to propagate the change to other types that refer to the two merged nodes, or that are parts of the merged nodes. For example, if we merge u and v then the nodes for `cell`<u> and `cell`<v> must also be merged. The propagation goes in other direction as well: if `cell`<u> and `cell`<v> were first merged, then u and v would need to be merged. A modified version of the `merge` algorithm from Nelson and Oppen [142] is shown in Figure 3. $P_u(G)$ denotes the set of all predecessors of the vertices equivalent to u in graph G.

Inserting type expressions into the graph. The DAG representation of the types is constructed incrementally as the compiler processes the $\mathcal{G}$ program. When a type expression τ is encountered it is inserted into the DAG. A new node u is created for τ and then the sub-types of τ are recursively inserted into the DAG, obtaining the nodes $v_1, \ldots, v_n$. Then the edges $(u, v_1), \ldots, (u, v_n)$ are added to the graph. Finally, if u is congruent to an existing vertex v, delete u and return v instead.

5.2.1.1. *Well-formed types.* The function `well_formed` checks whether a type is well formed and adds the type to the type DAG in the environment, returning the node representing the type. Figure 4 shows the pseudo-code for `well_formed`.

5.2.1.2. *Translating $\mathcal{G}$ types to C++ types.* The translation must convert from type expressions in $\mathcal{G}$ to type expressions in C++, for example, when translating the parameter type of function. We define a function that translates a $\mathcal{G}$ type τ in environment Γ to a C++ type

FIGURE 3. Merge procedure for congruence closure

```
merge(u,v,δ,G) modifies δ {
  if (find(u,δ) = find(v,δ))
    return;
  union(u,v,δ)
  k = outdegree(u)
  if (label(u) ≠ c<•>.t) // skip scoped-qualified types
    for i=1...k.
      merge(u[i], v[i], δ)
  for each (x,y) such that x ∈ P_u(G) and y ∈ P_v(G).
    if (find(x, δ) ≠ find(y,δ) and congruent(x,y,δ))
      merge(x, y, δ, G)
}
congruent(u,v,δ) {
  label(u) = label(v)
  and for i=1...outdegree(u). find(u[i],δ) = find(v[i],δ)
}
```

$[\![\tau]\!]_\Gamma$. For many types this translation is trivial, for example, $[\![\texttt{int}]\!]_\Gamma$ = `int`. We also define a function for translating a $\mathcal{G}$ type τ and a parameter passing mode m to a C++ type $[\![\tau, m]\!]_\Gamma$, which is used for translating the parameter types of a function.

The translation of type expressions is defined by recursion on the structure of types, but only for types that are representatives of their equivalence class. All other types are first mapped to their representative which is then translated to the C++ type. The function $[\![\tau]\!]_\Gamma$ = $[\![[\tau]_\Gamma]\!]$ where $[\![\cdot]\!]$ is defined as follows:

FIGURE 4. Well-formed types.

```
well_formed(‘t‘, Γ) {
  if (t ∈ dom(Γ.typevars)) return insert_node(t, Γ.dag)
  else raise error
}
well_formed(‘int‘, Γ) { return insert_node(‘int‘, Γ.dag) }
...
well_formed(‘τ*‘, Γ) {
  (τ′, Γ′) = well_formed(τ,Γ)
  return insert_node(‘τ*‘, Γ′)
}
well_formed(‘fun <ū> where { w̄ }(σ̄m̄) -> τm‘, Γ) {
  Γ′ = Γ, ū
  (w̄′, _, Γ′) = introduce_assumptions(w̄, Γ′)
  (σ̄′,Γ′) = well_formed(σ̄, Γ′)
  (τ′,Γ′) = well_formed(τ, Γ′)
  return insert_node(‘fun<t̄′> where { w̄′ }(σ̄′m̄ x̄) -> τ′m‘, Γ′)
}
well_formed(‘k<τ̄>‘, Γ) {
  τ̄′ = well_formed(τ̄, Γ)
  (t̄, w̄, _) = Γ.classes(k)
  if (length τ̄ ≠ length t̄) raise error
  satisfy_requirements([τ̄′/t̄]w̄, Γ)
  return insert_node(‘k<τ̄′>‘, Γ.dag)
}
well_formed(‘m.a‘, Γ) {
  if (m ∉ dom(Γ.modules)) raise error
  Γ′ = Γ.modules(m)
  if (t ∉ Γ′.typevars) raise error
  return insert_node(‘m.a‘, Γ)
}
well_formed(‘c<τ̄>.a‘, Γ) {
  τ̄′ = well_formed(τ̄, Γ)
  lookup_dict(c, τ̄′, Γ)
  find_associated_type(a, c)
  return insert_node(‘c<τ̄′>‘, Γ.dag)
}

find_associated_type(a, c) {
  (t̄, r̄) = Γ.concepts(c)
  if (type a ∈ r̄) return
  else
    for each ‘refine c′<σ̄>‘ in r̄.
      try { find_associated_type(a, c′); return }
      catch error { continue }
  raise error
}
```

$$[\![t]\!] = \texttt{any}$$

$$[\![c\texttt{<}\overline{\tau}\texttt{>}.a]\!] = \texttt{any}$$

$$[\![\texttt{int}]\!] = \texttt{int} \text{ (and likewise for all the basic types)}$$

$$[\![k]\!] = k \text{ when } k \text{ is a class, union, or struct identifier}$$

$$[\![k\texttt{<}\overline{\tau}\texttt{>}]\!] = k \text{ when } k \text{ is a class, union, or struct identifier}$$

$$[\![\tau\texttt{*}]\!] = \begin{cases} \texttt{any\_ptr} & \text{if } [\![\tau]\!] = \texttt{any} \\ [\![\tau]\!]\texttt{*} & \text{otherwise} \end{cases}$$

$$[\![\tau\ \texttt{const*}]\!] = \begin{cases} \texttt{any\_const\_ptr} & \text{if } [\![\tau]\!] = \texttt{any} \\ [\![\tau]\!]\texttt{const*} & \text{otherwise} \end{cases}$$

$$[\![\texttt{fun (}\overline{\sigma\ m}\texttt{) -> }\tau\ m]\!] = \texttt{function<}[\![\tau, m]\!]\texttt{(}\overline{[\![\sigma, m]\!]}\texttt{)>}$$

$$[\![\texttt{fun <}\overline{u}\texttt{> where \{ }\overline{w}\texttt{ \} (}\overline{\sigma\ m}\texttt{) -> }\tau\ m]\!] = \texttt{function<}\rho\texttt{(}\overline{C\texttt{*}}\texttt{)>}$$

where $\overline{C}$ is the list $\{C \mid C\texttt{<}\overline{\tau}\texttt{>} \in \bar{c}\}$

and $\rho = \texttt{function<}[\![\tau, m]\!]\texttt{(}\overline{[\![\sigma, m]\!]}\texttt{)>}$

The following defines the translation for parameters.

$$[\![\tau, !]\!] = \begin{cases} \texttt{any\_ref} & \text{if } [\![\tau]\!] = \texttt{any} \\ \texttt{any\_ptr\_ref} & \text{if } [\![\tau]\!] = \texttt{any\_ptr} \\ \texttt{any\_const\_ptr\_ref} & \text{if } [\![\tau]\!] = \texttt{any\_const\_ptr} \\ [\![\tau]\!]\texttt{\&} & \text{otherwise} \end{cases}$$

$$[\![\tau, \&]\!] = \begin{cases} \texttt{any\_const\_ref} & \text{if } [\![\tau]\!] = \texttt{any} \\ \texttt{any\_ptr\_const\_ref} & \text{if } [\![\tau]\!] = \texttt{any\_ptr} \\ \texttt{any\_const\_ptr\_const\_ref} & \text{if } [\![\tau]\!] = \texttt{any\_const\_ptr} \\ [\![\tau]\!]\ \texttt{const\&} & \text{otherwise} \end{cases}$$

$$[\![\tau, @]\!] = [\![\tau]\!]$$

5.2.2. Environment. The contextual information needed during the translation is maintained in an environment. The symbol Γ is used to denote the environment. An environment consists of:

$\Gamma.globals$ **and** $\Gamma.locals$**:** map from global variable names to bindings and map from local variable names to bindings, respectively. There are two kinds of bindings: for variables and for function overloads. A variable binding includes the $\mathcal{G}$ type (as a node), whether it is mutable or constant, the name to use for the variable in the C++ output, and whether the variable is public or private. The binding for a function overload contains a list of function types (nodes) and mangled names for the functions. The notation $\Gamma, (\texttt{global}\ x : (x', \tau, access))$ adds variable x to the global variable environment with type τ, the name x' for the C++ output, and $access$ specifies whether it is public or private. The notation $\Gamma, (\texttt{local}\ x : (x', \tau, access))$ adds the variable to the local environment. When a function named f is added to the environment, it is added to the set of overloads for f.

$\Gamma.classes$ **and** $\Gamma.structs$ **and** $\Gamma.unions$**:** maps from class, struct, and union names to their definitions, respectively.

$\Gamma.typevars$**:** maps from type variable names to their node in the type graph. The notation $\Gamma, (t : access)$ adds type variable t to the environment, mapping it to a new node, with the specified access (public or private).

$\Gamma.concepts$**:** maps from concept names to concept definitions. The notation $\Gamma, (c \mapsto (\bar{t}, \overline{r}, access))$ adds concept c to the environment, with type parameters $\bar{t}$ and requirements $\overline{r}$.

$\Gamma.models$**:** maps from concept names to a set of models. The information for each model includes the model head (a list of type nodes), the path for accessing the dictionary that corresponds to the model, and whether the model is public or private. The following notation adds a model to the environment.

$\Gamma,$ `model` $c\texttt{<}\overline{\tau'}\texttt{>} \mapsto (path, access)$

The following notation adds a parameterized model to the environment. In this case there is no dictionary, but we record the name of the derived class for the model.

$\Gamma,$ `model <`$\bar{t}$`> where { `$\overline{w}$` }` $c\texttt{<}\overline{\tau}\texttt{>} \mapsto (mclass, access)$

$\Gamma.dag$**:** is a directed acyclic graph that represents the types that appear in the program.

$\Gamma.\delta$**:** is a union-find (disjoint sets) data structure for maintaining equivalence classes of type expressions that denote the same type.

5.2.3. Auxiliary functions. The main compilation functions rely on several auxiliary functions. The two most important of these functions are used to process `where` clauses. The `introduce_assumptions` function is used in the compilation of function and model definitions. This function adds surrogate model and function signatures to the environment according to the contents of the `where` clause. The `satisfy_requirements` function is used in the instantiation of a generic function or model, and is used to check whether a model

satisfies the requirements of a concept. This function looks in the environment to see if the `where` clause is satisfied, and returns dictionaries and function definitions that satisfy the requirements.

Pseudo-code for `introduce_assumptions` is shown in Figure 5. The requirements in the `where` clause are processed in order; later requirements may depend on earlier requirements. For example, a later requirement may refer to an associated type that an earlier requirement brought into scope. If the requirement is a nested model requirement $c\langle\overline{\tau}\rangle$ we add the model to the environment and then introduce all the assumptions associated with the concept with a recursive call to `introduce_assumptions`. Refinements are processed in a similar way except associated types are brought into scope directly instead of being model-qualified. A function signature requirement adds to the overload set for that function, and a same type requirement causes the two types to be merged according to the congruence closure algorithm. Note that this merging may cause otherwise distinct model requirements to become the same requirement. Some care must be take to ensure that such models do not add duplicate functions into the overload set. The `introduce_assumptions` function returns the `where` clause (now containing pointers into the type DAG), the list of dictionary names, and the new environment.

The pseudo-code for `satisfy_requirements` is shown in Figure 6. For each model requirement or refinement we invoke `lookup_dict` to find the dictionary for the model. For each associated type we check that the type has been defined. For each same type constraint we check that the two type expressions are in the same equivalence class using the `find` function (of the union-find data structure). For each function signature we call `create_impl` which checks to see if there is a function defined that can be coerced to the signature and then creates a function that performs the coercion, if needed. The `coerce` function is responsible for inserting `any_casts` for converting from a polymorphic object to a concrete object, for wrapping functions when there needs to be coercions on the parameter or return type, and for choosing a particular overload from an overload set.

The `lookup_dict` function finds a model for a given concept and type arguments and returns the path to the dictionary for the model. Figure 7 shows pseudo-code for this function.

FIGURE 5. Pseudo-code for introducing `where` clause assumptions.

```
introduce_assumptions(w̄, Γ, path = [], scope=global, inref=false) {
  w̄' = []
  for i = 1, ..., length(w̄) {
    match w̄_i with
      c<τ̄> | require c<τ̄> ⇒
        if (c ∉ dom(Γ)) raise error;
        (τ̄', Γ) = well_formed(τ̄, Γ)
        w̄' = w̄', `require c<τ̄'>`
        d = fresh_name(); d̄ = d̄, d
        (t̄', w̄2) = Γ.concepts(c)
        (_,_,Γ) = introduce_assumptions([τ̄'/t̄']w̄2, Γ, path@[d], c<τ̄'>, false)
        Γ = Γ, model c<τ̄'> ↦ (path@[d], public)
    | refine c<τ̄> ⇒
        same as above except:
        (_,_,Γ') = introduce_assumptions([τ̄'/t̄']w̄2, Γ, path@[d], c<τ̄'>, inref)
        ...
    | type t ⇒
        w̄' = w̄', `type t`
        if (inref) Γ = Γ, t, scope.t
        else Γ = Γ, scope.t
    | fun<t̄> where { w̄ }(σ̄m̄) -> τm
    | fun<t̄> where { w̄ }(σ̄m̄) -> τm { s̄ } ⇒
        f' = fresh_name()
        Γ = Γ, local f : (f', fun<t̄> where { w̄ }(σ̄m̄) -> τm)
    | τ1 == τ2 ⇒
        (τ1',Γ) = well_formed(τ1, Γ); (τ2',Γ) = well_formed(τ2, Γ)
        w̄' = w̄', `τ1' == τ2'`
        merge(τ1', τ2', Γ.δ, Γ.dag)
  }
  return (w̄', d̄, Γ)
}
```

The function is mutually recursive with the `satisfy_requirements` function, for if a model is constrained by a `where` clause it must lookup dictionaries to satisfy those requirements. This recursion accomplishes a depth-first search for the requirements. Here we show the basic algorithm, but it can be enhanced to catch problems amongst model definitions such as catching circularity in model definitions and enhanced to prevent divergence.

FIGURE 6. Pseudo-code for satisfying requirements.

```
satisfy_requirements(w̄, Γ) {
  d̄ = []; impls = []
  for each w in w̄ {
    match w with
      c<τ̄> | require c<τ̄> | refine c<τ̄> ⇒
        let (d,_) = lookup_dict(c, S(τ̄), Γ) in d̄ = d̄,d
    | type t ⇒
      if (t ∉ dom(Γ.typevars)) raise error;
    | τ1 == τ2 ⇒
      if (find(τ1,Γ.δ) ≠ find(τ2,Γ.δ)) raise error;
    | fun f<t̄> where { w̄' }(σm̄) -> τ ⇒
      impls = impls, create_impl(f, ‘fun<t̄> where { w̄' }(σm̄) -> τ‘, Γ)
    | fun f<t̄> where { w̄' }(σm̄) -> τ { s̄ } ↦ default ⇒
      try { impls = impls, create_impl(f, fun<t̄> where { w̄' }(σm̄) -> τ, Γ) }
      catch error with { }
  }
  return (d̄, impls)
}
create_impl(f, fun<t̄> where { w̄ }(σm̄) -> τm, Γ) {
  Γ = Γ,t̄
  (_, _, Γ) = introduce_assumptions(w̄, Γ)
  (f',τ'') = resolve_overload(Γ(f), σm̄, Γ)
  if (τ'' ⋠ τ) raise error;
  p̄ = map (λσ. fresh_name()) σ̄
  f'' = coerce(f', τ'', fun(σm̄) -> τm, Γ)
  return ‘⟦τm⟧Γ f(⟦σm⟧Γ p̄) { return f''(p̄); }‘
}
```

In more detail, the `lookup_dict` function extracts all the models for concept c from the environment and then invokes `best_matching_model` to choose the most specific model. If the model is not generic we return the C++ expression for accessing the model. If the model is generic we must construct a new model object, passing in the dictionaries for the `where` clause and also the dictionaries for the refinements and requirements in concept c. We `unify` the type arguments with the model head to obtain a substitution which is applied to the `where` clause before calling `satisfy_requirements` to obtain the dictionaries. The unification algorithm used is that of ML^{F} [24]. The dictionaries for the refines and requires are obtained by recursive calls to `lookup_dict`.

FIGURE 7. Pseudo-code for finding the dictionary for a model.

```
lookup_dict(c, τ̄, Γ)
{
  match best_matching_model(τ̄, Γ.models(c), Γ) with
    model c<τ̄'> ↦ (path, access) ⇒
      return (make_dict_access(path), Γ')
  | model <t̄> where { w̄ } c<τ̄'> ↦ (mclass, access))
      S = unify(τ̄', τ̄, Γ, ∅)
      (d̄_w, _) = satisfy_requirements(S(w̄), Γ)
      (s̄,w̄_2) = Γ.concepts(c)
      d̄_r = map (λ c<τ̄>. let (d,_) = lookup_dict(c, [S(τ̄')/s̄]τ̄, Γ) in d)
                (refines and requires in w̄_2)
      return (‘new GC mclass(d̄_r, d̄_w)‘, [S(τ̄')/s̄]Γ')
}
make_dict_access([d]) = ‘d‘
make_dict_access(d :: path) =
  let rest = make_dict_access(path)
  in ‘d->rest‘
```

The pseudo-code for `best_matching_model` is shown in Figure 8. The input to this function is some type arguments, a list of models, and the environment; this function returns a model. First we find all models that match the type arguments $\overline{\tau}$. In the case of a generic model we try to unify the type arguments with the head of the model. Once the list of matching models is obtained, this function determines the most specific of the matches, if there is one, using the more-specific-model relation as defined in Section 4.6.2.

Figure 9 shows the pseudo-code for overload resolution. The input to this function is a list of function names with their types, the argument types, and the environment. This function is quite similar to the `best_matching_model` function, following the same basic pattern. The algorithm first filters out functions that are not applicable to the argument types and then tries to find the most specific function among the remaining overloads.

FIGURE 8. Algorithm for finding the most specific matching model.

```
best_matching_model(τ̄, models, Γ) {
  matches =
    filter (λm.
      match m with
        model c<τ̄> ↦ (path, access) ⇒ return true
      | model <t̄> where { w̄ } c<τ̄'> ↦ (mclass, access) ⇒
        try { S = unify(τ̄', τ̄); satisfy_requirements(S(w̄), Γ); return true }
        catch error { return false }
     ) models
  match matches with
    [] ⇒ raise error;
  | [m] ⇒ return m
  | m :: matches =>
    best = m
    while (matches ≠ []) {
      if (best more specific model than hd(matches)) ;
      else if (hd(matches) more specific model than best) best = hd(matches)
      else raise error;
      matches = tl(matches)
    }
    return best
}
```

FIGURE 9. Algorithm for function overload resolution.

```
resolve_overload(ovlds, σ̄m̄, Γ) {
  matches = filter (λ(f,τ).
                      α,β,γ,m′,r′ fresh variables
                      Q = {τ ≤ α, σ̄m̄ ≤ β̄m′}
                      try {
                        unify(α, fun(β̄)->γ, Γ, Q);
                        iter (λ c<τ̄>. lookup_dict(c, S(τ̄), Γ)) where(τ);
                        return true;
                      } catch error { return false; })
                ovlds
  match matches with
    [] ⇒ raise error;
  | [(f,τ)] ⇒ return (f,τ)
  | (f,τ) :: matches =>
    best = (f,τ)
    while (matches ≠ []) {
      if (snd(best) more specific overload than snd(hd(matches))) ;
      else if (snd(hd(matches)) more specific overload than snd(best))
        best = hd(matches)
      else raise error;
      matches = tl(matches)
    }
    return best
}
```

FIGURE 10. Pseudo-code for compiling function definitions.

```
compile(‘fun f<t̄> where { w̄ }(σ̄m x̄) -> τm { s̄ }‘, Γ, access) {
  Γ′ = Γ, t̄
  (w̄′, d̄_w, Γ′) = introduce_assumptions(w̄, Γ′)
  (σ̄′,Γ′) = well_formed(σ̄, Γ′); (τ′,Γ′) = well_formed(τ, Γ′)
  τ_f = fun<t̄′> where { w̄′ }(σ′m x̄) -> τ′m
  f′ = fresh_name()
  Γ′ = Γ′, x̄ : σ̄′, f : (f′, τ_f)
  (s̄′, _, _) = compile(s̄, τ′m, Γ′)
  if (t̄ = [])
    return (‘⟦τ′m⟧_Γ′ f′(⟦σ′m⟧_Γ′ x̄) { concat(s̄′) }‘, Γ, (global f : (f′, τ_f, access)))
  else
    c̄_w = map (λ c<τ̄>. c) w̄′
    f″ = fresh_name()
    return (‘class f″ {
              public:
                f″(c_w* d_w) : d_w(d_w) { }
                ⟦τ′m⟧_Γ′ operator()(⟦σ′ m⟧_Γ′ x) const { concat(s̄′) }
              private:
                c_w* d_w;
              };
              function<⟦τ′m⟧_Γ′(⟦σ′m⟧_Γ′)> f′(c_w* d_w) { return f″(d_w); }‘,
              Γ, (global f : (f′, τ_f, access)))

}
```

5.2.4. Declarations. In this section we describe the cases of the main `compile` function for declarations. The case for generic function definitions is shown in Figure 10. The type parameters of the generic function are added to the environment and the auxiliary function `introduce_assumptions` is used to augment the environment according to the `where` clause of the function. The parameters are then added to the environment and also the function itself to enable recursion. The body of the function is then compiled. If the function is generic, it is compiled to a curried function which takes the dictionaries corresponding to its `where` clause and returns a function object.

Figure 11 shows the pseudo-code for compiling concepts. The body of the concept is processed using the `introduce_assumptions` function to produce Γ' and then the function

FIGURE 11. Pseudo-code for compiling concepts.

```
compile(‘concept c<t̄>{ r̄ };‘, Γ, access) {
  Γ′ = Γ, t̄
  (t̄′, r̄′, _, Γ′) = introduce_assumptions(r̄, Γ′)
  c_r = map (λ c<τ>. c) (refines and requires in r̄′)
  d_r = map name_mangle (refines and requires in r̄′)
  (f̄, _) = map (λf. compile(f, Γ′)) (funsigs in r̄)
  (f̄′, _) = map (λf. compile(f, Γ′)) (fundefs in r̄)
  return (‘class c {
            public:
              c(c_r* d_r) : d_r(d_r) { }
              virtual f = 0;
              virtual f′
            private:
              c_r* d_r;
            };‘, Γ, (c ↦ (t̄′, r̄′, access)))
}
```

definitions in the concept are compiled in Γ'. The output is a class definition with pure virtual functions for each function signature in the concept, and a virtual function for each function definition. In addition, there are data members to point to the dictionaries for the refinements and nested model requirements. The environment is updated with an entry for the concept.

Figure 12 shows the pseudo-code for compiling model definitions. The definition is compiled in an environment Γ' that is extended with the type parameters $\bar{t}$ and also with the `where` clause with a call to `introduce_assumptions`. The definitions in the body of the model are compiled in Γ' and then added to Γ'. The model definition must satisfy the requirements of the concept, so we call `satisfy_requirements`. The generated C++ code consists of a class derived from the concept's abstract base class, and optionally a singleton object for the dictionary. If the model is generic, the compiler instead creates dictionary objects on-demand in `lookup_dict`.

The compilation of `let` declarations and type aliases is straightforward. A `let` compiles to a variable declaration, where the variable is given the type of the expression on the right

FIGURE 12. Compile model definitions.

```
compile(‘model <t̄> where { w̄ } c<τ̄>{ d̄ };‘, Γ, access) {
  Γ′ = Γ, t̄
  (w̄′, d̄_w, Γ′) = introduce_assumptions(w̄, Γ)
  (τ̄′, Γ′) = well_formed(τ̄, Γ′)
  (d̄′, Γ′) = compile(d̄, Γ′)
  (s̄,r̄) = Γ.concepts(c)
  (d̄_r, f̄) = satisfy_requirements([τ̄′/s̄]r̄, Γ′)
  c̄_r = filter_map (λ c<τ̄>. c) r̄
  c̄_w = filter_map (λ c<τ̄>. c) w̄′
  mclass = fresh_name(); d̄_r = map (λc. fresh_name()) c̄_r
  mdef = ‘class mclass : public c {
            public:
              m(c̄_r* r̄, c̄_w* d̄_w) : c(d̄_r), d̄_w(d̄_w) { }
              virtual f̄
            private:
              d̄′
              c̄_w* d̄_w;
            };‘
  if (t̄ = [])
    d_m = fresh_name();
    inst = ‘c* d_m = new mclass(d̄_r);‘
    return (mdef inst, Γ, model c<τ̄′> ↦ ([d_m], access))
  else
    return (mdef, Γ, model <t̄′> where { w̄′ } c<τ̄′> ↦ (mclass, acccess))
}
```

hand side. A type alias does not produce C++ output, but updates the environment with the equality $t = \tau$. Figure 13 shows the pseudo-code for compiling value and type aliases.

Class, struct, and union definitions are similar so we only discuss compiling classes. Figure 14 shows the pseudo-code. The type parameters and `where` clause are added to the environment to form Γ'. Class members are compiled in Γ'. The output C++ class contains extra data members for the dictionaries corresponding to the `where` clause and each constructor includes extra parameters for these dictionaries. The constructors themselves may be parameterized and constrained with `where` clauses, so two sets of dictionaries are passed to a constructor. Overload resolution between constructors is handled in the compilation from $\mathcal{G}$ to C++ and the normal C++ constructor overload resolution must be disabled. To

FIGURE 13. Compile value and type aliases.

```
compile(‘let x = e;‘, Γ, access) {
  (e′, τ) = compile(e, Γ)
  return (‘⟦τ⟧Γ x = e′‘, Γ, x : (x, τ, access))
}
compile(‘type t = τ‘, Γ, access) {
  (τ′, Γ) = well_formed(τ, Γ)
  Γ = Γ, (t : access)
  merge(t, τ′, Γ.δ, Γ.dag)
  return (‘‘, Γ)
}
```

this end each constructor has an extra parameter *consid* of a unique type that can be use to force C++ overload resolution to the correct constructor. Otherwise, the compilation of constructors is similar to the compilation of normal function definitions.

Figure 15 shows the compilation of `module` definitions and related declarations such as the `scope` alias and `public` and `private` declarations. A module in $\mathcal{G}$ is translated to a C++ namespace.

FIGURE 14. Compile class definition.

```
compile(‘class k<t̄> where { w̄ } { c̄  ~k(){s̄} τ̄ x̄; };‘, Γ, access) {
  Γ′ = Γ, t̄
  (w̄′, d̄_w, Γ′) = introduce_assumptions(w̄, Γ)
  c̄′ = map (λc. compile(c, c̄_w, d̄_w, Γ′)) c̄
  (s̄′, _) = compile(s̄, Γ′)
  τ̄′ = map (λτ. well_formed(τ, Γ′)) τ̄
  c̄_w = filter_map (λ c<τ̄>. c) w̄′
  return (‘class k {
            public:
              c̄′
              ~k() { s̄′ }
              ⟦τ′⟧_Γ′ x̄;
            private:
              c_w d_w;
            };‘, Γ, k ↦ (w̄′, c̄′, access))
}
compile(‘<t̄> where { w̄ } k(σ̄m ȳ) : x(e)‾ { s̄ }‘, c̄_k, d̄_k, Γ) {
  Γ′ = Γ, t̄
  (w̄′, d̄_w, Γ′) = introduce_assumptions(w̄, Γ′)
  (σ̄′,Γ′) = well_formed(σ̄, Γ′)
  Γ′ = Γ′, ȳ : σ̄′
  (ē′, _) = compile(ē, Γ′)
  (s̄′, _, _) = compile(s̄, void, Γ′)
  c̄_w = map (λ c<τ̄>. c) w̄′
  consid = fresh_name()
  return (‘k(c_k* d_k‾, c_w* d_w‾, ⟦σ′m⟧_Γ′ y‾, consid) : d_k(d_k)‾, d_w(d_w)‾, x(e′)‾ { s̄′ }‘ )
}
```

FIGURE 15. Compile module definition and related declarations.

```
compile(‘module m { d }‘, Γ, access) {
  (d', Γ') = compile(d, Γ, private)
  Γ'' = public members of Γ'
  return (‘namespace m { d' }‘, Γ, module m ↦ (Γ'', access))
}
compile(‘scope m = scope;‘, Γ, access) {
  Γ' = lookup_scope(global.scope, Γ)
  return (‘‘, Γ, module m ↦ (Γ', access))
}
compile(‘import scope.c<τ>;‘, Γ, access) {
  Γ' = lookup_scope(global.scope, Γ)
  (τ', Γ') = well_formed(τ, Γ')
  (d,_) = lookup_dict(c, τ', Γ')
  return (‘‘, Γ, model c<τ'> ↦ ([d], access))
}
compile(‘public: d‘, Γ, access) {
  (d', Γ') = compile(d, Γ, public)
  return (‘d'‘, Γ, Γ')
}
compile(‘private: d‘, Γ, access) {
  (d', Γ') = compile(d, Γ, private)
  return (‘d'‘, Γ, Γ')
}
```

5.2.5. Statements. This section defines the compilation of $\mathcal{G}$ statements to C++.

The `let` statement in $\mathcal{G}$ binds a name to an object. Thus it is similar to reference in C++. There is one small complication: in C++ a temporary cannot be bound to a non-const reference whereas the right hand side e of the `let` may be a temporary. Thus, the output C++ must first bind e' to a const reference, thereby extending its lifetime to the extent of the surrounding scope, and then assign the const reference to x, which is declared as either a const or non-const reference depending on the mutability of e.

```
compile(‘let x = e;‘, τm, Γ) {
  (e′, τm) = compile(e, Γ)
  τ′ = ⟦τm⟧Γ&
  return (‘⟦τ⟧Γ const& __x = e′;
            τ′ x = (τ′)__x;‘, [], Γ, local x : τm)
}
```

The `type` alias statement in $\mathcal{G}$ binds a name to a type. This introduces the type name t and merges it with the type τ. The `type` alias statement translates to an empty C++ statement.

```
compile(‘type t = τ;‘, τm, Γ) {
  (τ′, Γ) = well_formed(τ, Γ)
  Γ = Γ, t
  merge(t, τ′, Γ.δ, Γ.dag)
  return (‘;‘, Γ)
}
```

The compilation of the `return` statement depends on whether it is inside a function definition or a function expression. In the case of a function definition, there is a declared return type and the type of e must be convertible to the declared return type. In the case of a function expression there is no declared return type, and the `compile` function returns the type of e so that the return type of the function expression may be deduced.

FIGURE 16. Compilation of `if`, `while`, compound, and empty statements.

```
compile(‘if (e) s1 else s2‘, τm, Γ) {
  (e′, σm′) = compile(e, Γ)
  if (σ ⋠ bool) raise error;
  (s1′, rets1, _) = compile(s1, τm, Γ)
  (s2′, rets2, _) = compile(s2, τm, Γ)
  return (‘if (e′) { s1′ } else { s2′ }‘, rets1@rets2, Γ)
}
compile(‘while (e) s‘, τm, Γ) {
  (e′, σm′) = compile(e, Γ)
  if (σ ⋠ bool) raise error;
  (s′, rets, _) = compile(s, τm, Γ)
  return (‘while (e′) { s′ }‘, rets, Γ)
}
compile(‘{ s̄ }‘, τm, Γ) {
  (s̄′, rets, _) = compile(s̄, τm, Γ)
  return (‘{ concat(s̄′) }‘, concat(rets), Γ)
}
compile(‘e;‘, τm, Γ) {
  (e′, _) = compile(e, Γ);
  return (‘e′‘, [], Γ)
}
compile(‘;‘, τm, Γ) { return (‘;‘, [], Γ) }
```

```
compile(‘return e;‘, τm, Γ) {
  (e′, σm′) = compile(e, Γ)
  if (σm′ ⋠ τm) error
  return (‘return e′;‘, [], Γ) {
}
compile(‘return e;‘, void, Γ) {
  (e′, σm′) = compile(e, Γ)
  return (‘return e′;‘, [σm′], Γ) {
}
```

The compilation of `if`, `while`, compound, expression, and empty statements is shows in Figure 16. The compilation for each of these statements is straightforward.

FIGURE 17. Compile switch statement.

```
compile(‘switch (e) { c̄ }‘, τm, Γ) {
  (e′, τ) = compile(e, Γ)
  match τ with
    k<τ̄′> ⇒
      if (k ∉ Γ.unions(k)) raise error
      x = fresh_name()
      (c̄′, rets̄) = map (λc.
                        match c with
                          ‘case y: s̄‘ ⇒
                            (s̄′, rets, _) = compile(s̄, τm, Γ);
                            (t̄, w̄, mems) = Γ.unions(k);
                            σ = mems(y);
                            z = coerce(x->u->y, σ, [τ̄′/t̄]σ);
                            (‘case k::y: { σ& y = z; s̄′ break; }‘, rets)
                        | ‘default: s̄‘ ⇒
                            (s̄′, rets, _) = compile(s̄, τm, Γ);
                            (‘default: s̄′‘, rets))
                       c̄
      return (‘{ τ& x = e′; switch (x->tag) { c̄′ } }‘, concat(rets̄), Γ)
  | _ ⇒ raise error
}
```

The pseudo-code in Figure 17 describes the compilation of `switch` statements. The `switch` statement in $\mathcal{G}$ is specialized for use with `unions`. The `union` object has a tag that indicates which data member is present, and the `switch` statement dispatches based on this tag. The `union` class contains an `enum` with a constant for each data member.

FIGURE 18. Compilation of function application expressions.

```
compile(‘rator(rand…), Γ) {
  (rator′, τm) = compile(rator, Γ)
  (rand′…, σm…) = map (λe. compile(e, Γ)) rand…
  (rator″, τ′) = resolve_overload(τ, σm…, Γ)
  α, β, γ, m′, r′ fresh variables
  Q = {τ′ ≤ α, σm… ≤ βm′}
  S = unify(α, fun(β…)->γ, Γ, Q)
  rand″… = coerce(rand′…, σm…, paramtypes(τ′), Γ)
  if (where(τ′) = [])
    return (‘rator″(rand″…)‘, S(γ)m′)
  else {
    (d…, _) = satisfy_requirements(S(where(τ′)), Γ)
    return (‘(rator″(d…))(rand″…)‘, S(γ)m′)
  }
}
```

5.2.6. Expressions. This section describes the compilation of $\mathcal{G}$ expressions to C++.

Figure 18 shows the pseudo-code for compiling a function application. The *rator* may be a function or a function overload set. If it is a function then we treat it as an overload set with only a single overload. The `resolve_overload` function is called to determine the best overload. We then `unify` the arguments' types with the parameters' types to obtain a substitution S. A mismatch between argument and parameter types would cause `unify` to raise an error. If the function has a `where` clause, `satisfy_requirements` is called to obtain dictionaries. The C++ output is an application with the dictionaries and then a second application with the arguments. If the function does not have a `where` clause, the C++ translation is just an application with the arguments.

Figure 19 shows the pseudo-code compilation of object construction. This is similar to compiling a function application. The constructors of class k form an overload set from which the best match is chosen according to `resolve_overload`. Once the best constructor is chosen, `unify` is applied to deduce the type arguments for the constructor and then

FIGURE 19. Compilation of object construction.

```
compile(‘alloc k<τ̄>(rand)‘, Γ) {
  (t̄, w̄, mems) = Γ(k)
  (τ̄′, Γ′) = well_formed(τ̄)
  (d̄_k,_) = satisfy_requirements([τ̄′/t̄]w̄, Γ′)
  (rand′, σm) = compile(rand, Γ′)
  (consid, σ′) = resolve_overload(mems,σ̄, Γ′)
  α, β, γ, m′, r′ fresh variables
  Q = {σ′ ≤ α, σm ≤ βm′}
  S = unify(α, fun(β)->γ, Γ′, Q)
  rand″ = coerce(rand′, σm, paramtypes(σ′), Γ′)
  if (where(σ′) = [])
    return (‘k(rand″, consid)‘, k)
  else {
    (d̄_c, _) = satisfy_requirements(S(where(σ′)), Γ′)
    return (‘[[alloc]] k(d̄_k, d̄_c, rand″, consid)‘, k<τ̄′>)
  }
}
[[@]]Γ = ‘‘
[[new]]Γ = ‘new‘
[[new GC]]Γ = ‘new (GC)‘
[[new (e)]]Γ = let (e′,_) = compile(e, Γ) in ‘new (e′)‘
```

satisfy_constraints is applied to obtain dictionaries for the where clause of the constructor. The compiler must also obtain dictionaries for the where clause of class k and pass these to the constructor.

The pseudo-code for compiling an explicit instantiation is shown in Figure 20. The type of expression e must be a generic function type. The compiler invokes satisfy_requirements to check that the requirements of the where clause are satisfied and to obtain dictionaries. The output C++ is the compilation of e, that is e', applied to the dictionaries.

The compilation of variables is somewhat complicated by the distinction between global and local variables. Further, we treat function overload sets specially by combining the local and global overloads. Figure 21 shows the pseudo code for compiling a variable. The returned expression for a function overload set is unused, the actual translation will be determined by overload resolution, so we return ‘0‘ as the expression.

FIGURE 20. Compilation of explicit instantiation.

```
compile(‘e<|σ̄|>‘, Γ) {
  (σ̄′, Γ′) = well_formed(σ̄, Γ)
  (e′, τ) = compile(e, Γ)
  match τ with
    fun<t̄> where { w̄ }(ρ̄m̄) -> τ′m ⇒
    (d̄, _) = satisfy_requirements([σ̄′/t̄]w̄, Γ)
    return (‘e′(d̄)‘, fun([σ̄′/t̄]ρ̄m̄) -> [σ̄′/t̄]τ′m)
  | _ ⇒ error
}
```

FIGURE 21. Compile variable.

```
compile (‘x‘, Γ) {
  if (x ∈ dom(Γ.locals))
    match Γ.locals(x) with
      (x′, τ) ⇒ return (x′, τ)
    | ovlds ⇒
      match Γ.globals(x) with
        (x′, τ) ⇒ return (‘0‘, ovlds)
      | ovlds′ ⇒ return (‘0‘, ovlds@ovlds′)
  else if (x ∈ dom(Γ.globals))
    match Γ.locals(x) with
      (x′, τ) ⇒ return (x′, τ)
    | ovlds ⇒ return (‘0‘, ovlds)
  else error;
}
```

Figure 22 shows the pseudo-code for compiling a scope access expression. There are two kinds of scopes in $\mathcal{G}$, models and modules. In $\mathcal{G}$ the dot operator is used to access members of model and scopes, whereas in the C++ translation we must use `::` to access members of modules because they are translated to a C++ namespaces and we must use `->` to access members of a model because they are translated to objects. For simplicity, Figure 22 only shows the code for accessing into a single un-nested scope. This can be extended to handle nested scopes by iterating the process. However, the access of a model member

FIGURE 22. Compilation of scope member access expressions.

```
compile(‘m.x‘, Γ) {
  Γ′ = Γ.modules(m)
  if ((x : (x′, τ)) ∈ Γ′) return (‘m::x′‘, τ)
  else raise error
}
compile(‘c<τ̄>.x‘, Γ) {
  (τ̄′, Γ′) = well_formed(τ̄, Γ)
  (d, Γ′) = lookup_dict(c, τ̄′, Γ′)
  return access_model_member(x, c, τ̄′, d, [])
}
access_model_member(c<τ̄′>.x, path, Γ) {
  (t̄, r̄) = Γ.concepts(c)
  if (x ∈ dom(r̄))
    d = make_dict_access(path)
    return coerce(‘d->x‘, r̄(x), [τ̄′/t̄]r̄(x))
  else
    for each ‘refine c′<σ̄>‘ in r̄.
      try {
        m = name_mangle(c′<σ̄>)
        return access_model_member(c′, [τ̄′/t̄]σ̄, x, path@[m], Γ)
      } catch error { continue }
  raise error;
}
```

is still complicated by the fact that the member may be in a refinement, so the recursive `access_model_member` function is needed to search through the refinement hierarchy.

Figure 23 shows the pseudo-code to compile the access of an object member. The coercion is necessary, for example, to unbox the member if it is polymorphic.

5.3. Compiler implementation details

This section discusses some details of the implementation of the prototype compiler for $\mathcal{G}$. The implementation is written in Objective Caml [115]. We chose Objective Caml because it has several features that speed compiler implementation:

- Algebraic data types and pattern matching facilitate the manipulation of abstract syntax trees.

FIGURE 23. Pseudo-code for compiling access to an object member.

```
compile(‘e.x‘, Γ) {
  (e′, τ) = compile(e, Γ)
  match τ with
    k<σ̄> ⇒
      (t̄, w̄, mems) = Γ.classes(k)
      τ′ = [σ̄/t̄]mems(x)
      return (coerce(‘e′.x‘, mems(x), τ′), τ′)
  | _ ⇒
    error
}
```

- The Ocamllex and Ocamlyacc tools for lexical analysis and parsing are particularly easy to use.
- Automatic memory reclamation removes the work of manual memory management.

One disadvantage of Objective Caml with respect to Scheme for compiler construction is that Objective Caml does not have quasi-quote. In Scheme, quasi-quote provides a convenient way to form abstract syntax trees.

Ocamlyacc is an LALR(1) parser. The grammar of $\mathcal{G}$ is similar to that of C++ but differs in several respects to make the grammar LALR(1). For example, explicit instantiation uses <| and |> instead of < and > to avoid ambiguities with the less-than and greater-than operators. In addition, great care was taken to separate type expressions and normal expression in the grammar, thereby avoiding ambiguity between the < and > used for parameterized classes and the less-than and greater-than operators.

The translation of $\mathcal{G}$ to C++ is accomplished in two stages. The first stage performs type checking, translating polymorphic functions to monomorphic functions and models to dictionaries. These tasks are combined in a single stage because they are interdependent. The second stage lowers function expressions to function objects.

5.4. Summary

This chapter defined $\mathcal{G}$ by a translation to C++. The main technique is translating `where` clauses to extra dictionary parameters that contains operations implementing the requirements of the concepts. The basic idea is similar to the standard compilation strategy for type classes in Haskell, though here the target language is C++. As such, concepts are mapped to abstract base classes and models are mapped to objects of derived classes.

The proof of the pudding is in the eating.
Miguel de Cervantes Saavedra, *Don Quixote* [16]

6

Case studies: generic libraries in $\mathcal{G}$

This chapter evaluates the design of $\mathcal{G}$ with respect to two case studies: prototype implementations of the STL and the Boost Graph Library [169]. The STL case study was reported in [173] and the BGL study is new. The STL and BGL are large generic libraries that exercise a wide range of language features. Both libraries exhibit considerable internal reuse and the BGL makes heavy use of the STL, so these prototypes stress the language features that support the development and use of generic libraries. The approach taken with the STL prototype was to copy the algorithm implementations from the GNU C++ Standard Library, fixing the syntax here and there, and then to write the `where` clause for each algorithm based on the specifications in the C++ Standard and in *Generic Programming and the STL* by Austern [11]. The type system of $\mathcal{G}$ proved its worth during this process: several bugs were

found in the C++ Standard's specification and in the GNU implementation of the STL. Model definitions were a useful form of first test for data structure implementations. At a model definition, the compiler checks that the implementation matches the expected interfaces. Further, the experience of using the generic libraries was much improved compared to C++. Error messages due to misuse of the library were shorter and more accurate and compile times were shorter due to separate compilation. A couple of challenges were encountered while implementing the STL in $\mathcal{G}$. The first challenge concerned algorithm dispatching, and we developed an idiom to accomplish this in $\mathcal{G}$, but there is still room for improvement. The second challenge concerned code reuse within the STL data structures. It seems that a separate *generative* mechanism is needed to complement the generic features of $\mathcal{G}$. As a temporary solution, we used the m4 macro system to factor the common code.

6.1. The Standard Template Library

In this section we analyze the interdependence of the language features of $\mathcal{G}$ and generic library design in light of implementing the STL. A primary goal of generic programming is to express algorithms with minimal assumptions about data abstractions, so we first look at how the polymorphic functions of $\mathcal{G}$ can be used to accomplish this. Another goal of generic programming is efficiency, so we investigate the use of function overloading in $\mathcal{G}$ to accomplish automatic algorithm selection. We conclude this section with a brief look at implementing generic containers and adaptors in $\mathcal{G}$.

6.1.1. Algorithms. Figure 1 depicts a few simple STL algorithms implemented using polymorphic functions in $\mathcal{G}$. The STL provides two versions of most algorithms, such as the overloads for `find` in Figure 1. The first version is higher-order, taking a predicate function as its third parameter while the second version relies on `operator==`. The higher-order version is more general but for many uses the second version is more convenient. Functions are first-class in $\mathcal{G}$, so the higher-order version is straightforward to express: a function type is used for the third parameter. As is typical in the STL, there is a high-degree of internal reuse: `remove` uses `remove_copy` and `find`.

FIGURE 1. Some STL Algorithms in $\mathcal{G}$.

```
fun find<Iter> where { InputIterator<Iter> }
(Iter@ first, Iter last,
 fun(InputIterator<Iter>.value)->bool@ pred) -> Iter@ {
  while (first != last and not pred(*first)) ++first;
  return first;
}
fun find<Iter> where { InputIterator<Iter>,
    EqualityComparable<InputIterator<Iter>.value> }
(Iter@ first, Iter last, InputIterator<Iter>.value value) -> Iter@ {
  while (first != last and not (*first == value)) ++first;
  return first;
}
fun remove<Iter> where { MutableForwardIterator<Iter>,
    EqualityComparable<InputIterator<Iter>.value> }
(Iter@ first, Iter last, InputIterator<Iter>.value value) -> Iter@ {
  first = find(first, last, value);
  let i = @Iter(first);
  return first == last ? first : remove_copy(++i, last, first, value);
}
```

At the time of finishing this thesis, we have not yet implemented all of the algorithms in the STL, but we have implemented a significant portion, including several of the more involved algorithms such as `stable_sort`. The following is the list of algorithms implemented at this time: `min`, `max`, `swap`, `iter_swap`, `copy`, `copy_backward`, `advance`, `distance`, `for_each`, `find`, `search`, `find_end`, `adjacent_find`, `count`, `mismatch`, `search_n`, `equal`, `max_element`, `min_element`, `fill`, `fill_n`, `swap_ranges`, `reverse`, `rotate`, `replace_copy`, `remove`, `remove_copy`, `merge`, `merge_backward`, `lower_bound`, `upper_bound`, `inplace_merge`, `inplace_stable_sort`, `stable_sort`, and `accumulate`.

6.1.2. Iterators. Figure 2 shows the STL iterator hierarchy as represented in $\mathcal{G}$. Required operations are expressed in terms of function signatures, and associated types are expressed with a nested `type` requirement. The refinement hierarchy is established with the `refines` clauses and nested model requirements with `require`. In the previous example, the calls to `find` and `remove_copy` inside `remove` type check because the `MutableForwardIterator`

FIGURE 2. The STL Iterator Concepts in $\mathcal{G}$ (`Iterator` has been abbreviated to `Iter`).

```
concept InputIter<X> {
  type value;
  type difference;
  refines EqualityComparable<X>;
  refines Regular<X>;
  require SignedIntegral<difference>;
  fun operator*(X) -> value@;
  fun operator++(X!) -> X!;
};
concept OutputIter<X,T> {
  refines Regular<X>;
  fun operator<<(X!, T) -> X!;
};
concept ForwardIter<X> {
  refines DefaultConstructible<X>;
  refines InputIter<X>;
  fun operator*(X) -> value;
};
concept MutableForwardIter<X> {
  refines ForwardIter<X>;
  refines OutputIter<X,value>;
  require Regular<value>;
  fun operator*(X) -> value!;
};
```

```
concept BidirectionalIter<X> {
  refines ForwardIter<X>;
  fun operator--(X!) -> X!;
};
concept MutableBidirectionalIter<X> {
  refines BidirectionalIter<X>;
  refines MutableForwardIter<X>;
};
concept RandomAccessIter<X> {
  refines BidirectionalIter<X>;
  refines LessThanComparable<X>;
  fun operator+(X, difference) -> X@;
  fun operator-(X, difference) -> X@;
  fun operator-(X, X) -> difference@;
};
concept MutableRandomAccessIter<X> {
  refines RandomAccessIter<X>;
  refines MutableBidirectionalIter<X>;
};
```

concept refines `InputIterator` and `OutputIterator`. There are no examples of nested same-type requirements in the iterator concepts, but the STL Container concept includes such constraints. Semantic invariants and complexity guarantees are not expressible in $\mathcal{G}$: they are beyond the scope of its type system.

6.1.3. Automatic algorithm selection. To realize the generic programming efficiency goals, $\mathcal{G}$ provides mechanisms for automatic algorithm selection. The following code shows two overloads for `copy`. (We omit the third overload to save space.) The first version is for input iterators and the second for random access iterators. The second version uses an integer counter for the loop thereby allowing some compilers to better optimize the loop. The two signatures are the same except for the `where` clause.

```
fun copy<Iter1,Iter2> where { InputIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
(Iter1@ first, Iter1 last, Iter2@ result) -> Iter2@ {
  for (; first != last; ++first) result << *first;
  return result;
}
fun copy<Iter1,Iter2> where { RandomAccessIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
(Iter1@ first, Iter1 last, Iter2@ result) -> Iter2@ {
  for (n = last - first; n > zero(); --n, ++first) result << *first;
  return result;
}
```

The use of dispatching algorithms such as `copy` inside other generic algorithms is challenging because overload resolution is based on the proxy models in the `where` clause and not on the models defined for the instantiating type arguments. (This rule is needed to enable separate type checking and compilation.) Thus, a call to an overloaded function such as `copy` may resolve to a non-optimal overload. Consider the following implementation of `merge`. The `Iter1` and `Iter2` types are required to model `InputIterator` and the body of `merge` contains two calls to `copy`.

```
fun merge<Iter1,Iter2,Iter3>
where { InputIterator<Iter1>, InputIterator<Iter2>,
        LessThanComparable<InputIterator<Iter1>.value>,
        InputIterator<Iter1>.value == InputIterator<Iter2>.value,
        OutputIterator<Iter3, InputIterator<Iter1>.value> }
(Iter1@ first1, Iter1 last1, Iter2@ first2, Iter2 last2, Iter3@ result)
   -> Iter3@ {
  ...
  return copy(first2, last2, copy(first1, last1, result));
}
```

The `merge` function always calls the slow version of `copy`, even though the actual iterators may be random access. In C++, with tag dispatching, the fast version of `copy` is called because the overload resolution occurs after template instantiation. However, C++ does not provide separate type checking for templates.

To enable dispatching for `copy` the information available at the instantiation of `merge` must be carried into the body of `merge` (suppose it is instantiated with a random access

iterator). This can be accomplished using a combination of concept and model declarations. First, define a concept with a single operation that corresponds to the algorithm.

```
concept CopyRange<I1,I2> {
  fun copy_range(I1,I1,I2) -> I2@;
};
```

Next, add a requirement for this concept to the type requirements of merge and replace the calls to copy with the concept operation copy_range.

```
fun merge<Iter1,Iter2,Iter3>
where { ..., CopyRange<Iter2,Iter3>, CopyRange<Iter1,Iter3> }
(Iter1@ first1, Iter1 last1, Iter2@ first2, Iter2 last2, Iter3@ result)
   -> Iter3@ { ...
  return copy_range(first2, last2, copy_range(first1, last1, result));
}
```

The last part of the this idiom is to create parameterized model declarations for CopyRange. The where clauses of the model definitions match the where clauses of the respective overloads for copy. In the body of each copy_range there is a call to copy which resolves to the appropriate overload.

```
model <Iter1,Iter2> where { InputIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
CopyRange<Iter1,Iter2> {
  fun copy_range(Iter1 first, Iter1 last, Iter2 result) -> Iter2@
    { return copy(first, last, result); }
};
model <Iter1,Iter2> where { RandomAccessIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
CopyRange<Iter1,Iter2> {
  fun copy_range(Iter1 first, Iter1 last, Iter2 result) -> Iter2@
    { return copy(first, last, result); }
};
```

A call to merge with a random access iterator uses the second model to satisfy the requirement for CopyRange. Thus, when copy_range is invoked inside merge, the fast version of copy is called. A nice property of this idiom is that calls to generic algorithms need not change. A disadvantage of this idiom is that the interface of the generic algorithms becomes more complex.

FIGURE 3. Excerpt from a doubly-linked list container in $\mathcal{G}$.

```
struct list_node<T> where { Regular<T>, DefaultConstructible<T> } {
  list_node<T>* next; list_node<T>* prev; T data;
};
class list<T> where { Regular<T>, DefaultConstructible<T> } {
  list() : n(new list_node<T>()) { n->next = n; n->prev = n; }
  ~list() { ... }
  list_node<T>* n;
};
class list_iterator<T> where { Regular<T>, DefaultConstructible<T> } {
  ... list_node<T>* node;
};
fun operator*<T> where { Regular<T>, DefaultConstructible<T> }
(list_iterator<T> x) -> T { return x.node->data; }

fun operator++<T> where { Regular<T>, DefaultConstructible<T> }
(list_iterator<T>! x) -> list_iterator<T>!
    { x.node = x.node->next; return x; }

fun begin<T> where { Regular<T>, DefaultConstructible<T> }
(list<T> l) -> list_iterator<T>@
    { return @list_iterator<T>(l.n->next); }

fun end<T> where { Regular<T>, DefaultConstructible<T> }
(list<T> l) -> list_iterator<T>@ { return @list_iterator<T>(l.n); }
```

6.1.4. Containers. The containers of the STL are implemented in $\mathcal{G}$ using polymorphic types. Figure 3 shows an excerpt of the doubly-linked `list` container in $\mathcal{G}$. As usual, a dummy sentinel node is used in the implementation. With each STL container comes iterator types that translate between the uniform iterator interface and data structure specific operations. Figure 3 shows the `list_iterator` which translates `operator*` to `x.node->data` and `operator++` to `x.node = x.node->next`.

Not shown in Figure 3 is the implementation of the mutable iterator for `list` (the `list_iterator` provides read-only access). The definitions of the two iterator types are nearly identical, the only difference is that `operator*` returns by read-only reference for the constant iterator whereas it returns by read-write reference for the mutable iterator. The

code for these two iterators should be reused but $\mathcal{G}$ does not yet have a language mechanism for this kind of reuse.

In C++ this kind of reuse can be expressed using the Curiously Recurring Template Pattern (CRTP) and by parameterizing the base iterator class on the return type of `operator*`. This approach can not be used in $\mathcal{G}$ because the parameter passing mode may not be parameterized. Further, the semantics of polymorphism in $\mathcal{G}$ does not match the intended use here, we want to *generate* code for the two iterator types at library construction time. A separate *generative* mechanism is needed to compliment the generic features of $\mathcal{G}$. Similar limitations in the ability to express reuse in terms of generics are discussed in [17], where they suggest using the XVCL meta-programming system [202] to capture reuse. As a temporary solution we used the m4 macro system to factor the common code from the iterators. The following is an excerpt from the implementation of the iterator operators.

```
define(‘forward_iter_ops’,
‘fun operator*<T> where { Regular<T>, DefaultConstructible<T> }
($1<T> x) -> T $2 { return x.node->data; } ...’)
forward_iter_ops(list_iterator, &) /* read-only */
forward_iter_ops(mutable_list_iter, !) /* read-write */
```

At the time of finishing this thesis, the STL implementation in $\mathcal{G}$ includes the doubly-linked `list` class, a singly-linked `slist` class, and the `vector` class. The `map`, `set`, `multimap`, and `multiset` containers have not yet been implemented, but look to be straightforward to implement in $\mathcal{G}$.

6.1.5. Adaptors. The `reverse_iterator` class is a representative example of an STL adaptor.

```
class reverse_iterator<Iter>
  where { Regular<Iter>, DefaultConstructible<Iter> } {
  reverse_iterator(Iter base) : curr(base) { }
  reverse_iterator(reverse_iterator<Iter> other) : curr(other.curr) { }
  Iter curr;
};
```

The `Regular` requirement on the underlying iterator is needed for the copy constructor and `DefaultConstructible` for the default constructor. This adaptor flips the direction of

traversal of the underlying iterator, which is accomplished with the following `operator*` and `operator++`. There is a call to `operator--` on the underlying `Iter` type so `BidirectionalIterator` is required.

```
fun operator*<Iter> where { BidirectionalIterator<Iter> }
(reverse_iterator<Iter> r) -> BidirectionalIterator<Iter>.value
    { let tmp = @Iter(r.curr); return *--tmp; }

fun operator++<Iter> where { BidirectionalIterator<Iter> }
(reverse_iterator<Iter>! r) -> reverse_iterator<Iter>!
    { --r.curr; return r; }
```

Polymorphic model definitions are used to establish that `reverse_iterator` is a model of the iterator concepts. The following says that `reverse_iterator` is a model of `InputIterator` whenever the underlying iterator is a model of `BidirectionalIterator`.

```
model <Iter> where { BidirectionalIterator<Iter> }
InputIterator< reverse_iterator<Iter> > {
  type value = BidirectionalIterator<Iter>.value;
  type difference = BidirectionalIterator<Iter>.difference;
};
```

6.1.6. Function expressions. Most STL implementations implement two separate versions of `find_subsequence`, one written in terms of `operator==` and the in terms of a function object. The version using `operator==` could be written in terms of the one that takes a function object, but it is not written that way. The original reason for this was to improve efficiency, but with with a modern optimizing compiler there should be no difference in efficiency: all that is needed to erase the difference is some simple inlining. The $\mathcal{G}$ implementation we write the `operator==` version of `find_subsequence` in terms of the higher-order version. The following code shows how this is done and is a bit more complicated than we would have liked.

```
fun find_subsequence<Iter1,Iter2>
where { ForwardIterator<Iter1>, ForwardIterator<Iter2>,
        ForwardIterator<Iter1>.value == ForwardIterator<Iter2>.value,
        EqualityComparable<ForwardIterator<Iter1>.value> }
(Iter1 first1, Iter1 last1, Iter2 first2, Iter2 last2) -> Iter1@
{
```

```
  type T = ForwardIterator<Iter1>.value;
  let cmp = model EqualityComparable<T>.operator==;
  return find_subsequence(first1, last1, first2, last2,
                          fun(T a,T b) c=cmp: c(a, b));
}
```

It would have been simpler to write the function expression as

```
fun(T a, T b): a == b
```

However, this is an error in $\mathcal{G}$ because the `operator==` from the `EqualityComparable<..>` requirement is a local name, not a global one, and is therefore not in scope for the body of the function expression. The workaround is to store the comparison function as a data member of the function object. The expression

```
model EqualityComparable<T>.operator==
```

accesses the `operator==` member from the model of `EqualityComparable` for type `T`.

Examples such as these are a convincing argument that lexical scoping should be allowed in function expressions, and the next generation of $\mathcal{G}$ will support this feature.

6.1.7. Improved error messages. In Section 2.2.1.4 we showed an example of a hard to understand error message that resulted from a misuse of the STL `stable_sort` algorithm. The following code is the translation of that example to $\mathcal{G}$.

```
4  fun main() -> int@{
5    let v = @list<int>();
6    stable_sort(begin(v), end(v));
7    return 0;
8  }
```

The $\mathcal{G}$ compiler prints the following error message which is much shorter and easier to understand.

```
test/stable_sort_error.g:6:
In application stable_sort(begin(v), end(v)),
Model MutableRandomAccessIterator<mutable_list_iter<int>>
needed to satisfy requirement, but it is not defined.
```

6.1.8. Improved error detection. Another problem that plagues generic C++ libraries is that type errors often go unnoticed during library development. The errors go unnoticed because the type checking of template definitions is delayed until instantiation. A related problem is that the documented type requirements for a template may not be consistent with the implementation, which can result in unexpected compiler errors for the user.

These problems are directly addressed in $\mathcal{G}$: the implementation of a generic function is type-checked with respect to its `where` clause. Verifying that there are no type errors in a generic function and that the type requirements are consistent is trivial in G: the compiler does not accept generic functions invoked with inconsistent types.

Interestingly, while implementing the STL in $\mathcal{G}$, the type checker caught several errors in the STL as defined in C++. One such error was in `replace_copy`. The implementation below was translated directly from the GNU C++ Standard Library, with the `where` clause matching the requirements for `replace_copy` in the C++ Standard [86].

```
196 fun replace_copy<Iter1,Iter2, T>
197 where { InputIterator<Iter1>, Regular<T>, EqualityComparable<T>,
198         OutputIterator<Iter2, InputIterator<Iter1>.value>,
199         OutputIterator<Iter2, T>,
200         EqualityComparable2<InputIterator<Iter1>.value,T> }
201 (Iter1@ first, Iter1 last, Iter2@ result, T old, T neu) -> Iter2@ {
202   for ( ; first != last; ++first)
203     result << *first == old ? neu : *first;
204   return result;
205 }
```

The $\mathcal{G}$ compiler gives the following error message:

```
stl/sequence_mutation.g:203:
The two branches of the conditional expression must have the
same type or one must be coercible to the other.
```

This is a subtle bug, which explains why it has gone unnoticed for so long. The type requirements say that both the value type of the iterator and `T` must be writable to the output iterator, but the requirements do not say that the value type and `T` are the same type, or coercible to one another.

6.2. The Boost Graph Library

A group of us at the Open Systems Lab performed a comparative study of language support for generic programming [69]. We evaluated a half dozen modern programming languages by implementing a subset of the Boost Graph Library [169] in each language. We implemented a family of algorithms associated with breadth-first search, including Dijkstra's single-source shortest paths [52] and Prim's minimum spanning tree algorithms [153]. This section extends the previous study to include $\mathcal{G}$. We give a brief overview of the BGL, describe the implementation of the BGL in $\mathcal{G}$, and compare the results to those in our earlier study [69].

6.2.1. An overview of the BGL graph search algorithms. Figure 4 depicts some graph search algorithms from the BGL, their relationships, and how they are parameterized. Each large box represents an algorithm and the attached small boxes represent type parameters. An arrow labeled `<uses>` from one algorithm to another specifies that one algorithm is implemented using the other. An arrow labeled `<models>` from a type parameter to an unboxed name specifies that the type parameter must model that concept. For example, the breadth-first search algorithm has three type parameters: `G`, `C`, and `Vis`. Each of these has requirements: `G` must model the Vertex List Graph and Incidence Graph concepts, `C` must model the Read/Write Map concept, and `Vis` must model the BFS Visitor concept. The breadth-first search algorithm is implemented using the graph search algorithm.

The core algorithm of this library is graph search, which traverses a graph and performs user-defined operations at certain points in the search. The order in which vertices are visited is controlled by a type argument, `B`, that models the Bag concept. This concept abstracts a data structure with insert and remove operations but no requirements on the order in which items are removed. When `B` is bound to a FIFO queue, the traversal order is breadth-first. When it is bound to a priority queue based on distance to a source vertex, the order is closest-first, as in Dijkstra's single-source shortest paths algorithm. Graph search is also parameterized on actions to take at event points during the search, such as when a vertex is first discovered. This parameter, `Vis`, must model the Visitor concept (which is not

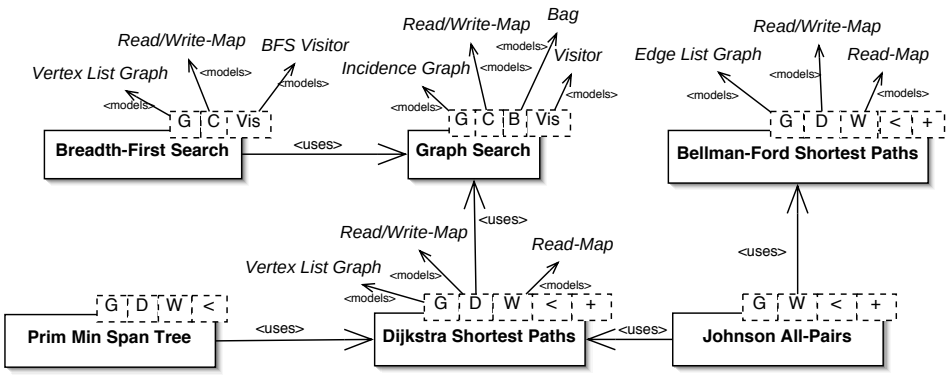

FIGURE 4. Graph algorithm parameterization and reuse within the Boost Graph Library. Arrows for redundant models relationships are not shown. For example, the type parameter `G` of breadth-first search must also model Incidence Graph because breadth-first search uses graph search.

to be confused with the Visitor design pattern). The graph search algorithm also takes a type parameter `C` for mapping each vertex to a color and `C` must model the Read/Write Map concept. The colors are used as markers to keep track of the progression of the algorithm through the graph.

The Read Map and Read/Write Map concepts represent variants of an important abstraction in the graph library: the ***property map***. In practice, graphs represent domain-specific entities. For example, a graph might depict the layout of a communication network, its vertices representing endpoints and its edges representing direct links. In addition to the number of vertices and the edges between them, a graph may associate values to its elements. Each vertex of a communication network graph might have a name and each edge a maximum transmission rate. Some algorithms require access to domain information associated with the graph representation. For example, Prim's minimum spanning tree algorithm requires "weight" information associated with each edge in a graph. Property maps provide a convenient implementation-agnostic means of expressing, to algorithms, relations between graph elements and domain-specific data. Some graph data structures directly contain associated values with each node; others use external associative data structures to implement these relationships. Interfaces based on property maps work equally well with both representations.

The graph algorithms are all parameterized on the graph type. Breadth-first search takes a type parameter `G`, which must model two concepts, Incidence Graph and Vertex List Graph. The Incidence Graph concept defines an interface for accessing out-edges of a vertex. Vertex List Graph specifies an interface for accessing the vertices of a graph in an unspecified order. The Bellman-Ford shortest paths algorithm [18] requires a model of the Edge List Graph concept, which provides access to all the edges of a graph.

That graph capabilities are partitioned among three concepts illustrates generic programming's emphasis on minimal algorithm requirements. The Bellman-Ford shortest paths algorithm requires of a graph only the operations described by the Edge List Graph concept. Breadth-first search, in contrast, requires the functionality of two separate concepts. By partitioning the functionality of graphs, each algorithm can be used with any data type that meets its minimum requirements. If the three fine-grained graph concepts were replaced with one monolithic concept, each algorithm would require more from its graph type parameter than necessary and would thus unnecessarily restrict the set of types with which it could be used.

The graph library design is suitable for evaluating generic programming capabilities of languages because its implementation involves a rich variety of generic programming techniques. Most of the algorithms are implemented using other library algorithms: breadth-first search and Dijkstra's shortest paths use graph search, Prim's minimum spanning tree algorithm uses Dijkstra's algorithm, and Johnson's all-pairs shortest paths algorithm [94] uses both Dijkstra's and Bellman-Ford shortest paths. Furthermore, type parameters for some algorithms, such as the `G` parameter to breadth-first search, must model multiple concepts. In addition, the algorithms require certain relationships between type parameters. For example, consider the graph search algorithm. The `C` type argument, as a model of Read/Write Map, is required to have an associated key type. The `G` type argument is required to have an associated vertex type. Graph search requires that these two types be the same.

As in our earlier study, we focus the evaluation on the interface of the breadth-first search algorithm and the infrastructure surrounding it, including concept definitions and an example use of the algorithm.

6.2.2. Implementation in $\mathcal{G}$. So far we have implemented breadth-first search and Dijkstra's single-source shortest paths in $\mathcal{G}$. This required defining several of the graph and property map concepts and an implementation of the `adjacency_list` class, a FIFO queue, and a priority queue.

The interface for the breadth-first search algorithm is straightforward to express in $\mathcal{G}$. It has three type parameters: the graph type `G`, the color map type `C`, and the visitor type `Vis`. The requirements on the type parameters are expressed with a `where` clause, using concepts that we describe below. In the interface of `breadth_first_search`, associated types and same-type constraints play an important role in accurately tracking the relationships between the graph type, its vertex descriptor type, and the color property map.

```
type Color = int;
let black = 0;
let gray  = 1;
let white = 2;

fun breadth_first_search<G, C, Vis>
  where { IncidenceGraph<G>, VertexListGraph<G>,
          ReadWritePropertyMap<C>,
          PropertyMap<C>.key == IncidenceGraph<G>.vertex_descriptor,
          PropertyMap<C>.value == Color,
          BFSVisitor<Vis,G> }
(G g, IncidenceGraph<G>.vertex_descriptor@ s, C c, Vis vis) { /* ... */ }
```

Figure 5 shows the definition of several graph concepts in $\mathcal{G}$. The `Graph` concept requires the associated types `vertex_descriptor` and `edge_descriptor` and some basic functionality for those types such as copy construction and equality comparison. This concept also includes the `source` and `target` functions. The `Graph` concept serves to factor common requirements out of the `IncidenceGraph` and `VertexListGraph` concepts.

The `IncidenceGraph` concept introduces the capability to access out-edges of a vertex. The access is provided by the `out_edge_iterator` associated type. The requirements for the out-edge iterator are slightly more than the standard `InputIterator` concept and slightly less than the `ForwardIterator` concept. The out-edge iterator must allow for multiple passes but dereferencing an out-edge iterator need not return a reference (for example, it

may return by-value instead). Thus we define the following new concept to express these requirements.

```
concept MultiPassIterator<Iter> {
    refines DefaultConstructible<Iter>;
    refines InputIterator<Iter>;
    // semantic requirement: allow multiple passes through the range
};
```

In Figure 5, the `IncidenceGraph` concept uses same-type constraints to require that the `value` type of the iterator to be the same type as the `edge_descriptor`. The `VertexListGraph` concepts adds the capability of traversing all the vertices in the graph using the associated `vertex_iterator`.

Figure 6 shows the implementation of a graph in terms of a vector of singly-linked lists. Vertex descriptors are integers and edge descriptors are pairs of integers. The out-edge iterator is implemented with the `vg_out_edge_iter` class whose implementation is shown in Figure 7. The basic idea behind this iterator is to provide a different view of the list of target vertices, making it appear as a list of source-target pairs.

The property map concepts are defined in Figure 8. The `ReadWritePropertyMap` is a refinement of the `ReadablePropertyMap` concept, which requires the `get` function, and the `WritablePropertyMap` concept, which requires the `put` function. Both of these concepts refine the `PropertyMap` concept which includes the associated `key` and `value` types.

Figure 9 shows the definition of the `BFSVisitor` concept. This concept is naturally expressed as a multi-parameter concept because the visitor and graph types are independent: a particular visitor may be used with many different concrete graph types and vice versa. The use of `refines` for `Graph` in `BFSVisitor` is somewhat odd, `require` would be more natural, but the refinement provides direct (and convenient) access to the vertex and edge descriptor types. An alternative would be use to `require` and some type aliases, but type aliases have not yet been added to concept definitions.

Figure 10 presents an example use of the `breadth_first_search` function to output vertices in breadth-first order. To do so, the `test_vis` visitor overrides the function `discover_vertex`; empty implementations of the other visitor functions are provided by `default_bfs_visitor`.

FIGURE 5. Graph concepts in $\mathcal{G}$.

```
concept Graph<G> {
    type vertex_descriptor;
    require DefaultConstructible<vertex_descriptor>;
    require Regular<vertex_descriptor>;
    require EqualityComparable<vertex_descriptor>;

    type edge_descriptor;
    require DefaultConstructible<edge_descriptor>;
    require Regular<edge_descriptor>;
    require EqualityComparable<edge_descriptor>;

    fun source(edge_descriptor, G) -> vertex_descriptor@;
    fun target(edge_descriptor, G) -> vertex_descriptor@;
};
concept IncidenceGraph<G> {
    refines Graph<G>;

    type out_edge_iterator;
    require MultiPassIterator<out_edge_iterator>;
    edge_descriptor == InputIterator<out_edge_iterator>.value;

    fun out_edges(vertex_descriptor, G)
        -> pair<out_edge_iterator, out_edge_iterator>@;
    fun out_degree(vertex_descriptor, G) -> int@;
};
concept VertexListGraph<G> {
    refines Graph<G>;

    type vertex_iterator;
    require MultiPassIterator<vertex_iterator>;
    vertex_descriptor == InputIterator<vertex_iterator>.value;

    fun vertices(G) -> pair<vertex_iterator, vertex_iterator>@;
    fun num_vertices(G) -> int@;
};
```

A graph is constructed using the `AdjacencyList` class, and then `breadth_first_search` is called.

FIGURE 6. Implementation of a graph with a vector of lists.

```
fun source(pair<int,int> e, vector< slist<int> >) -> int@ { return e.first; }
fun target(pair<int,int> e, vector< slist<int> >) -> int@ { return e.second; }

model Graph< vector< slist<int> > > {
  type vertex_descriptor = int;
  type edge_descriptor = pair<int,int>;
};

fun out_edges(int src, vector< slist<int> > G)
    -> pair<vg_out_edge_iter, vg_out_edge_iter>@ {
  return make_pair(@vg_out_edge_iter(src, begin(G[src])),
                   @vg_out_edge_iter(src, end(G[src])));
}
fun out_degree(int src, vector< slist<int> > G) -> int@ { return size(G[src]); }

model IncidenceGraph< vector< slist<int> > > {
  type out_edge_iterator = vg_out_edge_iter;
};

fun vertices(vector< slist<int> > G) -> pair<counting_iter,counting_iter>@
    { return make_pair(@counting_iter(0), @counting_iter(size(G))); }
fun num_vertices(vector< slist<int> > G) -> int@ { return size(G); }

model VertexListGraph< vector< slist<int> > > {
  type vertices_size_type = int;
  type vertex_iterator = counting_iter;
};
```

6.3. Summary

This chapter evaluated the design of $\mathcal{G}$ with respect to implementing representative portions of the STL and the BGL. The evaluation showed that implementing generic algorithms in $\mathcal{G}$ is straightforward. The `concept` and `where` clause features of $\mathcal{G}$ enable the direct expression of the ideas of generic programming. The use of generic libraries is made easier by the improvement in error messages and the development of generic algorithms is aided by separate type checking. With respect to building generic containers, $\mathcal{G}$ provides some support with its parameterized classes, but the kind of code reuse typical of inheritance or

FIGURE 7. Out-edge iterator for the vector of lists.

```
class vg_out_edge_iter {
  vg_out_edge_iter() { }
  vg_out_edge_iter(int src, slist_iterator<int> iter) : src(src), iter(iter) { }
  vg_out_edge_iter(vg_out_edge_iter x) : iter(x.iter), src(x.src) { }
  slist_iterator<int> iter;
  int src;
};
fun operator=(vg_out_edge_iter! me, vg_out_edge_iter other) -> vg_out_edge_iter!
    { me.iter = other.iter; me.src = other.src; return me; }
model DefaultConstructible<vg_out_edge_iter> { };
model Regular<vg_out_edge_iter> { };

fun operator==(vg_out_edge_iter x, vg_out_edge_iter y) -> bool@
    { return x.iter == y.iter; }
fun operator!=(vg_out_edge_iter x, vg_out_edge_iter y) -> bool@
    { return x.iter != y.iter; }
model EqualityComparable<vg_out_edge_iter> { };

fun operator*(vg_out_edge_iter x) -> pair<int,int>@
    { return make_pair(x.src, *x.iter); }
fun operator++(vg_out_edge_iter! x) -> vg_out_edge_iter!
    { ++x.iter; return x; }
model InputIterator<vg_out_edge_iter> {
  type value = pair<int,int>;
  type difference = ptrdiff_t;
};
model MultiPassIterator<vg_out_edge_iter> { };
```

mixins is not easily expressible in $\mathcal{G}$, so such language features would make a good addition to $\mathcal{G}$.

FIGURE 8. Property map concepts in $\mathcal{G}$.

```
concept PropertyMap<Map> {
    type key;
    type value;
};
concept ReadablePropertyMap<Map> {
    refines PropertyMap<Map>;
    fun get(Map, key) -> value;
};
concept WritablePropertyMap<Map> {
    refines PropertyMap<Map>;
    fun put(Map, key, value);
};
concept ReadWritePropertyMap<Map>  {
    refines ReadablePropertyMap<Map>;
    refines WritablePropertyMap<Map>;
};
```

FIGURE 9. Breadth-first search visitor concept.

```
concept BFSVisitor<Vis, G> {
    refines Regular<Vis>;
    refines Graph<G>;

    fun initialize_vertex(Vis v, vertex_descriptor d, G g) {}
    fun discover_vertex(Vis v, vertex_descriptor d, G g) {}
    fun examine_vertex(Vis v, vertex_descriptor d, G g) {}
    fun examine_edge(Vis v, edge_descriptor d, G g) {}
    fun tree_edge(Vis v, edge_descriptor d, G g) {}
    fun non_tree_edge(Vis v, edge_descriptor d, G g) {}
    fun gray_target(Vis v, edge_descriptor d, G g) {}
    fun black_target(Vis v, edge_descriptor d, G g) {}
    fun finish_vertex(Vis v, vertex_descriptor d, G g) {}
};
```

FIGURE 10. Example use of the BFS generic function.

```
struct test_vis { };
fun discover_vertex<G>(test_vis, int v, G g) { printf("%d ", v); }

model <G> where { Graph<G>,  Graph<G>.vertex_descriptor == int }
BFSVisitor<test_vis, G> { };

fun main() -> int@ {
  let n = 7;
  let g = @vector< slist<int> >(n);
  push_front(1, g[0]); push_front(4, g[0]);
  push_front(2, g[1]); push_front(3, g[1]);
  push_front(4, g[3]); push_front(6, g[3]);
  push_front(5, g[4]);

  let src = 0;
  let color = new Color[n];
  for (let i = 0; i != n; ++i)
    color[i] = white;
  breadth_first_search(g, src, color, @test_vis());
  return 0;
}
```

This type system has been designed to facilitate program verification on a modular basis. The general principle is that a module writer should not have to look outside his module to verify its correctness.

James H. Morris, Jr. [131]

7

Type Safety of $F^{\mathcal{G}}$

Type safety does not hold for $\mathcal{G}$ because $\mathcal{G}$ inherits many type safety holes from C++. For example, a dangling pointer is created when `delete` is invoked on a pointer and the type system does not prevent such a pointer from being dereferenced. Another example is that a stack allocated object may be returned by-reference from a function, thereby creating a dangling reference. There has been considerable research related to type-safe manual memory management. This research includes memory management via regions [81, 188, 197] and using type systems to track aliasing [26, 27, 44, 60]. Memory management is not the focus of this dissertation, so we leave for future work the application of the above research to define a type safe version of $\mathcal{G}$.

FIGURE 1. Types and Terms of System F

$$
\begin{array}{lll}
s, t & \in & \text{Type Variables} \\
x, y, d & \in & \text{Term Variables} \\
n & \in & \mathbb{N} \\
\tau & ::= & t \mid \texttt{fun}\ \overline{\tau} \to \tau \mid \tau \times \cdots \times \tau \mid \forall \overline{t}.\ \tau \\
f & ::= & x \mid f(\overline{f}) \mid \lambda \overline{y : \tau}.\ f \mid \Lambda \overline{t}.\ f \mid f[\overline{\tau}] \\
 & & \mid\ \texttt{let}\ x = f\ \texttt{in}\ f \mid (f, \ldots, f) \mid\ \texttt{nth}\ f\ n
\end{array}
$$

However, we still want to know whether the design for generics presented in this thesis creates holes in the type system, or whether it is sound with respect to type safety. To this end we embed the design for generics in System F [71, 157] to create a calculus named F$^{\mathcal{G}}$. System F is a small language that captures the essence of parametric polymorphism and is a standard tool in programming language research. The semantics of F$^{\mathcal{G}}$ is defined with respect to System F. That is, we define a translation from F$^{\mathcal{G}}$ to System F. This translation parallels the translation of $\mathcal{G}$ to C++. The type safety of F$^{\mathcal{G}}$ is proved by showing that well-typed terms of F$^{\mathcal{G}}$ translate to well-typed terms of System F. Therefore, because System F is type safe, so is F$^{\mathcal{G}}$. The property of type safety is important because when a language is type safe and a program passes type checking, any execution of that program will be guaranteed to be free of type errors. Thus type checking is a useful form of lightweight validation. The presentation of F$^{\mathcal{G}}$ here includes the material from [172] and adds the proof that the translation of F$^{\mathcal{G}}$ with associated types to System F preserves typing.

7.1. F$^{\mathcal{G}}$ = System F + concepts, models, and constraints

System F, the polymorphic lambda calculus, is the prototypical tool for studying type parameterization. The syntax of System F is shown in Figure 1 and the type rules for System F are in Figure 2. The variable f ranges over System F expressions; we reserve e for System F$^{\mathcal{G}}$ expressions. We use an over-bar, such as $\overline{\tau}$, to denote repetition: $\tau_1, \ldots, \tau_n$. We use mult-parameter functions and type abstractions in System F to ease the translation from F$^{\mathcal{G}}$ to F. We also include a `let` expression.

FIGURE 2. Type rules and well-formed types for System F

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

$$\text{(TABS)}\ \frac{\text{distinct } \overline{t} \quad \overline{t} \cap \text{FTV}(\Gamma) = \emptyset \quad \Gamma, \overline{t} \vdash f : \tau}{\Gamma \vdash \Lambda \overline{t}.\ f : \forall \overline{t}.\ \tau} \qquad \text{(TAPP)}\ \frac{\Gamma \vdash \overline{\sigma} \quad \Gamma \vdash f : \forall \overline{t}.\ \tau}{\Gamma \vdash f[\overline{\sigma}] : [\overline{t} \mapsto \overline{\sigma}]\tau}$$

$$\text{(VAR)}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{(ABS)}\ \frac{\Gamma, \overline{x : \sigma} \vdash f : \tau \quad \Gamma \vdash \overline{\sigma}}{\Gamma \vdash \lambda \overline{x : \sigma}.\ f : \texttt{fun}\ \overline{\sigma} \to \tau}$$

$$\text{(APP)}\ \frac{\Gamma \vdash f_1 : \texttt{fun}\ \overline{\sigma} \to \tau \quad \Gamma \vdash \overline{f_2 : \sigma}}{\Gamma \vdash f_1(\overline{f_2}) : \tau} \qquad \text{(LET)}\ \frac{\Sigma \vdash f_1 : \sigma \quad \Sigma, x : \sigma \vdash f_2 : \tau}{\Sigma \vdash \texttt{let}\ x = f_1\ \texttt{in}\ f_2 : \tau}$$

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

$$\frac{t \in \Gamma}{\Gamma \vdash t} \qquad \frac{\Gamma \vdash \overline{\tau} \quad \Gamma \vdash \tau}{\Gamma \vdash \texttt{fun}\ \overline{\tau} \to \tau} \qquad \frac{\Gamma \vdash \tau_1 \quad \cdots \quad \Gamma \vdash \tau_n}{\Gamma \vdash \tau_1 \times \cdots \times \tau_n} \qquad \frac{\text{distinct } \overline{t} \quad \Gamma, \overline{t} \vdash \tau}{\Gamma \vdash \forall \overline{t}.\ \tau}$$

FIGURE 3. Higher Order Sum in System F

```
let sum =
  (Λ t.
    fix (λ sum : fun(list t, fun(t,t)→t, t)→t.
         λ ls : list t, add : fun(t,t)→t, zero : t.
         if null[t](ls) then zero
         else add(car[t](ls), sum(cdr[t](ls), add, zero)))) in

let ls = cons[int](1, cons[int](2, nil[int])) in
sum[int](ls, iadd, 0)
```

It is possible to write generic algorithms in System F, as is demonstrated in Figure 3, which shows the implementation of a polymorphic `sum` function. The function is written in higher-order style, passing the type-specific `add` and `zero` as parameters. However, this approach does not scale: algorithms of any interest typically require dozens of type-specific operations.

FIGURE 4. Types and Terms of F$^{\mathcal{G}}$

$$
\begin{array}{lcl}
c & \in & \text{Concept Names} \\
s, t & \in & \text{Type Variables} \\
x, y, z & \in & \text{Term Variables} \\
\rho, \sigma, \tau & ::= & t \mid \texttt{fun}\ (\overline{\tau}) \to \tau \mid \forall \overline{t}\ \texttt{where}\ \overline{c\texttt{<}\overline{\sigma}\texttt{>}}.\ \tau \\
e & ::= & x \mid e(\overline{e}) \mid \lambda y : \tau.\ e \\
 & \mid & \Lambda \overline{t}\ \texttt{where}\ \overline{c\texttt{<}\overline{\sigma}\texttt{>}}.\ e \mid e[\overline{\tau}] \\
 & \mid & \texttt{concept}\ c\texttt{<}\overline{t}\texttt{>}\{\texttt{refines}\ \overline{c\texttt{<}\overline{\sigma}\texttt{>}};\ \overline{x : \tau};\}\ \texttt{in}\ e \\
 & \mid & \texttt{model}\ c\texttt{<}\overline{\tau}\texttt{>}\ \{\overline{x = e};\}\ \texttt{in}\ e \\
 & \mid & c\texttt{<}\overline{\tau}\texttt{>}.x
\end{array}
$$

7.1.1. Adding concepts, models, and constraints. F$^{\mathcal{G}}$ adds concepts, models, and `where` clauses to System F. These three features provide the core support for generic programming in $\mathcal{G}$. Figure 4 shows the abstract syntax of the basic formulation of F$^{\mathcal{G}}$. Associated types and same-type constraints are added to F$^{\mathcal{G}}$ in Section 7.4. While the core features of $\mathcal{G}$ are present in F$^{\mathcal{G}}$, are are several aspects of the generics of $\mathcal{G}$ that are left out for the sake of simplicity.

Function overloading: is not present in F$^{\mathcal{G}}$. Formalizing function overloading is straightforward but complicated and the kind of static overload resolution present in $\mathcal{G}$ poses no problems for type safety. For example, Java has static overload resolution and is type safe.

Parameterized models: are not present in F$^{\mathcal{G}}$. The presence of parameterized models in $\mathcal{G}$ makes its type system undecidable because the model lookup algorithm becomes much more powerful and is not guaranteed to terminate (see Section 4.6.2 for details). However, the type soundness property is unaffected: if a program type checks (and all the right models are found) then execution is still guaranteed to be free of type errors.

Implicit instantiation: is not present in F$^{\mathcal{G}}$. $\mathcal{G}$ uses the same approach to implicit instantiation as ML$^{\mathrm{F}}$ [24], and that approach was already proved to be type sound and decidable.

To illustrate the features of F$^{\mathcal{G}}$, we evolve the `sum` function defined above. To be generic, the `sum` function should work for any element type that supports addition, so we capture this requirement in a concept. As in Section 2.1 we define Semigroup and Monoid concepts as follows.

```
concept Semigroup<t> {
  binary_op : fun(t,t)→t;
} in
concept Monoid<t> {
  refines Semigroup<t>;
  identity_elt : t;
} in ...
```

As with System F, F$^{\mathcal{G}}$ is an expression-oriented programming language. These `concept` definitions are like `let`: they add to the lexical environment for the enclosed expression (after the `in`).

The following code declares `int` to be a model of Semigroup and Monoid, using integer addition for the binary operation and `0` for the identity element. The type system of F$^{\mathcal{G}}$ checks the body of the model against the concept definition to ensure all required operations are provided and that there are model declarations in scope for each refinement.

```
model Semigroup<int> {
  binary_op = iadd;
}
model Monoid<int> {
  identity_elt = 0;
}
```

A model is found via the concept name and type, and members of the model are extracted with the dot operator. For example, the following returns the `iadd` function.

```
Monoid<int>.binary_op
```

With the `Monoid` concept defined, we are ready to write a generic `sum` function. The function is generalized to work with any type that has an associative binary operation with an identity element (no longer necessarily addition), so a more appropriate name for this function is `accumulate`. As in System F, type parameterization in $F^{\mathcal{G}}$ is provided by the Λ expression. $F^{\mathcal{G}}$ adds a `where` clause to the Λ expression for listing requirements.

```
let accumulate = (Λ t where Monoid<t>. /*body*/)
```

The concepts, models, and `where` clauses collaborate to provide a mechanism for implicitly passing operations into a generic function. As in System F, a generic function is instantiated by providing type arguments for each type parameter.

```
accumulate[int]
```

In System F, instantiation substitutes `int` for `t` in the body of the Λ. In $F^{\mathcal{G}}$, instantiation also involves the following steps:

(1) `int` is substituted for `t` in the `where` clause.
(2) For each requirement in the `where` clause, the lexical scope of the instantiation is searched for a matching model declaration.
(3) The models are implicitly passed into the generic function.

Consider the body of the `accumulate` function listed below. The model requirements in the `where` clause serve as proxies for actual model declarations. Thus, the body of `accumulate` is type-checked as if there were a model declaration `model Monoid<t>` in the enclosing scope. The dot operator is used inside the body to access the binary operator and identity element of the `Monoid`.

```
let accumulate =
  (Λ t where Monoid<t>.
    fix (λ accum : fun(list t)→ t.
         λ ls : list t.
         let binary_op = Monoid<t>.binary_op in
         let identity_elt = Monoid<t>.identity_elt in
```

```
if null[t](ls) then identity_elt
else binary_op(car[t](ls), accum(cdr[t](ls)))))
```

It would be more convenient to write `binary_op` instead of the explicit member access: `Monoid<t>.binary_op`. However, such a statement could be ambiguous without the incorporation of overloading. For example, suppose that a generic function has two type parameters, `s` and `t`, and requires each to be a `Monoid`. Then a call to `binary_op` might refer to either `Monoid<s>.binary_op` or `Monoid<t>.binary_op`. While the convenience of function overloading is important, we did not wish to complicate F$^{\mathcal{G}}$ with this additional feature. Function overloading is present in the full language $\mathcal{G}$. Function overloading in $\mathcal{G}$ is described in Section 4.7 and an algorithm for overload resolution is defined in Section 5.2.3.

The complete program for this example is in Figure 5.

7.1.2. Lexically scoped models and model overlapping. The lexical scoping of models declarations is an important feature of F$^{\mathcal{G}}$, and one that distinguishes it from Haskell. We illustrate this distinction with an example. There are multiple ways for the set of integers to model Monoid besides addition with the zero identity element. For example, in F$^{\mathcal{G}}$, the Monoid consisting of integers with multiplication for the binary operation and `1` for the identity element would be declared as follows.

```
model Semigroup<int> {
  binary_op = imult;
}
model Monoid<int> {
  identity_elt = 1;
}
```

Borrowing from Haskell terminology, this creates overlapping model declarations, since there are now two model declarations for the `Semigroup<int>` and `Monoid<int>` concepts. Overlapping model declarations are problematic since they introduce ambiguity: when

FIGURE 5. Generic Accumulate

```
concept Semigroup<t> {
  binary_op : fun(t,t)→t;
} in
concept Monoid<t> {
  refines Semigroup<t>;
  identity_elt : t;
} in

let accumulate =
  (Λ t where Monoid<t>.
    fix (λ accum : fun(list t)→ t.
         λ ls : list t.
         let binary_op = Monoid<t>.binary_op in
         let identity_elt = Monoid<t>.identity_elt in
         if null[t](ls) then identity_elt
         else binary_op(car[t](ls), accum(cdr[t](ls))))) in


model Semigroup<int> {
  binary_op = iadd;
} in
model Monoid<int> {
  identity_elt = 0;
} in

let ls = cons[int](1, cons[int](2, nil[int])) in
accumulate[int](ls)
```

`accumulate` is instantiated, which model (with its corresponding binary operation and identity element) should be used?

In F$^{\mathcal{G}}$, overlapping models declarations may co-exist if they appear in separate lexical scopes. In Figure 6 we create `sum` and `product` functions by instantiating `accumulate` in the presence of different model declarations. This example would not type check in Haskell, even if the two instance declarations were to be placed in different modules, because instance declarations implicitly leak out of a module when anything in the module is used by another module.

FIGURE 6. Intentionally overlapping models.

```
let sum =
  model Semigroup<int> {
    binary_op = iadd;
  } in
  model Monoid<int> {
    identity_elt = 0;
  } in accumulate[int] in

let product =
  model Semigroup<int> {
    binary_op = imult;
  } in
  model Monoid<int> {
    identity_elt = 1;
  } in accumulate[int] in

let ls = cons[int](1, cons[int](2, nil[int])) in
(sum(ls), product(ls))
```

7.2. Translation of F$^{\mathcal{G}}$ to System F

We describe a translation from F$^{\mathcal{G}}$ to System F similar to the type-directed translation of Haskell type classes presented in [78]. The translation described here is intentionally simple; its purpose is to communicate the semantics of F$^{\mathcal{G}}$ and to aid in the proof of type safety. We show that the translation from F$^{\mathcal{G}}$ to System F preserves typing, which together with the fact that System F is type safe [151], ensures the type safety of F$^{\mathcal{G}}$. The main idea behind the translation is to represent models with dictionaries that map member names to values, and to pass these dictionaries as extra arguments to generic functions. Here, we use tuples to represent dictionaries. Thus, the model declarations for `Semigroup<int>` and `Monoid<int>` translate to a pair of `let` expressions that bind freshly generated dictionary names to the dictionaries (tuples) for the models. We show a diagram of the dictionary representation of these models in Figure 7 and we show the translation to System F below.

```
model Semigroup<int> {
  binary_op = iadd;
```

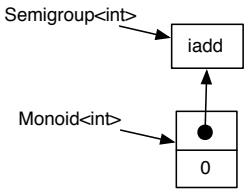

FIGURE 7. Dictionaries for `Semigroup<int>` and `Monoid<int>`.

```
} in
model Monoid<int> {
  identity_elt = 0;
} in /* rest */
==>
let Semigroup_61 = (iadd) in
let Monoid_67 = (Semigroup_61,0) in /* rest */
```

The `accumulate` function is translated by removing the `where` clause and wrapping the body in a λ expression with a parameter for each model requirement in the `where` clause.

```
let accumulate = (Λ t where Monoid<t>. /*body*/)
==>
let accumulate =
  (Λ t. (λ Monoid_18:(fn(t,t)→t)*t. /* body */)
```

The accumulate function is now curried, first taking a dictionary argument and then taking the normal arguments.

```
accumulate[int](ls)
==>
((accumulate[int])(Monoid_67))(ls)
```

In the body of `accumulate` there are model member accesses. These are translated into tuple member accesses.

```
let binary_op = Monoid<t>.binary_op in
```

```
let identity_elt = Monoid<t>.identity_elt in
==>
let binary_op = (nth (nth Monoid_18 0) 0) in
let identity_elt = (nth Monoid_18 1) in
```

The formal translation rules are in Figure 9. We write $[t \mapsto \sigma]\tau$ for the capture avoiding substitution of σ for t in τ. We write $[\overline{t} \mapsto \overline{\sigma}]\tau$ for simultaneous substitution. The function FTV returns the set of free type variables and CV returns the concept names occurring in the `where` clauses within a type. We write $\mathrm{distinct}\ \overline{t}$ to mean that each item in the list appears at most once. We subscript a nested tuple type with a non-empty sequence of natural numbers to mean the following:

$$(\tau_1 \times \ldots \times \tau_k)_i = \tau_i$$

$$(\tau_1 \times \ldots \times \tau_k)_{i,\overline{n}} = (\tau_i)_{\overline{n}}$$

The environment Γ consists of four parts: 1) the usual type assignment for variables, 2) the set of type variables currently in scope, 3) information about concepts and their corresponding dictionary types, and 4) information about models, including the identifier and path to the corresponding dictionary in the translation.

The (MEM) rule uses the auxiliary function $\flat(c, \overline{\rho}, \overline{n}, \Gamma)$ to obtain a set of concept members together with their types and the paths (sequences of natural numbers) to the members through the dictionary. A path instead of a single index is necessary because dictionaries may be nested due to concept refinement.

$\flat(c, \overline{\rho}, \overline{n}, \Gamma) =$

$M := \emptyset$

for $i = 0, \ldots, |\overline{c'}| - 1$

$M := M \cup \flat(c'_i, [\overline{t} \mapsto \overline{\rho}]\overline{\rho'}_i, (\overline{n}, i), \Gamma)$

for $i = 0, \ldots, |\overline{x}| - 1$

$M := M \cup \{x_i : ([\overline{t} \mapsto \overline{\rho}]\sigma_i, (\overline{n}, |\overline{c'}| + i))\}$

return M

where concept c<$\overline{t}$>{refines $\overline{c'<\overline{\rho'}>}$; $\overline{x : \sigma}$; } $\mapsto \delta \in \Gamma$

The (TABS) rule uses the auxiliary function $\flat^w$ to collect proxy model definitions from the where clause of a type abstraction and also computes the dictionary type for each requirement. The function $\flat^m$, defined below, is applied to each concept requirement.

$\flat^w([], \Gamma) = (\Gamma, [])$
$\flat^w((c<\overline{\rho}>, \overline{c'<\overline{\rho'}>}), \Gamma) =$
  generate fresh d
  $(\Gamma, \delta) := \flat^m(c, \overline{\rho}, d, [], \Gamma)$
  $(\Gamma, \overline{\delta'}) := \flat^w(\overline{c'<[\overline{t} \mapsto \overline{\rho}]\overline{\rho'}>}, \Gamma)$
  return $(\Gamma, (\delta, \overline{\delta'}))$
where concept c<$\overline{t}$>{refines $\overline{c'<\overline{\rho'}>}$; $\overline{x : \sigma}$; } $\mapsto \delta \in \Gamma$

The function $\flat^m(c, \overline{\rho}, d, \overline{n}, \Gamma)$ collects the model definitions and dictionary type for the model c<$\overline{\rho}$>. The model information inserted into the environment includes a dictionary name d and a path $\overline{n}$ that gives the location inside d for the dictionary of $c(\overline{\tau})$.

$\flat^m(c, \overline{\rho}, d, \overline{n}, \Gamma) =$
  check $\Gamma \vdash \overline{\rho} \rightsquigarrow -$
  $\overline{\tau} := []$
  for $i = 0, \ldots, |\overline{c'}| - 1$
    $(\Gamma, \delta') := \flat^m(c'_i, [\overline{t} \mapsto \overline{\rho}]\overline{\rho'}_i, d, (\overline{n}, i), \Gamma)$
    $\overline{\tau} := \overline{\tau}, \delta'$
  $\overline{\tau} := \overline{\tau}@[\overline{t} \mapsto \overline{\rho}]\overline{\sigma}$
  $\Gamma := \Gamma, (\text{model } c<\overline{\rho}> \mapsto (d, \overline{n}))$
  return $(\Gamma, \overline{\tau})$
where concept c<$\overline{t}$>{refines $\overline{c'<\overline{\rho'}>}$; $\overline{x : \sigma}$; } $\mapsto \delta \in \Gamma$

Figure 8 defines the translation from F$^{\mathcal{G}}$ types to System F types.

FIGURE 8. Well-formedness of F$^{\mathcal{G}}$ types and translation to System F types.

$$\boxed{\Gamma \vdash \tau \rightsquigarrow \tau'}$$

$$\text{(TyVar)}\ \frac{t \in \Gamma}{\Gamma \vdash t \rightsquigarrow t}$$

$$\text{(TyAbs)}\ \frac{\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \qquad \Gamma \vdash \tau \rightsquigarrow \tau'}{\Gamma \vdash \texttt{fun}\ \overline{\sigma} \to \tau \rightsquigarrow \texttt{fun}\ \overline{\sigma'} \to \tau'}$$

$$\text{(TyTAbs)}\ \frac{(\Gamma', \overline{\delta}) = \flat^w(\overline{c\texttt{<}\rho\texttt{>}}, (\Gamma, \overline{t})) \qquad \Gamma' \vdash \tau \rightsquigarrow \tau'}{\Gamma \vdash \forall \overline{t}\ \texttt{where}\ \overline{c\texttt{<}\rho\texttt{>}}.\ \tau \rightsquigarrow \forall \overline{t}.\texttt{fun}\ \overline{\delta} \to \tau'}$$

We now come to our main result for this section: translation produces well typed terms of System F, or more precisely, if $\Gamma \vdash e : \tau \rightsquigarrow f$ and Σ is a System F environment corresponding to Γ, then there exists some type τ' such that $\Sigma \vdash f : \tau'$. Figure 10 defines what we mean by correspondence between an F$^{\mathcal{G}}$ environment and System F environment.

Several lemmas are used in the theorem. The proofs of these lemmas are omitted here but appear in a technical report [171]. The technical report formalizes the lemmas and theorem in the Isar proof language [143] and the Isabelle proof assistant [144] was used to validate the proofs. We give an overview of that formalization in Section 7.3.

The first lemma relates the type of a model member returned by the $\flat$ function to the member type in the dictionary for the model given by the $\flat^m$.

LEMMA 1.

If $(x : (\tau, \overline{n'})) \in \flat(c, \overline{\rho}, \overline{n}, \Gamma)$ *and* $(-, \delta_{\overline{n}}) = \flat^m(c, \overline{\rho}, -, -, \Gamma)$

then $\Gamma \vdash \tau \rightsquigarrow \delta_{\overline{n'}}$

The next lemma states that the type of the dictionaries in the environment match the concept's dictionary type δ. The purpose of the sequence $\overline{n}$ is to map from the dictionary d for a "derived" concept to the nested tuple for the "super" concept c.

FIGURE 9. Type Rules for F$^{\mathcal{G}}$ and Translation to System F

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

$$\text{(CPT)}\ \frac{\begin{array}{c}\text{distinct } \bar{t} \qquad (\Gamma', -) = \flat^w(\overline{c'\texttt{<}\overline{\rho}\texttt{>}}, (\Gamma, \bar{t})) \\ \Gamma' \vdash \overline{\tau} \rightsquigarrow \overline{\tau'} \qquad \delta = (\overline{[\bar{t}' \mapsto \overline{\rho'}]\delta'})@\overline{\tau'} \\ \Gamma, (\texttt{concept } c\texttt{<}\bar{t}\texttt{>}\{\texttt{refines } \overline{c'\texttt{<}\overline{\rho}\texttt{>}};\ \overline{x : \tau};\} \mapsto \delta) \vdash e : \tau \rightsquigarrow f \qquad c \notin \mathrm{CV}(\tau)\end{array}}{\Gamma \vdash \texttt{concept } c\texttt{<}\bar{t}\texttt{>}\{\texttt{refines } \overline{c'\texttt{<}\overline{\rho}\texttt{>}};\ \overline{x : \tau};\} \texttt{ in } e : \tau \rightsquigarrow f}$$

$$\text{(MDL)}\ \frac{\begin{array}{c}\overline{\texttt{concept } c\texttt{<}\bar{t}\texttt{>}\{\texttt{refines } \overline{c'\texttt{<}\overline{\rho'}\texttt{>}};\ \overline{x : \tau};\} \mapsto \delta} \in \Gamma \qquad \Gamma \vdash \overline{\rho} \rightsquigarrow \overline{\tau'} \qquad \Gamma \vdash \overline{e : \sigma} \rightsquigarrow \overline{f} \\ \overline{\texttt{model } c'\texttt{<}[\bar{t} \mapsto \overline{\rho}]\overline{\rho'}\texttt{>} \mapsto (d', \overline{n})} \in \Gamma \qquad \overline{x : [\bar{t} \mapsto \overline{\rho}]\tau} \subseteq \overline{y : \sigma} \qquad d \text{ fresh} \\ \Gamma, (\texttt{model } c\texttt{<}\overline{\rho}\texttt{>} \mapsto (d, [])) \vdash e : \tau \rightsquigarrow f \qquad \overline{d''} = \overline{(\texttt{ nth } \ldots (\texttt{ nth } d' \ n_1) \ldots n_k)}\end{array}}{\Gamma \vdash \texttt{model } c\texttt{<}\overline{\rho}\texttt{>}\ \{\overline{y = e};\} \texttt{ in } e : \tau \rightsquigarrow \texttt{let } d = (\overline{d''}@[\overline{y} \mapsto \overline{f}]\overline{x}) \texttt{ in } f}$$

$$\text{(TABS)}\ \frac{\text{distinct } \bar{t} \qquad \bar{t} \cap \mathrm{FTV}(\Gamma) = \emptyset \qquad (\Gamma', \overline{\delta}) = \flat^w(\overline{c\texttt{<}\overline{\rho}\texttt{>}}, (\Gamma, \bar{t})) \qquad \Gamma' \vdash e : \tau \rightsquigarrow f}{\Gamma \vdash \Lambda\bar{t} \texttt{ where } \overline{c\texttt{<}\overline{\rho}\texttt{>}}.\ e : \forall\bar{t} \texttt{ where } \overline{c\texttt{<}\overline{\rho}\texttt{>}}.\ \tau \rightsquigarrow \Lambda\bar{t}.\ \lambda\overline{d : \delta}.\ f}$$

$$\text{(TAPP)}\ \frac{\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \qquad \Gamma \vdash e : \forall\bar{t} \texttt{ where } \overline{c\texttt{<}\overline{\rho}\texttt{>}}.\ \tau \rightsquigarrow f \qquad \overline{\texttt{model } c\texttt{<}[\bar{t} \mapsto \overline{\sigma}]\overline{\rho}\texttt{>} \mapsto (d, \overline{n})} \in \Gamma}{\Gamma \vdash e[\overline{\sigma}] : [\bar{t} \mapsto \overline{\sigma}]\tau \rightsquigarrow f[\overline{\sigma'}](\overline{\texttt{ nth } \ldots (\texttt{ nth } d \ n_1) \ldots n_k})}$$

$$\text{(MEM)}\ \frac{\Gamma \vdash \overline{\rho} \rightsquigarrow \overline{\rho'} \qquad (\texttt{model } c\texttt{<}\overline{\rho}\texttt{>} \mapsto (d, \overline{n})) \in \Gamma \qquad (x : (\tau, \overline{n'})) \in \flat(c, \overline{\rho}, \overline{n}, \Gamma)}{\Gamma \vdash c\texttt{<}\overline{\rho}\texttt{>}.x : \tau \rightsquigarrow (\texttt{ nth } \ldots (\texttt{ nth } d \ n'_1) \ldots n'_k)}$$

$$\text{(VAR)}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow x} \qquad \text{(ABS)}\ \frac{\Gamma, \overline{x : \sigma} \vdash e : \tau \rightsquigarrow f \qquad \Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'}}{\Gamma \vdash \lambda\overline{x : \sigma}.\ e : \texttt{fun } \overline{\sigma} \to \tau \rightsquigarrow \lambda\overline{x : \sigma'}.\ f}$$

$$\text{(APP)}\ \frac{\Gamma \vdash e_1 : \texttt{fun } \overline{\sigma} \to \tau \rightsquigarrow f_1 \qquad \Gamma \vdash \overline{e_2 : \sigma} \rightsquigarrow \overline{f_2}}{\Gamma \vdash e_1(\overline{e_2}) : \tau \rightsquigarrow f_1(\overline{f_2})}$$

LEMMA 2.

If $(\texttt{model } c\texttt{<}\overline{\tau}\texttt{>} \mapsto (d, \overline{n})) \in \Gamma$ *and* $\Gamma \rightsquigarrow \Sigma$ *and* $(-, \delta) = \flat^m(c, \overline{\tau}, -, -, \Gamma)$

then $\Sigma \vdash (\texttt{ nth } \ldots (\texttt{ nth } d \ n_1) \ldots n_k) : \delta$

FIGURE 10. Well-formed F$^{\mathcal{G}}$ environment in correspondence with a System F environment.

$$\boxed{\Gamma \rightsquigarrow \Sigma}$$

$$\frac{}{\emptyset \rightsquigarrow \emptyset} \qquad \frac{\Gamma \rightsquigarrow \Sigma \quad \Gamma \vdash \tau \rightsquigarrow \tau'}{\Gamma, x : \tau \rightsquigarrow \Sigma, x : \tau'} \qquad \frac{\Gamma \rightsquigarrow \Sigma}{\Gamma, t \rightsquigarrow \Sigma, t}$$

$$\frac{\Gamma \rightsquigarrow \Sigma \quad (-, \delta) = \flat^m(c, \overline{\tau}, -, -, \Gamma)}{\Gamma, (\texttt{model}\ c\texttt{<}\overline{\tau}\texttt{>} \mapsto (d, [])) \rightsquigarrow \Sigma, d : \delta}$$

$$\frac{\Gamma \rightsquigarrow \Sigma \quad 0 < |\overline{n}| \quad d : \delta \in \Sigma \quad (-, \delta_{\overline{n}}) = \flat^m(c, \overline{\tau}, -, -, \Gamma)}{\Gamma, (\texttt{model}\ c\texttt{<}\overline{\tau}\texttt{>} \mapsto (d, \overline{n})) \rightsquigarrow \Sigma}$$

$$\frac{\Gamma \rightsquigarrow \Sigma \quad (\Gamma', \overline{\delta'}) = \flat^w(\overline{c'\texttt{<}\overline{\tau}\texttt{>}}, (\Gamma, \overline{t})) \quad \Gamma' \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'}}{\Gamma, (\texttt{concept}\ c\texttt{<}\overline{t}\texttt{>}\{\texttt{refines}\ \overline{c'\texttt{<}\overline{\tau}\texttt{>}};\ \overline{x : \sigma};\} \mapsto \overline{\delta'}@\overline{\sigma'}) \rightsquigarrow \Sigma}$$

The following lemma states that extending the F$^{\mathcal{G}}$ environment with proxy models from a `where` clause, and extending the System F environment with $\overline{d} : \overline{\delta}$, preserves the environment correspondence.

LEMMA 3.

If $\Gamma \rightsquigarrow \Sigma$ *and* $(\Gamma', \overline{\delta}) = \flat^w(\overline{c\texttt{<}\overline{\rho}\texttt{>}}, \Gamma)$ *then* $\Gamma' \rightsquigarrow \Sigma, \overline{d} : \overline{\delta}$

We now state and prove that the translation preserves well typing.

THEOREM 1 (Translation preserves well typed programs).

If $\Gamma \vdash e : \tau \rightsquigarrow f$ *and* $\Gamma \rightsquigarrow \Sigma$ *then there exists* τ' *such that* $\Sigma \vdash f : \tau'$ *and* $\Gamma \vdash \tau \rightsquigarrow \tau'$

PROOF. (of Theorem 1) The proof is by induction on the derivation of $\Gamma \vdash e : \tau \rightsquigarrow f$.

Cpt: Let $\Gamma' = \Gamma, \texttt{concept}\ c\texttt{<}\overline{t}\texttt{>}\{\texttt{refines}\ \overline{c'\texttt{<}\overline{\nu}\texttt{>}};\ \overline{x : \tau};\}$. By inversion we have:

(2) $\overline{\texttt{concept}\ c'\texttt{<}\overline{t'}\texttt{>}\{\ldots\} \mapsto \delta} \in \Gamma$

(3) $\Gamma, \overline{t} \vdash \overline{\tau} \rightsquigarrow \overline{\tau'}$

(4) $\Gamma' \vdash e : \tau \rightsquigarrow f$

(5) $c \notin \mathrm{CV}(\tau)$

From the assumption $\Gamma \rightsquigarrow \Sigma$ and from (2) and (3) we have $\Gamma' \rightsquigarrow \Sigma$. Then by (4) and the induction hypothesis we have $\Sigma \vdash f : \tau'$ and $\Gamma' \vdash \tau \rightsquigarrow \tau'$. Then from (5) we have $\Gamma \vdash \tau \rightsquigarrow \tau'$.

Mdl: Let $\Gamma' = \Gamma, (\texttt{model}\ c\texttt{<}\overline{\rho}\texttt{>}) \mapsto (d, [])$. We have the following by inversion:

(6) $$\Gamma \vdash \overline{e : \sigma} \rightsquigarrow \overline{f}$$

(7) $$\overline{\texttt{model}\ c'\texttt{<}[\overline{t} \mapsto \overline{\rho}]\overline{\rho'}\texttt{>} \mapsto (d', \overline{n'})} \subseteq \Gamma$$

(8) $$\overline{x : [\overline{t} \mapsto \overline{\rho}]\tau} \subseteq \overline{y : \sigma}$$

(9) $$\Gamma' \vdash e : \tau \rightsquigarrow f$$

(10) $$\texttt{concept}\ c\texttt{<}\overline{t}\texttt{>}\{\texttt{refines}\ \overline{c'\texttt{<}\overline{\rho'}\texttt{>}};\ \overline{x : \tau};\} \mapsto \delta \in \Gamma$$

Let Σ such that $\Gamma \rightsquigarrow \Sigma$. With (6) and the induction hypothesis there exists σ' such that $\Sigma \vdash \overline{f} : \overline{\sigma'}$ and $\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'}$. Next, let

$$\overline{r} = \overline{(\ \texttt{nth}\ \ldots (\ \texttt{nth}\ d'\ n'_1) \ldots n'_k)}$$

From $\Gamma \rightsquigarrow \Sigma$ and (10) we have $(-, \overline{\delta'}) = \flat^w(\overline{c'\texttt{<}\overline{\rho'}\texttt{>}}, \Gamma)$. and therefore $(-, [\overline{t} \mapsto \overline{\rho}]\overline{\delta'}) = \flat^w(\overline{c'\texttt{<}[\overline{t} \mapsto \overline{\rho}]\overline{\rho'}\texttt{>}}, \Gamma)$. Together with (7) and Lemma 2 we have $\Sigma \vdash \overline{r} : [\overline{t} \mapsto \overline{\rho}]\overline{\delta'}$. With (8) we have a well typed dictionary:

(11) $$\Sigma \vdash (\overline{r}@[\overline{y} \mapsto \overline{f}]\overline{x}) : \delta$$

Let Σ' be $\Sigma, d : \delta$ so $\Gamma' \rightsquigarrow \Sigma'$. Then with (9) and the induction hypothesis there exists τ' such that $\Sigma' \vdash f : \tau'$ and $\Gamma' \vdash \tau \rightsquigarrow \tau'$. From (11) we show $\Sigma \vdash \texttt{let}\ d = (\overline{r}@[\overline{y} \mapsto \overline{f}]\overline{x})\ \texttt{in}\ f : \tau'$.

TAbs: By inversion we have:

(12) $$(\Gamma', \overline{\delta}) = \flat^w(\overline{c\texttt{<}\overline{\rho}\texttt{>}}, (\Gamma, \overline{t}))$$

(13) $$\Gamma', \overline{t}, \overline{M} \vdash e : \tau \rightsquigarrow f$$

From the assumption $\Gamma \rightsquigarrow \Sigma$ we have $\Gamma, \overline{t} \rightsquigarrow \Sigma, \overline{t}$. Then with (12) we apply Lemma 3 to get $\Gamma' \rightsquigarrow \Sigma, \overline{t}, \overline{d} : \overline{\delta}$. We then apply the induction hypothesis with (13), so there exists τ' such that $\Sigma, \overline{t}, \overline{d} : \overline{\delta} \vdash f : \tau'$ and $\Gamma' \vdash \tau \rightsquigarrow \tau'$. Hence we have

$\Sigma, \overline{t} \vdash \lambda \overline{d} : \overline{\delta}.\ f : \texttt{fun}\ \overline{\delta} \to \tau'$ and therefore $\Sigma \vdash \Lambda \overline{t}.\ \lambda \overline{d} : \overline{\delta}.\ f : \forall \overline{t}.\texttt{fun}\ \overline{\delta} \to \tau'$. Also, from $\Gamma' \vdash \tau \rightsquigarrow \tau'$ we have $\Gamma, \overline{t} \vdash \tau \rightsquigarrow \tau'$. Then with (12) we have $\Gamma \vdash \forall \overline{t}\ \texttt{where}\ \overline{c\texttt{<}\rho\texttt{>}}.\ \tau \rightsquigarrow \forall \overline{t}.\texttt{fun}\ \overline{\delta} \to \tau'$.

TApp: By inversion of the (TAPP) rule we have:

$$\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \tag{14}$$

$$\Gamma \vdash e : \forall \overline{t}.\ \texttt{where}\ \overline{c\texttt{<}\rho\texttt{>}}.\ \tau \rightsquigarrow f \tag{15}$$

$$\overline{\texttt{model}\ c\texttt{<}[\overline{t} \mapsto \overline{\sigma}]\overline{\rho}\texttt{>} \mapsto (d, \overline{n})} \in \Gamma \tag{16}$$

From (15) and the induction hypothesis there exists τ' such that $\Sigma \vdash f : \tau'$ and $\Gamma \vdash \forall \overline{t}\ \texttt{where}\ \overline{c\texttt{<}\rho\texttt{>}}.\ \tau \rightsquigarrow \tau'$. By inversion there exists $\overline{\delta}$, τ'', and Γ' such that

$$\tau' = \forall \overline{t}.\ \texttt{fun}\ \overline{\delta} \to \tau'' \tag{17}$$

$$(\Gamma', \overline{\delta}) = \flat^w(\overline{c\texttt{<}\rho\texttt{>}}, (\Gamma, \overline{t})) \tag{18}$$

$$\Gamma' \vdash \tau \rightsquigarrow \tau'' \tag{19}$$

Using (17) we have

$$\Sigma \vdash f[\overline{\sigma'}] : [\overline{t} \mapsto \overline{\sigma'}](\texttt{fun}\ \overline{\delta} \to \tau'') \tag{20}$$

From (18) and (14) we have

$$(\Gamma', [\overline{t} \mapsto \overline{\sigma'}]\overline{\delta}) = \flat^w(\overline{c\texttt{<}[\overline{t} \mapsto \overline{\sigma}]\overline{\rho}\texttt{>}}, \Gamma)) \tag{21}$$

Let $\overline{d'} = \overline{(\ \texttt{nth}\ \ldots (\ \texttt{nth}\ d\ n_1) \ldots n_k)}$. From the assumption $\Gamma \rightsquigarrow \Sigma$, (16), and (21) we apply Lemma 2 to get $\Sigma \vdash \overline{d'} : [\overline{t} \mapsto \overline{\sigma'}]\overline{\delta}$. Then with (20) we have $\Sigma \vdash f[\overline{\sigma'}](\overline{d'}) : [\overline{t} \mapsto \overline{\sigma}]\tau''$ and from (14) and (19) we have $\Gamma \vdash [\overline{t} \mapsto \overline{\sigma}]\tau \rightsquigarrow [\overline{t} \mapsto \overline{\sigma'}]\tau''$.

Mem: By inversion we have

$$(\texttt{model}\ c\texttt{<}\overline{\tau}\texttt{>} \mapsto (d, \overline{n})) \in \Gamma \tag{22}$$

$$x : (\tau, \overline{n'}) \in \flat(c, \overline{\tau}, \overline{n}, \Gamma) \tag{23}$$

From the assumption $\Gamma \rightsquigarrow \Sigma$ and (22), we have the following by inversion.

$$(d : \delta) \in \Sigma \tag{24}$$

$$(-, \delta_{\overline{n}}) = \flat^m(c, \overline{\tau}, -, -, \Gamma) \tag{25}$$

From (24) we have $\Sigma \vdash d : \delta$ and with (23) we show

$$\Sigma \vdash (\ \texttt{nth}\ \ldots (\ \texttt{nth}\ d\ n'_1) \ldots n'_k) : \delta_{\overline{n'}}$$

From (23), (25), and Lemma 1 we have $\Gamma \vdash \tau \rightsquigarrow \delta_{\overline{n'}}$.

Var: By inversion we have $x : \tau \in \Gamma$. Then from $\Gamma \rightsquigarrow \Sigma$ there exists τ' such that $\Gamma \vdash \tau \rightsquigarrow \tau'$ and $x : \tau' \in \Sigma$. Thus $\Sigma \vdash x : \tau'$.

Abs: By inversion we have $\Gamma, \overline{x : \sigma} \vdash e : \tau \rightsquigarrow f$ and $\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'}$. With $\Gamma \rightsquigarrow \Sigma$ we have $\Gamma, \overline{x : \sigma} \rightsquigarrow \Sigma, \overline{x : \sigma'}$ and then from the induction hypothesis there exists τ' such that $\Sigma, \overline{x : \sigma'} \vdash f : \tau'$ and $\Gamma \vdash \tau \rightsquigarrow \tau'$. So $\Sigma \vdash \lambda \overline{x : \sigma'}.\ f : \texttt{fun}\ \overline{\sigma'} \to \tau'$ and $\Gamma \vdash \texttt{fun}\ \overline{\sigma} \to \tau \rightsquigarrow \texttt{fun}\ \overline{\sigma'} \to \tau'$.

App: By inversion there exists $\overline{\sigma}$ such that $\Gamma \vdash e_1 : \texttt{fun}\ \overline{\sigma} \to \tau \rightsquigarrow f_1$ and $\Gamma \vdash \overline{e_2} : \overline{\sigma} \rightsquigarrow \overline{f_2}$. By the induction hypothesis there exists ρ_1 such that $\Sigma \vdash f_1 : \rho_1$ and $\Gamma \vdash \texttt{fun}\ \overline{\sigma} \to \tau \rightsquigarrow \rho_1$. Then by inversion there exists $\overline{\sigma'}$ and τ' such that $\rho_1 = \texttt{fun}\ \overline{\sigma'} \to \tau'$ and $\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'}$ and $\Gamma \vdash \tau \rightsquigarrow \tau'$. Also by the induction hypothesis there exists $\overline{\rho_2}$ such that $\Sigma \vdash \overline{f_2} : \overline{\rho_2}$ and $\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\rho_2}$. Then because type translation is a function, $\overline{\sigma'} = \overline{\rho_2}$ and so $\Gamma \vdash \overline{f_2} : \overline{\sigma'}$. Thus $\Sigma \vdash f_1(\overline{f_2}) : \tau'$.

□

7.3. Isabelle/Isar formalization

Isar [143] is a language for writing proofs and is the language we used to formalize the translation of F$^{\mathcal{G}}$ to System F and the proof of Theorem 1. Figure 11 is simple example of a proof in Isar which shows that the length of the concatenation of two lists is equal to the sum of the lengths of each list. The Isabelle proof assistant [144] can be used to check proofs written in Isar, and the Proof General interface [8] is useful for incrementally developing Isar proofs.

FIGURE 11. Example Isar proof.

```
lemma length_append: ∀ls2. length (ls1@ls2) = length ls1 + length ls2
proof (induct ls1)
  show ∀ls2. length ([] @ ls2) = length [] + length ls2 by simp
next
  fix x xs
  assume IH: ∀ls2. length (xs @ ls2) = length xs + length ls2
  show ∀ls2. length ((x#xs) @ ls2) = length (x#xs) + length ls2
  proof clarify
    fix ls2
    have length ((x#xs) @ ls2) = length (x#(xs@ls2)) by simp
    also have ... = 1 + length (xs@ls2) by simp
    also from IH have ... = 1 + length xs + length ls2 by simp
    ultimately have length ((x#xs) @ ls2) = 1 + length xs + length ls2 by simp
    thus length ((x#xs) @ ls2) = length (x#xs) + length ls2 by simp
  qed
qed
```

The main advantage of the Isabelle/Isar system is that allows for the straightforward modification of large proofs. The majority of other theorem proving systems are tactic based, which means that the proofs are not truly human readable, and even small changes to a proof often require changes to all of the remaining steps in the proof. The development of the proof for F$^{\mathcal{G}}$ was fairly large, the technical report is 70 pages, so it was critical to be able to make incremental changes to the proof.

7.3.0.1. *Induction.* The `length_append` proof is a typical example of performing induction on an inductively defined data type. The first case of the induction handles when `ls1` is the empty list and the second case handles when `ls1=x#xs` for some `x` and `xs`. The induction hypothesis, which is labeled `IH`, says that the proposition holds for `xs`, which we use in the equational reasoning about the length of `(x#xs) @ ls2`. In the proof, the `by` keyword is followed by the rule or tactic used to prove the preceding proposition. The `simp` tactic of Isabelle includes a rewriting engine which among other things will unfold definitions. In this proof it is used to unfold the definition of `@` and `length`.

FIGURE 12. A type system as an inductively defined set.

```
consts well_typed :: ((nat ⇒ stlc_type) × stlc_term × stlc_type) set
inductive well_typed intros
  stlc_var: (Γ, 'x, Γ x) ∈ well_typed
  stlc_app: ⟦ (Γ, e1, τ→τ') ∈ well_typed; (Γ, e2, τ) ∈ well_typed ⟧
              ⟹ (Γ, e1 · e2, τ') ∈ well_typed
  stlc_abs: (Γ(x:=τ), e, τ') ∈ well_typed ⟹ (Γ, λx. e, τ→τ') ∈ well_typed
```

Isabelle also provides a mechanism for inductively defined sets. This facility is useful for defining type systems. For example, the type judgment $\Gamma \vdash e : \tau$ for the simply-typed lambda calculus is encoded as the inductively defined set `well_typed` as shown in Figure 12. As with datatypes, Isabelle provides proof by induction on these inductively defined sets. Theorem 1 is an example of such an induction, as are many of the lemmas.

7.3.0.2. *Variables and substitution.* One of the necessary but annoying aspects of formalizing the type system and semantics of a programming language is handling variables and substitution. De Bruijn indices are a popular choice for representing variables in formal systems, and early on we used them in the formalization of $F^{\mathcal{G}}$. While De Bruijn indices are manageable in the context of the lambda calculus, we found that using them in a more complex language, with both type variables and normal variable, to be quite burdensome, making the resulting proofs much more complex and difficult to reason about. We switched to using naive substitution in combination with the Barendregt convention [14] made explicit. This approach made it straightforward to reason about variables in proofs but it has a couple drawbacks:

- Type equality had to be explicitly formalized to allow for α-conversion, we could not rely on Isabelle's built-in equality. Defining type equality was straightforward but uses of type equality in the proof was more cumbersome. For example, we could no longer rely on Isabelle's equational reasoning.
- To make the Barendregt convention explicit we had to add several extra premises to most lemmas, and the proofs had to be augmented with steps that reason about free variables and sometimes α-rename types or terms. Normally fresh variables

are used when renaming, that is, variables known to be globally unique. However, it is difficult to track global properties in a proof, so instead we generate new variables that are fresh with respect to the types or terms involved. This can be achieved by computing the maximum natural number used as a variable in the types or terms, and then choosing the next larger natural number. In several places in our Isabelle proofs we skip the tedious renaming step and cheat by using Isabelle's `sorry` command, but it should be straightforward to dot all the i's and cross all the t's.

Despite these drawbacks we were satisfied with this approach to variables and substitution.

7.3.0.3. *Evaluation of Isabelle/Isar.* Isabelle/Isar is a big step forward in technology for formalizing programming languages and validating proofs about languages. However, it seems that the difficulty of formalizing proofs in Isabelle/Isar is still greater than it should be, mainly due to user-interface issues. One of the problems is that Isar is built as a thin layer over Isabelle's tactic system, and the layer is transparent, not opaque. A user must understand both systems and be able to switch back and forth between them. Another problem is that when a proof step fails, the error message is rarely helpful in identifying the source of the problem.

7.4. Associated types and same-type constraints

The syntax of $F^{\mathcal{G}}$ with associated types and same-type constraints is given in Figure 13 with the additions highlighted in gray. The syntax for concepts is extended to include requirements for associated types and for type equalities. We add type assignments to model declarations. In addition, where clauses are extended with type equalities.

We have also added an expression for creating type aliases. Type aliases were singled out in [69] as an important feature and the semantics of type aliases is naturally expressed using the type equality infrastructure for same-type constraints.

FIGURE 13. F$^{\mathcal{G}}$ with Associated Types and Same Type Constraints

$$
\begin{array}{rcl}
c & \in & \text{Concept Names} \\
s, t & \in & \text{Type Variables} \\
x, y & \in & \text{Term Variables} \\
\rho, \sigma, \tau & ::= & t \mid \texttt{fun}\ \overline{\tau} \to \tau \mid \forall \overline{t}\ \texttt{where}\ \overline{c\texttt{<}\overline{\sigma}\texttt{>}};\ \overline{\sigma = \tau}\ .\ \tau \\
& \mid & c\texttt{<}\overline{\tau}\texttt{>}.t \\
e & ::= & x \mid e(\overline{e}) \mid \lambda \overline{y : \tau}.\ e \\
& \mid & \Lambda \overline{t}\ \texttt{where}\ \overline{c\texttt{<}\overline{\sigma}\texttt{>}};\ \overline{\sigma = \tau}\ .\ e \mid e[\overline{\tau}] \\
& \mid & \texttt{concept}\ c\texttt{<}\overline{t}\texttt{>}\ \{ \\
& & \quad \texttt{types}\ \overline{s};\ \ \texttt{refines}\ \overline{c\texttt{<}\overline{\sigma}\texttt{>}}; \\
& & \quad \overline{x : \tau};\ \ \overline{\sigma = \tau}; \\
& & \} \ \texttt{in}\ e \\
& \mid & \texttt{model}\ c\texttt{<}\overline{\tau}\texttt{>}\ \{ \\
& & \quad \texttt{types}\ \overline{t = \sigma}; \\
& & \quad \overline{x = e}; \\
& & \} \ \texttt{in}\ e \\
& \mid & c\texttt{<}\overline{\tau}\texttt{>}.x \\
& \mid & \texttt{type}\ t = \tau\ \texttt{in}\ e
\end{array}
$$

Type checking is complicated by the addition of same-type constraints because type equality is no longer syntactic equality: it must take into account the same-type declarations. We extend environments to include type equalities, and introduce a new type equality relation $\Gamma \vdash \sigma = \tau$ which is defined in Figure 14. This relation is the congruence that includes all the type equalities in Γ. Deciding type equality is equivalent to the quantifier free theory of equality with uninterpreted function symbols, for which there is an $O(n \log n)$ average time algorithm [142] ($O(n^2)$ time complexity in the worst case). We prefix operations on sets of types and type assignments with $\Gamma \vdash$ because type equality now depends on the environment Γ.

Figure 17 gives the typing rules for F$^{\mathcal{G}}$ with associated types and same-type constraints and the translation to System F. The (MDL) rule must check that all required associated types are given type assignments and that the same-type requirements of the concept are satisfied. Also, when comparing the model's operations to the operations in the concept, in addition to substituting $\overline{\rho}$ for the concept parameters $\overline{t}$, occurrences of associated types

FIGURE 14. Type equality for F$^{\mathcal{G}}$.

$$\text{(REFL)}\frac{}{\Gamma \vdash \tau = \tau} \quad \text{(SYMM)}\frac{\Gamma \vdash \sigma = \tau}{\Gamma \vdash \tau = \sigma} \quad \text{(TRANS)}\frac{\Gamma \vdash \sigma = \rho \quad \Gamma \vdash \rho = \tau}{\Gamma \vdash \sigma = \tau}$$

$$\text{(HYP)}\frac{\sigma = \tau \in \Gamma}{\Gamma \vdash \sigma = \tau} \quad \text{(FNEQ)}\frac{\Gamma \vdash \overline{\sigma} = \overline{\tau} \quad \Gamma \vdash \sigma = \tau}{\Gamma \vdash \texttt{fun}\ \overline{\sigma} \to \sigma = \texttt{fun}\ \overline{\tau} \to \tau} \quad \text{(ASCEQ)}\frac{\Gamma \vdash \overline{\sigma} = \overline{\tau}}{\Gamma \vdash c\texttt{<}\overline{\sigma}\texttt{>}.t = c\texttt{<}\overline{\tau}\texttt{>}.t}$$

$$\text{(ALLEQ)}\frac{\begin{array}{c}\Gamma \vdash \overline{\rho_1} = [\overline{t_1}/\overline{t_2}]\overline{\rho_2} \quad \Gamma \vdash \overline{\sigma_1} = [\overline{t_1}/\overline{t_2}]\overline{\sigma_2} \quad \Gamma \vdash \overline{\tau_1} = [\overline{t_1}/\overline{t_2}]\overline{\tau_2} \\ \Gamma, \overline{\sigma_1 = \tau_1} \vdash \tau_3 = [\overline{t_1}/\overline{t_2}]\tau_4\end{array}}{\Gamma \vdash \forall \overline{t_1}\ \texttt{where}\ \overline{c\texttt{<}\overline{\rho_1}\texttt{>}}; \overline{\sigma_1 = \tau_1}.\ \tau_3 = \forall \overline{t_2}\ \texttt{where}\ \overline{c\texttt{<}\overline{\rho_2}\texttt{>}}; \overline{\sigma_2 = \tau_2}.\ \tau_4}$$

must be replaced with their type assignments from the body of the model and from models of the concepts c refines. The (TABS) and (TAPP) rules are changed to introduce same-type constraints into the environment and to check same-type constraints respectively. The (APP) rule has been changed from requiring syntactic equality between the parameter and argument types to requiring type equality based on the congruence of the type equalities in the environment. The new rule (ALS) for type aliasing checks the body in an environment extended with a type equality that expresses the aliasing.

The main idea of the translation is to turn associated types into extra type parameters on type abstractions, an approach we first outlined in [89] and which is also used in [38]. The following code shows an example of this translation. The `copy` function requires a model of `Iterator`, which has an associated type `elt`.

```
let copy = (Λ Iter, OutIter where Iterator<Iter>,
   OutputIterator<OutIter, Iterator<Iter>.elt>. /* body */)
```

An extra type parameter for the associated type is added to the translated version of `copy`.

```
let copy =
  (Λ Iter, OutIter, elt.
  (λ Iterator_21:(fun(Iter)→Iter)*(fun(Iter)→elt)*(fun(Iter)→bool),
    OutputIterator_23:(fun(OutIter,elt)→OutIter).
    /* body */)
```

However, there are two complications here that are not present in [38]: same-type constraints and concept refinement. Due to the same-type constraints, all type expressions in the same equivalence class must be translated to the same System F type. Fortunately, the congruence closure algorithm for type equality [142] is based on a union-find data structure that maintains a representative for each type class. Therefore the translation outputs the representative for each type expression. The translation of the `merge` function shows an example of this. There are two type parameters `elt1` and `elt2` for each of the two Iterator constraints. Note that in the types for the three dictionaries, only `elt1` is used, since it was chosen as the representative.

```
let merge =
  (Λ In1, In2, Out, elt1, elt2.
  (λ Iterator_78:(fun(In1)→In1)*(fun(In1)→elt1)*(fun(In1)→bool),
     Iterator_80:(fun(In2)→In2)*(fun(In2)→elt1)*(fun(In2)→bool),
     OutputIterator_84:(fun(Out,elt1)→Out),
     LessThanComparable_88:(fun(elt1,elt1)→bool).  /* body */))
```

The second complication is the presence of concept refinement. As mentioned in [38], extra type parameters are needed not just for the associated types of a concept c mentioned in the `where` clause, but also for every associated type in concepts that c refines. Furthermore, there may be diamonds in the refinement diagram. To preclude duplicate associated types we keep track of which concepts (with particular type arguments) have already been processed.

Figure 17 presents the translation from F$^{\mathcal{G}}$ with associated types and same-type constraints to System F. We omit the (Mem), (Var), and (Abs) rules since they do not change. The functions $\flat$ and $\flat^m$ need to be changed to take into account associated types that may appear in the type of a concept member or refinement. For example, in the body of function below, the expression `<B(r)>.bar(x)` has type `<B(r)>.z`, not just `z`. Also, the refinement for `A<z>` in `B` translates to `B<r>.z` modeling `A`.

```
concept A<u> { foo : fun(u)→u; } in
```

```
concept B<t> {
  types z;
  refines A<z>;
  bar : fun(t)→z;
} in
(Λ r where B<r>.
 λ x:r. A<B<r>.z>.foo(B<r>.bar(x)))
```

We define a function $\flat^a$ to collect all the associated types from a concept c and from the concepts refined by c and map them to their concept-qualified names.

$\flat^a(c, \overline{\tau}) =$

$S := \overline{s : c\texttt{<}\overline{\tau}\texttt{>}.s}$

for $i = 0, \ldots, |\overline{c'}| - 1$

$S := S, \flat^a(c'_i, S(\overline{\tau'}_i))$

return S

where

$\texttt{concept}\ c\texttt{<}\overline{t}\texttt{>}\{\texttt{types}\ \overline{s}\ ;\ \texttt{refines}\ \overline{c'\texttt{<}\overline{\tau'}\texttt{>}};\ \overline{x : \sigma};\ \overline{\rho = \rho'}\} \in \Gamma$

Here is the new definition of $\flat$.

$\flat(c, \overline{\tau}, \overline{n}, \Gamma) =$

$S := \flat^a(c, \overline{\tau}), \overline{t : \tau}$

$M := \emptyset$

for $i = 0, \ldots, |\overline{c'}| - 1$

$M := M \cup \flat(c'_i, S(\overline{\tau'}_i), (\overline{n}, i), \Gamma)$

for $i = 0, \ldots, |\overline{x}| - 1$

$M := M \cup \{x_i : (S(\sigma_i), (\overline{n}, |\overline{c'}| + i))\}$

return M

where

$\texttt{concept}\ c\texttt{<}\overline{t}\texttt{>}\{\texttt{types}\ \overline{s}\ ;\ \texttt{refines}\ \overline{c'\texttt{<}\overline{\tau'}\texttt{>}};\ \overline{x : \sigma};\ \overline{\rho = \rho'}\} \in \Gamma$

We used $\flat^m$ in Section 7.2 to collect the the models from a concept c and the concepts that c refines. We change $\flat^m$ to also collect the same-type constraints from the concepts. In addition, for every associated type s in c we generate a fresh type variable s' and add the same-type constraint $s' = c\texttt{<}\overline{\tau}\texttt{>}.s$. The function $\flat^m$ also returns the type variables generated for the associated types.

$\flat^m(c, \overline{\rho}, d, \overline{n}, \Gamma)$ =

check $\Gamma \vdash \overline{\rho} \rightsquigarrow -$ and generate fresh variables $\overline{s'}$

$\Gamma := \Gamma, \overline{s' = c\texttt{<}\overline{\rho}\texttt{>}.s}$

$A := \flat^a(c, \overline{\rho}), \overline{t : \rho}$

$\overline{s''} := []$; $\overline{\tau} := []$

for $i = 0, \ldots, |\overline{c'}| - 1$

$(\Gamma, \overline{a}, \delta') := \flat^m(c'_i, A(\overline{\rho'}_i), d, (\overline{n}, i), \Gamma)$

$\overline{s''} := \overline{s''}, \overline{a}$; $\overline{\tau} := \overline{\tau}, \delta'$

$\overline{\tau} := \overline{\tau}@A(\overline{\sigma})$

$\Gamma := \Gamma, \overline{A(\eta)} = \overline{A(\eta')}$

$\Gamma := \Gamma, \texttt{model}\ c\texttt{<}\overline{\rho}\texttt{>} \mapsto (d, \overline{n}, \flat^a(c, \overline{\rho}))$

return $(\Gamma, (\overline{s''}, \overline{s'}), \overline{\tau})$

where

$\texttt{concept}\ c\texttt{<}\overline{t}\texttt{>}\{\texttt{types}\ \overline{s}\ ;\ \texttt{refines}\ \overline{c'\texttt{<}\overline{\rho'}\texttt{>}};\ \overline{x : \sigma};\ \overline{\eta = \eta'}\} \in \Gamma$

The `where` clause of a type abstraction is processed sequentially so that later requirements in the `where` clause may refer to requirements (e.g., their associated types) that appear earlier in the list.

$\flat^w([], \Gamma) = (\Gamma, [])$

$\flat^w((c\texttt{<}\overline{\rho}\texttt{>}, \overline{c'\texttt{<}\overline{\rho'}\texttt{>}}), \Gamma)$ =

generate fresh d

$(\Gamma, \overline{s}, \delta) := \flat^m(c, \overline{\rho}, d, [], \Gamma)$

$(\Gamma, \overline{s'}, \overline{\delta'}) := \flat^w(\overline{c'\texttt{<}[\overline{t} \mapsto \overline{\rho}]\overline{\rho'}\texttt{>}}, \Gamma)$

return $(\Gamma, (\overline{s}, \overline{s'}), (\delta, \overline{\delta'}))$

FIGURE 15. Well-formed F$^{\mathcal{G}}$ types (with associated types) and translation to System F.

$$\boxed{\Gamma \vdash \tau \rightsquigarrow \tau'}$$

$$\text{(TyVar)}\ \frac{t \in \Gamma}{\Gamma \vdash t \rightsquigarrow [t]_\Gamma}$$

$$\text{(TyAbs)}\ \frac{\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \qquad \Gamma \vdash \tau \rightsquigarrow \tau'}{\Gamma \vdash \texttt{fun}\ \overline{\sigma} \to \tau \rightsquigarrow \texttt{fun}\ \overline{\sigma'} \to \tau'}$$

$$\text{(TyTAbs)}\ \frac{(\Gamma', \overline{s}, \overline{\delta}) = \flat^w(\overline{c\texttt{<}\overline{\rho}\texttt{>}}, (\Gamma, \overline{t})) \qquad \Gamma', \overline{\eta = \eta'} \vdash \tau \rightsquigarrow \tau'}{\Gamma \vdash \forall \overline{t}\ \texttt{where}\ \overline{c\texttt{<}\overline{\rho}\texttt{>}}, \overline{\eta = \eta'}\ .\ \tau \rightsquigarrow \forall \overline{t}, \overline{s}\ .\texttt{fun}\ \overline{\delta} \to \tau'}$$

$$\text{(TyAsc)}\ \frac{\Gamma \vdash \overline{\rho} \rightsquigarrow \overline{\rho'} \qquad \Gamma \vdash \texttt{model}\ c\texttt{<}\overline{\rho}\texttt{>} \ldots \in \Gamma}{\Gamma \vdash c\texttt{<}\overline{\rho}\texttt{>}.x \rightsquigarrow [c\texttt{<}\overline{\rho}\texttt{>}.x]_\Gamma}$$

where

$$\texttt{concept}\ c\texttt{<}\overline{t}\texttt{>}\{\texttt{types}\ \overline{s}\ ;\ \texttt{refines}\ \overline{c'\texttt{<}\overline{\rho'}\texttt{>}};\ \overline{x : \sigma};\ \overline{\eta = \eta'}\} \in \Gamma$$

Figure 15 shows the changes to the translation of F$^{\mathcal{G}}$ types to System F types. Type variables and member access types are mapped to their representative, written as $[-]_\Gamma$.

The proof that the translation to System F preserves well typing can be modified to take into account the changes we have made for associated types and same-type constraints. The proof relies on the following lemma which establishes the correspondence between type equality judgments and type translation. Whenever two F$^{\mathcal{G}}$ types are equal they translate to the same System F type.

LEMMA 4 (Correspondence of type equality and translation).

If $\Gamma \vdash \sigma = \tau$ *and* $\Gamma \vdash \sigma \rightsquigarrow \rho$ *then* $\Gamma \vdash \tau \rightsquigarrow \rho$.

The F$^{\mathcal{G}}$ environment now contains information about associated types and same-type constraints, so the correspondence with System F environments is updated in Figure 16.

FIGURE 16. Well-formed F$^{\mathcal{G}}$ environment in correspondence with a System F environment.

$$\boxed{\Gamma \rightsquigarrow \Sigma}$$

$$\frac{}{\emptyset \rightsquigarrow \emptyset} \qquad \frac{\Gamma \rightsquigarrow \Sigma \quad \Gamma \vdash \tau \rightsquigarrow \tau'}{\Gamma, x : \tau \rightsquigarrow \Sigma, x : \tau'} \qquad \frac{\Gamma \rightsquigarrow \Sigma}{\Gamma, t \rightsquigarrow \Sigma, t}$$

$$\frac{\Gamma \rightsquigarrow \Sigma \quad (-,-,\delta) = \flat^m(c, \overline{\tau}, -, -, \Gamma)}{\Gamma, (\texttt{model}\ c\texttt{<}\overline{\tau}\texttt{>} \mapsto (d, [], \overline{s : \sigma})) \rightsquigarrow \Sigma, d : \delta}$$

$$\frac{\Gamma \rightsquigarrow \Sigma \quad 0 < |\overline{n}| \quad d : \delta \in \Sigma \quad (-,-,\delta_{\overline{n}}) = \flat^m(c, \overline{\tau}, -, -, \Gamma)}{\Gamma, (\texttt{model}\ c\texttt{<}\overline{\tau}\texttt{>} \mapsto (d, \overline{n}, \overline{s : \sigma})) \rightsquigarrow \Sigma}$$

$$\frac{\Gamma \rightsquigarrow \Sigma \quad (\Gamma', -, \overline{\delta'}) = \flat^w(\overline{c\texttt{<}\overline{\tau}\texttt{>}}, (\Gamma, \bar{t})) \quad \Gamma' \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \quad \Gamma' \vdash \overline{\rho} \rightsquigarrow \overline{\nu} \quad \Gamma' \vdash \overline{\rho'} \rightsquigarrow \overline{\nu'}}{\Gamma, (\texttt{concept}\ c\texttt{<}\bar{t}\texttt{>}\{\texttt{types}\ \overline{s}\ ;\ \texttt{refines}\ \overline{c'\texttt{<}\overline{\tau}\texttt{>}};\ \overline{x : \sigma};\ \overline{\rho = \rho'}\} \mapsto \overline{\delta'}@\overline{\sigma'}) \rightsquigarrow \Sigma}$$

THEOREM 2 (Translation preserves well typing).

If $\Gamma \vdash e : \tau \rightsquigarrow f$ *and* $\Gamma \rightsquigarrow \Sigma$ *then there exists* τ' *such that* $\Sigma \vdash f : \tau'$ *and* $\Gamma \vdash \tau \rightsquigarrow \tau'$.

PROOF. Like the proof of Theorem 1, this proof is by induction on the derivation of $\Gamma \vdash e : \tau \rightsquigarrow f$. The cases for (MDL), (TAPP), and (APP) rules differ because they rely on the type equality judgment.

Mdl: Let $\Gamma' = \Gamma, (\texttt{model}\ c\texttt{<}\overline{\rho}\texttt{>} \mapsto (d, [], (\cup\overline{A'}, \overline{s' : [\overline{s} \mapsto \overline{\nu}]s'})))$. We have the following by inversion:

(26) $\Gamma \vdash \overline{e : \sigma} \rightsquigarrow \overline{f}$

(27) $\overline{\texttt{model}\ c'\texttt{<}S(\overline{\rho'})\texttt{>} \mapsto (d', \overline{n'}, A')} \subseteq \Gamma$

(28) $\overline{x} \subseteq \overline{y}$

(29) $\Gamma \vdash [\overline{y} \mapsto \overline{\sigma}]\overline{x} = \overline{S'(\tau)}$

(30) $\Gamma \vdash \overline{S'(\eta)} = \overline{S'(\eta')}$

(31) $\Gamma' \vdash e : \tau \rightsquigarrow f$

(32) $\texttt{concept}\ c\texttt{<}\bar{t}\texttt{>}\{\texttt{types}\ \overline{s'}\ ;\ \texttt{refines}\ \overline{c'\texttt{<}\overline{\rho'}\texttt{>}};\ \overline{x : \tau};\ \overline{\eta = \eta'}\} \mapsto \delta \in \Gamma$

From $\Gamma \rightsquigarrow \Sigma$, (26), and the induction hypothesis there exists σ' such that $\Sigma \vdash \overline{f} : \overline{\sigma'}$ and $\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'}$. Next, let $\overline{r} = \overline{(\texttt{nth} \ldots (\texttt{nth}\ d'\ n'_1) \ldots n'_k)}$. From $\Gamma \rightsquigarrow \Sigma$ and (32) we have $(-, \overline{s}, \overline{\delta'}) = \flat^w(\overline{c'\texttt{<}\overline{\rho'}\texttt{>}}, \Gamma)$. and therefore $(-, \overline{s}, [\overline{t} \mapsto \overline{\rho}]\overline{\delta'}) = \flat^w(\overline{c'\texttt{<}[\overline{t} \mapsto \overline{\rho}]\overline{\rho'}\texttt{>}}, \Gamma)$. Together with (27) and Lemma 2 we have $\Sigma \vdash \overline{r} : [\overline{t} \mapsto \overline{\rho}]\overline{\delta'}$. With (28), (29), and (30), we have a well typed dictionary:

$$\Sigma \vdash (\overline{r}@[\overline{y} \mapsto \overline{f}]\overline{x}) : \delta \tag{33}$$

Let Σ' be $\Sigma, d : \delta$ so $\Gamma' \rightsquigarrow \Sigma'$. Then with (31) and the induction hypothesis there exists τ' such that $\Sigma' \vdash f : \tau'$ and $\Gamma' \vdash \tau \rightsquigarrow \tau'$. From (33) we show $\Sigma \vdash \texttt{let}\ d = (\overline{r}@[\overline{y} \mapsto \overline{f}]\overline{x})\ \texttt{in}\ f : \tau'$.

TApp: By inversion of the (TAPP) rule we have:

$$\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \tag{34}$$

$$\Gamma \vdash e : \forall \overline{t}.\ \texttt{where}\ \overline{c\texttt{<}\overline{\rho}\texttt{>}}, \overline{\eta = \eta'}.\ \tau \rightsquigarrow f \tag{35}$$

$$\overline{\texttt{model}\ c\texttt{<}[\overline{t} \mapsto \overline{\sigma}]\overline{\rho}\texttt{>} \mapsto (d, \overline{n}, \overline{s : \nu})} \in \Gamma \tag{36}$$

$$\Gamma \vdash \overline{[\overline{t} \mapsto \overline{\sigma}]\eta = [\overline{t} \mapsto \overline{\sigma}]\eta'} \tag{37}$$

From (35) and the induction hypothesis there exists τ' such that $\Sigma \vdash f : \tau'$ and $\Gamma \vdash \forall \overline{t}\ \texttt{where}\ \overline{c\texttt{<}\overline{\rho}\texttt{>}}, \overline{\eta = \eta'}.\ \tau \rightsquigarrow \tau'$. By inversion there exists $\overline{\delta}$, τ'', and Γ' such that

$$\tau' = \forall \overline{t}, \overline{s'}.\ \texttt{fun}\ \overline{\delta} \to \tau'' \tag{38}$$

$$(\Gamma', -, \overline{\delta}) = \flat^w(\overline{c\texttt{<}\overline{\rho}\texttt{>}}, (\Gamma, \overline{t})) \tag{39}$$

$$\Gamma', \overline{\eta = \eta'} \vdash \tau \rightsquigarrow \tau'' \tag{40}$$

Using (38) we have

$$\Sigma \vdash f[\overline{\sigma'}, \overline{\nu}] : [\overline{t} \mapsto \overline{\sigma'}][\overline{s'} \mapsto \overline{\nu}](\texttt{fun}\ \overline{\delta} \to \tau'') \tag{41}$$

From (39) and (34) we have

$$(\Gamma', -, [\overline{t} \mapsto \overline{\sigma'}][\overline{s'} \mapsto \overline{\nu}]\overline{\delta}) = \flat^w(\overline{c\texttt{<}[\overline{t} \mapsto \overline{\sigma}]\overline{\rho}\texttt{>}}, (\Gamma, \overline{t}))) \tag{42}$$

Let $\overline{d'} = \overline{(\texttt{nth} \ldots (\texttt{nth}\ d\ n_1) \ldots n_k)}$. From the assumption $\Gamma \rightsquigarrow \Sigma$, (36), and (42) we apply Lemma 2 to get $\Sigma \vdash \overline{d'} : [\overline{t} \mapsto \overline{\sigma'}][\overline{s'} \mapsto \overline{\nu}]\overline{\delta}$. Then with (41) we have $\Sigma \vdash f[\overline{\sigma'}, \overline{\nu}](\overline{d'}) : [\overline{t} \mapsto \overline{\sigma'}][\overline{s'} \mapsto \overline{\nu}]\tau''$ and from (34) and (40) we have $\Gamma \vdash [\overline{t} \mapsto \overline{\sigma}]\tau \rightsquigarrow [\overline{t} \mapsto \overline{\sigma'}][\overline{s'} \mapsto \overline{\nu}]\tau''$.

App: By inversion there exists $\overline{\sigma_1}$ and $\overline{\sigma_2}$ such that $\Gamma \vdash e_1 : \texttt{fun}\ \overline{\sigma_1} \to \tau \rightsquigarrow f_1$ and $\Gamma \vdash \overline{e_2} : \overline{\sigma_2} \rightsquigarrow \overline{f_2}$ and $\Gamma \vdash \overline{\sigma_1} = \overline{\sigma_2}$. By the induction hypothesis there exists ρ_1 such that $\Sigma \vdash f_1 : \rho_1$ and $\Gamma \vdash \texttt{fun}\ \overline{\sigma_1} \to \tau \rightsquigarrow \rho_1$. Then by inversion there exists $\overline{\sigma_1'}$ and τ' such that $\rho_1 = \texttt{fun}\ \overline{\sigma_1'} \to \tau'$ and $\Gamma \vdash \overline{\sigma_1} \rightsquigarrow \overline{\sigma_1'}$ and $\Gamma \vdash \tau \rightsquigarrow \tau'$. Also by the induction hypothesis there exists $\overline{\rho_2}$ such that $\Sigma \vdash \overline{f_2} : \overline{\rho_2}$ and $\Gamma \vdash \overline{\sigma_2} \rightsquigarrow \overline{\rho_2}$. Then with $\Gamma \vdash \overline{\sigma_1} = \overline{\sigma_2}$ and Lemma 4 we have $\overline{\sigma_1'} = \overline{\rho_2}$ and so $\Gamma \vdash \overline{f_2} : \overline{\sigma_1'}$. Thus $\Sigma \vdash f_1(\overline{f_2}) : \tau'$.

□

7.5. Summary

This chapter showed that the design for generics presented in this thesis is type safe. The language $\mathcal{G}$ is not type safe, due to the aspects of the language unrelated to generics: the presence of pointer, manual memory allocation, and also stack allocation. To show type safety of the design for generics, we embed the design in System F, a type safe language, creating the language F$^{\mathcal{G}}$. The language F$^{\mathcal{G}}$ is defined by translation to System F, and we show that if an F$^{\mathcal{G}}$ program is well typed, the translation will result in a well typed term of System F, thereby ensuring that execution of the System F term will not result in a type error.

FIGURE 17. Type rules for F$^{\mathcal{G}}$ with associated types and translation to System F.

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

$$\text{(CPT)}\ \frac{\begin{array}{c}\text{distinct } \overline{t} \quad \text{distinct } \overline{s} \quad \overline{\texttt{concept } c\texttt{<}\overline{t'}\texttt{>}\{\ldots\} \mapsto \delta'} \in \Gamma \quad \Gamma, \overline{t}, \overline{s} \vdash \overline{\rho} \rightsquigarrow \overline{\rho'} \\ \Gamma, \overline{t}, \overline{s} \vdash \overline{\tau} \rightsquigarrow \overline{\tau'} \quad \Gamma, \overline{t}, \overline{s} \vdash \overline{\sigma} \rightsquigarrow \overline{\nu} \quad \Gamma, \overline{t}, \overline{s} \vdash \overline{\sigma'} \rightsquigarrow \overline{\nu'} \\ \delta = (\overline{[\overline{t'} \mapsto \overline{\rho'}]\delta'})@\overline{\tau'} \\ \Gamma, (\texttt{concept } c\texttt{<}\overline{t}\texttt{>}\{\texttt{types } \overline{s}\ ;\ \texttt{refines } \overline{c'\texttt{<}\overline{\rho}\texttt{>}};\ \overline{x : \tau};\ \overline{\sigma = \sigma'}\} \mapsto \delta) \vdash e : \tau \rightsquigarrow f\end{array}}{\Gamma \vdash \texttt{concept } c\texttt{<}\overline{t}\texttt{>}\{\texttt{types } \overline{s}\ ;\ \texttt{refines } \overline{c'\texttt{<}\overline{\rho}\texttt{>}};\ \overline{x : \tau};\ \overline{\sigma = \sigma'}\}\ \texttt{in}\ e : \tau \rightsquigarrow f}$$

$$\text{(MDL)}\ \frac{\begin{array}{c}\texttt{concept } c\texttt{<}\overline{t}\texttt{>}\{\texttt{types } \overline{s'}\ ;\ \texttt{refines } \overline{c'\texttt{<}\overline{\rho'}\texttt{>}};\ \overline{x : \tau};\ \overline{\eta = \eta'}\} \mapsto \delta \in \Gamma \quad \Gamma \vdash \overline{\rho} \rightsquigarrow \overline{\tau'} \\ \Gamma \vdash \overline{\nu} \rightsquigarrow \overline{\nu'} \quad \Gamma \vdash \overline{e : \sigma} \rightsquigarrow \overline{f} \quad \overline{s'} \subseteq \overline{s} \\ S = \overline{t : \rho}, \overline{s' : [\overline{s} \mapsto \overline{\nu}]s'} \quad \Gamma \vdash \overline{\texttt{model } c'\texttt{<}S(\overline{\rho'})\texttt{>} \mapsto (d', \overline{n},\ A')} \in \Gamma \\ S' = S, \cup\overline{A'} \quad \overline{x} \subseteq \overline{y} \quad \Gamma \vdash \overline{[\overline{y} \mapsto \overline{\sigma}]\overline{x} = S'(\tau)} \quad \Gamma \vdash \overline{S'(\eta) = S'(\eta')} \\ d \text{ fresh} \quad \Gamma, (\texttt{model } c\texttt{<}\overline{\rho}\texttt{>} \mapsto (d, [],\ (\cup\overline{A'}, \overline{s' : [\overline{s} \mapsto \overline{\nu'}]s'})\ )) \vdash e : \tau \rightsquigarrow f \\ \overline{d''} = \overline{(\ \texttt{nth}\ \ldots (\ \texttt{nth}\ d'\ n_1) \ldots n_k)}\end{array}}{\Gamma \vdash \texttt{model } c\texttt{<}\overline{\rho}\texttt{>}\ \{\ \texttt{types } \overline{s = \nu}; \overline{y = e}\}\ \texttt{in}\ e : \tau \rightsquigarrow \texttt{let } d = (\overline{d''}@[\overline{y} \mapsto \overline{f}]\overline{x})\ \texttt{in}\ f}$$

$$\text{(TABS)}\ \frac{\text{distinct } \overline{t} \quad \overline{t} \cap \text{FTV}(\Gamma) = \emptyset \quad (\Gamma',\ \overline{s}\ , \overline{\delta}) = \flat^w(\overline{c\texttt{<}\overline{\rho}\texttt{>}}, (\Gamma, \overline{t})) \quad \Gamma',\ \overline{\tau = \tau'} \vdash e : \tau \rightsquigarrow f}{\Gamma \vdash \Lambda\overline{t}\ \texttt{where } \overline{c\texttt{<}\overline{\rho}\texttt{>}},\ \overline{\tau = \tau'}\ .\ e : \forall\overline{t}\ \texttt{where } \overline{c\texttt{<}\overline{\rho}\texttt{>}},\ \overline{\tau = \tau'}\ .\ \tau \rightsquigarrow \Lambda\overline{t},\ \overline{s}\ .\ \lambda\overline{d : \delta}.\ f}$$

$$\text{(TAPP)}\ \frac{\begin{array}{c}\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \quad \Gamma \vdash e : \forall\overline{t}\ \texttt{where } \overline{c\texttt{<}\overline{\rho}\texttt{>}},\ \overline{\eta = \eta'}\ .\ \tau \rightsquigarrow f \\ \Gamma \vdash \overline{\texttt{model } c\texttt{<}[\overline{t} \mapsto \overline{\sigma}]\overline{\rho}\texttt{>} \mapsto (d, \overline{n},\ \overline{s : \nu}\ )} \in \Gamma \quad \overline{[\overline{t} \mapsto \overline{\sigma}]\eta = [\overline{t} \mapsto \overline{\sigma}]\eta'}\end{array}}{\Gamma \vdash e[\overline{\sigma}] : [\overline{t} \mapsto \overline{\sigma}]\tau \rightsquigarrow f[\overline{\sigma'},\ \overline{\overline{\nu}}\ ](\overline{\texttt{nth}\ \ldots (\ \texttt{nth}\ d\ n_1) \ldots n_k})}$$

$$\text{(ALS)}\ \frac{t \notin \text{FTV}(\Gamma) \quad \Gamma, t = \tau \vdash e : \tau \rightsquigarrow f}{\Gamma \vdash \texttt{type } t = \tau\ \texttt{in}\ e : \tau \rightsquigarrow f}$$

$$\text{(APP)}\ \frac{\Gamma \vdash e_1 : \texttt{fun } \overline{\sigma} \to \tau \rightsquigarrow f_1 \quad \Gamma \vdash \overline{e_2 : \sigma'} \rightsquigarrow \overline{f_2} \quad \Gamma \vdash \overline{\sigma = \sigma'}}{\Gamma \vdash e_1\ \overline{e_2} : \tau \rightsquigarrow f_1(\overline{f_2})}$$

8

Conclusion

This thesis presents and evaluates a design for language support for generic programming, embodied in the programming language $\mathcal{G}$. The design formalizes the current practice of generic programming in C++, replacing the semi-formal specification language used to document C++ libraries with a formal interface description language integrated with the type system of a full programming language. The advantage is that an automated tool (the $\mathcal{G}$ type system) checks uses of generic components against their interfaces, and on the other side, checks implementations of generic components against their interfaces.

Of course, many languages provide this kind of modularity, but what is unique about $\mathcal{G}$ is that 1) its interface description language is expressive enough to describe the rich interfaces of generic libraries such as the Standard Template Library and the Boost Graph

Library, and 2) using generic components in $\mathcal{G}$ is convenient, even when dealing with large and complex abstractions. Both of these points were demonstrated in Chapter 6. The central features of $\mathcal{G}$, `concept`'s, `model`'s, and `where` clauses, cooperate to provide a mechanism for implicitly passing type-specific operations to generic functions, thereby relieving users of this task. Implicit mechanisms are often dangerous, so in $\mathcal{G}$ the connection between the implementations of type-specific operations and the concepts they fulfill is established by explicit `model` definitions. `model` definitions are lexically scoped, so it is always possible for a programmer to determine which model will be used by examining the program text of just the module under construction and the public interface of any imported modules.

Chapter 5 described a compiler for $\mathcal{G}$ that can separately compile generic functions. This is a critical point concerning the scalability of reuse-oriented software construction. Separate compilation allows the compile time of a component to be a function of the size of just that component and not a function of everything used by the component. Of course, there is an inherent performance penalty associated with separate compilation (which is not particular to $\mathcal{G}$). The design of $\mathcal{G}$ allows for optimizations such as function specialization and inlining to be applied in situations where the programmer does not want separate compilation, but instead desires the greatest possible performance. Implementing these optimizations in the compiler for $\mathcal{G}$ is planned for future work.

In conclusion, the design of $\mathcal{G}$ successfully satisfies the goals set down in Chapter 1: it supports the modular construction of software, it makes generic components easier to use and to build, it provides support for implementing and dispatching between efficient algorithms, and it allows for efficient compilation.

There are several directions for future work on the language $\mathcal{G}$: 1) further refinements in the support for generic programming, 2) support for generative programming and 3) improved compilation.

Support for Generic Programming. Chapter 6 presented an idiom for dispatching between specialized versions of an algorithm. While this idiom incurs little burden on users of generic algorithms, it does expose unnecessary details in the interface of the generic algorithms and can lead to large `where` clauses. One solution that we have envisioned for this

is to add support for optional requirements in a `where` clause. The rules for concept-based overload resolution would then take this into account and allow for run-time or link-time dispatching based on whether the optional requirements were satisfied at a particular call site. There are some technical challenges and open questions concerning the compilation of optional requirements that will be the focus of future work.

Another area where there is room for improvement is in implicit instantiation. As discussed in Section 4.6.1 it would be nice to allow coercions on function types (use the (ARROW) subtyping rule). I have done some research into creating a semi-decision procedure for this subtyping problem, but it remains to prove that the procedure is sound and to demonstrate whether it is effective and efficient in practice.

An important aspect of concepts are their semantic requirements. The language $\mathcal{G}$ does not yet provide mechanisms for expressing semantics, but this in an extremely interesting area for future research. Semantic requirements could be an aid for program correctness and for optimization. If $\mathcal{G}$ were outfitted with a program logic one could prove correctness of generic algorithms based on the assumptions (semantic guarantees) provided by concepts. Similarly, models of concepts could be proved correct by showing that the implementation functions meet the requirements of the concept. Semantic requirements can also be used in the context of compiler optimization. Many optimizations that are currently applied only to scalar values could also be applied to user-defined types, such as constant folding and constant propagation, if model definitions can assert that the necessary semantic properties hold for the user-defined types.

Generative Programming. While the design of generics for $\mathcal{G}$ provides language support for the implementation and use of generic algorithms, it does not provide language support for *generative* programming, which is often used in generic libraries to allow for code reuse in the implementation of data structures. We will be investigating the addition of metaprogramming features to $\mathcal{G}$ to provide support for generative programming. There is considerable challenge with respect to integrating metaprogramming facilities and parametric polymorphism. Metaprogramming typically relies on information from the context in which a library is used, whereas parametric polymorphism blocks out information from

the context. Thus, at a fundamental level metaprogramming and parametric polymorphism are at odds with each other, so finding a way to bring them together will be challenging.

Improved Compilation. The compiler for $\mathcal{G}$ does not yet include an optimization pass. Many traditional optimizations would increase the efficiency of $\mathcal{G}$ programs, but the most critical optimizations are those that fall under the heading of partial evaluation. Those include function specialization, function inlining, constant folding, and constant propagation. One other critical optimization for $\mathcal{G}$ programs is the scalar replacement of aggregates [132].

The compiler for $\mathcal{G}$ currently translates to C++. This translation took advantage of many features of C++ to reduce the amount of work done by the compiler. However, it would be useful to compile all the way to C, thereby gaining more portability. In particular, replacing the use of the `any` class with `void*` would likely speed up compilation of the resulting C/C++ code. Similarly, compiling to Java byte code or to .Net would allow for better interoperability with other languages and component frameworks.

A

Grammar of $\mathcal{G}$

This appendix defines the syntax for $\mathcal{G}$. We start with some preliminaries concerning the lexical structure of identifiers and literals and then describe the grammars for type expressions, declarations, statements, and expressions.

There are several kinds of identifiers that appear in $\mathcal{G}$ programs but they all share the same lexical structure as given by the following regular expression:

```
['A'-'Z' 'a'-'z' '_'] ['A'-'Z' 'a'-'z' '_' '0'-'9' '\'']*
```

The grammar variable id stands for value variables, $tyvar$ for type variables, $clid$ for class, struct, and union names, cid for concept names, and mid for module names.

The integer literals $intlit$ are sequences of digits

```
['0'-'9']+
```

and the floating point literals *floatlit* are sequences of digits followed by a period and an optional second sequence of digits.

```
['0'-'9']+ '.' ['0'-'9']*
```

A.1. Type expressions

The type expressions of $\mathcal{G}$ differ from those of C++ in several respects. Instead of function pointers $\mathcal{G}$ has first-class functions, so $\mathcal{G}$ has function types, not function pointer types. Also, $\mathcal{G}$ has type expressions for referring to associated types of a model using the dot notation. Two other minor differences are that there are no reference types and `const` is not a general type qualifier.

type	::=	*tyvar*	type variable
		`fun` *polyhdr* `(`*type mode*`, ...)`[`->` *type mode*]	function
		clid[`<`*type*`, ...>`]	class, struct, or union
		scope`.`*tyvar*	scope-qualified type
		type [`const`] `*`	pointer
		`(`*type*`)`	parenthesized type
		btype	basic type
mode	::=	*mut* [`&`]	pass by reference
		`@@`	pass by value
mut	::=	[`const`]	constant
		`!`	mutable
polyhdr	::=	[`<`*tyvar*`, ...>`][`where {`*constraint*`, ...}`]	polymorphic header
constraint	::=	*cid*`<`*type*`, ...>`	model constraint
		type `==` *type*	same-type constraint
		funsig	function constraint
scope	::=	*scopeid*	
		scope`.`*scopeid*	scope member
scopeid	::=	*mid*	module identifier
		cid`<`*type*`, ...>`	model identifier
btype	::=	[`signed`] *intty* | `unsigned` *intty*	
		`float` | `double` | `long double`	
		`char` | `string` | `bool` | `void`	
intty	::=	`int` | `short` | `long` | `long long`	

A.2. Declarations

The main declarations of interest in $\mathcal{G}$ are concepts, models, and `where` clauses, which can appear in function, model, class, struct, and union definitions. For now, classes in $\mathcal{G}$ are basic, consisting only of constructors, a destructor, and data members. A struct in $\mathcal{G}$ consists only of data members.

decl	::=	`concept` *cid*`<`*tyvar*, ...`> { `*cmem* ... `};`	concept
		`model` *polyhdr* `<`*type*, ...`> { `*decl* ... `};`	model
		`class` *clid polyhdr* `{`*clmem* ...`};`	class
		`struct` *clid polyhdr* `{`*type id*`; ...};`	struct
		`union` *clid polyhdr* `{`*type id*`; ...};`	union
		fundef	
		funsig	
		`let` *id* `=` *expr*`;`	global variable binding
		`type` *tyvar* `=` *type*`;`	type alias
		`module` *mid* `{` *decl* ... `}`	module
		`scope` *id* `=` *scope*`;`	scope alias
		`import` *scope*.*c*`<`$\overline{\tau}$`>;`	import model
		`public:` *decl* ...	public region
		`private:` *decl* ...	private region
fundef	::=	`fun` *id polyhdr* `(`*type mode* [*id*]`, ...)` `->` *type mode* `{` *stmt* ... `}`	Function definition
funsig	::=	`fun` *id polyhdr* `(`*type mode* [*id*]`, ...)` `->` *type mode*`;`	Function signature
cmem	::=	*funsig*	Function requirement
		fundef	" with default impl.
		`type` *tyvar*`;`	Associated type
		type `==` *type*`;`	Same-type requirement
		`refines` *cid*`<`*type*`, ...>;`	Refinement
		`require` *cid*`<`*type*`, ...>;`	Nested requirement
clmem	::=	*type id*`;`	data member
		polyhdr clid`(`*type mode* [*id*]`, ...)`{*stmt* ...`}`	constructor
		`~`*clid*`()`{*stmt* ...`}`	destructor

A.3. Statements and expressions

Local variables are introduced with the `let` statement, with the type of the variable deduced from the right-hand side expression. The `switch` statement is quite different from that of C++, for it provides type-safe decomposition of unions. There is an expression for

initializing a struct object by field, and there is the dot notation for accessing members of a model. The syntax for explicit instantiation includes extra bars as a concession to ease parsing with Yacc [95]. Without the bars, the syntax is ambiguous with the less-than operator.

stmt	::=	`let` *id* `=` *expr* `;`	local variable binding
		`type` *tyvar* `=` *type* `;`	type alias
		expr `;`	expression
		`return` [*expr*] `;`	return from function
		`if (`*expr*`)` *stmt* [`else` *stmt*]	conditional
		`while (`*expr*`)` *stmt*	loop
		`{` *stmt* ... `}`	compound
		`;`	empty
		`switch (`*expr*`){` *case* ... `}`	switch on union
case	::=	`case` *id* `:` *stmt* ...	case
		`default:` *stmt* ...	default case
expr	::=	*id*	variable
		expr`(`*expr*`,` ...`)`	function application
		`fun` *polyhdr* `(`*type mode* [*id*]`,` ...`)`	function expression
		id`=`*expr*`,` ...`(`{*stmt* ...}|`:`*expr*`)`	
		scope`.`*id*	scope member
		expr`.`*id*	object member
		expr`<|`*type*`,` ...`|>`	explicit instantiation
		expr`,` ...	sequence
		expr `?` *expr* `:` *expr*	conditional
		`(`*expr*`)`	parenthesized expression
		alloc clid`(`*expr*`,` ...`)`	class instance
		alloc clid`{`*id*`=`*expr*`,` ...`}`	struct or union instance
		alloc type `[`*expr*`]`	array allocation
		`delete` *expr*	invoke destructor and release memory
		`destroy` *expr*	invoke destructor
		literal	literals
alloc	::=	`@@`	stack allocation
		`new`	manual heap allocation
		`new GC`	garbage collected heap allocation
		`new (`*expr*`)`	construct in place
literal	::=	`true` | `false`	Boolean constants
		intlit	integer constant
		floatlit	floating point constant
		`'`*char*`'`	character constant
		`"`*char* ...`"`	string literal

A.4. Derived forms

for (s_1 e_1; e_2) s_2 $\Longrightarrow$ { s_1 while (e_1) { s_2 e_2; } }

do s while (e) $\Longrightarrow$ { s while (e) s }

e_1 = e_2 $\Longrightarrow$ __assign(e_1,e_2)

*e $\Longrightarrow$ __star(e)
e→x $\Longrightarrow$ __star(e).x
e_1[e_2] $\Longrightarrow$ __arrayelt(e_1,e_2)

e_1 + e_2 $\Longrightarrow$ __add(e_1,e_2)
e_1 - e_2 $\Longrightarrow$ __sub(e_1,e_2)

- e $\Longrightarrow$ __sub(e)

++e $\Longrightarrow$ __increment(e)
--e $\Longrightarrow$ __decrement(e)

e_1 * e_2 $\Longrightarrow$ __star(e_1,e_2)
e_1 / e_2 $\Longrightarrow$ __div(e_1,e_2)
e_1 % e_2 $\Longrightarrow$ __mod(e_1,e_2)

e_1 == e_2 $\Longrightarrow$ __equal(e_1,e_2)
e_1 != e_2 $\Longrightarrow$ __not_equal(e_1,e_2)

e_1 < e_2 $\Longrightarrow$ __less_than(e_1,e_2)
e_1 <= e_2 $\Longrightarrow$ __less_equal(e_1,e_2)
e_1 > e_2 $\Longrightarrow$ __greater_than(e_1,e_2)
e_1 >= e_2 $\Longrightarrow$ __greater_equal(e_1,e_2)

e_1 and e_2 $\Longrightarrow$ __and(e_1,e_2)
e_1 or e_2 $\Longrightarrow$ __or(e_1,e_2)
not e $\Longrightarrow$ __not(e)

e_1 << e_2 $\Longrightarrow$ __output(e_1,e_2)
e_1 >> e_2 $\Longrightarrow$ __input(e_1,e_2)

B

Definition of $F^{\mathcal{G}}$

The syntax of $F^{\mathcal{G}}$ is defined below. The language $F^{\mathcal{G}}$ is an extension of System F (refer to Section 7.1 for the definition of System F) that captures the core features for generic programming: concepts with associated types, models, and generic functions with `where` clauses.

$$
\begin{array}{lll}
c & \in & \text{Concept Names} \\
s, t & \in & \text{Type Variables} \\
x, y & \in & \text{Term Variables} \\
\rho, \sigma, \tau & ::= & t \mid \texttt{fun}\ \overline{\tau} \to \tau \mid \forall \bar{t}\ \texttt{where}\ \overline{c\texttt{<}\overline{\sigma}\texttt{>}}; \overline{\sigma = \tau}.\ \tau \\
 & \mid & c\texttt{<}\overline{\tau}\texttt{>}.t \\
e & ::= & x \mid e(\overline{e}) \mid \lambda \overline{y : \tau}.\ e \\
 & \mid & \Lambda \bar{t}\ \texttt{where}\ \overline{c\texttt{<}\overline{\sigma}\texttt{>}}; \overline{\sigma = \tau}.\ e \mid e[\overline{\tau}] \\
 & \mid & \texttt{concept}\ c\texttt{<}\bar{t}\texttt{>}\ \{ \\
 & & \quad \texttt{types}\ \overline{s};\ \ \texttt{refines}\ \overline{c\texttt{<}\overline{\sigma}\texttt{>}}; \\
 & & \quad \overline{x : \tau};\ \ \overline{\sigma = \tau}; \\
 & & \} \ \texttt{in}\ e \\
 & \mid & \texttt{model}\ c\texttt{<}\overline{\tau}\texttt{>}\ \{ \\
 & & \quad \texttt{types}\ \overline{t = \sigma}; \\
 & & \quad \overline{x = e}; \\
 & & \} \ \texttt{in}\ e \\
 & \mid & c\texttt{<}\overline{\tau}\texttt{>}.x \\
 & \mid & \texttt{type}\ t = \tau\ \texttt{in}\ e
\end{array}
$$

Figure 1 defines the type system for F$^{\mathcal{G}}$ and defines the semantics of F$^{\mathcal{G}}$ in terms of System F. Several auxiliary functions are used in Figure 1 and they are defined as follows.

$\flat^a(c, \overline{\tau}) =$
  $S := \overline{s : c\texttt{<}\overline{\tau}\texttt{>}.s}$
  for $i = 0, \ldots, |\overline{c'}| - 1$
    $S := S, \flat^a(c'_i, S(\overline{\tau'}_i))$
  return S
where
  $\texttt{concept}\ c\texttt{<}\bar{t}\texttt{>}\{\texttt{types}\ \overline{s}\ ;\ \texttt{refines}\ \overline{c'\texttt{<}\overline{\tau'}\texttt{>}};\ \overline{x : \sigma};\ \overline{\rho = \rho'}\} \in \Gamma$

$\flat(c, \overline{\tau}, \overline{n}, \Gamma) =$
  $S := \flat^a(c, \overline{\tau}), \overline{t : \tau}$
  $M := \emptyset$
  for $i = 0, \ldots, |\overline{c'}| - 1$
    $M := M \cup \flat(c'_i, S(\overline{\tau'}_i), (\overline{n}, i), \Gamma)$
  for $i = 0, \ldots, |\overline{x}| - 1$
    $M := M \cup \{x_i : (S(\sigma_i), (\overline{n}, |\overline{c'}| + i))\}$
  return M
where
  $\texttt{concept}\ c\texttt{<}\bar{t}\texttt{>}\{\texttt{types}\ \overline{s}\ ;\ \texttt{refines}\ \overline{c'\texttt{<}\overline{\tau'}\texttt{>}};\ \overline{x : \sigma};\ \overline{\rho = \rho'}\} \in \Gamma$

$\flat^m(c, \overline{\rho}, d, \overline{n}, \Gamma) =$
  check $\Gamma \vdash \overline{\rho} \rightsquigarrow -$ and generate fresh variables $\overline{s'}$

FIGURE 1. Semantics of F$^{\mathcal{G}}$ defined by translation to System F.

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

$$\text{(CPT)}\ \frac{\begin{array}{c}\text{distinct } \overline{t} \qquad \text{distinct } \overline{s} \qquad \overline{\texttt{concept } c'\texttt{<}\overline{t'}\texttt{>}\{\ldots\} \mapsto \delta'} \in \Gamma \qquad \Gamma, \overline{t}, \overline{s} \vdash \overline{\rho} \rightsquigarrow \overline{\rho'} \\ \Gamma, \overline{t}, \overline{s} \vdash \overline{\tau} \rightsquigarrow \overline{\tau'} \qquad \Gamma, \overline{t}, \overline{s} \vdash \overline{\sigma} \rightsquigarrow \overline{\nu} \qquad \Gamma, \overline{t}, \overline{s} \vdash \overline{\sigma'} \rightsquigarrow \overline{\nu'} \\ \delta = (\overline{[\overline{t'} \mapsto \overline{\rho'}]\delta'})@\overline{\tau'} \\ \Gamma, (\texttt{concept } c\texttt{<}\overline{t}\texttt{>}\{\texttt{types } \overline{s}\ ;\ \texttt{refines } \overline{c'\texttt{<}\overline{\rho}\texttt{>}};\ \overline{x : \tau};\ \overline{\sigma = \sigma'}\} \mapsto \delta) \vdash e : \tau \rightsquigarrow f\end{array}}{\Gamma \vdash \texttt{concept } c\texttt{<}\overline{t}\texttt{>}\{\texttt{types } \overline{s}\ ;\ \texttt{refines } \overline{c'\texttt{<}\overline{\rho}\texttt{>}};\ \overline{x : \tau};\ \overline{\sigma = \sigma'}\}\ \texttt{in } e : \tau \rightsquigarrow f}$$

$$\text{(MDL)}\ \frac{\begin{array}{c}\texttt{concept } c\texttt{<}\overline{t}\texttt{>}\{\texttt{types } \overline{s'}\ ;\ \texttt{refines } \overline{c'\texttt{<}\overline{\rho'}\texttt{>}};\ \overline{x : \tau};\ \overline{\eta = \eta'}\} \mapsto \delta \in \Gamma \qquad \Gamma \vdash \overline{\rho} \rightsquigarrow \overline{\tau'} \\ \Gamma \vdash \overline{\nu} \rightsquigarrow \overline{\nu'} \qquad \Gamma \vdash \overline{e : \sigma} \rightsquigarrow \overline{f} \qquad \overline{s'} \subseteq \overline{s} \\ S = \overline{t : \rho}, \overline{s' : [\overline{s} \mapsto \overline{\nu}]s'} \qquad \Gamma \vdash \overline{\texttt{model } c'\texttt{<}S(\overline{\rho'})\texttt{>} \mapsto (d', \overline{n}, A')} \in \Gamma \\ S' = S, \cup\overline{A'} \qquad \overline{x} \subseteq \overline{y} \qquad \Gamma \vdash \overline{[\overline{y} \mapsto \overline{\sigma}]\overline{x} = S'(\tau)} \qquad \Gamma \vdash \overline{S'(\eta) = S'(\eta')} \\ d \text{ fresh} \qquad \Gamma, (\texttt{model } c\texttt{<}\overline{\rho}\texttt{>} \mapsto (d, [], (\cup\overline{A'}, \overline{s' : [\overline{s} \mapsto \overline{\nu'}]s'}))) \vdash e : \tau \rightsquigarrow f \\ \overline{d''} = \overline{(\ \texttt{nth}\ \ldots (\ \texttt{nth}\ d'\ n_1) \ldots n_k)}\end{array}}{\Gamma \vdash \texttt{model } c\texttt{<}\overline{\rho}\texttt{>}\ \{\ \texttt{types } \overline{s = \nu}; \overline{y = e}\}\ \texttt{in } e : \tau \rightsquigarrow \texttt{let } d = (\overline{d''}@[\overline{y} \mapsto \overline{f}]\overline{x})\ \texttt{in } f}$$

$$\text{(TABS)}\ \frac{\text{distinct } \overline{t} \qquad \overline{t} \cap \mathrm{FTV}(\Gamma) = \emptyset \qquad (\Gamma', \overline{s}, \overline{\delta}) = \flat^w(\overline{c\texttt{<}\overline{\rho}\texttt{>}}, (\Gamma, \overline{t})) \qquad \Gamma', \overline{\tau = \tau'} \vdash e : \tau \rightsquigarrow f}{\Gamma \vdash \Lambda\overline{t}\ \texttt{where } \overline{c\texttt{<}\overline{\rho}\texttt{>}}, \overline{\tau = \tau'}.\ e : \forall\overline{t}\ \texttt{where } \overline{c\texttt{<}\overline{\rho}\texttt{>}}, \overline{\tau = \tau'}.\ \tau \rightsquigarrow \Lambda\overline{t}, \overline{s}.\ \lambda\overline{d : \delta}.\ f}$$

$$\text{(TAPP)}\ \frac{\begin{array}{c}\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \qquad \Gamma \vdash e : \forall\overline{t}\ \texttt{where } \overline{c\texttt{<}\overline{\rho}\texttt{>}}, \overline{\eta = \eta'}.\ \tau \rightsquigarrow f \\ \Gamma \vdash \overline{\texttt{model } c\texttt{<}[\overline{t} \mapsto \overline{\sigma}]\overline{\rho}\texttt{>} \mapsto (d, \overline{n}, \overline{s : \nu})} \in \Gamma \qquad \Gamma \vdash \overline{[\overline{t} \mapsto \overline{\sigma}]\eta = [\overline{t} \mapsto \overline{\sigma}]\eta'}\end{array}}{\Gamma \vdash e[\overline{\sigma}] : [\overline{t} \mapsto \overline{\sigma}]\tau \rightsquigarrow f[\overline{\sigma'}, \overline{\nu}](\overline{\ \texttt{nth}\ \ldots (\ \texttt{nth}\ d\ n_1) \ldots n_k})}$$

$$\text{(ALS)}\ \frac{t \notin \mathrm{FTV}(\Gamma) \qquad \Gamma, t = \tau \vdash e : \tau \rightsquigarrow f}{\Gamma \vdash \texttt{type } t = \tau\ \texttt{in } e : \tau \rightsquigarrow f}$$

$$\text{(APP)}\ \frac{\Gamma \vdash e_1 : \texttt{fun } \overline{\sigma} \to \tau \rightsquigarrow f_1 \qquad \Gamma \vdash \overline{e_2 : \sigma' \rightsquigarrow f_2} \qquad \Gamma \vdash \overline{\sigma = \sigma'}}{\Gamma \vdash e_1\ \overline{e_2} : \tau \rightsquigarrow f_1(\overline{f_2})}$$

$\Gamma := \Gamma, \overline{s' = c\texttt{<}\overline{\rho}\texttt{>}.s}$
$A := \flat^a(c, \overline{\rho}), \overline{t : \rho}$
$\overline{s''} := []$; $\overline{\tau} := []$
for $i = 0, \ldots, |\overline{c'}| - 1$
  $(\Gamma, \overline{a}, \delta') := \flat^m(c'_i, A(\overline{\rho'}_i), d, (\overline{n}, i), \Gamma)$
  $\overline{s''} := \overline{s''}, \overline{a}$; $\overline{\tau} := \overline{\tau}, \delta'$
$\overline{\tau} := \overline{\tau}@A(\overline{\sigma})$
$\Gamma := \Gamma, \overline{A(\eta) = A(\eta')}$

FIGURE 2. Type equality for F$^{\mathcal{G}}$.

$$\text{(REFL)}\frac{}{\Gamma \vdash \tau = \tau} \qquad \text{(SYMM)}\frac{\Gamma \vdash \sigma = \tau}{\Gamma \vdash \tau = \sigma} \qquad \text{(TRANS)}\frac{\Gamma \vdash \sigma = \rho \qquad \Gamma \vdash \rho = \tau}{\Gamma \vdash \sigma = \tau}$$

$$\text{(HYP)}\frac{\sigma = \tau \in \Gamma}{\Gamma \vdash \sigma = \tau} \qquad \text{(FNEQ)}\frac{\Gamma \vdash \overline{\sigma} = \overline{\tau} \qquad \Gamma \vdash \sigma = \tau}{\Gamma \vdash \texttt{fun}\ \overline{\sigma} \to \sigma = \texttt{fun}\ \overline{\tau} \to \tau} \qquad \text{(ASCEQ)}\frac{\Gamma \vdash \overline{\sigma} = \overline{\tau}}{\Gamma \vdash c\texttt{<}\overline{\sigma}\texttt{>}.t = c\texttt{<}\overline{\tau}\texttt{>}.t}$$

$$\text{(ALLEQ)}\frac{\begin{array}{c}\Gamma \vdash \overline{\rho_1} = [\overline{t_1}/\overline{t_2}]\overline{\rho_2} \qquad \Gamma \vdash \overline{\sigma_1} = [\overline{t_1}/\overline{t_2}]\overline{\sigma_2} \qquad \Gamma \vdash \overline{\tau_1} = [\overline{t_1}/\overline{t_2}]\overline{\tau_2} \\ \Gamma, \overline{\sigma_1 = \tau_1} \vdash \tau_3 = [\overline{t_1}/\overline{t_2}]\tau_4\end{array}}{\Gamma \vdash \forall \overline{t_1}\ \texttt{where}\ \overline{c\texttt{<}\overline{\rho_1}\texttt{>}}; \overline{\sigma_1 = \tau_1}.\ \tau_3 = \forall \overline{t_2}\ \texttt{where}\ \overline{c\texttt{<}\overline{\rho_2}\texttt{>}}; \overline{\sigma_2 = \tau_2}.\ \tau_4}$$

$\Gamma := \Gamma, \texttt{model}\ c\texttt{<}\overline{\rho}\texttt{>} \mapsto (d, \overline{n}, \flat^a(c, \overline{\rho}))$
return $(\Gamma, (\overline{s''}, \overline{s'}), \overline{\tau})$
where
$\texttt{concept}\ c\texttt{<}\overline{t}\texttt{>}\{\texttt{types}\ \overline{s}\ ;\ \texttt{refines}\ \overline{c'\texttt{<}\overline{\rho'}\texttt{>}};\ \overline{x : \sigma};\ \overline{\eta = \eta'}\} \in \Gamma$

$\flat^w([], \Gamma) = (\Gamma, [])$
$\flat^w((c\texttt{<}\overline{\rho}\texttt{>}, \overline{c'\texttt{<}\overline{\rho'}\texttt{>}}), \Gamma) =$
generate fresh d
$(\Gamma, \overline{s}, \delta) := \flat^m(c, \overline{\rho}, d, [], \Gamma)$
$(\Gamma, \overline{s'}, \overline{\delta'}) := \flat^w(\overline{c'\texttt{<}[\overline{t} \mapsto \overline{\rho}]\overline{\rho'}\texttt{>}}, \Gamma)$
return $(\Gamma, (\overline{s}, \overline{s'}), (\delta, \overline{\delta'}))$
where
$\texttt{concept}\ c\texttt{<}\overline{t}\texttt{>}\{\texttt{types}\ \overline{s}\ ;\ \texttt{refines}\ \overline{c'\texttt{<}\overline{\rho'}\texttt{>}};\ \overline{x : \sigma};\ \overline{\eta = \eta'}\} \in \Gamma$

Type equality in F$^{\mathcal{G}}$ is defined in Figure 2.

Bibliography

[1] *Ada 95 Reference Manual*, 1997.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[3] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams Iv, D. P. Friedman, E. Kohlbecker, Jr. G. L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[4] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[5] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[6] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[7] Konstantine Arkoudas. *Denotational Proof Languages*. PhD thesis, MIT, 2000.

[8] David Aspinall. Proof general: A generic tool for proof development. In *(TACAS 2000) Tools and Algorithms for the Construction and Analysis of Systems*, number 1785 in LNCS, 2000.

[9] Matt Austern. (draft) technical report on standard library extensions. Technical Report N1711=04-0151, ISO/IEC JTC 1, Information Technology, Subcommittee SC

22, Programming Language C++, 2004.

[10] Matt Austern. Proposed draft technical report on C++ library extensions. Technical Report PDTR 19768, n1745 05-0005, ISO/IEC, January 2005.

[11] Matthew H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.

[12] Bruno Bachelet, Antoine Mahul, and Loïc Yon. Designing Generic Algorithms for Operations Research. *Software: Practice and Experience*, 2005. submitted.

[13] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[14] H.P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic*. Elsevier, 1984.

[15] Bruce H. Barnes and Terry B. Bollinger. Making reuse cost-effective. *IEEE Software*, 8(1):13–24, 1991.

[16] John Bartlett. *Familiar Quotations*. Little Brown, 1919.

[17] Hamid Abdul Basit, Damith C. Rajapakse, and Stan Jarzabek. Beyond templates: a study of clones in the STL and some general implications. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 451–459, New York, NY, USA, 2005. ACM Press.

[18] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[19] K. L. Bernstein and E. W. Stark. Debugging type errors. Technical report, State University of New York at Stony Brook, 1995.

[20] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. Technical report, Pittsburgh, PA, USA, 1993.

[21] Jean-Daniel Boissonnat, Frederic Cazals, Frank Da, Olivier Devillers, Sylvain Pion, Francois Rebufat, Monique Teillaud, and Mariette Yvinec. Programming with CGAL: the example of triangulations. In *Proceedings of the fifteenth annual symposium on Computational geometry*, pages 421–422. ACM Press, 1999.

[22] Boost. *Boost C++ Libraries*. `http://www.boost.org/`.

[23] Richard Bornat. Proving pointer programs in hoare logic. In *MPC '00: Proceedings of the 5th International Conference on Mathematics of Program Construction*, pages 102–126, London, UK, 2000. Springer-Verlag.

[24] Didier Le Botlan and Didier Remy. MLF: raising ML to the power of system F. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 27–38, New York, NY, USA, 2003. ACM Press.

[25] Nicolas Bourbaki. *Elements of Mathematics. Theory of Sets*. Springer, 1968.

[26] Chandrasekhar Boyapati, Alexandru Salcianu, Jr. William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM Press.

[27] John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 283–295, New York, NY, USA, 2005. ACM Press.

[28] Kim B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety. Technical report, Williams College, 1996.

[29] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[30] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127. Springer-Verlag, 1997.

[31] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. In *Proceedings of the international symposium on Semantics of data types*, pages 1–50, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[32] Rod M. Burstall and Joseph A. Goguen. Putting theories together to make specifications. In *IJCAI*, pages 1045–1058, 1977.

[33] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280, New York, NY, USA, 1989. ACM Press.

[34] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.

[35] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[36] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI*, June 1991.

[37] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 56–71, London, UK, 2000. Springer-Verlag.

[38] Manuel M. T. Chakravarty, Gabrielle Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, New York, NY, USA, 2005. ACM Press.

[39] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 146–160, New York, NY, USA, 1989. ACM Press.

[40] Craig Chambers and the Cecil Group. *The Cecil Language: Specification and Rationale, Version 3.1*. University of Washington, Computer Science and Engineering, December 2002. `http://www.cs.washington.edu/research/projects/cecil/`.

[41] Craig Chambers and David Ungar. Interative type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 150–164, New York, NY, USA, 1990. ACM Press.

[42] Chung chieh Shan. Sexy types in action. *SIGPLAN Notices*, 39(5):15–22, 2004.

[43] Olaf Chitil, Frank Huch, and Axel Simon. Typeview: A tool for understanding type errors. In *12th International Workshop on Implementation of Functional Languages*, 2000.

[44] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64, New York, NY, USA, 1998. ACM Press.

[45] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[46] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, Reading, MA, 2002.

[47] CoFI Language Design Task Group. CASL—the CoFI algebraic specification language—summary, 2001. `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.

[48] William R. Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):304–311, 1989.

[49] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[50] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 2000.

[51] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[52] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[53] Glen Jeffrey Ditchfield. Overview of Cforall. University of Waterloo, August 1996.

[54] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM (JACM)*, 27(4):758–771, 1980.

[55] Pavol Droba. Boost string algorithms library, July 2004. `http://www.boost.org/doc/html/string_algo.html`.

[56] R. Kent Dybvig. *The Scheme Programming Language: ANSI Scheme*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1996.

[57] H. Eichelberger and J. Wolff v. Gudenberg. UML description of the STL. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[58] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[59] Erik Ernst. Family polymorphism. In *ECOOP '01*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, June 2001.

[60] Manuel Fahndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 13–24, New York, NY, USA, 2002. ACM Press.

[61] A.D. Falkoff and D.L. Orth. Development of an apl standard. Technical Report RC 7542, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, February 1979.

[62] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, New York, NY, USA, 2002. ACM Press.

[63] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Behavioral contracts and behavioral subtyping. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 229–236, New York, NY, USA, 2001. ACM Press.

[64] Jr. Frederick P. Brooks. *The Mythical Man-Month: Essays on Softw*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[65] Daniel P. Friedman and Matthias Felleisen. *The Little Schemer*. MIT Press, fourth edition, 1996.

[66] B. A. Galler and A. J. Perlis. A proposal for definitions in ALGOL. *Communications of the ACM*, 9(7):481–482, 1966.

[67] B. A. Galler and A. J. Perlis. *A View of Programming Languages*. Computer science and information processing. Addison-Wesley, 1970.

[68] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[69] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 115–134, New York, NY, USA, 2003. ACM Press.

[70] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 2005. submitted.

[71] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, Paris, France, 1972.

[72] J. A. Goguen. Parameterized programming and software architecture. In *ICSR '96: Proceedings of the 4th International Conference on Software Reuse*, page 2, Washington, DC, USA, 1996. IEEE Computer Society.

[73] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[74] Joseph A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-IO, No(5):528–543, September 1984.

[75] Miguel Guerrero, Edward Pizzi, Robert Rosenbaum, Kedar Swadi, and Walid Taha. Implementing DSLs in metaOCaml. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 41–42, New York, NY, USA, 2004. ACM Press.

[76] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[77] John V. Guttag, Ellis Horowitz, and David R. Musser. The design of data type specifications. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 414–420, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[78] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[79] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141, New York, NY, USA, 1995. ACM Press.

[80] Bastiaan Heeren, Johan Jeuring, Doaitse Swierstra, and Pablo Azero Alcocer. Improving type-error messages in functional languages. Technical report, Utrecht Univesity, February 2002.

[81] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 73–84, New York, NY, USA, 2004. ACM Press.

[82] Ralf Hinze. A simple implementation technique for priority search queues. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 110–121, New York, NY, USA, 2001. ACM Press.

[83] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[84] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16:14–21, 1951.

[85] Mark Howard, Eric Bezault, Bertrand Meyer, Dominique Colnet, Emmanuel Stapf, Karine Arnout, and Markus Keller. Type-safe covariance: competent compilers can catch all catcalls. `http://www.inf.ethz.ch/~meyer/`, April 2003.

[86] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. Geneva, Switzerland, September 1998.

[87] Kenneth E. Iverson. Operators. *ACM Trans. Program. Lang. Syst.*, 1(2):161–176, 1979.

[88] Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 193–205, New York, NY, USA, 1996. ACM Press.

[89] Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An analysis of constrained polymorphism for generic programming. In Kei Davis and Jörg Striegnitz, editors, *Multiparadigm Programming in Object-Oriented Languages Workshop (MPOOL) at OOPSLA*, Anaheim, CA, October 2003.

[90] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Algorithm specialization and concept constrained genericity. In *Concepts: a Linguistic Foundation of Generic Programming*. Adobe Systems, April 2004.

[91] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, 2005. To appear.

[92] Mehdi Jazayeri, Rüdiger Loos, David Musser, and Alexander Stepanov. Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, Schloss Dagstuhl, Germany, April 1998.

[93] Richard D. Jenks and Barry M. Trager. A language for computational algebra. In *SYMSAC '81: Proceedings of the fourth ACM symposium on Symbolic and algebraic computation*, pages 6–13, New York, NY, USA, 1981. ACM Press.

[94] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[95] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[96] Mark P. Jones. *Qualified Types: Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[97] Mark P. Jones. First-class polymorphism with type inference. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 483–496, New York, NY, USA, 1997. ACM Press.

[98] M.P. Jones. Dictionary-free overloading by partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 107–117, 1994.

[99] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[100] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. submitted to the Journal of Functional Programming, April 2004.

[101] D. Kapur and D. Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report RPI–92–20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180, July 1992.

[102] D. Kapur, D. R. Musser, and X. Nie. An overview of the tecton proof system. *Theoretical Computer Science*, 133:307–339, October 1994.

[103] D. Kapur, D. R. Musser, and A. A. Stepanov. Tecton: A language for manipulating generic objects. In J. Staunstrup, editor, *Proceedings of a Workshop on Program Specification*, volume 134 of *LNCS*, pages 402–414, Aarhus, Denmark, August 1981. Springer.

[104] Deepak Kapur, David R. Musser, and Alexander Stepanov. Operators and algebraic structures. In *Proc. of the Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*. ACM, 1981.

[105] A. Kershenbaum, D. Musser, and A. Stepanov. Higher order imperative programming. Technical Report 88-10, Rensselaer Polytechnic Institute, 1988.

[106] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with ASPECTJ. *Communications of the ACM*, 44(10):59–65, 2001.

[107] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM Press.

[108] Ullrich Köthe. *Handbook on Computer Vision and Applications*, volume 3, chapter Reusable Software in Computer Vision. Acadamic Press, 1999.

[109] Bernd Krieg-Brückner and David C. Luckham. ANNA: towards a language for annotating ada programs. In *SIGPLAN '80: Proceeding of the ACM-SIGPLAN symposium on Ada programming language*, pages 128–138, New York, NY, USA, 1980. ACM Press.

[110] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Abstraction mechanisms in the BETA programming language. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 1983. ACM Press.

[111] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[112] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.

[113] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The Generic Graph Component Library. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 399–414, New York, NY, USA, 1999. ACM Press.

[114] Xavier Leroy. Unboxed objects and polymorphic typing. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 177–188, New York, NY, USA, 1992. ACM Press.

[115] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jerome Vouillon. *The Objective Caml Documentation and User's Manual*, September 2003.

[116] Wayne C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Softw.*, 11(5):23–30, 1994.

[117] Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. CLU reference manual. Technical Report LCS-TR-225, Cambridge, MA, USA, October 1979.

[118] B.H. Liskov and S. N. Zilles. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering*, SE-1(1):7–18, March 1975.

[119] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of aop with generative programming in aspectc++. In G. Karsai and E. Visser, editors, *Generative Programming and Component Engineering*, number 3286 in LNCS, pages 55–74, Heidelberg, 2004. Springer-Verlag.

[120] Andrew Lumsdaine and Brian C. McCandless. The matrix template library. BLAIS Working Note #2, University of Notre Dame, 1996.

[121] Andrew Lumsdaine and Brian C. McCandless. The role of abstraction in high performance computing. In *Proceedings, 1997 Internantional Conference on Scientific Computing in Object-Oriented Parallel Computing*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[122] John Maddock. A proposal to add regular expressions to the standard library. Technical Report J16/03-0011= WG21/N1429, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, March 2003. `http://www.open-std.org/jtc1/sc22/wg21`.

[123] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York,

NY, USA, 1989. ACM Press.

[124] Boris Magnusson. Code reuse considered harmful. *Journal of Object-Oriented Programming*, 4(3), November 1991.

[125] Johan Margono and Thomas E. Rhoads. Software reuse economics: cost-benefit analysis on a large-scale ada project. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 338–348, New York, NY, USA, 1992. ACM Press.

[126] M. Douglas McIlroy. Mass-produced software components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Proceedings of Software Engineering Concepts and Techniques, 1968 NATO Conference on Software Engineering*, pages 138–155, January 1969. `http://www.cs.dartmouth.edu/~doug/components.txt`.

[127] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1997.

[128] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[129] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2-3):211–249, 1988.

[130] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[131] James H. Morris, Jr. Types are not sets. In *Conference Record of ACM Symposium on Principles of Programming Languages*, pages 120–124, New York, 1973. ACM.

[132] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[133] David R. Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27(8):983–993, 1997.

[134] David R. Musser. Formal methods for generic libraries or toward semantic concept checking. In *Workshop on Software Libraries: Design and Evaluation*, Dagstuhl, Germany, March 2005. `http://www.cs.chalmers.se/~tveldhui/tmp/lwg/proceedings/DavidMusser.pdf`.

[135] David R. Musser. Generic programming and formal methods. In *Workshop on The Verification Grand Challenge*, Menlo Park, CA, February 2005. `http://www.csl.sri.com/users/shankar/VGC05/`.

[136] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 2nd edition, 2001.

[137] David R. Musser and Alex Stepanov. Generic programming. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, 1988.

[138] David R. Musser and Alexander A. Stepanov. A library of generic algorithms in Ada. In *Using Ada (1987 International Ada Conference)*, pages 216–225, New York, NY, December 1987. ACM SIGAda.

[139] David R. Musser and Alexander A. Stepanov. Generic programming. In P. (Patrizia) Gianni, editor, *Symbolic and algebraic computation: ISSAC '88, Rome, Italy, July 4–8, 1988: Proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Berlin, 1989. Springer Verlag.

[140] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[141] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall Series in Innovative Technology. Prentice Hall, 1991.

[142] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[143] Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646, pages 259–278, 2003.

[144] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[145] Object Management Group. *OMG Unified Modeling Language Specification*, 1.5 edition, March 2003.

[146] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[147] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, 2003.

[148] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–67, New York, NY, USA, 1996. ACM Press.

[149] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[150] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 2004. submitted.

[151] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[152] W. R. Pitt, M. A. Williams, M. Steven, B. Sweeney, A. J. Bleasby, and D. S. Moss. The bioinformatics template library: generic components for biocomputing. *Bioinformatics*, 17(8):729–737, 2001.

[153] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[154] B. Randell. Software engineering in 1968. In *ICSE '79: Proceedings of the 4th international conference on Software engineering*, pages 1–10, Piscataway, NJ, USA, 1979. IEEE Press.

[155] Didier Remy. Exploring partial type inference for predicative fragments of system-F. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, September 2005. ACM Press.

[156] Nicolas Remy. GsTL: The geostatistical template library in C++. Master's thesis, Stanford University, March 2001. `http://pangea.stanford.edu/~nremy/GTL/`.

[157] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *LNCS*, pages 408–425, Berlin, 1974. Springer-Verlag.

[158] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[159] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

[160] Graziano Lo Russo. An interview with a. stepanov. `http://www.stlport.org/resources/StepanovUSA.html`.

[161] Owre Sam and Shankar Natarajan. Theory interpretations in PVS. Technical report, 2001.

[162] Sriram Sankar, David Rosenblum, and Randall Neff. An implementation of anna. In *SIGAda '85: Proceedings of the 1985 annual ACM SIGAda international conference on Ada*, pages 285–296, New York, NY, USA, 1985. Cambridge University Press.

[163] Sibylle Schupp, Douglas Gregor, David R. Musser, and Shin-Ming Liu. User-extensible simplification: Type-based optimizer generators. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 86–101, London, UK, 2001. Springer-Verlag.

[164] Christoph Schwarzweller. Towards formal support for generic programming. `http://www.math.univ.gda.pl/~schwarzw`, 2003. Habilitation thesis, Wilhelm-Schickard-Institute for Computer Science, University of Tübingen.

[165] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Haskell '02: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM Press.

[166] Jeremy Siek. A modern framework for portable high performance numerical linear algebra. Master's thesis, University of Notre Dame, 1999.

[167] Jeremy Siek. *Boost Concept Check Library*. Boost, 2000. `http://www.boost.org/libs/concept_check/`.

[168] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concepts for C++0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, January 2005.

[169] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[170] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.

[171] Jeremy Siek and Andrew Lumsdaine. Essential language support for generic programming: Formalization part 1. Technical Report 605, Indiana University, December 2004.

[172] Jeremy Siek and Andrew Lumsdaine. Essential language support for generic programming. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, pages 73–84, New York, NY, USA, June 2005. ACM Press.

[173] Jeremy Siek and Andrew Lumsdaine. Language requirements for large-scale generic libraries. In *GPCE '05: Proceedings of the fourth international conference on Generative Programming and Component Engineering*, September 2005. accepted for publication.

[174] Jeremy G. Siek and Andrew Lumsdaine. *Advances in Software Tools for Scientific Computing*, chapter A Modern Framework for Portable High Performance Numerical Linear Algebra. Springer, 2000.

[175] Raul Silaghi and Alfred Strohmeier. Better generative programming with generic aspects. Technical report, Swiss Federal Institute of Technology in Lausanne, December 2003. http://icwww.epfl.ch/publications/abstract.php?ID=200380.

[176] Silicon Graphics, Inc. *SGI Implementation of the Standard Template Library*, 2004. http://www.sgi.com/tech/stl/.

[177] Richard Soley and the OMG Staff Strategy Group. Model driven architecture. Technical report, Object Management Group, November 2000. http://www.omg.org/~soley/mda.html.

[178] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[179] Alexander Stepanov. gclib. http://www.stepanovpapers.com, 1987.

[180] Alexander A. Stepanov, Aaron Kershenbaum, and David R. Musser. Higher order programming. http://www.stepanovpapers.com/Higher%20Order%20Programming.

pdf, March 1987.

[181] Alexander A. Stepanov and Meng Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[182] Christopher Strachey. Fundamental concepts in programming languages, August 1967.

[183] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. Technical report, 1999.

[184] Robert Endre Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[185] J. W. Thatcher, E. G. Wagner, and J. B. Wright. Data type specification: Parameterization and the power of specification techniques. *ACM Trans. Program. Lang. Syst.*, 4(4):711–732, 1982.

[186] Kresten Krab Thorup. Genericity in Java with virtual types. In *ECOOP '97*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471, 1997.

[187] Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179(1):1–18, 2002.

[188] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[189] Mads Torgersen. Virtual types are statically safe. In *FOOL 5: The Fifth International Workshop on Foundations of Object-Oriented Languages*, January 1998.

[190] Matthias Troyer, Synge Todo, Simon Trebst, and Alet Fabien and. *ALPS: Algorithms and Libraries for Physics Simulations*. `http://alps.comp-phys.org/`.

[191] Franklyn Turbak, Allyn Dimock, Robert Muller, and J. B. Wells. Compiling with polymorphic and polyvariant flow types.

[192] B. L. van der Waerden. *Algebra*. Frederick Ungar Publishing, 1970.

[193] Todd L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, volume 1505 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[194] Friedrich W. von Henke, David Luckham, Bernd Krieg-Brueckner, and Olaf Owe. Semantic specification of ada packages. In *SIGAda '85: Proceedings of the 1985 annual ACM SIGAda international conference on Ada*, pages 185–196, New York, NY, USA, 1985. Cambridge University Press.

[195] Oscar Waddell and R. Kent Dybvig. Fast and effective procedure inlining. In *Proceedings of the Fourth International Symposium on Static Analysis (SAS '97)*, volume 1302 of *Lecture Notes in Computer Science*, pages 35–52. Springer-Verlag, September 1997.

[196] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[197] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[198] Joerg Walter and Mathias Koch. *uBLAS*. Boost. `http://www.boost.org/libs/numeric/ublas/doc/index.htm`.

[199] M. Wenzel. Using axiomatic type classes in Isabelle (manual), 1995. `www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`.

[200] Jeremiah Willcock, Jaakko Järvi, Andrew Lumsdaine, and David Musser. A formalization of concepts for generic programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit*. Adobe Systems, April 2004.

[201] J. Yang, J. Wells, P. Trinder, and G. Michaelson. Improved type error reporting, 2000.

[202] Hongyu Zhang and Stan Jarzabek. XVCL: a mechanism for handling variants in software product lines. *Science of Computer Programming*, 53(3):381–407, 2004.

[203] S.N. Zilles. Algebraic specification of data types. Technical Report Project MAC Progress Report 11, Mass. Inst. Technology, 1975.

Index

$\mathcal{LOOM}$, 80

ML^{F}, 104

`find_end`, 47

`iterator_traits`, 27

`replace_copy`, 36

`reverse_iterator`, 43

`where` clause, 93

Bidirectional Iterator, 30

Binary Function, 39

Forward Iterator, 30

Input Iterator, 18, 28

Output Iterator, 29

Random Access Iterator, 30

`accumulate`, 36

`advance`, 38

`merge`, 34

`min`, 23

`stable_sort`, 31

`unique`, 30

`count`, 25

`deque`, 41

`map`, 41

`multimap`, 41

`multiset`, 41

`priority_queue`, 48

`queue`, 47

`set`, 41

`stack`, 47

`vector`, 41

`count`, 25

`list`, 41

abstract base class, 126

abstract data type, 110

accidental conformance, 84

Ada, 86

alias, 184

annotated type, 134

anonymous function, 112

any, 121

archetype classes, 34

argument dependent lookup, 25

associated types, 19, 94, 99

backward chaining, 107

BETA, 63

binary method problem, 60

callable from, 109

Cforall, 83

class, 110

CLU, 83, 121
compilation, 118
complexity guarantees, 19
concept, 95, 126
concept-based overloading, 110, 166
concepts, 17
conditional model, 46, 97
congruence relation, 100
conversion requirements, 38

declaration, 221

environment, 134
equivalence relation, 100
evidence, 120
expression, 221

first-class polymorphism, 113
function
  anonymous, 122
  expressions, 122
  generic, 121
  parameters, 123
  pure virtual, 126
  types, 123
  virtual, 126
function expression, 112
function object, 8, 37
function overloading, 108
function specialization, 69
functor, 86

gbeta, 63, 80
generic function, 92
generics, 9, 55
grammar, 219

higher-order functions, 8
Horn clause, 106

implicit instantiation, 7, 102, 125
implicit model passing, 105
instantiated, 24
intensional type analysis, 72, 125
interface, 83

macro-like parameterization, 65
matching, 80
Maude, 86
ML, 86
model, 96, 126
model head, 106
model lookup, 105
model passing, 86
models, 18
monomorphization 69
more specific model, 107
more specific overload, 109
multi-parameter concept, 35

nominal conformance, 83

OBJ, 86
object types, 83
Objective Caml, 83, 86

parameteric polymorphism, 65
parameterized model, 97
partial evaluation, 70
partial template specialization, 29

Pebble, 86
pointers, 124
predecessor, 135
Prolog, 106
property map, 175

refinement, 26
regions, 184
requirements on associated types, 30

same-type constraints, 35, 93, 99, 178
Scala, 63, 80
scalar replacement of aggregates, 218
separate type checking, 8
separately compiled, 8
signature, 83, 87
statement, 221
struct, 110
structural conformance, 83
structure, 86
subsumption principle, 59, 102
syntax, 219

tag dispatching idiom, 40
template specialization, 27
theory, 53
traits class, 27
type, 220
type class, 83
type sets, 83
type argument deduction, 102
type equality, 98
type expression, 220
type sharing, 82
unification, 106, 144
unify, 144
union, 110

valid expressions, 24
value semantics, 42
virtual classes, 63
virtual patterns, 63
virtual types, 63

Jeremy G. Siek

Open Systems Lab
Department of Computer Science
Indiana University
Bloomington, IN 47405

Tel: (812) 855-3608
Fax: (812) 855-4829
jsiek@osl.iu.edu

Objective To advance both current practice and the state of the art in software engineering in the area of library construction and programming languages with an emphasis on generic programming.

Education

University of Notre Dame 1993–2001
B.S. Mathematics 1997.
M.S. Computer Science and Engineering 1999.

Indiana University 2001–2005
Enrolled in the Ph.D. program. Defended thesis in July 2004.

Awards: NCAA Post-Graduate Scholarship and National Merit Scholar.

Experience

Rice University, Houston *Post-doctoral research associate* 2005–present
Working with Prof. Walid Taha on multi-stage programming and with Prof. Ken Kennedy on research related to telescoping languages

AT&T Labs–Research, Florham Park *Summer Manager* Summer 2001
Worked with Bjarne Stroustrup on the Extended Type Information (XTI) library, a system for compile-time reflection of type information for C++. Applied XTI to the construction of a remote-procedure invocation system.

C++ Standards Committee *Representative for Indiana University* 2001–present
Worked on the iterator concept and iterator adaptor proposals for standard library extension. Worked on a proposal for adding support for generic programming to C++ through the addition of "concepts".

Boost C++ Group *Contributing Member* 1999–present
Worked on the graph, concept check, property map, operator, iterator adaptor, and dynamic bitset libraries.

SGI C++ Compiler Group *Intern* 1999–2000
Worked on the port of the SGI iostream library to Linux. Developed the concept checking library in collaboration with Alexander Stepanov and applied it to the SGI STL implementation. Contributed to bootstrapping the SGI C++ compiler in the IA64 Itanium processor.

Professional Interests Generic library construction, especially in the domains of graph theory and high-performance linear algebra. Language design and implementation for the support of generic programming techniques, including type systems and logics for program correctness.

Selected publications Language Requirements for Large-Scale Generic Libraries. With Andrew Lumsdaine. In GPCE '05: Proceedings of the fourth international conference on Generative Programming and Component Engineering. September, 2005.

Essential Language Support for Generic Programming. With Andrew Lumsdaine. In PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation. June, 2005.

Concepts for C++0x. With Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++. Report number N1758=05-0018. 2005.

A Comparative Study of Language Support for Generic Programming. With Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, and Jeremiah Willcock. In proceedings OOPSLA, 2003.

Improving the Lazy Krivine Machine. With Daniel P. Friedman, Abdulaziz Ghuloum, and Lynn Winebarger. Accepted for publication in Higher-Order and Symbolic Computation, 2003.

The Boost Graph Library: User Guide and Reference. Addison-Wesley. With Lie-Quan Lee and Andrew Lumsdaine. December 20, 2001.

Concept checking: binding parametric polymorphism in C++. Jeremy Siek and Andrew Lumsdaine. In the First Workshop on C++ Template Programming. Erfurt, Germany, October 10, 2000.

The generic graph component library. With Lie-Quan Lee and Andrew Lumsdaine. In Proceedings OOPSLA, 1999.

A Modern Framework for Portable High Performance Numerical Linear Algebra. Jeremy Siek and Andrew Lumsdaine. Chapter in Modern Software tools for Scientific Computer. Birkhauser 1999.

Generic Graph Algorithms for Sparse Matrix Ordering. Lie-Quan Lee and Jeremy G. Siek and Andrew Lumsdaine. In Proceedings of ISCOPE, Lecture Notes in Computer Science Springer-Verlag, 1999.

The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra. Jeremy G. Siek and Andrew Lumsdaine. In the International Symposium on Computing in Object-Oriented Parallel Environments, 1998.