Modular Generics

Jeremy Siek Andrew Lumsdaine
Open Systems Lab
Indiana University Bloomington
Bloomington, IN USA
{jsiek,lums}@osl.iu.edu

ABSTRACT

This paper presents the design of $\mathcal{G}$, a new language specifically created for generic programming. We review and identify important language features of C++ and Haskell in light of the past decade of generic library research and development. Based on this analysis we propose and evaluate relevant language design decisions for $\mathcal{G}$. Generic programming is concerned with the construction of libraries of reusable software components and is inherently about programming "in the large." Thus, the design of $\mathcal{G}$ places its greatest emphasis on modularity and safety, while also providing runtime efficiency and programmer convenience. This paper focuses on name scoping and type checking for generic functions, support for dispatching to algorithm specializations, support for type associations among abstractions, and separate compilation. The resulting design for $\mathcal{G}$ includes three novel aspects: scoped models declarations, nested types in concepts, and optional type constraints on generic functions.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*reusable libraries*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types, constraints, polymorphism*

General Terms

Languages, Design, Standardization

Keywords

Generics, Generic Programming, Polymorphism, C++, Haskell

1. INTRODUCTION

Generic programming is an increasingly popular and important paradigm for the development of software libraries. David Musser and Alexander Stepanov developed the methodology of generic programming in the late 1980's [21,32,33] and applied it to the construction of sequence and graph algorithms in Scheme, Ada, and C. Since then the methodology has evolved to meet the needs of new algorithms and problem domains, and to take advantage of new programming language features, such as templates in C++ [34, 46].

Terminology

Figure 1 gives the definition of generic programming from [16]. Here we review the standard terminology from [4] for the key elements of generic programming.

The notion of abstraction is fundamental to generic programming: generic algorithms are specified in terms of abstract properties of types, not in terms of particular types. A ***concept*** is the formalization of an abstraction as a set of requirements on a type (or several types). These requirements may be semantic as well as syntactic. A concept may incorporate the requirements of another concept, in which case the first concept is said to ***refine*** the second. A type (or list of types) that meets the requirements of a concept is said to ***model*** the concept.

> 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

Concepts are used to specify interfaces to generic algorithms by ***constraining*** the type parameters of an algorithm. A generic algorithm may only be used with type arguments that model its constraining concepts.

Traditionally, a concept consists of associated types, operations, semantic invariants, and complexity guarantees. The associated types of a concept specify mappings from the modeling type to other collaborating types. The operations specify the functionality that must be implemented for the modeling type. At this point in the state of the art, type systems typically do not include semantic invariants and complexity guarantees. In this paper we adopt the convention that for a type to properly model a concept, the associ-

ated types and operations specified by the concept must be defined.

Progress in Generic Library Development

Generic programming has become an integral part of modern C++ programming, especially in the area of library design and implementation. The Standard Template Library (STL) was the first of such C++ libraries [45], and now there are numerous generic C++ libraries covering domains as diverse as computer vision [22], regular expressions [28], numerical linear algebra [27, 44, 53], graph theory [42, 43], computational geometry [5], bioinformatics [40], and physics [6, 50].

In Haskell, type classes [52] are used to write generic functions and have become an important tool for reuse and the organization of abstractions within the Haskell community. They have found use in libraries for domains such as sequence algorithms [1, 38], databases [25], graphs [10], graphical user interfaces [24], symbolic mathematics [29], and micro-architecture design [23].

Language Support

When constructing generic libraries, the primary elements of generic programming—i.e., generic algorithms, concepts, refinement, modeling, and constraints—are mapped to programming language features. Table 1 shows these mapping for C++ and Haskell.

Role	C++	Haskell
Generic algorithm	function template	polymorphic function
Concept	documentation	type class
Refinement	documentation	subclass ($\Rightarrow$)
Modeling	documentation	instance
Constraint	documentation	context ($\Rightarrow$)

Table 1: The roles of language features for generic programming.

In [12] we compared the support for generic programming of several languages. We concluded that while many languages offered some support, C++ and Haskell had the best support for generic programming. Therefore, in this paper we present our design for $\mathcal{G}$ by first reviewing the strengths and weakness of C++ and Haskell, and show how the design of $\mathcal{G}$ builds on those strengths while eliminating weaknesses.

The analysis centers on the following issues that have proven critical for supporting generic programming.

Name Scoping and Type Checking

One of the primary ways that a language can support modularity is to provide a means for creating separate name scopes. When a language fails to provide such features, programming in the large becomes highly error prone. For instance, the C preprocessor provides no name scoping and thus name clashes between separately developed libraries are common. Another example is dynamic scoping in Lisp and the resulting FUNARG problem [30]. In this paper we are concerned with name scoping rules for generic functions. Generic functions present a special challenge because some calls within a generic function are intended to resolve to functions provided by the client, while other calls are intended to resolve to helper functions in the generic library.

Type checking can also improve modularity. The use of types to describe interfaces facilitates the discovery of errors at the boundaries between modules. We say that a language supports *separate type checking* if the body of a function is type checked with respect to its interface and if a call to that function is type checked with respect to the interface of the function. Separate type checking is important for library developers because it aids in catching bugs before the library is distributed to clients. Separate type checking is important for clients because it catches misuses of the library that might otherwise require knowledge of library internals to debug.

Type checking and name lookup are intertwined. Various forms of overloading require name lookup to be dependent on type checking, and type checking is of course dependent on name lookup since the type system must assign types to variables in a way that mirrors the run-time behavior. In Section 2 we analyze the name scoping and type checking rules for generic functions in C++ and Haskell and then present the rules for $\mathcal{G}$.

Generic Algorithm Specialization

The definition of generic programming in Figure 1 requires providing more than one generic algorithm for the same purpose when none dominates the others in efficiency for all inputs. These generic algorithms differ in which concepts the input types are required to model. The most efficient algorithm typically requires more refined concepts. Beyond being simply a programming convenience, automatic algorithm selection is an important control on the complexity of implementing large generic algorithms that use other generic algorithms. Section 3 discusses a commonly used C++ idiom for automatic algorithm selection and considers the ramifications to library construction if automatic algorithm selection is not used. We then discuss how automatic algorithm selection is supported in $\mathcal{G}$.

Associated types

Concepts often include operations that involve more than one type. Typically, some of these types are helper types that are determined by the other types. For example, the Container concept of the STL includes `begin()` and `end()` operations that return iterators. The iterator type is an *associated type* determined by the container type. In Section 4 we discuss the support for associated types in C++ and Haskell, point out disadvantages of those approaches, and propose a language feature to support associated types in $\mathcal{G}$.

Separate Compilation and Efficiency

A programming language supports *separate compilation* if modules can be independently compiled to object files and then linked together to form an executable. If a change is made in the implementation of one module that does not affect its interface, then a client module need not be recompiled but instead can simply be relinked with the updated object file. All of the information needed to compile the client is contained in a header or interface file. Separate compilation can reduce compilation times for large programs and allows software library vendors to distribute binaries instead of source, thereby protecting the intellectual property of the vendor.

In many application areas run-time efficiency is a primary concern. We would like the use of generic libraries to be a viable option in such scenarios. To accomplish this, a generic function must be just as efficient as a normal function. One of the reasons for the success of generic programming in C++ is that several production compilers allow generic libraries to approach this ideal.

There is often a tension between separate compilation and efficiency. Separate compilation requires indirection for function calls that are resolved at link or run-time and indirection has a cost. The ideal is to let the programmer decide which is more important on a case by case basis. We discuss these issues in Section 5 and present a straightforward compilation of $\mathcal{G}$ to C++ that supports separate compilation of generic functions.

Related work is discussed in Section 6, future directions in Section 7, and Section 8 concludes.

Contribution

The following are novel aspects to the design of $\mathcal{G}$:

Scoped models declarations provide better modularity, by preventing accidental clashes, and allow for intentional overlapping of models (§ 2).

Optional constraints together with concept-based function overloading enable generic algorithm specialization (§ 3).

Nested types in concepts provide support for associated types (§ 4).

2. NAME SCOPING AND TYPE CHECKING

This section reviews the name lookup and type checking rules with regards to generic functions in C++ and Haskell and discusses problems that arise. The rules for $\mathcal{G}$ are introduced and shown to address these problems.

2.1 Scoping and Typing in C++

The type checking of a function template occurs in two phases, the first phase at the point of definition and the second at the point of instantiation (at a call to the function). The first phase resolves names and type checks expressions that do not depend on the template parameters. Typically there are not many of these, so phase two is of greater interest.

In phase two the function template is instantiated and concrete types are substituted for the template parameters. After that, the body is type checked, during which overload resolution is performed for function applications. Overload resolution considers functions defined in the lexical scope of the function template and also functions defined in namespaces where the argument types for the call were defined, called *argument-dependent name lookup* (ADL) [11].

ADL provides the mechanism for generic programming in C++ by which concept-associated operations can be found. Consider the following program that defines the function template `accumulate`, a struct `point`, and calls `accumulate` on a list of points. Inside `main`, `accumulate` is instantiated with `T` bound to `B::point`. The body of `accumulate` is then type checked, during which overload resolution is performed for the call to `operator+`. The argument expressions `init` and `*first` have type `B::point`, so the definition of `operator+` in namespace `B` is eligible.

```
#include <list>
#include <iostream>

namespace A {
  template <class T>
  T accumulate(const std::list<T>& xs, T init) {
    typename std::list<T>::const_iterator
      first = xs.begin(), last = xs.end();
    for ( ; first != last; ++first)
      init = init + *first;
    return init;
  }
}

namespace B {
  struct point { float x, y; };
  point operator+(point a , point b) {
    point c = { a.x + b.x, a.y + b.y };
    return c;
  }
}

int main() {
  B::point array[] = { {1,1}, {2,2}, {3,3} };
  std::list<B::point> points(array, array + 3);
  B::point zero = { 0, 0 };
  B::point s = A::accumulate(points, zero);
  std::cout << "(" << s.x << "," << s.y << ")\n";
}
```

The problem with ADL is that it does not distinguish between calls to functions that are intended to be provided by the client, and calls to internal helper functions. As a result, a call that was intended for an internal helper function can be hijacked by a function with the same name in a different namespace. The C++ Standard Library defect report 225 gives the following example [3]. The `unique` function of the Standard Library replaces a subrange of duplicate elements with the first element in the subrange. A typical implementation of `unique` calls another Standard Library function `unique_copy` that is similar to `unique`, except that instead of working in-place it copies the result to another range specified by the third iterator argument.

```
namespace std {
  template <class InIter, class OutIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);

  template <class FwdIter>
  FwdIter unique(FwdIter first, FwdIter last) {
    first = adjacent_find(first, last);
    return unique_copy(first, last, first);
  }
}
```

Suppose that a client of `unique` also defines a function named `unique_copy` that checks whether the second range is a copy of the first range, modulo duplicated subranges.

```
namespace user {
  class my_iter;
  bool unique_copy(my_iter a, my_iter b, my_iter c);
}
```

If `std::unique` is called with `my_iter`, the call to `unique_copy` inside `unique` will resolve via ADL to `user::unique_copy`, which does not have the same semantics as `std::unique_copy`. Therefore the call to `std::unique` will fail to accomplish its goal and there will be a bug in the program. Conscientious authors of generic C++ libraries have started to explicitly namespace-qualify all calls to internal helper functions to prevent this kind of problem, which clutters the library code. Because of ADL, a C++ `namespace` does not create a truly separate "name space" (i.e., scope).

2.1.1 Lack of Separate Type Checking

As described above, the second phase of type checking a generic function occurs after instantiation. Thus C++ does not provide separate type checking of function templates. Two serious and common problems result. First, when a client misuses a generic function by not satisfying the function's constraints, the compiler error message points inside the body of the generic function. The text of the error message is typically long and difficult to understand, often referring to internal names used in the function template. Clients are often led to believe there is a bug in the function template.

The second problem, though less troublesome to the average user, is more insidious. Consider an implementation of a generic function that overlooks the use of some operation and fails to document it as a constraint. For example, suppose that the documentation for `accumulate` only constrained parameter `T` to be a model

of an Addable concept that includes the operation +. This constraint is insufficient because T must also be Copy Constructible: the third parameter of accumulate is passed by value. If a client attempts to call accumulate with a type that is not Copy Constructible, an error will be reported. Again, the error message will be hard to understand, but even worse, the wrong person will see it: the client. This time the library author was at fault, but since the generic function was not separately type checked, the error was not caught during library construction. As a result, the interface as documented was incorrect.

The Standard C++ Library specification, now six years old, still has errors of this nature, primarily because there is no means of automatically checking implementations against their specifications.

2.1.2 Intentionally Overlapping Models

A variation on the accumulate example brings up an interesting issue. Suppose that instead of requiring operator+ for type T, the concept Monoid is used as a constraint. The Monoid concept consists of any associative binary operator and a unit object. Some example of models of the Monoid concept are:

- int with + as the binary operator and 0 as the unit.
- int with * as the binary operator and 1 as the unit.

The above models are "overlapping" because they both include the same type int (this terminology comes from Haskell).

One way to accommodate overlapping models is to take the approach used in the C++ Standard Library: add extra parameters to accumulate for the concept members.

```
template<class InIter, class T, class BinOp>
T accumulate(InIter first, InIter last, T unit,
             BinOp bin_op)
```

However, it is inconvenient to supply concept members as parameters, especially for more complex concepts. One could bundle all of the members of a concept into a single data structure, so that only one extra parameter would be needed, but this is still inconvenient.

A common approach for accessing concept-associated operations is to use a traits class [35]. The following code shows accumulate implemented in this style.

```
namespace A {
  template <class T> struct monoid_traits;

  template <class T>
  T accumulate(const std::list<T>& xs) {
    typename std::list<T>::const_iterator
      first = xs.begin(), last = xs.end();
    T sum = monoid_traits<T>::unit();
    for ( ; first != last; ++first)
      sum = monoid_traits<T>::bin_op(sum, *first);
    return sum;
  }
}
```

A specialization of monoid_traits is then used to make int a model of Monoid.

```
namespace A {
  template <> struct monoid_traits<int> {
    static int bin_op(int x, int y)
      { return x + y }
    static int unit;
  };
  B::point monoid_traits<B::point>::unit = 0;
}
```

Unfortunately, a second model of Monoid cannot be created—e.g., using * and 1—with this approach.

2.2 Scoping and Typing in Haskell

In Haskell, concepts are defined with type class declarations. Below is the definition for a type class named Addable, which includes the add operation that is needed for accumulate. Function types are denoted with arrows (->). It is common practice in Haskell to write functions in curried form, taking one argument at a time and returning a function that expects more arguments.

```
module A where

  class Addable t where
    add :: t -> t -> t

  accumulate [] init = init
  accumulate (x:xs) init =
    add x (accumulate xs init)
```

The definition of the accumulate function consists of two alternatives. The input pattern [] matches empty lists and the pattern (x:xs) matches non-empty lists, binding the variable x to the first element of the list and xs to the rest of the list.

Haskell's type system performs type inferencing, not just type checking, so the type of accumulate need not be supplied by the programmer; instead it can be deduced. In the following, the notation e :: t means that expression e has type t. The first alternative is type checked with the following name bindings in scope:

```
add :: Addable t => t -> t -> t
init :: b
accumulate :: [a] -> b -> c
```

The symbols a, b, c, and t are type variables. The entry for add comes from the type class declaration for Addable, and the context Addable t => signifies that at a call to add, the type bound to t must be an instance of Addable. There is an entry for accumulate to allow for recursive functions.

The type inference process takes as input an environment and an expression and returns a type, a unifier, and a list of constraints [17]. The unifier specifies how the type variables are related to each other and how they should be instantiated. Type inferencing applied to the expression init computes the type b. The type b needs to unify with the return type of accumulate, thus the unifier (b,c) is computed. A unifier such as (b,c) means that b is substituted for c. Thus, the first alternative of accumulate has type [a] -> b -> b.

The second alternative of accumulate is type checked in a lexical environment that contains the following bindings:

```
add :: Addable t => t -> t -> t
x :: a
xs :: [a]
init :: b
accumulate :: [a] -> b -> c
```

The notation [a] denotes a list containing elements of type a. For the expression (accumulate xs init) the type c is inferred, with no unifiers or constraints. For the expression (add x ...) the type a is inferred, and the unifier (a,t) is computed with constraint Addable a. The type a is unified with the function's return type c to get the unifier (a,c). Thus, the second alternative has the type Addable a => [a] -> b -> a. The types for two alternatives of accumulate are unified, which results in the type Addable a => [a] -> a -> a with the unifier (a,b).

Thus, the accumulate function has been type checked independently of any call to it. The distinction between calls to client-provided functions and calls to internal functions is determined

by which type class definitions are in scope at the definition of `accumulate`. Functions that correspond to type class members must be defined by clients, and all other function calls resolve to functions in the lexical scope of the definition of `accumulate`.

Note how type classes introduce names into the scope of the module where they are defined. For example, `Addable` introduces the name `add` into the scope of module `A`. In Haskell, two type classes in the same module may not have members with the same name, since that causes a name clash when the member names are added to the scope of the module.

In Haskell, calls to generic functions are type checked with respect to the interface (type) of the function. The following describes how this works with the `accumulate` example. The type of the `accumulate` function contains the constraint `Addable a`, so if `accumulate` is to be applied to a list of points, the `Point` type must be a model of `Addable`.

In Haskell an instance declaration is required to assert that a type models a concept. Instance declarations must include the definitions of the type class methods. Below is the definition of a `Point` type and an instance declaration that makes `Point` an instance of `A.Addable`.

```
module B where
  import qualified A(Addable)

  data Point = Point Float Float deriving Show

  instance A.Addable Point where
    add (Point ax ay) (Point bx by) =
      Point (ax + bx) (ay + by)
```

The following module uses `accumulate` to add a list of points.

```
module Main where
  import qualified A(accumulate)
  import qualified B(Point)

  points = [ B.Point 1 1, B.Point 2 2, B.Point 3 3]
  main = A.accumulate points (B.Point 0 0)
```

The function `main` is type checked in the following environment.

```
A.accumulate :: A.Addable a => [a] -> a -> a
points :: [B.Point]
A.Addable ::i B.Point
```

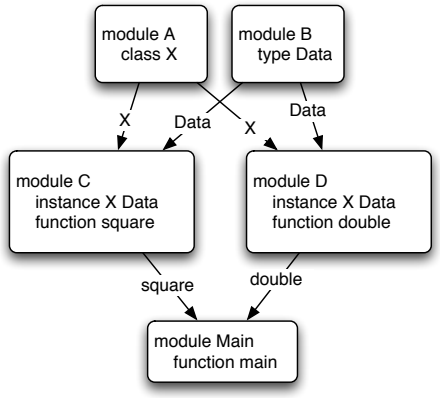
The third entry in the environment indicates that `B.Point` is an instance of `A.Addable`. The instance declaration from module `B` is in scope because any import from a module also brings in all instance declarations, and there is an import of `B(Point)`.

For the expression `A.accumulate points (B.Point 0 0)`, the type `Point` is inferred and the type parameter `a` is unified with `Point`. The resulting constraint `A.Addable B.Point` is satisfied by the environment. Thus, the call to `accumulate` has been type checked with respect to the type of the function, separately from it implementation.

However, there are three problems with the rule that any import from a module brings in all instance declarations, which we discuss in the following sections.

2.2.1 Accidental Instance Clash

Consider the scenario shown below, with five modules and arrows that represent the flow of imported definitions. Module `Main` imports the `square` function from module `C` and the `double` function from module `D`.




```
module Main where
  import C(square)
  import D(double)
  main = C.square 3 + D.double 4
```

Compilation of this program results in the following error:

```
ERROR D.hs:4 - Overlapping instances for class "X"
*** This instance   : X Data
*** Overlaps with   : X Data
*** Common instance : X Data
```

The problem is that modules `C` and `D` contain instance declarations for type class `X` with the same type `Data`, which in separate modules is fine, but when module `Main` imports function `square` from `C` and `double` from `D`, the instance declarations are also brought into `Main`, where they clash.

For completeness, the definitions of modules `A`, `B`, `C`, and `D` are as follows.

```
module A where
  class X a where
    f :: a -> a

module B where
  data Data = MkData

module C where
  import A(X)
  import B(Data)
  instance X Data where
    f d = d
  square x = x * x

module D where
  import A(X)
  import B(Data)
  instance X Data where
    f d = d
  double x = x + x
```

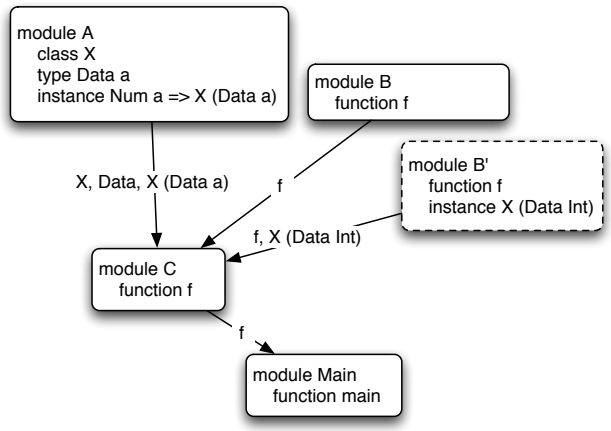
One seemingly obvious solution to the clashing instance declarations is to merge them into a single declaration in a common module, such as `B`. However, if modules `C` and `D` are independently developed, it may not be feasible to make such changes.

2.2.2 Accidental Instance Override

A popular extension to Haskell's type classes is to allow overlapping instance declarations. This allows, for example, a more specific concrete instance declaration to override a more general polymorphic instance declaration. This extension is quite useful,

but when combined with Haskell's scoping rules it can lead to surprising behavior.

Consider a scenario involving a common module `A`, two separately developed libraries `B` and `C`, and an application module `Main`. The following is the import graph for these modules.



Module `C` uses function `f` from `B` and also the data type and instance declaration in `A`. The application module `Main` uses function `f` from `C`. The result of this program is the value `6`.

```
module A where
  class X a where
    g :: a -> a -> a

  data Data a = MkData a

  instance Num a => X (Data a) where
     g (MkData x) (MkData y) = MkData (x - y)

module B (f) where
  f :: Int -> Int
  f x = x + x

module C (f) where
  import A
  import qualified B(f)

  f :: Int -> Int
  f x = (B.f x) + let a = g (MkData x) (MkData x)
                  in case a of MkData y -> y

module Main where
  import C(f)
  main = f (3::Int)
```

At some later time, the author of module `B` changes the implementation of `f` and in so doing, adds an instance declaration for type class `X`.

```
module B (f) where -- updated version
  import A

  instance X (Data Int) where
    g (MkData x) (MkData y) = MkData (x + y)

  f :: Int -> Int
  f x = case (g (MkData x) (MkData x))
        of MkData y -> y
```

Note that the semantics of `f` is unchanged, so one would expect no change to clients of module `B`. However, the result of running `Main` using this new version of `B` is now the value `12`. The reason is that the new instance declaration in `B` leaks out to module `C` and the call to `g` resolves to the new instance declaration in `B` instead of the original one in `A`, changing the behavior of the program.

2.2.3 Intentionally Overlapping Instances

The problem of overlapping models in C++ was discussed in Section 2.1.2. Haskell has the same problem because any import brings in all instance declarations. The program below attempts to define and use overlapping instances, but fails to type check. Note that the overlapping instances are defined and used in separate modules, so one might hope that they would not clash. However, they do indeed clash when module `Main` tries to import `prod` from module `D` and `sum` from module `E`.

```
module Algebra where

  class Monoid t where
    mult :: t -> t -> t
    unit :: t

module Algo where
  import Algebra(Monoid(..))

  accumulate :: Monoid t => [t] -> t
  accumulate xs =  foldl mult unit xs

module D (prod) where
  import Algo(accumulate)

  instance Monoid Int where
    mult x y = x * y
    unit = 1

  prod :: [Int] -> Int
  prod c = accumulate c

module E (sum) where
  import Algo(accumulate)

  instance Monoid Int where
    mult x y = x + y
    unit = 0

  sum :: [Int] -> Int
  sum c = accumulate c

module Main where
  import qualified D(prod)
  import qualified E(sum)

  nums = [1,2,3,4,5::Int]
  main = (E.sum nums, D.prod nums)
```

2.3 Scoping and Typing in $\mathcal{G}$

Experience with C++ has taught us the need for separate type checking and for name lookup rules that allow a generic library author to decide which function calls are to client-provided functions and which calls are to internal functions. Haskell presents a design that satisfies these needs, however the rule for importing instance declarations allows for accidental instance clashes and instance overrides and fails to provide a mechanism for intentionally overlapping instances. Furthermore, Haskell's type classes were designed to make type inferencing possible, which is not a requirement for $\mathcal{G}$.

Like Haskell, $\mathcal{G}$ provides language support for expressing concepts. The following code in $\mathcal{G}$ defines an `accumulate` function and the concepts necessary to specify its constraints: `Addable` and

`Regular`. Concept operations are specified with signatures (function types), and not valid expressions (as is typical in C++ documentation). The reason for this choice is that signatures were straightforward to implement whereas we had doubts about the valid expression approach. Like C++, $\mathcal{G}$ has reference types, however `&` denotes a constant reference and `&!` a mutable reference. In the definition of `accumulate`, the parameter types, return type, and constraints (the `where` clause) are required since $\mathcal{G}$ does not perform type inferencing.

```
module A {
  concept Addable<X> {
    sig operator+(X&, X&) -> X;
  };
  concept Regular<U> {
    sig operator copy(U&) -> U;
    sig operator =(U&, U&!) -> void;
    sig operator delete(U&!) -> void;
  };

  fun accumulate<T>(list<T>& xs, T init) -> T
    where Addable<T>, Regular<T>
  {
    let first = begin(xs), last = end(xs);
    while (first != last) {
      init = init + *first;
      ++first;
    }
    return init;
  }
}
```

As with Haskell, type parameters of generic functions, such as `Iter` and `T`, are treated in an opaque fashion; the only information known about the type parameters is specified in the `where` clause.

Unlike Haskell, a concept declaration does not introduce the names of its operations into the module's scope. Instead, when a concept is used in the `where` clause of a function, the concept members are introduced into the scope of the function body. The reason for this difference is twofold. First, the scope of names should be restricted to where they are needed, thereby reducing the chance of name clashes. Second, identifying the scope of concept operations with the bodies of generic functions better matches typical generic programming practice.

The body of `accumulate` is type checked in the following environment, which includes the parameters and also the operations from the concepts in the `where` clause.

```
xs :: list<T>
init :: T
operator+ :: fun (T&,T&) -> T
operator copy :: fun (T&) -> T
operator assign :: fun (T&, T&!) -> void
operator delete :: fun (T&!) -> void
```

Type checking the body of `accumulate` is done in the standard syntax-directed fashion. The `let` statement introduces new variables that can be accessed in the remainder of the enclosing scope. The type of the variable is that of the right-hand side expression. If the function has a non-void return type, every control flow path through the function must have a return statement and the type of the returned expression must match the return type of the function.

As before, the `accumulate` function is applied to a list of points. The `Point` type must be a model of the `Addable` and `Regular` concepts. Since `Point` is a struct and all of its component types are models of `Regular`, it too is a model of `Regular`. However, `Point` must also model the `Addable` concept. Analogous to Haskell's `instance` declarations, there are `model` declarations in $\mathcal{G}$ for establishing that a type (or types) model a given concept. All of the operations required by the concept must either be defined in the body of the `model` declaration or in an enclosing scope.

```
module B {
  struct Point { float x; float y; };

  model A.Addable<Point> {
    fun operator+(Point p, Point q) -> Point {
      return struct Point { x = p.x + q.x,
                            y = p.y + q.y };
    }
  };
}
```

The following `main` function uses `accumulate` to add the list of points. The `import` statement brings the `model` declaration into scope, which is needed to satisfy the requirements of `accumulate`. The arguments' types are `list<B.Point>` and `B.Point`, so the call type checks with `B.Point` deduced for the type parameter `T`.

```
fun main() -> int {
  import A.Addable<B.Point> from B;
  let points = class list<B.Point>;
  push_back(points, B.Point{x=1,y=1});
  push_back(points, B.Point{x=2,y=2});
  push_back(points, B.Point{x=2,y=2});
  let s = A.accumulate(points,
                       struct B.Point{x=0,y=0});
  print("(" + f2str(s.x) + ","
        + f2str(s.y) + ")\n");
  return 0;
}
```

$\mathcal{G}$ does not have Haskell's rule that any import also brings in all `instance` declarations. Instead $\mathcal{G}$'s `model` declarations obey the normal scoping rules for entities in a module, an explicit import declaration is required to import them.

2.3.1 Intentionally Overlapping Models

There is a straightforward solution in $\mathcal{G}$ to the problem of overlapping models from Sections 2.1.2 and 2.2.3. The overlapping `model` declarations are placed in separate modules and only imported into separate local scopes. The following example revisits `accumulate` in the style of Section 2.2.3, using the `Monoid` concept as a constraint.

```
concept Monoid<T> {
  sig operator*(T&, T&) -> T;
  sig unit() -> T;
};
fun accumulate<T>(list<T>& xs) -> T
  where Monoid<T>, Regular<T>
{
  let first = begin(xs), last = end(xs);
  let total = unit();
  while (first != last) {
    total = total * *first;
    ++first;
  }
  return total;
}
```

In the code below, two overlapping models of `Monoid` are defined in different modules and imported into separate scopes. In the first scope we use `accumulate` to compute the sum of a list of numbers and in the second scope to compute the product.

```
module IntMult {
  model Monoid<int> {
```

```
    fun unit() -> int { return 1; }
  }
}
module IntAdd {
  model Monoid<int> {
    fun operator*(int x, int y) -> int
        { return x + y; }
    fun unit() -> int { return 0; }
  }
}
fun main() -> int {
  ...
  let sum = 0, prod = 0;
  {
    import Monoid<int> from IntAdd;
    sum = accumulate(nums);
  }
  {
    import Monoid<int> from IntMult;
    prod = accumulate(nums);
  }
  ...
}
```

A slightly more convenient syntax for the above is to use named models. In the following different names are assigned to the overlapping models and they are imported by name in the two local scopes.

```
IntMultMonoid: model Monoid<int> {
  fun unit() -> int { return 1; }
}
IntAddMonoid: model Monoid<int> {
  fun operator*(int x, int y) -> int
      { return x + y; }
  fun unit() -> int { return 0; }
}
...

{
  import IntAddMonoid;
  sum = accumulate(nums);
}
{
  import IntMultMonoid;
  prod = accumulate(nums);
}
```

3. GENERIC ALGORITHM SPECIALIZATION

One goal of generic programming is to make generic algorithms as efficient as non-generic algorithms. For many problems one generic algorithm provides the best efficiency for all input types. However, for other problems there is not a single best algorithm, but instead there are several best algorithms for different subsets of input types. Typically, each of the best algorithms takes advantage of properties of its input types to obtain efficiency gains. The `advance` function of the STL, shown below, is an example of such a generic function. The input to `advance` is any iterator type that models the Input Iterator concept. A traits class [35] is used to query whether the iterator type supports more functionality, such as random access, and dispatches accordingly to one of three different algorithms. This idiom is called "tag dispatching".

```
template <class InputIter, class Distance>
void advance(InputIter& i, Distance n) {
  typename iterator_traits<InputIter>
    ::iterator_category cat;
  __advance(i, n, cat);
}
template <class InputIter, class Distance>
void __advance(InputIter& i, Distance n,
               input_iterator_tag) {
  while (n--) ++i;
}
template <class BidirectionalIter, class Distance>
void __advance(BidirectionalIter& i, Distance n,
               bidirectional_iterator_tag) {
  if (n > 0)
    while (n--) ++i;
  else
    while (n++) --i;
}
template <class RandomAccessIter, class Distance>
void __advance(RandomAccessIter& i, Distance n,
               random_access_iterator_tag) {
  i += n;
}
```

Overloading on type properties is an important technique for managing the complexity of large generic functions. In the following we compare implementations of the `inplace_merge` function of the STL. The first implementation uses overloading while the second version does not. Instead, the client of the function chooses between the specializations. For example, `advance` is replaced by three functions: `advance_input`, `advance_bidir`, and `advance_rand`, that the client can choose from.

Figure 2 shows the call graph for the `inplace_merge` function of the STL. Tag dispatching occurs near the leaves of the tree, in the low level functions. Figure 3 shows the call graph for an implementation that does not use overloading. The specialization must be "bubbled up" to every helper function and to the top level function. For example, there are three versions of `lower_bound`, each calling the appropriate version of `advance`. The resulting call graph is much more complex and represents a large amount of code duplication.

3.1 Function Overloading in $\mathcal{G}$

In $\mathcal{G}$, functions with the same name and different signatures may be defined in the same scope. All functions with the same name in a scope form a function overload set. Unlike C++, function overloads are first class objects. The type of an overload set is an intersection type [39] that consists of the types of the functions in the overload set.

We use the term *concept-based overloading* when there are two functions with the same parameter and return types but different `where` clauses. The following code shows the overloads for the `__advance` function written in $\mathcal{G}$, using concept-based overloading instead of the tag dispatching of C++.

```
fun __advance<Iter, D>(Iter&! i, D n) -> void
  where InputIter<Iter>, Integral<D>
{
  while (n--) ++i;
}
fun __advance<Iter, D>(Iter&! i, D n) -> void
  where BidirectionalIter<Iter>, Integral<D>
{
  if (n > 0)
    while (n--) ++i;
  else
    while (n++) --i;
}
fun __advance<Iter, D>(Iter&! i, D n) -> void
  where RandomAccessIter<Iter>, Integral<D>
{
  i += n;
}
```

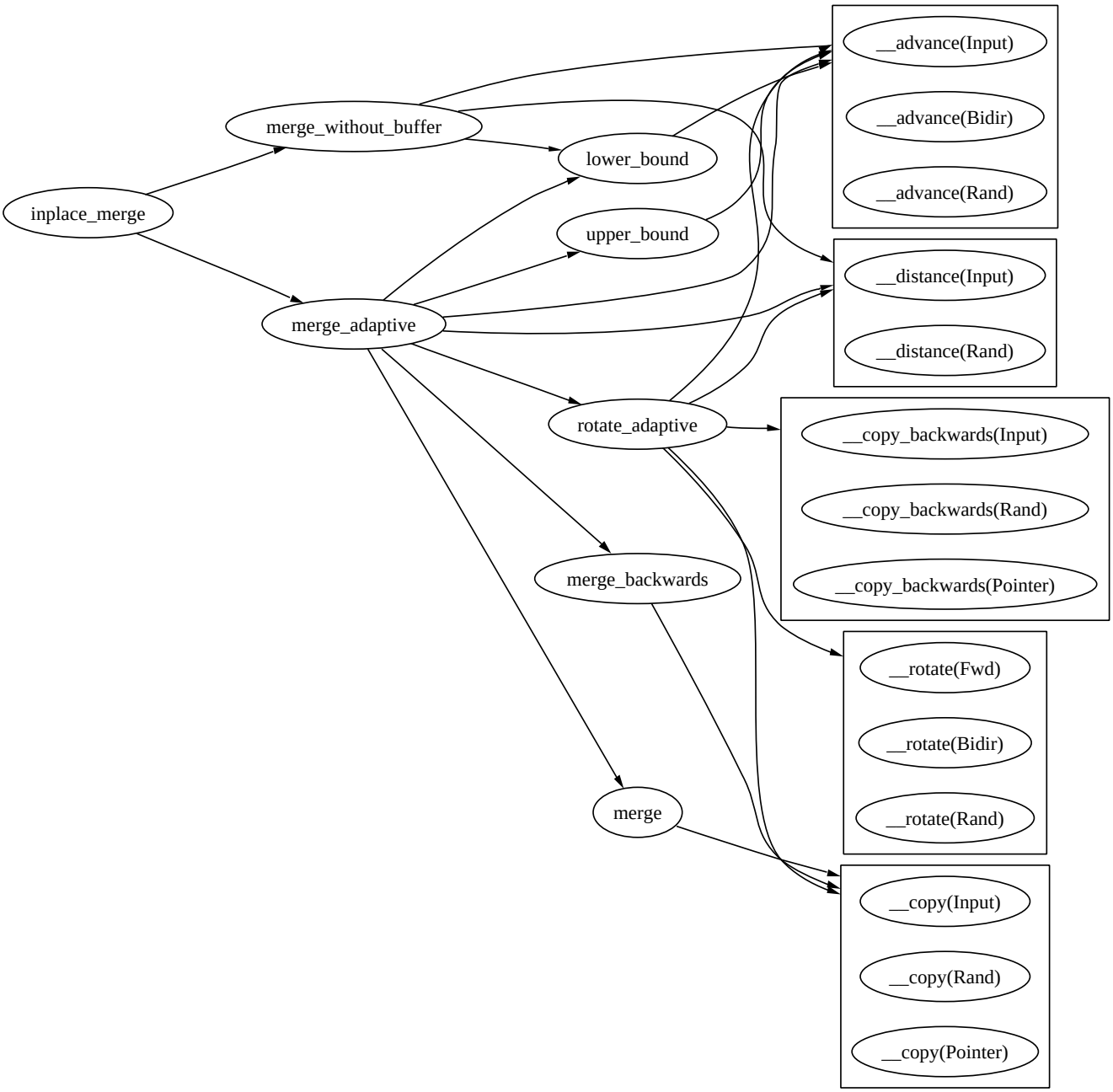


Figure 2: Call graph for inplace merge with overloading.

Figure 3: Call graph for inplace merge without overloading.

The following outlines the overload resolution process at the call site of an overloaded function.

1. Remove from consideration functions in the overload set having the wrong arity.
2. Remove from consideration functions whose parameter types do not match the argument types. If the function is polymorphic the matching process includes deducing types for the type parameters.
3. Remove from consideration functions whose type requirements can not be satisfied in the calling scope, that is, if the appropriate models declarations can not be found and if any same-type constraints are not satisfied.
4. Of the remaining functions, determine whether there is one that is most specific. If there is a most specific function, it will be called. Otherwise, emit a compiler error. The intuition behind the ordering is that if a call to function `f` will type check in the body of `g`, but not vice-versa, then `g` is more specific. The following factors determine specificity:
   (a) A concrete type is more specific than a type parameter.
   (b) A struct, class, or union type of the form $n\texttt{<}t_1, \ldots, t_i\texttt{>}$ is more specific than another type $m\texttt{<}s_1, \ldots, s_i\texttt{>}$ if $n = m$ and if t_k is more specific than s_k for $k = 1, \ldots, i$.
   (c) A requirement for $C\texttt{<}t_1, \ldots, t_i\texttt{>}$ is a *refinement* of the requirement $D\texttt{<}s_1, \ldots, s_j\texttt{>}$ if the concept C contains a refines clause that matches $D\texttt{<}s_1, \ldots, s_j\texttt{>}$ once the type parameters of C have been substituted for $t_1, \ldots, t_i$. A type requirement $C\texttt{<}t_1, \ldots, t_i\texttt{>}$ is more specific than the requirement $D\texttt{<}s_1, \ldots, s_j\texttt{>}$ if they are in the transitive closure of the refinement relation.

3.1.1 Optional Type Requirements

Consider the following attempt to write an `advance` function that will dispatch to the `__advance` overloads.

```
fun advance<Iter, D>(Iter&! i, D n) -> void
  where InputIter<Iter>, Integral<D>
{
  __advance(i, n);
}
```

The above call to `__advance` will always resolve to the version that requires `InputIter`, not the other two overloads. The dispatching inside `advance` is based purely on information provided by the `where` clause, in this case `Iter` models `InputIter`. This problem is solved by the addition of optional requirements. An optional type requirement is a constraint that need not be satisfied by the caller, but if it is satisfied, the information is propagated into the generic function and used for dispatching. The following code shows the `advance` function with optional constraints, which are denoted by a trailing question mark.

```
fun advance<Iter, D>(Iter&! i, D n) -> void
  where InputIter<Iter>, Integral<D>,
    BidirectionalIter<Iter>?,
    RandomAccessIter<Iter>?
{
  __advance(i, n);
}
```

The above call to `__advance` will dispatch to the appropriate overload depending on whether models for the more refined concepts were available at the call site.

An alternative design for dealing with concept-based overloading is to propagate *all* information about models from the call site into the generic function, thus making it possible to use the full information about a type when resolving overloads. However this approach gives up separate compilation and separate type checking in exchange for the convenience of not writing out the optional requirements. As stated in the introduction, modularity and safety are higher priorities than convenience for $\mathcal{G}$, so we have chosen optional requirements as the solution to this problem. For more discussion of the alternative design, see [15].

4. ASSOCIATED TYPES

Associated types are types that play a role in the operations of a concept, vary from model to model, and are determined by the type parameters of the concept. A familiar example of this is the element type of a container. In fact, the Container concept of the C++ Standard Library includes 8 associated types:

- `value_type`
- `iterator`
- `const_iterator`
- `reference`
- `const_reference`
- `pointer`
- `difference_type`
- `size_type`

It is often necessary to refer to an associated type in the signature or body of a generic function. The following code for an `accumulate` function is an example. The return type of the function and the type of the local variable `sum` must be the value type of the iterator. In C++ the traits class idiom is used to access associated types. In this example, `iterator_traits` is used to access the value type.

```
template <class InputIter>
typename std::iterator_traits<InputIter>::value_type
accumulate(InputIter first, InputIter last) {
  typedef typename
    std::iterator_traits<InputIter>::value_type T;
  T sum = monoid_traits<T>::unit();
  for ( ; first != last; ++first)
    sum = sum + *first;
  return sum;
}
```

The following code shows a call to `accumulate`, with `InputIter` bound to `B::point*`.

```
int main() {
  B::point array[] = { {1,1}, {2,2}, {3,3} };
  B::point s = A::accumulate(array, array + 3);
  std::cout << "(" << s.x << "," << s.y << ")\n";
}
```

The C++ Standard Library supplies a partial specialization of the `iterator_traits` class for pointers, as shown below.

```
namespace std {
  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;
  };
}
```

Thus, the type

```
std::iterator_traits<B::point*>::value_type
```

resolves to `B::point`. When `accumulate` is instantiated, the return type and the type of the variable `sum` will be `B::point`.

The traits class idiom suffers from the following two disadvantages.

1. The use of template specialization inside of generic function relies on template parameters being transparent: the actual type must be known to find the matching templates specialization. Thus, the traits class idiom is not compatible with a language that has opaque template parameters and separate type checking (like Haskell and $\mathcal{G}$).

2. Template specialization is an advanced technique and rarely taught in introductory C++ courses. Thus, many beginning C++ programmers have difficulty understanding the uses of traits classes in the C++ Standard Library.

4.1 Associated Types in Haskell

Some dialects of Haskell have partial support for associated types via the ability to express functional dependencies between type parameters [19]. Below is a type class with two type parameters, the collection type `c` and the element type `t`.

```
class Collection c t | c -> t where
  member :: Eq t => t -> c -> Bool
  fold :: (a -> t -> a) -> a -> c -> a
```

The notation `c -> t` means that `t` is determined by `c`. The use of functional dependencies informs the Haskell type system that a type argument for `t` need not be deduced at the call site of a generic function. Function dependencies are important for generic functions that have a parameter of the collection type but not of the element type. The following `accumulate` function has a single parameter `cont` of type `c`. The type bound to `c` must be an instance of the `Collection` type class. The type parameter `t` is for the element type.

```
class Monoid t where
  mult :: t -> t -> t
  unit :: t

accumulate :: (Collection c t, Monoid t) => c -> t
accumulate cont =  fold mult unit cont

instance Collection [t] t where
  member x xs = elem x xs
  fold f x xs = foldl f x xs

data Point = Point Float Float deriving Show
instance Monoid Point where
  mult (Point ax ay) (Point bx by) =
    Point (ax + bx) (ay + by)
  unit = (Point 0 0)

points = [ Point 1 1, Point 2 2, Point 3 3]
main = accumulate points
```

At the call to `accumulate`, the type system deduces type `[Point]` for the type parameter `c`. It does not need to deduce a type for parameter `t` because `t` appears in a constraint where it is functionally dependent. The type assignment for `t` is instead deduced from the instance declaration `Collection [t] t`, so in this case `t` is bound to `Point`.

The disadvantage of using type parameters for associated types is that every function must have a type parameter for every associated type. As the number of associated types grows, and as type classes are composed via sub-classing, the number of type parameters needed for a generic function grows. For example, the Reversible Container concept from the STL has 10 associated types (8 of which are inherited from Container). A generic function with two parameters required to model Reversible Container would have 22 type parameters.

Type classes allow the programmer to define a *relation* between types, but for associated types, a *function* is needed. The traits class idiom of C++ is a mechanism for defining compile-time functions on types using specialization of templates. However, traits classes depend upon post-instantiation type checking; the concrete argument type must be known to find the correct specialization of the traits class. Thus, traits classes will not work for $\mathcal{G}$, where generic functions are type checked prior to instantiation. Another option is to allow nested type place-holders in a type class, which is explored in the next section. There is also work in progress to add nested types to Haskell [8].

4.2 Associated Types in $\mathcal{G}$

The `InputIter` concept defined below contains a declaration of an associated `value` type.

```
concept InputIter<X> {
  type value;
  refines Regular<X>;
  sig operator++(X&! c) -> X&!;
  sig operator*(X& b) -> value;
};
```

(The `refines` declaration means that `InputIter` "inherits" the requirements from the `Regular` concepts.)

In a model declaration for `InputIter`, the associated type `value` must be assigned a type. The following models declaration is parameterized on `T`, and says that pointers to `T` are models of `InputIter` with a value type of `T`. The increment and dereference operators are the built-in operators for pointers.

```
model<T> InputIter<T*> {
  type value = T;
};
```

The following is an iterator version of the `accumulate` function. The dot notation is used to refer to an associated type. As with type parameters, nothing is known about an associated type except for what is declared in the `where` clause.

```
fun accumulate<Iter>(Iter first, Iter last)
  -> InputIter<Iter>.value
  where InputIter<Iter>,
        Monoid<InputIter<Iter>.value>
{
  InputIter<Iter>.value sum = unit();
  while (first != last) {
    sum = sum + *first;
    ++first;
  }
  return sum;
}
```

For many generic functions, two type expressions must denote the same type. For example, in the below `merge` function the associated value type of `Iter1` and `Iter2` must be the same type.

```
fun merge<Iter1,Iter2,Iter3>
  (Iter1 first1, Iter1 last1,
   Iter2 first2, Iter2 last2,
   Iter3 result) -> Iter3
  where InputIter<Iter1>, InputIter<Iter2>,
    OutputIter<Iter3, InputIter<Iter1>.value>,
    InputIter<Iter1>.value
      == InputIter<Iter2>.value,
    Ordered<InputIter<Iter1>.value>
{
  ...
}
```

Associated types and same-type constraints increase the difficulty of comparing type expressions for equality while type checking a generic function. Each type expression is normalized to the representative type expression for its equivalence class and the normalized type expressions are compared for syntactic equality.

5. SEPARATE COMPILATION AND EFFICIENCY

There are many ways to compile polymorphic functions. The following approaches represent the two ends of the spectrum.

- Stamp out a specialization of the function for each set of argument types.
- Compile to a single procedure that treats objects of different types in a uniform way. Type specific operations are accessed via indirection through run-time data structures such as a vtable or a dictionary.

Most C++ compilers use specialization and inlining to compile templates. Compilers for Haskell typically start by compiling to a single procedure and then perform selective specialization and inlining as an optimization.

There are complex time and space tradeoffs between the two approaches.

- Specialization and inlining reduce or remove function call overheads, reducing run-time [2].
- Specialization creates multiple copies of the same function, increasing code size. Increased code size can affect time by reducing the effectiveness of the instruction cache.
- Specialized versions of functions often take up less space, since they need not manipulate dictionaries, and because dead code can be eliminated [18]. This reduces code size.
- Specialization increases compile times [18].

Another issue to consider is that to specialize a generic function, the compiler must have access to the implementation of the function. If the generic function is in a separately compiled library, then specialization may not be an option. Also, if a client does not wish to recompile when a library is updated, then specialization should not be performed.

For $\mathcal{G}$, we plan to implement a compiler that provides both specialization and single function compilation of generic functions, controlled by flags that control whether modules are compiled together or separately. The semantics of $\mathcal{G}$ is such that both approaches can be used, resulting in executables with the same behavior (in terms of outputs, etc.), but with different time and space characteristics.

As a first step we have begun implementing the single function approach to compiling generic functions. This approach supports separate compilation. As a second step we plan to add function specialization and inlining in an optimization pass. We also plan to implement the small object optimization, an important optimization for reducing abstraction penalty [31, 41, 51].

5.1 Compilation to C++

To illustrate the basic ideas, this section steps through the compilation of the following small $\mathcal{G}$ program.

```
concept Comparable<U> {
  sig operator<(U x, U y) -> bool;
};
fun min<T>(T x, T y) -> T
  where Comparable<T>, Regular<T>
{
  if (x < y) return x;
  else return y;
}
model Comparable<int>;
fun main() -> int {
  return min(1, 0);
}
```

None of the compilation techniques discussed in this section are novel; they are the standard techniques for compiling polymorphic functions and for implementing type classes via dictionary passing. However, one difference between $\mathcal{G}$ and languages such as ML and Haskell is that parameters are passed by value in $\mathcal{G}$, as they are in C and C++. Thus even a polymorphic identity function has type requirements: it needs to copy the input object.

`boost::function` [13] is used to represent functions because it can store both global functions and function objects. For each concept a struct definition is output for the dictionary (like a vtable) of its operations. The below `__Comparable` struct is the dictionary for the `Comparable` concept. Type parameters such as `U` are compiled to `void*`.

```
struct __Comparable {
  function<bool(void*,void*)> __operator_lt;
};
```

The model declaration for `Comparable<int>` is compiled to the following variable declaration, which creates a dictionary object containing one function.

```
__Comparable __int__Comparable = { op_lt_int };
```

The function in the dictionary should be the built-in `operator<` for `int`. However, that function takes two `int`'s whereas the function in the `__Comparable` dictionary must must take two arguments of type `void*`. We therefore create a wrapper function that unboxes its arguments and then boxes the result if necessary. The following `op_lt_int` function is the wrapper for `operator<`.

```
bool op_lt_int(void* x, void* y) {
  return *(int*)x < *(int*)y;
}
```

There are several interesting aspects to compiling the call to `min`. First, the dictionaries for models that satisfy the type requirements

are explicitly passed as extra arguments. Second, the auxiliary __box function (which allocates space on the heap and initializes the space with its argument) is used to convert the arguments 1 and 2 of type int to void*. The min function is responsible for deallocating the memory associated with its parameters. The return type of the compiled min function will be void*, so a cast and dereference is required to return the underlying integer. The caller is responsible for deallocating the returned temporary object, so a boost::scoped_ptr [9] is used.

```
scoped_ptr<int> tmp;
return *(tmp.reset((int*)min(__box(1), __box(2),
                             &__int_Comparable,
                             &__int_Regular)),
         tmp.get());
```

The following is the compiled form of the min function. The type parameter T has been replaced with void* and two parameters have been added for the dictionaries. Since min is responsible for deallocating the memory associated with its parameters, scoped_ptr is used with the del function provided by the dictionary for the Regular concept. The use of operator< has become an explicit call to the function in the __T_Comparable dictionary. There are four calls to copy, which are needed to implement the call-by-value semantics of $\mathcal{G}$.

```
void* min(void* x, void* y,
          __Comparable* __T_Comparable,
          __Regular* __T_Regular)
{
  scoped_ptr<void> __del_x(x, __T_Regular->del),
                   __del_y(y, __T_Regular->del);
  if (__T_Comparable->__operator_lt
        (__T_Regular->copy(x),
         __T_Regular->copy(y)))
    return __T_Regular->copy(x);
  else
    return __T_Regular->copy(y);
}
```

5.2 Concept Refinement

One obvious design for the run-time representation for refinement is to have the struct of the refining concept inherit from the struct for the refined concept. For example:

```
struct __Eq {
  function<bool(void*,void*)> __operator_eq;
  function<bool(void*,void*)> __operator_neq;
};
struct __Ordered : public __Eq {
  function<bool(void*,void*)> __operator_lt;
  ...
};
```

However, suppose a concept refines the same concept in two different ways, as in the concept below.

```
concept A<T> { ... };
concept B<T> {
  type X;
  sig foo(T&) -> X;
  refines A<T>;
  refines A<X>;
};
```

This would result in a struct that inherits two times from the same class, which is not legal C++.

Instead refinement is represented by placing a pointer to the model for the refined concept in the struct for the refining concept. This approach has a performance disadvantage since pointer dereferences are required, so alternatives are being explored. The run-time representation for models of the above concepts would be:

```
struct __A { ... };
struct __B {
  function<void*(void*)> foo;
  __A* __T_A;
  __A* __X_A;
};
```

Refinement adds a minor complication to the compilation of generic functions that call other generic functions. In the following example the generic function g calls the generic function f.

```
fun f<T>(T& x) -> void
  where A<T> { ... }

fun g<U>(U& u) -> void
  where B<U>
{
  f(u);
  f(foo(u));
}
```

Function f has the type requirement A<T>, and g has the requirement B<U> . Inside g we call f twice, once with an argument of type U and again with an argument of type B<U>.X. Both calls type check since the type requirements A<U> and A<B<U>.X> are both satisfied due to the refinements in concept B.

The following code is the result of compiling f and g.

```
void f(void* x, __A* __T_A) { ... }

void g(void* u, __B* __U_B) {
  f(u, __U_B->__T_A);
  f(__U_B->foo(u), __U_B->__X_A);
}
```

At a call site such as f(u) it must be determined whether there is a model available to satisfy the type requirements of f, and further, the refinement path to the model must be known so that the appropriate sequence of dictionary member accesses can be output, such as __U_B->__T_A.

6. RELATED WORK

6.1 Concept Proposals for C++

Stroustrup and Dos Reis have written several preliminary design documents for extending C++ with concepts [47–49]. Several ways to constrain template parameters are compared: base-classes (subtyping), signatures, and usage patterns (valid expressions). The base-class approach is rejected because it suffers from the binary method problem [7]. The usage pattern approach is preferred to signatures because it is a better fit with existing C++ coding practices, which implement the same expression via functions with widely varying signatures.

In terms of syntax, Stroustrup proposes using the concept name as a place-holder for the modeling type. The authors of this paper prefer to use an explicit type parameter for the modeling type (or types). Stroustrup proposes to support associated types by parameterized concepts. However, as described in Section 4, this leads to an increase in the number of type parameters needed for generic functions. The alternative solution taken by $\mathcal{G}$ and also in a future version of Haskell (see the next section) includes nested type placeholders in concepts. Stroustrup proposes operators for composing concepts: && (and), || (or), and ! (not). The "or" would provide an alternative to the optional type requirements present in $\mathcal{G}$. For example, the advance function's requirements could be written as

```
InputIter<Iter> || BidirectionalIter<Iter>
  || RandomAccessIter<Iter>
```

6.2 Associated Data Types in Haskell

In a rapid response to [12] Chakravarty, et. al. [8] have developed a proposal for nesting both type synonyms and data types within type classes. It was straightforward for Chakravarty to extend the typing rules of Haskell, however, the type inferencing algorithm became more involved as the type unification algorithm was extended to maintain a set of pending unification constraints. In $\mathcal{G}$, supporting associated types was somewhat simpler because we have only type checking in $\mathcal{G}$, not full fledged type inference.

6.3 Named Instances in Haskell

The named `model` declarations of $\mathcal{G}$, discussed in Section 2.3.1, were inspired by a proposal for named instances in Haskell [20].

6.4 Nested Types in Modules for ML

There is a rough correspondence between concepts in $\mathcal{G}$ and signatures in ML, and between models in $\mathcal{G}$ and structures in ML. In Standard ML, signatures have nested types, but the nested types are transparent. That is, type checking is performed after functor instantiation, after concrete types have been assigned to the nested types. Thus ML functors are in some respects similar to class templates with nested types in C++. There have been several proposals [14, 26] for changing the module system to also support opaque nested types, which would correspond to the associated types of $\mathcal{G}$.

7. FUTURE DIRECTIONS

Work on $\mathcal{G}$ proceeds in three directions: the compiler, the formal definition of $\mathcal{G}$, and evaluation of $\mathcal{G}$. The compiler currently accepts only a subset of $\mathcal{G}$, so the compiler must be extended to the full language. Once that is complete, optimization passes will be added to support function inlining and specialization (and related supporting optimizations) and small object optimization. A formal definition of the type system and dynamic semantics of $\mathcal{G}$ is being developed with the Isabelle/Isar proof system [36, 37]. The goal is to provide a rigorous definition of $\mathcal{G}$ and to prove it type safe. The design and implementation of $\mathcal{G}$ will be evaluated by constructing several generic libraries in $\mathcal{G}$, including a subset of the STL and the Boost Graph Library [43]. We will evaluate $\mathcal{G}$ using the criteria from our comparative study [12].

8. CONCLUSION

In this paper we have reviewed the support for generic programming in C++ and Haskell, analyzing their strengths and weaknesses in four key areas: name scoping and type checking, algorithm specialization, associated types, and compilation (separate compilation and efficiency). Based on this analysis we have presented the design for $\mathcal{G}$ and discussed how the problems encountered in C++ and Haskell are overcome. To this end, three novel language features are proposed: scoped models declarations, nested types in concepts, and optional constraints.

Acknowledgments

Most of the design issues of $\mathcal{G}$ were worked out during many fruitful discussions with Jaakko Järvi and Jeremiah Willcock. The formulation of concepts in $\mathcal{G}$ was inspired by how concepts were documented in the SGI STL documentation, which was written by Matthew Austern and based on work by Alexander Stepanov and David Musser. Discussions with Bjarne Stroustrup were also very helpful. The design of $\mathcal{G}$ was heavily influenced by Haskell's type classes, so many thanks to Philip Wadler, Stephen Blott, and the rest of the Haskell community. This work was supported by NSF grant EIA-0131354 and a grant from the Lilly Endowment.

9. REFERENCES

[1] *Haskell 98 Language and Libraries: The Revised Report*, December 2002. http://www.haskell.org/onlinereport/index.html.

[2] L. Augustsson. Implementing Haskell overloading. In *Functional Programming Languages and Computer Architecture*, pages 65–73, 1993.

[3] M. Austern. C++ standard library defect report list (revision 28). Technical Report N1538=03-0121, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, November 2003. http://www.open-std.org/jtc1/sc22/wg21/prot/14882fdis/lwg-defects.html#%225.

[4] M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.

[5] J.-D. Boissonnat, F. Cazals, F. Da, O. Devillers, S. Pion, F. Rebufat, M. Teillaud, and M. Yvinec. Programming with cgal: the example of triangulations. In *Proceedings of the fifteenth annual symposium on Computational geometry*, pages 421–422. ACM Press, 1999.

[6] G. Brown, H. Lee, and T. Schulthess. C++ and generic programming for rapid development of monte carlo simulations. In *17th Annual Workshop on Recent Developments in Computers Simulations Studies in Condensed Matter Physics*. Springer-Verlag, 2004.

[7] K. B. Bruce, L. Cardelli, G. Castagna, J. Eifrig, S. F. Smith, V. Trifonov, G. T. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[8] M. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *ICFP*, 2004. submitted.

[9] G. Colvin and B. Dawes. *Smart Pointers*. Boost. http://www.boost.org/libs/smart_ptr/smart_ptr.htm.

[10] M. Erwig. *FGL/Haskell - A Functional Graph Library for Haskell*, January 2004. http://web.engr.oregonstate.edu/~erwig/fgl/haskell/.

[11] I. O. for Standardization (ISO). *ISO/IEC Final Draft International Standard 14882: Programming Language C++*. 1 rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland, 1998.

[12] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 115–134. ACM Press, Oct. 2003.

[13] D. Gregor. *Boost.Function*. Boost. http://www.boost.org/doc/html/function.html.

[14] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. pages 123–137, Portland, OR, January 1994.

[15] J. Järvi, J. Willcock, and A. Lumsdaine. Algorithm specialization and concept constrained genericity. In *Concepts: a Linguistic Foundation of Generic Programming*. Adobe Systems, Apr. 2004.

[16] M. Jazayeri, R. Loos, D. Musser, and A. Stepanov. Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, Schloss Dagstuhl, Germany, Apr. 1998.

[17] M. Jones. Typing haskell in haskell. In *Haskell Workshop*, September 1999. http://citeseer.ist.psu.edu/424440.html.

[18] M. P. Jones. Partial evaluation for dictionary-free overloading. Technical Report YALEU/DCS/RR-959, 1993.

[19] M. P. Jones. Type classes with functional dependencies. In *European Symposium on Programming*, number 1782 in LNCS, pages 230–244. Springer-Verlag, March 2000.

[20] W. Kahl and J. Scheffczyk. Named instances for Haskell type classes. In R. Hinze, editor, *Proc. Haskell Workshop 2001*, volume 59 of *ENTCS*, 2001. See also: http://ist.unibw-muenchen.de/Haskell/NamedInstances/.

[21] A. Kershenbaum, D. Musser, and A. Stepanov. Higher order imperative programming. Technical Report 88-10, Rensselaer Polytechnic Institute, 1988.

[22] U. Köthe. *Handbook on Computer Vision and Applications*, volume 3, chapter Reusable Software in Computer Vision. Acadamic Press, 1999.

[23] J. Launchbury, J. Lewis, and B. Cook. On embedding a microarchitectural design language within haskell. http://www.cse.ogi.edu/PacSoft/projects/Hawk/.

[24] D. Leijen. *wxHaskell*. http://wxhaskell.sourceforge.net/.

[25] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Domain-Specific Languages*. USENIX, October 1999. http://www.usenix.org/events/dsl99/leijen.html.

[26] X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 109–122, 1994.

[27] A. Lumsdaine, L.-Q. Lee, and J. Siek. The iterative template library home page. `http://ww.osl.iu.edu/research/itl`.

[28] J. 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://anubis.dkuug.dk/jtc1/sc22/wg21.

[29] S. Mechveliani. *DoCon the Algebraic Domain Constructor*. Program Systems Institute, Pereslavl-Zalessky, Russia, 2002. http://haskell.org/docon/.

[30] J. Moses. The function of function in lisp. *SIGSAM Bulletin*, (15), 1970.

[31] M. Müller. Abstraction benchmarks and performance of C++ applications. In *Proceedings of the Fourth International Conference on Supercomputing in Nuclear Applications*, 2000.

[32] D. R. Musser and A. A. Stepanov. A library of generic algorithms in Ada. In *Using Ada (1987 International Ada Conference)*, pages 216–225, New York, NY, Dec. 1987. ACM SIGAda.

[33] D. R. Musser and A. A. Stepanov. Generic programming. In P. P. Gianni, editor, *Symbolic and algebraic computation: International Symposium ISSAC '88, Rome, Italy, July 4–8, 1988: Proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Berlin, 1989. Springer Verlag.

[34] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software — Practice and Experience*, 24(7):623–642, July 1994.

[35] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[36] T. 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.

[37] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[38] C. Okasaki. An overview of edison. In *Hakell Workshop*, September 2000. http://www.cs.nott.ac.uk/~gmh/papers/5.ps.

[39] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 11, 1996.

[40] W. R. Pitt, M. A. Williams, M. Steven, B. Sweeney, A. J. Bleasby, and D. S. Moss. The bioinformatics template librarygeneric components for biocomputing. *Bioinformatics*, 17(8):729–737, 2001.

[41] A. D. Robison. The abstraction penalty for small objects in C++. In *POOMA'96: The Parallel Object-Oriented Methods and Applications Conference*, Feb. 28 - Mar. 1 1996. Santa Fe, New Mexico.

[42] J. Siek, L.-Q. Lee, and A. Lumsdaine. The generic graph component library. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 399–414. ACM Press, 1999.

[43] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[44] J. Siek and A. Lumsdaine. *Modern Software Tools for Scientific Computing*, chapter A Modern Framework for Portable High Performance Numerical Linear Algebra. Birkhäuser, 1999.

[45] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[46] B. Stroustrup. Parameterized types for C++. In *USENIX C++ Conference*, October 1988.

[47] B. Stroustrup. Concepts – a more abstract complement to type checking. Technical Report N1510=03-0093, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Oct. 2003. http://anubis.dkuug.dk/jtc1/sc22/wg21.

[48] B. Stroustrup and G. Dos Reis. Concepts – design choices for template argument checking. Technical Report N1522=03-0105, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Oct. 2003. http://anubis.dkuug.dk/jtc1/sc22/wg21.

[49] B. Stroustrup and G. Dos Reis. Concepts – syntax and composition. Technical Report N1536=03-0119, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Oct. 2003. http://anubis.dkuug.dk/jtc1/sc22/wg21.

[50] M. Troyer, S. Todo, S. Trebst, and A. F. and. *ALPS: Algorithms and Libraries for Physics Simulations*. http://alps.comp-phys.org/.

[51] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Berlin, Heidelberg, New York, Tokyo, 1997. Springer-Verlag.

[52] 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, Jan. 1989.

[53] J. Walter and M. Koch. *uBLAS*. Boost. http://www.boost.org/libs/numeric/ublas/doc/index.htm.