Language Requirements for Large-Scale Generic Libraries

Jeremy Siek and Andrew Lumsdaine
{jsiek,lums}@osl.iu.edu

Open Systems Laboratory
Indiana University

Abstract. The past decade of experience has demonstrated that the generic programming methodology is highly effective for the design, implementation, and use of large-scale software libraries. The fundamental principle of generic programming is the realization of interfaces for entire sets of components, based on their essential syntactic and semantic requirements, rather than for any particular components. Many programming languages have features for describing interfaces between software components, but none completely support the approach used in generic programming. We have recently developed G, a language designed to provide first-class language support for generic programming and large-scale libraries. In this paper, we present an overview of G and analyze the interdependence between language features and libraries design in light of a complete implementation of the Standard Template Library using G. In addition, we discuss important issues related to modularity and encapsulation in large-scale libraries and how language support for validation of components in isolation can prevent many common problems in component integration.

1 Introduction

In the 1980s Musser and Stepanov developed a methodology for creating highly reusable algorithm libraries [1–4], using the term "generic programming" for their work.[1] Their approach was novel in that the algorithms in their libraries were not written based on any particular data structure. Rather, the algorithms were written based on specifications of requirements that a structure would have to meet in order for the algorithm to be correct. Such algorithms ("generic algorithms") were therefore able to operate on any data structure at all, provided the structure met the specified requirements. For example, a given generic algorithm could operate on linked lists, arrays, red-black trees (representing ordered sequences), and any other structure (in particular, structures developed independently of the generic library) meeting the requirements of that algorithm. Early versions of generic algorithm libraries were implemented in Scheme, Ada, and C.

In the early 1990s Stepanov and Musser took advantage of the template system in C++ [5] to construct the Standard Template Library (STL) [6,7]. The STL became part of the C++ Standard, which brought their style of generic programming into the mainstream. Since then, the methodology has been successfully applied to the creation of libraries for numerous domains [8–12].

[1] 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 ("polytypic programming"). Here, we use the term "generic programming" solely in the sense of Musser and Stepanov.

The ease with which programmers implement and use generic libraries varies greatly depending on the language features available for expressing polymorphism and requirements on type parameters. In [13] we performed a comparative study of modern language support for generic programming, implementing a representative subset of the Boost Graph Library [9] in each of six languages. While some languages performed quite well, none were ideal for generic programming.

The language presented here, named G, was designed explicitly for generic programming. In [14] we laid the foundation for G, defining a core calculus, named F^G, based on System F [15, 16]. In F^G we captured the essential features for generic programming in a small formal system and proved type safety. The language G applies the ideas from F^G to a full programming language capable of implementing the entire STL.

1.1 Contributions

The contributions of this paper are the design and evaluation of a language for generic programming:

- We give a high-level and intuitive description of the language G. A formal description of the idealized core of G, named F^G, is presented in [14]. We leave a formal description of the full language G for future work.
- We evaluate the design of G with respect to implementing the Standard Template Library. The STL is a large generic library that exercises all aspects of the generic programming methodology. The STL is therefore a fitting first test for validating the design of G.
- We evaluate the design of G with respect to scalability issues in software development. In particular, we show how G provides support for the independent validation of components and support for component integration.

Many elements of G can be found in other programming languages, but G is unique in providing a carefully selected combination of language features for generic programming. In terms of interface description, the closest relative to G is Haskell's type classes. However, G differs in that 1) the concept feature in G integrates nested types and type sharing (similar to ML), 2) model definitions in G obey normal scoping rules, and 3) G explores design issues of type classes for non-type-inferencing languages.

1.2 Road Map

In Section 2 we review the essential ideas and terminology of generic programming and present an overview of the language G in Section 3. We review the high-level structure of the Standard Template Library in Section 4 and analyze the use of language features in G to implement the STL in Section 5. In Section 6 we show how component development and integration is facilitated by the G type system. Related work is discussed in Section 7. We conclude the paper in Section 8.

2 Generic Programming

Defining characteristics of the generic programming methodology are the following:

- Algorithms are expressed with minimal assumptions about data abstractions, and vice versa, making them maximally interoperable. This is accomplished by taking a concrete algorithm and lifting the non-essential requirements. For example, an algorithm on linked-lists becomes an algorithm on forward iterators.

- Absolute efficiency is required. Algorithms are never lifted to the point where they lose efficiency. When a single generic algorithm can not achieve the best efficiency for all input types, multiple generic algorithms are implemented and automatic algorithm selection is provided.

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 a set of requirements on a type (or on several types) and 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.

Concepts are used to specify interfaces to a generic algorithm and thus ***constrain*** the type parameters to that algorithm. A generic algorithm may only be used with type arguments that model its constraining concepts.

An example of a concept from the STL is Input Iterator (a sans-serif font is used to distinguish names of concepts). This concept is a refinement of the Assignable, Copy Constructible, and Equality Comparable concepts. In addition, a type `X` is a model of Input Iterator if it satisfies the following:

- It has increment and dereference functions;
- It has two associated helper types: a value type, which is the return type of the dereference operator, and a difference type, which is some model of SignedIntegral suitable for measuring distances between iterators; and
- Given objects `a` and `b` of type `X`, `a == b` implies `*a` is equivalent to `*b`; and
- The increment and dereference operators must be constant time.

In general, a concept consists of five kinds of requirements: operations, ***associated types***, nested requirements, semantic invariants, and complexity guarantees. To further elaborate, associated types are types needed for the operations required by the concept and that are determined by the modeling type but which may vary from one model of the concept to another.

In C++, the types `int*` and `list<char>::iterator` are examples of types that model Input Iterator. The associated value type for `int*` is `int` and the associated value type for `list<char>::iterator` is `char`. The concept Input Iterator is directly used as a type requirement in over 28 of the STL algorithms. One example is the `copy` algorithm, which requires the first range to be a model of Input Iterator. The following is the signature for the C++ STL `copy` algorithm:

```
template<class InputIterator, class OutputIterator>
OutputIterator
copy(InputIterator first, InputIterator last, OutputIterator result);
```

3 Overview of G

G is a statically typed imperative language with syntax and memory model similar to C++. We have implemented a compiler that translates G to C++, but G could also be interpreted or compiled to byte-code. Compilation units are separately type checked and may be separately compiled, relying only on forward declarations from other compilation units (even compilation units containing generic functions and classes). The languages features of G that support generic programming are the following:

- Concept and model definitions;
- Constrained polymorphic functions, classes, structs, and type-safe unions;
- Implicit instantiation of polymorphic functions; and
- Concept-based function overloading.

In addition, G includes the usual basic types and control constructs of a general purpose programming language.

Concepts are defined using the following syntax:

```
decl ← concept id <id,...> { cmem ... };
cmem ← funsig | fundef       // Operations
       | type id;            // Associated types
       | ty == ty;           // Same-type constraints
       | refines id<ty, ...>; | require id<ty, ...>;
```

The identifiers in the `<>`'s are place holders for the modeling type (or list of types). The distinction between `refines` and `require` is that refinement brings in the associated types from the "super" concept and also plays a role in function overloading. The following is the definition of the `InputIterator` concept in 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!;
};
```

The modeling relation between a type and a concept is established with a model definition using the following syntax.

```
decl ← model [<id,...>] [where { constraint, ... }] id <ty,...> { decl ... };
```

A model may be parameterized: the identifiers in the `<>`'s are type parameters and the `where` clause introduces concept and same-type constraints:

```
constraint ← id<ty, ...> | ty == ty
```

The following statement establishes that all pointer types are models of `InputIterator`.

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

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. Refinements and nested requirements are satisfied by preceding model definitions.

The syntax for polymorphic functions is shown below. The return type of a function is delimited by `->`.

```
fundef ← fun id [<id,...>] [where { constraint, ... }]
                (ty pass [id], ...) -> ty pass { stmt ... }
funsig ← fun id [<id,...>] [where { constraint, ... }]
               (ty pass [id], ...) -> ty pass;
decl ← fundef | funsig
pass ← ! | @ | /* nothing */ | &
```

The default parameter passing mode in 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 by `@`. The body of a polymorphic function is type checked separately from any instantiation of the function. The `where` clause introduces surrogate model definitions and signatures (for all required concept operations) into the scope of the function. The generic `distance` function is a simple example.

```
fun distance<Iter> where { InputIterator<Iter> }
(Iter@ first, Iter last) -> InputIterator<Iter>.difference@ {
  let n = zero();
  while (first != last) { ++first; ++n; }
  return n;
}
```

The dot notation used in the return type refers to an associated type, in this case the `difference` type of the iterator.

```
assoc ← id<ty, ...>.id | id<ty, ...>.assoc
ty ← assoc
```

Inside `distance` we use the following three kinds of statements. The `let` statement introduces a variable bound to the value of the expression on the right-hand side. The scope is to the end of the enclosing block and the type of the variable is the type of the right-hand side. G includes `while`, `return`, and the usual control constructs of C++.

```
stmt ← let id = expr; | while (expr) stmt | return expr; | ...
```

Multiple functions with the same name may be defined, and static overload resolution is performed by G to decide which function to invoke at a particular call site depending on the argument types and also depending on which model definitions are in scope. When more than one overload may be called, the more specific overload is called if one exists ("more specific" is a partial order). The `where` clause and the concept refinement hierarchy are a factor in the partial ordering.

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

```
decl ← class id polyhdr { classmem ... };
decl ← struct id polyhdr { mem ... };
decl ← union id polyhdr { mem ... };
mem ← ty id;
classmem ← mem
         | polyhdr id(ty pass [id], ...) { stmt ... }
         | ~id() { stmt ... }
polyhdr ← [<id,...>] [where { constraint, ... }]
```

Classes consist of data members, constructors, and a destructor. There are no member functions; normal functions are used instead. Data encapsulation (`public`/`private`) is

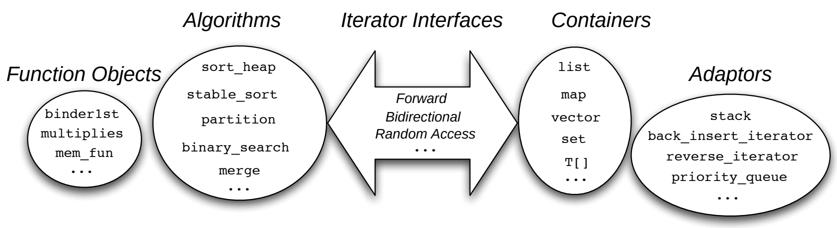

Fig. 1. High-level structure of the STL.

specified at the module level instead of inside the class. Class, struct, and unions are used as types 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.

$ty \leftarrow id[\texttt{<}ty\texttt{,} \ldots\texttt{>}]$

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

$expr \leftarrow expr\texttt{(}expr\texttt{,} \ldots\texttt{)}$

Arguments for the type parameters of a polymorphic function need not be supplied at the call site: G will deduce the type arguments by unifying the types of the arguments with the types of the parameters and ***implicitly instantiate*** the polymorphic function. All of the requirements in the `where` clause must be satisfied by model definitions in the lexical scope preceding the function call. The following is a program that calls the `distance` function, applying it to iterators of type `int*`.

```
fun main() -> int@ {
  let p = new int[8];
  let d = distance(p, p + 4);
  return d == 4 ? 0 : -1;
}
```

A polymorphic function may be explicitly instantiated using this syntax:

$expr \leftarrow expr\texttt{<|}ty\texttt{,} \ldots\texttt{|>}$

4 Overview of the STL

The high-level structure of the STL is shown in Fig. 1. The STL contains over fifty generic algorithms. Prior to the STL, algorithms such as these were implemented in terms of concrete data structures such as linked lists and arrays. The STL generic algorithms abstract away from the non-essential characteristics of these data structures, implementing them in terms of a family of iterator abstractions. As a result, the STL algorithms may be used with an infinite set of concrete data structures, i.e., any data structure that exports iterators with the required capabilities.

Fig. 2 shows the hierarchy of STL's iterator concepts. An arrow indicates that the source concept is a refinement of the target. The iterator concepts arose from the requirements of algorithms: the need to express the minimal requirements for each algorithm. For example, the `merge` algorithm passes through a sequence once, so it only requires the basic requirements of Input Iterator. On the other hand, `sort_heap` requires iterators that can jump arbitrary distances, so it requires Random Access Iterator.

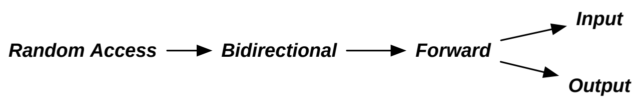

Fig. 2. Iterator concept hierarchy.

The STL includes a handful of common data structures. When one of these data structures does not fulfill some specialized purpose, the programmer is encouraged to implement the appropriate specialized data structure. All of the STL algorithms can then be made available for the new data structure at the small cost of implementing iterators for the specialized data structure.

Many of the STL algorithms are higher-order: they take functions as parameters, allowing the user to customize the algorithm to their own needs. The STL defines over 25 function objects for creating and composing functions.

The STL also contains a collection of adaptor classes, which are parameterized classes that implement some concept in terms of the type parameter (which is the adapted type). For example, the `back_insert_iterator` adaptor implements Output Iterator in terms of any model of Back Insertion Sequence. The generic `copy` algorithm can then be used with `back_insert_iterator<list<int>>` to append some integers to a list. Adaptors play an important role in the plug-and-play nature of the STL and enable a high degree of reuse. For example, the `find_last_subsequence` function is implemented using `find_subsequence` and the `reverse_iterator` adaptor.

5 Analysis of G and the STL

In this section we analyze the interdependence of the language features of 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 G can be used to accomplish this. Another goal of generic programming is efficiency, so we investigate the use of function overloading in G to accomplish automatic algorithm selection. We conclude this section with a brief look at implementing generic containers and adaptors in G.

Algorithms. Fig. 3 depicts a few simple STL algorithms implemented using polymorphic functions in G. The STL provides two versions of most algorithms, such as the overloads for `find` in Fig. 3. 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 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 and `find`.

Iterators. Fig. 4 shows the STL iterator hierarchy as represented in 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` concept refines `InputIterator` and `OutputIterator`.

```
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);
}
```

Fig. 3. Some STL Algorithms in G.

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 G: they are beyond the scope of its type system.

Automatic Algorithm Selection. To realize the generic programming efficiency goals, 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, which 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. We call this ***concept-based overloading***.

```
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 for 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`.

```
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>;
};
```

Fig. 4. The STL Iterator Concepts in G (`Iterator` has been abbreviated to `Iter`).

```
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 have 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 will resolve 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 will use 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.

Containers. The containers of the STL are implemented in G using polymorphic types. Fig. 5 shows an excerpt of the doubly-linked `list` container in 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. Fig. 5 shows the `list_iterator` which translates `operator*` to `x.node->data` and `operator++` to `x.node = x.node->next`.

Not shown in Fig. 5 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 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 G because the parameter passing mode may not be parameterized. Further, the semantics of polymorphism in 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 G. As a temporary solution, we used the m4 macro system to factor the

```
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); }
```

Fig. 5. Excerpt from a doubly-linked list container in G.

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 */
```

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 Component Development Benefits

Generic programming has enabled programmers from all over the world to construct and share interchangeable components. An example of this is the Boost collection of C++ libraries [17]. While this has benefited programmer productivity, there is room to improve: the cost of reuse is still too high. Programmers routinely run into component integration problems such as namespace pollution, libraries with type errors, documentation inconsistencies, long compile times, and hard to understand error messages. Many languages provide the necessary modularity to solve these problems, but lack the abstractions to express the STL. On the other hand, C++ can easily express the STL but lacks modularity. The point of this section is to show that not only is G suitable for expressing the STL but it also provides modularity.

Namespace pollution issues related to `cpp` macros are an old story, but generic programming brings with it new and subtle issues. For example, function templates in C++ rely on argument dependent lookup (ADL) [18] to access user-defined operations, but ADL breaks namespace modularity. There is tension in between the need to allow for rich interfaces and user-supplied operations while at the same time ensuring modularity. In G this tension is resolved through the use of concepts and `where` clauses that provide a mechanism for specifying rich interfaces while at the same time separating library and user namespaces.

Users of generic libraries in C++ are plagued by long compile times and hard to understand error messages. The reason is C++'s lack of separate compilation and separate type checking. G addresses both of these problems. In G, generic libraries can be compiled to object code so the user need only link them to the executable. Many of the hard to understand error messages in C++ come from misuses of generic algorithms. For example, the following G program misuses `stable_sort`: it requires a random access iterator but `list` only provides bidirectional.

```
4  fun main() -> int@{
5    let v = @list<int>();
6    stable_sort(begin(v), end(v));
7    return 0;
8  }
```

In C++ this would evoke pages of error messages with line numbers pointing deep inside the implementation of `stable_sort`. In contrast, the G compiler prints the following:

```
test/stable_sort_error.hic:6:
```

```
In application stable_sort(begin(v), end(v)),
Model MutableRandomAccessIterator<mutable_list_iter<int>>
needed to satisfy requirement, but it is not defined.
```

Another problem that plagues generic C++ libraries is that type errors often go unnoticed during library development. This is because type checking of templates 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 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 will not accept generic functions invoked with inconsistent types.

Interestingly, while implementing the STL in 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 [18].

```
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 G compiler gives the following error message:

```
stl/sequence_mutation.hic: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.

7 Related Work

There is a long history of programming language support for polymorphism, dating back to the 1970s [15, 16, 19, 20]. An early precursor to G's concept feature can be seen in CLU's type set feature [19]. In mathematics, the equivalent notion of algebraic structure has been in use for an even longer time [21].

The concept feature in G is heavily influenced the type class feature of Haskell [22], with its nominal conformance and explicit model definitions. However, G's support for associated types, same type constraints, and concept-based overloading is novel. Also, G's type system is fundamentally different from Haskell's: it is based on System F [15, 16] instead of Hindley-Milner type inferencing [20]. This difference has some repercussions. In G there is more control over the scope of concept operations because `where` clauses introduce concept operations into the scope of the body. This difference allows Haskell to infer type requirements but induces the restriction that two type classes in the same module may not have operations with the same name. A difference we discuss in [14] is that in G, overlapping models may coexist in separate scopes but still be used in the same program, whereas in Haskell overlapping models may not be used in the same program. In [13] we performed a comparative study of support for generic programming in several language and Haskell performed quite well. We pointed out that Haskell was missing support for associated types, and work to remedy this has been reported in [23]. The approach in [23] adds datatype definitions to type classes, whereas G's associated types are closer to nested types in ML signatures [24].

Less closely related to G are languages based on subtype-bounded polymorphism [25, 26] such as Java, C#, and Eiffel. We found subtype-bounded polymorphism less suitable for generic programming and refer the reader to [13] for an in-depth discussion. More recently, the object-oriented language Scala [27, 28] has added abstract type members based of the theory of dependent types. A comparison of this with G's associated types is planned for future work.

8 Conclusion

This paper presented the design of a new programming language named G and demonstrated with an implementation of the Standard Template Library that this language is well-suited to generic programming. We were able to implement all of the abstractions in the STL in a straightforward manner. Further, G is particularly well-suited for the development of reusable components due to its support of separate type checking and compilation. G's strong type system provides support for the independent validation of components and G's system of concepts and constraints allows for rich interactions between components without sacrificing namespace safety. As a result, the language features present in G hold some promise to increase programmer productivity with respect to the development and use of generic components.

Acknowledgments

We would like to thank Ronald Garcia, Jeremiah Willcock, Doug Gregor, Jaakko Järvi, Dave Abrahams, Dave Musser, and Alexander Stepanov for many discussions and collaborations that informed this work. This work was supported by NSF grant EIA-0131354 and by a grant from the Lilly Endowment.

References

1. Kapur, D., Musser, D.R., Stepanov, A.: Operators and algebraic structures. In: Proc. of the Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire, ACM (1981)
2. Musser, D.R., Stepanov, A.A.: Generic programming. In Gianni, P.P., ed.: Symbolic and algebraic computation: ISSAC '88, Rome, Italy, July 4–8, 1988: Proceedings. Volume 358 of Lecture Notes in Computer Science., Berlin, Springer Verlag (1989) 13–25
3. Musser, D.R., Stepanov, A.A.: A library of generic algorithms in Ada. In: Using Ada (1987 International Ada Conference), New York, NY, ACM SIGAda (1987) 216–225
4. Kershenbaum, A., Musser, D., Stepanov, A.: Higher order imperative programming. Technical Report 88-10, Rensselaer Polytechnic Institute (1988)

5. Stroustrup, B.: Parameterized types for C++. In: USENIX C++ Conference. (1988)
6. Stepanov, A.A., Lee, M.: The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project (1994)
7. Austern, M.H.: Generic Programming and the STL. Professional computing series. Addison-Wesley (1999)
8. Köthe, U.: Reusable Software in Computer Vision. In: Handbook on Computer Vision and Applications. Volume 3. Acadamic Press (1999)
9. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley (2002)
10. Boissonnat, J.D., Cazals, F., Da, F., Devillers, O., Pion, S., Rebufat, F., Teillaud, M., Yvinec, M.: Programming with CGAL: the example of triangulations. In: Proceedings of the fifteenth annual symposium on Computational geometry, ACM Press (1999) 421–422
11. Pitt, W.R., Williams, M.A., Steven, M., Sweeney, B., Bleasby, A.J., Moss, D.S.: The bioinformatics template library: generic components for biocomputing. Bioinformatics **17** (2001) 729–737
12. Troyer, M., Todo, S., Trebst, S., and, A.F.: (ALPS: Algorithms and Libraries for Physics Simulations) `http://alps.comp-phys.org/`.
13. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: 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, ACM Press (2003) 115–134
14. Siek, J., Lumsdaine, A.: Essential language support for generic programming. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'05. (2005) accepted for publication.
15. Girard, J.Y.: Interprtation Fonctionnelle et Élimination des Coupures de l'Arithmtique d'Ordre Suprieur. Thse de doctorat d'tat, Universit Paris VII, Paris, France (1972)
16. Reynolds, J.C.: Towards a theory of type structure. In Robinet, B., ed.: Programming Symposium. Volume 19 of LNCS., Berlin, Springer-Verlag (1974) 408–425
17. Boost: (Boost C++ Libraries) `http://www.boost.org/`.
18. International Standardization Organization (ISO): ANSI/ISO Standard 14882, Programming Language C++, 1 rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland (1998)
19. Liskov, B., Snyder, A., Atkinson, R., Schaffert, C.: Abstraction mechanisms in CLU. Communications of the ACM **20** (1977) 564–576
20. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–375
21. Bourbaki, N.: Elements of Mathematics. Theory of Sets. Springer (1968)
22. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: ACM Symposium on Principles of Programming Languages, ACM (1989) 60–76
23. Chakravarty, M., Keller, G., Jones, S.P., Marlow, S.: Associated types with class. In: Proceedings of the 32nd ACM-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, ACM (2005) 1–13
24. Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press (1990)
25. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Computing Surveys **17** (1985) 471–522
26. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: Proceedings of the fourth international conference on functional programming languages and computer architecture. (1989)
27. Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: Proc. ECOOP'03. Springer LNCS (2003)
28. Odersky, M., al.: An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland (2004)