Concepts: Linguistic Support for Generic Programming in C++

Douglas Gregor
Indiana University
dgregor@osl.iu.edu

Jaakko Järvi
Texas A&M University
jarvi@cs.tamu.edu

Jeremy Siek
Rice University
Jeremy.G.Siek@rice.edu

Bjarne Stroustrup
Texas A&M University
bs@cs.tamu.edu

Gabriel Dos Reis
Texas A&M University
gdr@cs.tamu.edu

Andrew Lumsdaine
Indiana University
lums@osl.iu.edu

Abstract

Generic programming has emerged as an important technique for the development of highly reusable and efficient software libraries. In C++, generic programming is enabled by the flexibility of templates, the C++ type parametrization mechanism. However, the power of templates comes with a price: generic (template) libraries can be more difficult to use and develop than non-template libraries and their misuse results in notoriously confusing error messages. As currently defined in C++98, templates are unconstrained, and type-checking of templates is performed late in the compilation process, i.e., after the use of a template has been combined with its definition. To improve the support for generic programming in C++, we introduce *concepts* to express the syntactic and semantic behavior of types and to constrain the type parameters in a C++ template. Using concepts, type-checking of template definitions is separated from their uses, thereby making templates easier to use and easier to compile. These improvements are achieved without limiting the flexibility of templates or decreasing their performance—in fact their expressive power is increased. This paper describes the language extensions supporting concepts, their use in the expression of the C++ Standard Template Library, and their implementation in the ConceptGCC compiler. Concepts are candidates for inclusion in the upcoming revision of the ISO C++ standard, C++0x.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

General Terms Design, Languages

Keywords Generic programming, constrained generics, parametric polymorphism, C++ templates, C++0x, concepts

1. Introduction

The C++ language [25, 62] supports parametrized types and functions in the form of *templates*. Templates provide a unique combination of features that have allowed them to be used for many different programming paradigms, including Generic Programming [3,44], Generative Programming [11], and Template Metaprogramming [1, 66]. Much of the flexibility of C++ templates comes from their unconstrained nature: a template can perform any operation on its template parameters, including compile-time type computations, allowing the emulation of the basic features required for diverse programming paradigms. Another essential part of templates is their ability to provide abstraction without performance degradation: templates provide sufficient information to a compiler's optimizers (especially the inliner) to generate code that is optimal in both time and space.

Consequently, templates have become the preferred implementation style for a vast array of reusable, efficient C++ libraries [2, 6, 14, 20, 32, 54, 55, 65], many of which are built upon the Generic Programming methodology exemplified by the C++ Standard Template Library (STL) [42, 60]. Aided by the discovery of numerous *ad hoc* template techniques [28, 46, 56, 66, 67], C++ libraries are becoming more powerful, more flexible, and more expressive.

However, these improvements come at the cost of implementation complexity [61, 63]: authors of C++ libraries typically rely on a grab-bag of template tricks, many of which are complex and poorly documented. Where library interfaces are rigorously separated from library implementation, the complexity of implementation of a library is not a problem for its users. However, templates rely on the absence of modular (separate) type-checking for flexibility and performance. Therefore, the complexities of library implementation leak through to library users. This problem manifests itself most visibly in spectacularly poor error messages for simple mistakes. Consider:

```
list<int> lst;
sort(lst.begin(), lst.end());
```

Attempting to compile this code with a recent version of the GNU C++ compiler [17] produces more than two kilobytes of output, containing six different error messages. Worse, the errors reported provide line numbers and file names that point to the implementation of the STL sort() function and its helper functions. The only clue provided to users that this error was triggered by their own code (rather than by a bug in the STL implementation) is the following innocuous line of output:

```
sort_list.cpp:8:   instantiated from here
```

The actual error, in this case, is that the STL sort() requires a pair of Random Access Iterators, i.e., iterators that can move any number of steps forward or backward in constant time. The STL

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

```
sort_list.cpp: In function 'int main()':
sort_list.cpp:8: error: no matching function for call to 'sort(std::_List_iterator<int>, std::_List_iterator<int>)'
.../stl_algo.h:2835: note: candidates are: void std::sort(_Iter, _Iter) [with _Iter = std::_List_iterator<int>]
sort_list.cpp:8: note:   no concept map for requirement 'std::MutableRandomAccessIterator<std::_List_iterator<int> >'
```

Figure 1. ConceptGCC error message produced when attempting to compile a call to the STL sort() with list iterators.

list container, however, provides only Bidirectional Iterators, which can move forward or backward only one step at a time. Random Access Iterator and Bidirectional Iterator are *concepts* used in the design and documentation of the STL. A *concept* describes a set of requirements that a generic algorithm (implemented as a C++ function template) places on its template parameters. Concepts have no representation within the C++ language, so a user's failure to meet the concept requirements of an STL algorithm will only be detected when the compiler attempts to instantiate the algorithm.

The absence of modular type-checking for templates also means that the definitions of templates are not type checked independently of their uses. This causes serious problems for implementers of generic libraries because bugs can go uncaught prior to deployment. In particular, it is common for inconsistencies to exist between the documented type requirements and the implementation of a function template. A form of separate type-checking can be somewhat approximated using existing C++ language mechanisms [56]. However, with this approach ("concept checking" or "constraints checking"), concepts are not language entities. As a result, concept-checking libraries are quite difficult to design, must be manually verified, and suffer from portability problems.

Modern template libraries rely on a set of naming and documentation conventions to express their fundamental design ideas, such as the Bidirectional Iterator and Random Access Iterator abstractions, and state the requirements of generic algorithms in terms of those ideas. The root cause of most of the problems with templates is that these fundamental design ideas cannot be directly expressed using C++98 templates, so compilers cannot detect errors early and cannot report them in the terms used by the library documentation. Direct language support for concepts addresses this problem.

This paper describes a set of extensions to C++ that directly supports the notion of *concepts*; thus providing greater expressive power (e.g., in overloading) and improved modular type checking for C++ templates. For example, *concepts* directly support the expression and checking of concepts such as Bidirectional Iterator and Random Access Iterator. Using *concepts* we can place constraints on template parameters, enabling modular type-checking for templates. Having concepts directly expressible in C++ makes templates easier to write and use. For example, instead of two kilobytes of irrelevant information, our experimental compiler supporting concepts, ConceptGCC [19], produces the error message shown in Figure 1 when given the erroneous STL code above.

Besides its brevity, there are several improvements in this error message. Most importantly, the error message (there is only one) refers directly to the user's code, at the erroneous call to sort(). The user is informed that no sort() function matches, and that the cause of the failure is an unsatisfied requirement for the Mutable Random Access Iterator concept, i.e., a list iterator is not a Random Access Iterator, and therefore sort() cannot be used. By completely type-checking the call to sort() at the call site, the problem with STL implementation details "leaking" into the user's error message has been eliminated.

The improvement to error messages is a result of fundamental changes to the way templates are specified: the requirements of a (templated) algorithm on its arguments are made explicit. This allows us to reason about template code and (almost incidentally) gives the compiler the information it needs to produce radically better error messages early. It also gives information needed to improve overloading.

The primary challenge in designing a system of constrained generics for C++ templates is in providing language features that support current successful uses of the C++ template system without compromising the ideal of modular type checking for templates or damaging performance. We focus on supporting the Generic Programming paradigm, which has been used to develop many generic C++ libraries.

Early discussions on constrained generics for C++ can be found in [61]. At the time, no solution was found to the stated problem of providing constraints without unacceptably limiting flexibility or performance. The objective that standard C++ should be extended with constrained generics was made explicit in 2003 with a report laying out the main design goals [63]. There are many existing programming languages that support "generics" in some form or other. Early attempts at building the Standard Template Library focused on Scheme, then Ada [43] before settling on C++. More recently, we evaluated many more languages, including C#, Java, Cecil, ML, and Haskell, based on their ability to express the ideas of Generic Programming [15, 16]. Building on these results and the experience gained from the Generic Programming language $\mathcal{G}$ [57–59], we designed *concepts* with the following goals in mind:

- To provide direct support for Generic Programming, so that programs and libraries developed with the Generic Programming methodology can be expressed clearly and without relying on "template tricks" or documentation conventions.
- To make templates easier to write and easier to use by making a template's requirements on its arguments explicit using concepts, thereby enabling modular type-checking and making template overloading more robust.
- To provide a clear transition path from today's unconstrained templates to templates constrained by concepts. Libraries built with the Generic Programming methodology should be upgradeable to use concepts without breaking source compatibility for the vast majority of user programs.
- To do so without making programs more verbose, less flexible, or less efficient.
- To retain compatibility with C++98, both by not breaking existing code and by not introducing language rules that differ significantly from existing rules.

The work reported here is part of a large on-going effort led by the authors to provide concepts for C++0x, the next ISO C++ standard [12, 22, 23, 53, 64]. This paper reports the first implementation of constrained generics for C++ using concepts, and the first extensive evaluation of their use in developing generic libraries. The primary aim of the resulting language and library, collectively called ConceptC++, is to gain practical experience with the use of concepts.

Our goals are ambitious and, in some cases, contradictory. We report elsewhere [27] on the inherent trade-off between modular type checking and the Generic Programming notion of specialization. Specialization will be discussed in Section 2 as part of a general introduction to Generic Programming. Section 3 presents the language extensions and Section 4 gives an overview of their implementation in our compiler. Finally, we evaluate the effective-

ness of our language support for concepts by providing a concept-enhanced STL implementation, both from the perspective of library users and library developers. This analysis, along with a discussion of the practical impact of our "nearly-modular" type checking for templates, will be discussed in Section 5.

2. Generic Programming

Generic programming is a systematic approach to designing and organizing software that focuses on finding the most general (or abstract) formulation of an algorithm together with an efficient implementation [30]. The primary goal of generic programming is to make *algorithms* more widely applicable, and thus generic programming is sometimes referred to as *algorithm-oriented* [45].

2.1 Lifting and Abstraction

The generic programming process derives generic algorithms from families of concrete (non-generic) implementations that exhibit some commonality. We *lift* away unnecessary requirements on types from an implementation, thereby raising the level of abstraction. Consider the following two concrete implementations of sum(). The first computes the sum of doubles stored in an array; the second computes the sum of integers in a linked list.

```
double sum(double* array, int n) {
  double s = 0;
  for (int i = 0; i < n; ++i )
    s = s + array[i];
  return s;
}

struct node { node* next; int data; };

int sum(node* first, node* last) {
  int s = 0;
  for (; first != last; first = first→ next)
    s = s + first→ data;
  return s;
}
```

Abstractly, both implementations do the same thing: traverse a collection of elements and sum the values. However, these implementations impose additional requirements (ones that are unnecessary for the purposes of summation). In the first implementation, the elements must be of type double and be stored in an array. In the second implementation, the elements must be of type int and be stored in the node* representation of a linked list.

Fundamentally, summing the elements in a collection only requires the ability to visit the elements of the collection and add elements. A generic algorithm should therefore be able to work correctly with any collection of elements supporting traversal and the addition of elements. Using concepts, we can specify these requirements:

```
template<InputIterator Iter>
  where Addable<Iter::value_type> &&
        Assignable<Iter::value_type>
value_type sum(Iter first, Iter last, Iter::value_type s) {
  for (; first != last; ++first)
    s = s + *first;
  return s;
}
```

The sum() algorithm is implemented as a function template, with the parameter Iter for the iterator type. The algorithm can be used with any type that is an Input Iterator, i.e., it supports the ++ operation for moving from one element to another, the * operation for accessing a value, and != for testing iterator positions. This constraint for the template parameter Iter is stated in the template parameter list. Similarly, that iterator's element type, Iter::value_type, must support addition and assignment, as stated in the separate where clause. Section 3 will explain the syntax and semantics of concepts and their use as constraints in more detail.

Suppose we broaden the family of concrete implementations to include computing the product of a list of integers:

```
int product(node* first, node* last) {
  int s = 1;
  for (; first != last; first = first→ next)
    s = s * first→ data;
  return s;
}
```

The sum() algorithm can be further generalized to to compute products by replacing addition with parameters for an arbitrary binary operator and initial element. With this change, we arrive at an implementation of the STL accumulate() algorithm:

```
template<
      InputIterator Iter,
      BinaryOperation<Iter::value_type, Iter::value_type> Op>
  where Assignable<Iter::value_type, Op::result_type>
Iter::value_type
  accumulate(Iter first, Iter last, Iter::value_type s, Op op) {
    for (; first != last; ++first)
      s = op(s, *first);
    return s;
}
```

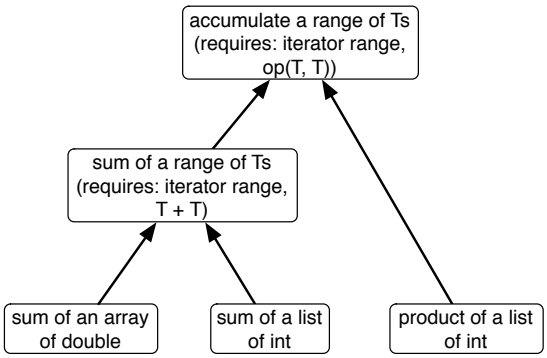
We can use the accumulate() algorithm with arrays, linked lists, or any other types that meet the requirements for Input Iterator and Binary Operation:

```
double x[10];
double a = accumulate(x, x+10, 0.0,  plus<double>());

node* n = cons(1, cons(2,  cons(2,  null )));
int  s = accumulate(n, null , 1,   multiplies <int>());

int  p = sum(n, null, 0,   plus<int>());
```

To review, we replaced requirements for particular types, such as double* or node*, with requirements for certain properties, such as InputIterator and BinaryOperation. We have specified policy—but we have not specified *how* these operations must be carried out. The lifting process can be summarized graphically as follows:



At the bottom of this figure lie the concrete implementations that form the basis of the lifting process. As we move further up the figure, we find increasingly generic implementations that subsume the concrete implementations below, each step removing unnecessary requirements. The lifting process is iterative, and at its limit lies the *ideal* of an algorithm: the generic implementation that subsumes all concrete implementations of the algorithm, specifying the minimum set of requirements to achieve maximum reusability without sacrificing efficiency.

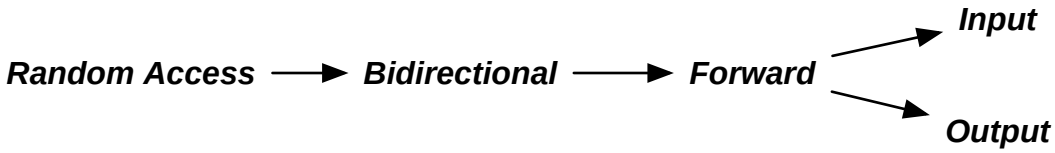


Figure 2. Refinement relationships among the STL iterator concepts, where an edge $A \rightarrow B$ is read as "A refines B."

2.2 Algorithm Specialization

In addition to building highly reusable algorithms, Generic Programming is concerned with building highly efficient algorithms. There is a natural tension between reusability and efficiency because sometimes greater efficiency (a better algorithm) is enabled by additional requirements (and adding requirements decreases opportunities for reuse). We refer to such algorithms as "specialized." When specialized algorithms provide better performance, we implement them in addition to the more generic algorithm and provide automatic dispatching to the appropriate algorithm based on the properties of the input types. Note that we do not give up on genericity: we still program in terms of properties of types and not in terms of particular types.

A simple, but important, example of algorithm specialization is the STL advance() function which moves an iterator forward n steps. The following shows two versions of advance(), the second more specialized than the first. There is a slow advance() for iterators that only provide increment, and a fast advance() for iterators that can jump forward arbitrary distances in constant time (using the $+=$ operator).

```
template<InputIterator Iter>
void advance(Iter& i, difference_type n)
{ while (n--) ++i; }

template<RandomAccessIterator Iter>
void advance(Iter& i, difference_type n)
{ i += n; }
```

At a call to advance(), the most specific overload is selected. Consider:

```
list<int> l;
advance(l.begin(), n); // dispatches to slow advance

vector<int> b;
advance(v.begin(), n); // dispatches to fast advance
```

The first call to advance() dispatches to the version for Input Iterator because that is the only overload that matches: a list iterator is not a Random Access Iterator. The second call to advance() could potentially dispatch to either version, because a vector iterator is both an Input Iterator and a Random Access Iterator. Since Random Access Iterator includes the requirements of Input Iterator, the second advance() is the more specialized algorithm, and is chosen in favor of the first.

2.3 Concepts

Generic algorithms are specified in terms of abstract properties of types, rather than in terms of particular types. In the terminology of generic programming, a *concept* is the formalization of an abstraction as a set of requirements on a type (or on a set of types, integers, operations, etc.) [3,31]. A type that implements the requirements of a concept is said to *model* the concept. These requirements may be syntactic, semantic, or performance-related. The requirements for operators $++$ and $*$ on iterators are examples of syntactic requirements. In the documentation of C++ libraries, such requirements are typically expressed in the form of *valid expressions* (also called *use patterns*); ConceptC++ allows the expression of requirements on operations directly in the language. The second kind of syntactic requirement is *associated types*: types that collaborate in the concept operations and that are determined by the modeling type. For example, the associated value and distance types of an iterator, which state the type of value that the iterator points to and the type that can be used to measure distances between iterators, respectively.

An example of a semantic requirement is that iterator equality should imply that dereferencing yields the same value: i $==$ j implies $*$i $==$ $*$j. The requirement that operator $+=$ execute in constant time is a performance requirement of the Random Access Iterator concept. We do not include support for semantic or performance requirements, though such support is under investigation.

A concept may incorporate the requirements of another concept, in which case the first concept is said to *refine* the second. Families of related concepts, called concept taxonomies, form a strong organizational basis for many generic libraries. In particular, concepts can describe the essential abstractions for an entire domain, provide common building blocks for implementing a large number of generic algorithms, and lead to a uniform interface to the generic algorithms within the domain. Figure 2 illustrates the concept taxonomy for the C++ Standard Template Library iterator concepts.

Often, a concept is parametrized over more than just one type: commonly a concept involves multiple types and specifies their relationships. For example, the algebraic concept of a Vector Space, involves two types, the vector type and the scalar type. Similarly, a concept can be parametrized by integer values and operations.

3. Language Support for Concepts

ConceptC++ translates the ideas of Generic Programming into a small set of C++ language extensions. Generic algorithms and data structures are expressed as templates with constraints on their template parameters. The constraints are expressed in terms of concepts in where clauses or the template parameter list. A concept is an interface-like definition of a set of requirements (associated types, signatures), while a concept_map definition provides a mechanism for declaring that a type satisfies a concept and for mapping a type to requirements (concepts) by providing required operations and associated types. We can illustrate the major concept features using the standard library min function:

```
concept LessThanComparable<typename T> {
  bool operator<(T x, T y);
}
template<typename T>
where LessThanComparable<T>
const T& min(const T& x, const T& y) {
  return x < y? x : y;
}
```

The first definition specifies the concept Less Than Comparable: any type T that models Less Than Comparable must provide a less-than

operator on two T objects that returns a boolean result. The second definition specifies a "constrained" function template min(), which can only be applied to types T that model Less Than Comparable. That is, we can find smaller of two objects using min provided we can compare them by <.

However, there are notions of "smaller" that are not conventionally expressed using <. For example, for complex numbers the distance from (0,0), abs, is for some algorithms suitable for comparisons. We can map abs for a complex number, dcomplex, to the < required by min:

```
concept_map LessThanComparable<complex> {
  bool operator<(complex a, complex b) {
    return abs(a)<abs(b);
  }
};
int f(dcomplex a, dcomplex b) {
  dcomplex& x = min(a, b);
  // ...
}
```

The first definition is a concept map, which establishes that the type dcomplex is a model of the Less Than Comparable concept when we use the supplied <. Finally, the f() function illustrates a call to min(): at the call site, the type T is bound to dcomplex and the concept map LessThanComparable<dcomplex> satisfies min()'s Less Than Comparable constraint.

3.1 Constraining Templates with Concepts

The syntax of C++ template declarations (and definitions) is extended to include a where clause consisting of a set of requirements. Any template that contains a where clause is called a *constrained template*. The following is a simple example of a where clause containing two requirements:

```
template<typename T>
where Assignable<T> && CopyConstructible<T>
void swap(T& a, T& b) {
  T tmp(a); // copy using constructor
  a = b; // copy using assignment
  b = tmp;
}
```

A concept name applied to a type, e.g., Assignable<T>, indicates that T must meet the requirements of the Assignable concept, e.g.,

```
int a = 10;
int b = 20;
swap(a,b);
```

Here, int must be Assignable (as well as Copy Constructible). Thus, the where clause, ensures that swap() can only be instantiated on a type that meets the requirements of the Assignable and Copy Constructible concepts. A where clause may also be used to constrain class templates and members of class templates. For example, the list template requires the element type to model Copy Constructible. Furthermore, the sort() member function imposes the further requirement that T model the Less Than Comparable concept:

```
template<CopyConstructible T>
class list {
public:
  where LessThanComparable<T> void sort();
};
```

Note that the conventional abbreviation for single-type concepts

```
template<CopyConstructible T>
```

is equivalent to

```
template<typename T> where CopyConstructible<T>
```

A concept constraint applies the requirements of a concept to a template parameter (or a set of template parameters). In doing so, it serves two roles:

1. It acts as a predicate: when a template identifier is used, all of the requirements in the template's where clause must be satisfied. For example, "does int* meet the requirements of a Forward Iterator?"
2. It provides a scope for the resolution of names used in a template. For example, "what is the value_type of int* when it is acting as a Forward Iterator?"

The ConceptGCC error message shown in Section 1 is the result of a type error when concepts are used in the first role. At the call to sort(), the compiler determines whether the actual type (a list iterator) meets the requirements of the concept stated in sort()'s where clause (MutableRandomAccessIterator<Iter>); since this is not the case, the error message shown in Figure 1 is emitted.

When the requirements of a where clause are used in the second role, they introduce assumptions into the context of compilation. For example, in the previous swap() example, the first line in the body of swap() uses the copy constructor for type T, the existence of which is guaranteed by CopyConstructible<T> in the where clause. Likewise, the two assignments in the body of swap() typecheck because of Assignable<T>.

Type checking fails when neither the requirements of a template nor the enclosing scope provide suitable declarations for operations used in the template. For example, consider the following (erroneous) implementation of an STL-like find() algorithm:

```
template<InputIterator Iter, typename Val>
where EqualityComparable<Iter::value_type, Val>
Iter find(Iter first, Iter last, Val v) {
  while (first < last && !(*first == v))
    ++first;
  return first;
}
```

The error in this example occurs in the comparison first < last: the Input Iterator concept (see Figure 6) provides comparison via == and !=, but not ordering via <. With unconstrained C++ templates, this error would go unnoticed until find() is instantiated with iterators that do not support <. Using concepts, the error is detected at template definition time and ConceptGCC produces the error message shown in Figure 3.

Note the Iter::value_type in the definition of find. That is an abbreviation of InputIterator<Iter>::value_type, meaning "find the name value_type in the concept map InputIterator<Iter>." Concept maps are explained in 3.4.

3.1.1 Same-type Constraints

All the concepts mentioned so far are "user-defined"; that is, they are defined using the facilities described in Section 3.2. The only built-in concept is SameType, which requires that two type expressions produce equivalent types. This concept is built-in because it plays a special role in the type checking of constrained templates. When the constraint SameType<S, T> appears in a where clause (where S and T are arbitrary type expressions), the type checker assumes that S and T denote the same type. For example, the STL iter_swap() template requires that the value types of the two iterators be equivalent.

```
template<MutableForwardIterator Iter1,
         MutableForwardIterator Iter2>
where SameType<Iter1::value_type, Iter2::value_type>
void iter_swap(Iter1 a, Iter2 b) {
  swap(*a, *b);
}
```

```
find.cpp: In function 'Iter find(Iter, Iter, Val)':
find.cpp:7: error: no match for 'operator<' in 'first < last'
.../concepts.h:169: note: candidates are:
 bool std::SignedIntegral<Iter::difference_type>::operator<(const Iter::difference_type&, const Iter::difference_type&)
```

Figure 3. ConceptGCC error message produced when attempting to compile the erroneous definition of find().

When the type checker considers the call swap(*a,*b), it knows that the type of *a and the type of *b are the same type, which allows template argument deduction for swap() to succeed.

The iter_swap function also illustrates how one template may use another template. In this case, swap() is implicitly instantiated on the iterator's value type and swap() requires Copy Constructible and Assignable. The Mutable Forward Iterator concept includes the requirement that its value type satisfy these concepts. The where clause of iter_swap therefore introduces these as assumptions in the body of iter_swap, allowing the instantiation of swap to type-check.

3.1.2 Negative Constraints

All of the constraints described thus far have been concept constraints which can be satisfied only when the concept arguments model the concept. For example, a constraint InputIterator<Iter> is only satisfied when the type Iter is a model of Input Iterator. Negative constraints, on the other hand, are only satisfied when the concept arguments do *not* model the concept. A negative constraint !ForwardIterator<Iter> can be satisfied by a type Iter that is only an Input Iterator, or not even an iterator at all.

Negative constraints are most useful for directing algorithm specialization when selection among specializations is ambiguous. In the following example, we provide three different implementations of sort() with different concept requirements:

```
template<Sequence S>
where LessThanComparable<S::value_type>
void sort(S& s) {
  vector<S::value_type> v(s.begin(), s.end());
  quick_sort(v);
  copy(v.begin(), v.end(), s.begin());
}

template<SortedSequence S> void sort(S&) {}

template<RandomAccessSequence S>
where LessThanComparable<S::value_type>
void sort(S& s) {
  quick_sort(s);
}
```

If this sort() routine were called with a sorted array, which models both the Sorted Sequence and Random Access Sequence concepts, compilation would fail with an ambiguity [27], because neither the second nor the third sort() is better than the other. Negative constraints can be used to break such ties, by excluding types that model certain concepts. We augment the where clause of the third sort() to reject sorted sequences, resolving the ambiguity:

```
template<RandomAccessSequence S>
where LessThanComparable<S::value_type> &&
      !SortedSequence<S>
void sort(S& s) {
  quick_sort(s);
}
```

3.1.3 Constraint Propagation

Constraint propagation is a language mechanism that gathers constraints on type parameters that are induced by other constraints on the same type parameter. Constraint propagation often allows the omission of "obvious" constraints, simplifying the expression of concept-constrained templates. In our experience, the lack of constraint propagation leads to verbose specifications of generic functions and interfaces [16, 29].

Concepts support constraint propagation by propagating the requirements from any concepts or templates used in the declaration of an entity (concept, generic function, or generic data structure) into the where clause of that entity. For example, consider the following simple wrapper function that sorts a list:

```
template<LessThanComparable T>
void sort_container(list<T>& c) {
  c.sort();
}
```

The list class template, shown in Section 3.1, requires that its type parameter T model the Copy Constructible concept. The list<T>::sort() method introduces the additional requirement that T be Less Than Comparable. Without constraint propagation, our sort_container() function would fail to compile, because it does not guarantee that T is a model of Copy Constructible, and therefore cannot use list<T>.

That T should be Copy Constructible is obvious from the declaration of sort_container(): if T were not Copy Constructible, the user could not have created an object of type std::list<T> to pass to sort_container(). Constraint propagation scans the *declaration* of constrained function templates identifying places where other templates are used, then adds the constraints of those templates to the where clause. These constraints can then be assumed to hold, when type-checking the *definition* of the constrained function template, as discussed in Section 4.4. With sort_container(), the presence of list<T> in the declaration causes the requirements of list<T> (i.e., CopyConstructible<T>) to be propagated, essentially replacing the declaration with the following:

```
template<LessThanComparable T>
where CopyConstructible<T>
void sort_container(list<T>& c);
```

Concepts provide a special form of constraint propagation for types that are passed to or returned from functions. For each type T that is passed by value, the constraint CopyConstructible<T> will be propagated into the where clause. The following min_value() function propagates CopyConstructible<T>, because T is both passed as a parameter by value and returned by value. Without constraint propagation, this generic function would fail to compile when the return statement attempts to copy the result.

```
template<LessThanComparable T>
T min_value(T x, T y) {
  return x < y? x : y;
}
```

3.2 Concept Definitions

A *concept definition* is a namespace-level entity that bundles together a set of requirements and names them. A concept consists of a parameter list, an (optional) refinement clause, an (optional) where clause, and a body (optionally) specifying further requirements specific to this concept. The body can specify three kinds of requirements on the concept parameters: signatures, associated types, and (nested) where clauses.

We can define a concept named Forward Iterator with a single type parameter, Iter, as follows:

```
concept ForwardIterator<typename Iter> { ... };
```

The parameter(s) of a concept are placed after the concept name, to emphasize that the parameters are mandatory. We could have used a template header, as is done for class and function templates. However, unlike class and function templates, a concept without any parameters does not make sense, because concepts describe requirements on their parameters. The typename keyword specifies a type parameter; concepts can also have template and non-type (e.g., integer) parameters.

3.2.1 Refinement

Typically, we define a concept in terms of previously defined concepts using a form of inheritance called *refinement*. Refinement is expressed using the same syntax as inheritance, to emphasize the "is-a" relationship between concepts. We can define a concept Bidirectional Iterator in terms of Forward Iterator:

```
concept BidirectionalIterator<typename Iter>
  : ForwardIterator<Iter> { ... };
```

A concept may refine any number of other concepts. Because concepts do not contain state, the repetition of refinements is not a problem either theoretically or practically. Concepts contain *sets* of requirements, and set union simply ignores duplicates.

3.2.2 Associated Types

Often, we design types that rely critically on related types. For example, a graph type may refer to the types of edges and vertices, which are used in most graph operations. Thus, the concept Graph used to express generic graph algorithms may require that a graph type name its edge and vertex types, so that the concept can refer to these types in its signatures:

```
concept Graph<typename G> {
  typename edge;
  typename vertex;
  ...
  edge find_edge(vertex u, vertex v, const G& g);
};
```

Associated types are are place-holders for actual types, that is, concept they are implicit parameters to the concept. However, associated types are nested inside the concept because they are entirely determined by the concept and its (explicit) parameters. For instance, if we are given a type G that is a graph, we can determine its edge and vertex types because they are part of the graph definition. The converse is not true: there may be many graph types that all share the same vertex and edge types, particularly for common indexing types (int) or opaque pointers (void*).

A concept may provide a default for an associated type. The default type need not be well-formed if the concept map provides the type definition so that the default isn't used. For example, we can define the Input Iterator concept to require four associated types that default to nested type definitions in the iterator:

```
concept InputIterator<typename Iter> {
  typename value_type = Iter::value_type;
  typename reference = Iter::reference;
  typename pointer = Iter::pointer;
  typename difference_type = Iter::difference_type;
  ...
};
```

It is possible to eliminate associated types by replacing each associated type with a new concept parameter. However, doing so makes the use of concepts much more verbose, because each reference to the concept will need to specify *all* of the type parameters, even when most of those parameters are not directly used by most generic algorithms [15,16]. For instance, in the following find() algorithm, we must mention the Reference, Pointer, and Difference type parameters even though they are unused:

```
template<typename Iter, typename Value,
         typename Reference, typename Pointer,
         typename Difference,
         EqualityComparable<Value> T>
where InputIterator<Iter, Value, Reference, Pointer, Difference>
Iter find(Iter first, Iter last, T v);
```

3.2.3 Nested Requirements

Nested requirements allow the use of other concepts to express requirements on a concept's type parameters and associated types. Consider the STL Container concept. Its associated iterator type satisfies the requirements of Input Iterator. Furthermore, the container's value_type must be the same type as the iterator's value_type.

```
concept Container<typename X> {
  typename value_type;
  InputIterator iterator;
  where SameType<value_type,
                 InputIterator<iterator>::value_type>;
};
```

Note the use of Input Iterator as the type of the associated type iterator. This is the shorthand for:

```
typename iterator; where InputIterator<iterator>;
```

3.2.4 Function Signatures

Signatures express requirements for specific functions or operators. During type checking, they serve two purposes. When checking a concept map definition, signatures specify the functions that must be implemented by a concept map. When checking a template definition, signatures specify some of the operations that may be legally used in its body.

Syntactically, an abstract signature is just a normal C++ function declaration or definition. A signature may be followed by a function body, providing a default implementation to be used if a concept map does not define the function (see Section 3.4 for more details and an example). The following definition of the Equality Comparable concept includes two signatures, the second of which has a default implementation:

```
concept EqualityComparable<typename T> {
  bool operator==(const T& x, const T& y);

  bool operator!=(const T& x, const T& y)
  {return !(x==y);}
};
```

An operator signature can be satisfied either with a free function definition (at global scope, in a namespace, or in a concept map) or a member function definition. Operators that normally are only allowed as member functions may be expressed as normal "free" functions in concept requirements. For instance, the Convertible concept describes the requirement for an implicit conversion from one type to another:

```
concept Convertible<typename T, typename U> {
  operator U(const T&);
};
```

The requirement that an operation must be implemented as a member function is expressed with an abstract signature qualified by the type. For example, in the following Container concept there is a requirement for an empty member function for type X. Constructor requirements are expressed similarly.

```
concept Container<typename X> {
  bool X::empty() const;
  X::X(int n);
};
```

A concept may require a function template via a signature template, which may itself be a constrained template. The following Sequence concept illustrates the use of a signature template to describe the STL sequence constructors that allow one to construct, for instance, a vector from a pair of list iterators.

```
concept Sequence<typename X> {
  typename value_type;

  template<InputIterator InIter>
    where Convertible<InIter::value_type, value_type>
    X::X(InIter first, InIter last);
};
```

We are investigating an alternative to abstract signatures, called use patterns [12], which express operation requirements via expressions that illustrate how concept parameters can be used, rather than declarations that state what concept parameters provide. Much of the design of concepts is independent of the syntax used to describe required operations.

3.3 Overloading and Specialization

Using concepts, the Generic Programming notion of specialization takes on two different forms. The most commonly used form of specialization is concept-based overloading, which allows the names of function templates to be overloaded on their where clauses. This form of specialization—referred to as concept-based overloading—permits the expression and selection of the most specific algorithm for a particular task. Here we revisit the discussion of the advance() function from Section 2.2 in more detail. There are three potential implementations of advance(), depending on the characteristics of the iterator: Input Iterators can be moved forward with n increments Bidirectional Iterators can be moved either forward or backward in $|n|$ steps, and Random Access Iterators can jump forward or backward any distance in constant time. These three functions can be expressed as overloaded function templates follows:

```
template<InputIterator Iter>
void advance(Iter& iter, Iter::difference_type n) {
  while (n−−) ++iter;
}

template<BidirectionalIterator Iter>
void advance(Iter& iter, Iter::difference_type n) {
  if (n > 0) { while (n−−) ++iter; }
  else { while (n++) −−iter; }
}

template<RandomAccessIterator Iter>
void advance(Iter& iter, Iter::difference_type n) {
  iter += n;
}
```

An invocation of the advance() function will select the most specific implementation of the advance() function based on the capabilities of the type provided. In the following three examples, the Input Iterator, Bidirectional Iterator, and Random Access Iterator versions of advance() will be invoked, respectively:

```
void advancement(istream_iterator<int> ii,
                 list<string>::iterator lsti,
                 vector<float>::iterator vi) {
  advance(ii, 17);
  advance(lsti, 17);
  advance(vi, 17);
}
```

Concept-based overloading is particularly interesting when an overloaded function such as advance() is called from within another generic function. The STL lower_bound() algorithm, for instance, performs a binary search for a given value within a sequence denoted by iterators. An implementation of lower_bound() (extracted from ConceptGCC) follows.

```
template<ForwardIterator Iter>
where LessThanComparable<Iter::value_type>
Iter
lower_bound(Iter first, Iter last, const Iter::value_type& value) {
  Iter::difference_type len = distance(first, last);
  Iter::difference_type half;
  Iter middle;

  while (len > 0) {
    half = len >> 1;
    middle = first;
    advance(middle, half);
    if (*middle < value) {
        first = middle;
        ++first;
        len = len − half − 1;
    } else len = half;
  }
  return first;
}
```

The implementation of lower_bound() relies on two auxiliary functions: advance() and distance(); advance() moves the iterator forward some number of steps (to find the new "middle"), while distance() determines the length of the sequence. The algorithm itself can operate on Forward Iterators, providing a logarithmic number of comparisons but a linear number of iterator increment operations due to the linear-time implementations of advance() and distance(). However, when the algorithm is provided with a Random Access Iterator, concept-based overloading selects the constant-time versions of advance() and distance() to effect a lower_bound() algorithm with a logarithmic number of comparisons and iterator movements:

```
list<int> lst;
binary_search(lst.begin, lst.end(), 17); // O(n) movements

vector<int> v;
binary_search(v.begin, v.end(), 17); // O(lg n) movements
```

The mechanism that ensures that the most efficient forms of advance() and distance() are selected is similar to the two-phase name lookup facility of C++ templates, because it performs some type-checking at template definition time but defers the final decision until instantiation time. When a template such as lower_bound() is initially parsed, overload resolution solution resolves calls by selecting a *seed* function, which is the most specific function that meets the minimal requirements of the template being parsed. The call to advance() therefore resolves to the Input Iterator variant of advance(), because lower_bound() is only guaranteed to pass a Forward Iterator to advance(). Later, when the template is instantiated with a given set of concrete template arguments, overload resolution for the call is performed a second time with an expanded set of candidate functions. The set of candidate functions includes the seed function selected in the first phase (at template definition time) and all other functions that meet the following three criteria:

- The function occurs within the same lexical scope as the seed function,
- the function's template parameters, return type, and argument types are identical to those of the seed function, and
- the requirements in the function's where clause are stricter than those of the seed function's where clause.

These criteria represent a compromise between the ideal of modular type checking for templates and the need to select the most efficient function for any given operation. By limiting the set of candidate functions to those with identical signatures, we ensure that type errors cannot occur when more specific functions are found. However, by allowing functions that are more specific than the seed to enter the candidate set—even if they are declared after the calling function—we free ourselves from ordering dependencies and allow the most efficient operations to be selected in the vast majority of important cases. Note, however, that even with the restrictions we are placing on the candidate set we have not eliminated all sources of instantiation-time errors: ambiguities in the overload resolution process can still occur, a topic which we discuss in a separate paper [27]. In practice, we have found that instantiation-time errors due to ambiguities are very rare.

By ensuring that the most efficient form of advance() is selected, algorithms can be written in a very generic style without sacrificing performance due to that genericity. The same "two-phase" strategy applies to the selection of class template partial specializations. In the following example we define a dictionary template that selects among three alternatives for lookup: a balanced binary tree, a hash table, or a hash table using sorted separate chains. These three alternatives are expressible as constrained partial specializations of the primary dictionary template:

```
template<EqualityComparable Key, Regular Value>
class dictionary;

template<EqualityComparable Key, Regular Value>
  where LessThanComparable<Key>
  class dictionary<Key, Value>
  { /* use balanced binary tree */ };

template<Hashable Key, Regular Value>
  where EqualityComparable<Key>
  class dictionary<Key, Value>
  { /* use hash table */ };

template<EqualityComparable Key, Regular Value>
  where Hashable<Key> && LessThanComparable<Key>
  class dictionary<Key, Value>
  { /* use hash table with sorted chains */ };
```

Concept-based overloading and specialization are crucial to support the Generic Programming ideal of always selecting the most efficient operation or data structure based on complete type information. These features follow naturally from the expression of constrains via where clauses, and integrate seamlessly into the existing C++ rules governing partial ordering of templates.

3.4 Concept Maps

One of the strengths of Generic Programming in C++ is that templates allows users to instantiate templates with types providing a variety of interfaces. For example, the sort algorithm accepts pointers and user-defined iterators and the vector container can hold both built-in types and user-defined types. Unconstrained templates permit a variety of interfaces by allowing a built-in range of variations in argument passing style, operations declared as member vs. free functions, etc. With concepts, we generalize this notion to an arbitrary, user-defined mapping from a type to a concept, established by a *concept map*.

A concept_map definition establishes that a type is a model of a concept and defines a mapping that states *how* the type models the concept. This mapping can include specifying associated types (e.g., the value_type of an iterator) or providing definitions for the operations required by a concept. Concept maps establish a modeling relationships, so a *concept map* is often called a *model*.

Consider the following example:

```
class student_record {
public:
  string id;
  string name;
  string address;

  bool id_equal(const student_record&);
  bool name_equal(const student_record&);
  bool address_equal(const student_record&);
};
```

While the student_record type is useful for storing information about students, from the point of view of many algorithms (e.g., find()) it lacks a way of comparing records. We could of course add such a comparison, but there are several ways of comparing records, so we prefer to define a comparison specifically for the Equality Comparable concept used by find() and its brethren:

```
concept_map EqualityComparable<student_record> {
  bool operator==(const student_record& a,
                  const student_record& b)
  { return a.id_equal(b); }
};
```

A map is defined from the student_record type to the Equality Comparable concept. This concept_map is used whenever a template requires student_record to be Equality Comparable. In particular, when an algorithm uses == on a parameter declared Equality Comparable that is instantiated with a student_record, the == defined in the concept map EqualityComparable<student_record> is used. We do not have to redesign student_record to match the algorithms requirements or separately define a new type to provide a mapping. Nor do we need to define the meaning of == for all code in our program. Concept maps allow us to provide an interface to our data types that is specific to a given concept without interfering with the interfaces of other concepts or making that interface global.

When a concept_map is declared, its definition is checked for consistency against the concept. Each of the signatures in the concept must be satisfied by a function definition in the concept map, each associated type must be satisfied by a type definition in the concept map, and, and all nested requirements must be satisfied. For any signatures or associated types not provided by the concept map, default versions will be synthesized either from the lexical scope of the concept map (for signatures) or from the defaults provided in the concept. Thus, when the default definitions for signatures and associated types are correct for a given set of types, the concept map definition may be empty. For example, in the following concept map definition the built-in integer == and != operators will be used to satisfy the requirements of Equality Comparable:

```
concept_map EqualityComparable<int> { };
```

Concept maps can be templated. For example, the following definition says that any pointer type can be used as a Mutable Random Access Iterator:

```
template<typename T>
concept_map MutableRandomAccessIterator<T*> {
  typedef T value_type;
  typedef T& reference;
  typedef T* pointer;
  typedef ptrdiff_t difference_type;
};
```

Concept map templates can express nontrivial relationships. For instance, the STL vector class is Equality Comparable whenever its type parameter T is Equality Comparable:

```
template<EqualityComparable T>
  concept_map EqualityComparable<vector<T> > { };
```

Possibly the most important use of concept map templates is to compose generic libraries. Often, two libraries will provide components that are related because they express the same fundamental idea, but syntactic differences prevent reuse of the components. Concept maps can be used to adapt the syntax of one concept to the syntax of the other. More importantly, this mapping can be performed even when the concepts represent entities from different application domains. Consider a graph $G = (V, E)$, where $E \subseteq V \times V$. Theoretically, one can view the graph G as a $|V| \times |V|$ one-zero matrix A, where $A_{i,j} = 1$ when $(i, j) \in E$ and $A_{i,j} = 0$ when $(i, j) \notin E$. Concept maps allow us to express this relationship directly, so that any data structure that is a model of Graph can also be used with a linear algebra library that expects a Matrix:

```
template<Graph G>
concept_map Matrix<G> {
  typedef int value_type;
  int rows(const G& g) { return num_vertices(g); }
  int columns(const G& g) { return num_vertices(g); }
  double operator()(const G& g, int i, int j) {
    if (edge e = find_edge(ith_vertex(i, g), ith_vertex(j, g), g))
      return 1;
    else return 0;
  }
};

my_graph g = read_graph();
vector<int> x = compute_ith_eigenvector(g, 0);
```

By expressing the mapping from graph theory into linear algebra, we are able to immediately reuse the algorithms from the domain of linear algebra to compute such properties as graph eigenvalues [7]. We only require lightweight mappings expressed via concept maps, which will be optimized away by the compiler's inliner. Concept maps therefore permit cross-domain fertilization through the composition of separately developed libraries.

3.4.1 Implicit and Explicit Concepts

Writing concept maps can be a burden to the user of constrained templates: before using a template a programmer must declare how the argument types map to the appropriate concepts. It would be convenient if instead the compiler would perform a check to see if a mapping is actually necessary. That is, the compiler could simply check if a type has the required types and operations. Unfortunately, such an implicit ("structural") check can (in our design) only take into account syntactic properties of a type, whereas significant differences can be semantic. This introduces the potential for run-time errors in what would otherwise be innocuous situations. Consider:

```
istream_iterator<int> first(cin), last;
vector<int> v(first, last);
```

This should initialize a vector from standard input, but if implicit matching is used, it instead results in an empty vector or a run-time error. The reason is that vector defines two overloads for its range constructor: a slow version that grows the vector incrementally, and a faster version that relies on the multi-pass capabilities of a Forward Iterator to determine the size of the range and resize the vector ahead of time.

```
template<CopyConstructible T>
class vector {
  template<InputIterator Iter>
    where Convertible<Iter::value_type, T>
    vector(Iter first, Iter last); // slow

  template<ForwardIterator Iter>
    where Convertible<Iter::value_type, T>
    vector(Iter first, Iter last); // fast
  ...
};
```

The istream_iterator does not provide multi-pass capabilities, however it structurally (syntactically) satisfies the requirements of Forward Iterator. Therefore, the constructor call would resolve to the more specialized (faster) version. This would read the input and reserve space in the vector, but it would not be able fill the vector in a second pass through the input range because the input will have been consumed in the first pass.

The author of a concept can choose between explicit and implicit concepts. Concepts with important semantic aspects that are used for overloading should be explicit; other concepts can be implicit. An implicit concept is identified by placing the auto keyword in front of concept. For example, the following is the Equality Comparable concept specified to allow implicit matching:

```
auto concept EqualityComparable<typename T> {
  bool operator==(const T& x, const T& y);

  bool operator!=(const T& x, const T& y)
  {return !(x==y);}
};
```

3.4.2 Refinements and Concept Maps

When a concept map is defined for a concept that has a refinement clause, concept maps for each of the refinements of that concept are implicitly defined. For example, the concept map in Section 3.4 that makes pointers a model of Mutable Random Access Iterator also makes pointers model Random Access Iterator, Bidirectional Iterator, Forward Iterator, etc., because Mutable Random Access Iterator refines all of the iterator concepts, either directly or indirectly.

The implicit generation of concept maps for refinements minimizes the number of concept maps that users must specify for a given refinement hierarchy. It suffices to provide a concept map for the most refined concept that the concept arguments model, as we have done for pointers in the iterator hierarchy.

Beyond the benefits of reducing the amount of effort users must expend writing concept maps, the implicit generation of concept maps allows refinement hierarchies to evolve without breaking existing code. As refinement hierarchies evolve, they tend to move from coarse-grained approximations to finer-grained approximations, as new models, whose behavior fits "in between" two existing concepts, are discovered. For example, the Forward Iterator introduces two new kinds of requirements on top of the Input Iterator concept it refines: the need for the reference associated type to be a true reference and the "multi-pass" property that allows repeated iteration through the sequence of values. Since the inception of the STL iterator concepts, iterator types that provide the latter property but not the former (such as a "counting iterator" that enumerates values) have become more prevalent, leading to the introduction of a Multi Pass Input Iterator concept [6] that sits between Input Iterator and Forward Iterator in the refinement hierarchy. Without the implicit generation of concept maps for refinement, the introduction of this new iterator concept would break code based on the existing hierarchy, because every Forward Iterator concept map would have to be augmented with a new concept map for Multi Pass Input Iterator. With implicit generation of concept maps for refinements, a concept hierarchy can evolve to a more fine-grained structure over time, without breaking backward compatibility.

4. ConceptGCC

To evaluate our proposed extensions to C++, we implemented concepts in the GNU C++ compiler [17] and reimplemented the STL using concepts (Section 5). Our concept-enhanced version of GCC is freely available from the ConceptGCC web site [19] . Here, we provide an overview of ConceptGCC and discuss several of the implementation techniques we employed. We refer the interested reader to [21] for additional details of concept compilation.

The aim of ConceptGCC is partially to provide us a tool with which to experiment with the use of concepts, and partially to demonstrate that concepts can be integrated into real compilers. For concepts to become accepted as part of C++0x we must make it plausible that such integration is possible and economical in all major C++ compilers, and ensure that there is a great deal of practical experience with concepts prior to standardization [24].

4.1 Compilation Model

There are many potential compilation models that could be applied to C++ templates [69]. However, existing C++ compilers implement the so-called *inclusion model* of templates, where the definition of a template must be available wherever the template is used. When a particular instance of a template is needed, the compiler *instantiates* the template by substituting the concrete types needed by the instance for the corresponding template parameters. The result is code that is specific to and optimized for a particular use of the template. Coupled with compiler optimizations such as inlining and copy propagation, template instantiation enables generic C++ libraries to produce code as efficient as hand-tuned FORTRAN [34, 55, 68].

ConceptGCC retains the template inclusion model of compilation. This decision was motivated by the need for backwards compatibility, the need to fit into existing C++ compilers, and the need to provide the performance currently delivered using unconstrained templates. The benefits of improved type-checking for templates would be weakened if they represented a trade-off in performance.

The adoption of the template inclusion model of compilation precludes separate compilation for C++ templates. In this C++ constrained generics (concepts) differ from C# and Java generics. In C# and Java the default is that all instances of a generic method share the same native code. However, C# has some flavor of the instantiation model: different code is generated for each different instantiation of a generic method whose type arguments are *value types*. To attain apparent separate compilation, this instantiation-specific code is generated at run time.

The $\mathcal{G}$ language [57–59] shares syntactic constructs with our design for C++ However, $\mathcal{G}$ differs in that it provides separate compilation and therefore implements a weaker form of concept-based overloading than we propose for C++ concepts. Concept-based overloading in $\mathcal{G}$ is resolved exclusively based on the lexical information available prior to template instantiation, whereas we postpone part of the overload resolution until after instantiation, when more information is available. This more powerful facility for dispatching prevents the separate compilation of template definitions, and also prevents completely separate type checking, as overload ambiguities may occur during instantiation. We report on these and other issues in another paper [27]. Overall, the inclusion model of templates is the best match for C++ at this time, although we plan to further explore the interaction between specialization and separate compilation.

4.2 Concepts and Concept Maps

The concept map lookup process shares so much with the normal C++ template instantiation process that compiling concepts and concept maps into class templates provides a reasonable implementation approach. Concepts are compiled to class templates and concept maps are compiled to (full) specializations of those class templates. Concept map templates are compiled to partial specializations of the class templates. ConceptGCC exploits this similarity completely, representing refinement via (virtual) inheritance, function signatures as static member functions, and associated types as nested typedefs.

Concept maps provide the same static member functions and typedefs as in their concept's class. Using this representation, ConceptGCC is able to directly reuse much of the existing C++ front end functionality, including qualified name lookup (used to find associated types and signatures in concepts and concept maps), name lookup in base classes (used to find associated types and signatures in refinements), template instantiation (for synthesizing concept maps from concept map templates and implicitly matching concepts), and partial ordering of templates (for concept map selection and concept-based overloading). Although concepts are semantically distinct from class templates, their structural similarities can greatly reduce the implementation cost of introducing concepts into an existing C++ compiler.

Figure 4 shows the compilation of a simple concept and concept map definition.

4.3 Compilation of Constrained Templates

The definition of a constrained template is compiled so that expressions that refer to function signatures from the where clause are translated into expressions that make explicit qualified calls into the class specializations for the appropriate concept map. For example, the expression:

```
s = s + *first
```

from the sum() example in Section 2.1 is translated into the following:

```
Addable<value_type>::operator+(s,
  InputIterator<Iter>::operator*(first))
```

The completely transformed version of the sum() algorithm looks like this:

```
template <typename Iter>
T sum(Iter first, Iter last, Iter::value_type s) {
  typedef InputIterator<Iter>::value_type value_type;
  for (; InputIterator<Iter>::operator!=(first, last);
         InputIterator<Iter>::operator++(first))
    Assignable<value_type>::operator=(
      s,
      Addable<value_type>::operator+(
        s,
        Convertible<reference>::operator value_type(
          InputIterator<Iter>::operator*(first))));
  return CopyConstructible<value_type>::value_type(s);
}
```

Note that this translation is reminiscent of dictionary-based schemes for separate compilation of generics: in effect, the instantiation process is acting as dictionary lookup, because resolving Addable<value_type> involves finding the best matching concept map (selecting a dictionary) and extracting its operator+ acts as lookup into the dictionary. The main difference between this translation and dictionary passing is that here the lookup is performed at compile-time and there is no run-time data-structure passed to the generic function.

4.4 Type-checking Templates

The visible benefits of concepts come from the introduction of modular type checking for templates. Not surprisingly, so do most of the implementation challenges. It is a common misconception that C++ template definitions are completely unchecked when they are initially parsed. In fact, type-checking templates is a two-stage process. In the first stage, when parsing the template definition, a C++ compiler will type-check any non-dependent expressions, i.e., those that do not depend on any template parameters. Any dependent expressions (i.e., those that somehow depend on a template parameter) will be stored in an abstract syntax tree without any type information. In the second stage, during template instantiation, concrete types are substituted for template parameters throughout the

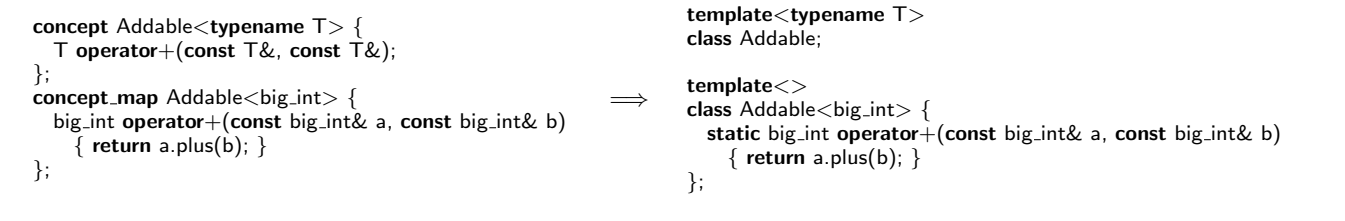


Figure 4. Compilation of concepts and concept maps.

abstract syntax tree. This process makes all dependent expressions non-dependent, thereby type-checking the entirety of the template with concrete types.

ConceptGCC implements modular type checking by making dependent expressions into non-dependent expressions in the first stage, so that the entire template definition will be type-checked when it is initially parsed. To do so, ConceptGCC generates an archetype [56] for each template parameter, using the archetype for type-checking in lieu of its corresponding template parameter. Archetypes are placeholder types that define only the operations that have been stated by concept requirements involving the template parameters. For instance, the archetype for template parameter T in the swap() function of Section 3.1 will only have two operations defined: a copy constructor, provided by CopyConstructible<T>, and an assignment operation, provided by Assignable<T>.

Due to the use of archetypes and making more expressions non-dependent, the implementation of modular type-checking in ConceptGCC required far fewer changes to the compiler or language semantics than we had anticipated. Moreover, by placing both constrained and unconstrained template compilation into the same framework of (non-)dependent expressions, we are able to compile partially-constrained templates where some parameters have been explicitly marked "unconstrained." These templates have been found useful for the introduction of concepts into existing C++ libraries, and may play a further role when template libraries need to break modular type checking in a localized way to perform some useful but type-unsafe operation.

4.4.1 Same-type Constraints

Same-type constraints are requirements (contained either in a where clause or as a nested requirement in a concept) that two types be equivalent. When type checking the definition of a template, same-type constraints in the where clause affect which types are considered equal. Consider the includes() function template:

```
template<InputIterator InIter1, InputIterator InIter2>
where SameType<InIter1::value_type, InIter2::value_type> &&
      LessThanComparable<InIter1::value_type>
bool includes(InIter1 first1, InIter1 last1,
              InIter2 first2, InIter2 last2) {
  ...
    if (*__first2 < *__first1)
  ...
}
```

In the body of includes(), the type InIter1::value_type is considered the same type as InIter2::value_type. This equivalence is important; consider $*$__first2 $<$ $*$__first1. Because the $<$ operator is defined as a type-symmetric operation on the first iterator's value_type. The compiler must use the equivalence between the two value_types to type-check this function template.

Type equality is an equivalence relation: it is reflexive, transitive, and symmetric. Thus, a same type constraint may imply many other type equalities. The following template is an example where transitivity is required for type checking: the compiler must deduce that R is the same type as T:

```
template<typename R, typename S, LessThanComparable T>
  where SameType<R,S> && SameType<S,T>
bool foo(R r, S s, T t) { return r<s && r<t; }
```

Type equality is also congruence relation. For example, if we have SameType<S,T> then SameType<vector<S>,vector<T>> holds. Conversely, SameType<vector<S>,vector<T>> implies SameType<S,T>. The compiler must also ensure that the same-type constraints appearing in a where clause do not conflict. For example, if the where clause contains (or implies) a constraint such as SameType<int,char>, the compiler should produce an error.

The problem of determining whether two types are equal given a set of same-type constraints is an instance of the congruence closure problem. The congruence closure problem already shows up in modern compilers, for example, in common sub-expression elimination. There are efficient algorithms for the congruence closure problem: the algorithm by Nelson and Oppen [48] is $O(n \log n)$ time complexity on average, where n is the number of type nodes. It has $O(n^2)$ time complexity in the worst case. This can be improved by instead using the slightly more complicated Downey-Sethi-Tarjan algorithm which is $O(n \log n)$ in the worst case [13].

The propagation of same-type constraints affects more than types. For instance, it can cause concept maps introduced by a where clause to become duplicated. This is particularly common with nested requirements. For instance, in the following example the same-type constraint that makes the two iterator's difference_types equivalent also means that the two SignedIntegral concept maps are now a single concept map. In ConceptGCC, we remove duplicates from the list of requirements once all same-type constraints have been processed.

```
concept InputIterator<typename Iter> {
  SignedIntegral difference_type;
  // ...
};

template<InputIterator Iter1, InputIterator Iter2>
where SameType<Iter1::difference_type, Iter2::difference_type>
void f(Iter1 first1, Iter1 last1, Iter2 first2);
```

Same-type constraints change the notion of type equivalence in C++. An efficient implementation of same-type constraints, based on congruence closure, is provided by the compiler for $\mathcal{G}$ [58, 59]. Within ConceptGCC, however, we were forced to implement a simple disjoint sets data structure augmented by deep type comparisons, because GCC's internal representation of types is not amenable to congruence closure algorithms.

4.4.2 Use of Class Templates and Specializations

Generic functions and data structures are often implemented using other, generic data structures. For instance, the STL mismatch() algorithm returns a pair of iterators:

```
template<InputIterator Iter1, InputIterator Iter2>
where EqualityComparable<Iter1::reference, Iter2::reference>
pair<Iter1, Iter2>
mismatch(Iter1 first1, Iter1 last1, Iter2 first2) {
  while (first1 != last1 && *first1 == *first2) {
    ++first1;
    ++first2;
  }
  return pair<Iter1, Iter2>(first1, first2);
}
```

When type-checking the body of this algorithm, the compiler must verify that the type pair<Iter1, Iter2> has a constructor accepting two parameters of type Iter1 and Iter2, respectively. To do so, the compile will need to look inside the definition of the class template pair, whose definition follows:

```
template<typename T, typename U>
class pair {
public:
  pair(const T&, const U&);
  // ...
};
```

ConceptGCC implements type-checking for uses of class templates within constrained templates via archetypes. When the definition of mismatch() is type-checked, the compiler generates archetypes A_Iter1 and A_Iter2 for the template type parameters Iter1 and Iter2, respectively. The compiler also generates a type equivalence between the template type parameters and their archetypes. At the point when the compiler requires a complete type for pair<Iter1, Iter2> (e.g., to search for a suitable constructor in the return statement), ConceptGCC instantiates the template class pair<A_Iter1, A_Iter2>. Doing so creates the constructor pair(const A_Iter1&, const A_Iter&), which is invoked by the return statement. Thus, by instantiating class templates with archetypes, ConceptGCC is able to type-check function templates such as mismatch() that use class templates.

There are some pitfalls with instantiating class templates inside a generic function. The definitions of those class templates are required for type-checking such instantiations, but this introduces the potential for instantiation-time failures that break modular type checking. In particular, it is possible that an instantiation of the algorithm will select a specialization of a class template that does not provide precisely the same members as the primary template. For pair this would be absurd. However, not all templates are as simple as pair. For example, containers sometimes provide optimized implementations for particular argument types. In particular, the C++ vector, optimizes storage of bool to a single bit per value via the vector<bool> specialization [25, §23.2.5]. This optimization, however, changes the interface in subtle ways that can lead to instantiation-time failures. Consider the following generic function:

```
template<Regular T>
void g(vector<T>& vec) {
  T& startval = vec.front();
  // ...
}
```

The definition of this function template is correct and will type-check properly with ConceptGCC. It will instantiate properly for vectors of integers, strings, lists, employees, and nearly every other data type. However, when invoked with a vector<bool> the instantiation will fail to compile, because the front() method of vector<bool> returns a proxy class type vector<bool>::reference, not a true reference. This is a design flaw of the C++98 STL caused by the inability of C++98 to express a perfect proxy for a reference.

How did the vector<bool> specialization manage to subvert modular type checking? When g() is defined and type-checked, its use of vector<T> is checked against the primary class template for vector; there front() returns a reference to T. The vector<bool> specialization may not even have been available to the compiler at the time this type-checking occurred. Later, during the instantiation of g<bool>, the compiler selects the most specialized form of vector, the vector<bool>. Since vector<bool> does not provide a compatible front() method, instantiation fails.

There are myriad workarounds that could be applied to fix this problem. We could provide a specialized g() for vector<bool>. We could design a Sequence concept to encapsulates the behavior required by g(), so that a call with a vector<bool> argument would fail because it is not a Sequence. We could eliminate the potential for these problems by placing language restrictions on specializations; Siek and Lumsdaine proved [57] with the system F^G that a language with concepts but without specialization can provide modular type checking. To obtain modular type checking, we must either restrict the use of specializations (thereby breaking the principle of always selecting the most-specific implementation) or restrict the implementations of specializations (thereby making it impossible to use template meta-programming in conjunction with constrained generics). Neither solution seems feasible for C++.

5. Evaluation

We evaluated the ability of our concept mechanisms to express the ideas of Generic Programming by enhancing the GNU implementation of the C++ Standard Template Library, libstdc++, with concepts. We could have developed a new, idealized "STL-like" library that kept all of the core ideas of the STL but adapted them to use concepts. However, we instead decided to model the existing STL (modulo errors) to produce an implementation that conforms closely to the ISO C++ standard [25]. Doing so emphasizes the evaluation of backward compatibility: existing C++ code using the STL should still compile and produce identical results with ConceptC++. Unfortunately, modeling the existing STL rather than inventing a new generic library comes with a price: some design decisions in the STL would have been very different if concepts were available at the time of its inception, but we are compelled to model the existing design rather than improve it.

The process of upgrading the Standard Template Library first involves migrating the semi-formal concept descriptions of the C++ standard (expressed via "requirements tables") into concept definitions. These are then used to constrain the templates—algorithms, data structures, and adaptors—of the STL.

5.1 Defining STL Concepts

The definition of STL concepts involves the translation from requirements tables, which are used to express the current standard, into concepts. Figure 5 compares the requirements table for the Copy Constructible concept (left-hand side, extracted from the ISO C++ standard [25]) with its ConceptC++ equivalent (right-hand side). The requirements table specifies the syntax of the concept using *valid expressions*, which illustrate the allowed uses of the types in a concept, while the "type" column provides information about the return type of the valid expression. Introducing concepts (shown on the right-hand side of Figure 6), we have preserved the semantics as closely as possible. Often, some reorganization of the requirements is necessary, but in this concept the translation from valid expressions to signatures is straightforward. The requirements tables for other simple concepts—Assignable, Default Constructible, Less Than Comparable, Equality Comparable, etc.—translate equally well into (implicit) concepts.

The most interesting set of requirements tables in the STL describe the iterator concepts. The left-hand side of Figure 6 illustrates one such requirements table, which describes Forward Iterators. In the translation to concepts (shown on the right-hand side of

Table 30, Copy Constructible requirements [25]

expression	return type
T(t)	
T(u)	
T::~T()	
&t	T*
&u	const T*

Type T is a model of Copy Constructible, t is a value of type T and u is a value of type const T.

```
auto concept CopyConstructible<typename T>
{
  T::T(const T&);
  T::~T();
  T* operator&(T&);
  const T* operator&(const T&);
};
```

Figure 5. The requirements table for Copy Constructible compared with its representation as a concept.

Figure 6), we again have preserved the semantics as closely as possible but now some reorganization of requirements was necessary:

- We have factored common requirements into refinements. Many of the valid expressions in the requirements table are already described in other tables (e.g., Default Constructible, Copy Constructible, Assignable, Equality Comparable, and Input Iterator). While refinement does exist within the requirements tables, it is implicit: when the requirements in table T_1 are a superset of the requirements in another table T_2, the concept associated with T_1 refines T_2. We have instead made refinement explicit.
- We have untangled the two distinct concepts described in the requirements table into separate Forward Iterator and Mutable Forward Iterator concepts. There is a natural refinement relationship between the two, as any mutable iterator (which can be read or written) is also a non-mutable iterator (which can only be read).
- We have collapsed the iterator_traits facility [25, §24.3.1], which provides access to the associated types of a given iterator, into the iterator concepts themselves. In doing so, we provide specific meanings for the reference and pointer associated types, which are required of STL iterators but have unspecified behavior.
- We have translated each valid expression into a corresponding signature. In some cases, such as the pre-increment expression ++r, the translation is direct and obvious. In other cases, we introduce a special associated type that describes the return type of the signature. The valid expressions in the requirements table do not specify the exact return type of the operators, and thus the exact value of this associated type is not specified. Instead, we place concept requirements on the associated type to state that it must be "convertible to" the type described in the requirements table. For instance, operator*() returns a value of type reference, which must meet the requirements of Convertible<reference, value_type>, i.e., the return type of operator*() is a value that is convertible to the value_type, as described in the requirements table.

Not all of the requirements in the table can be modeled precisely, although most differences are due to errors in the semi-formal specification. For example, the requirements table for forward iterators states that the return type of operator*() (called reference in the ConceptC++ formulation) must be exactly equal to const value_type& for non-mutable forward iterators or exactly equal to value_type& for mutable forward iterators. These two requirements are mutually exclusive, because no value type can be identical to both a reference and constant reference. However, it is assumed throughout the STL that every mutable iterator is a non-mutable iterator, i.e., mutable iterators refine their non-mutable counterparts by adding the ability to modify values. To remedy this situation, our non-mutable iterator concepts provide a weaker constraint on the reference associated type: it must be convertible to const value_type&, while the mutable iterator concepts state that the reference associated type must be exactly equal to value_type&. By weakening the non-mutable constraint, we are able to express the necessary refinement relationship between mutable and non-mutable iterators.

Translation of the iterator requirements into concepts proved to be the most challenging part of formalizing the STL. The source of most problems was the loose specification strategy in requirements tables (which permits "proxy" objects to be returned from many iterator operations), the tangling of non-mutable and mutable forms of the same concepts, and the presence of numerous small errors in the iterator specifications. Despite these challenges, the resulting concepts express the spirit of the existing STL closely enough that iterators, data structures, and algorithms written to the existing specification can be used with the ConceptC++ STL without any changes. Section 5.3 discusses the mapping from existing iterators into concept-enhanced STL iterators in more detail.

5.2 STL Algorithm Requirements

To introduce concepts into the algorithms and data structures of the STL, we must state the requirements of each STL algorithm (and data structure). This process involves introducing a where clause that covers the minimal requirements needed to implement the algorithm. We focus here on algorithms, although the same process also applies to generic data structures.

The left-hand side of Figure 7 illustrates the informal specification style used to describe the requirements on STL algorithms. Using the convention that parameters are named after the concepts they model, the function signatures in the specification state that find() and find_if() require models of the Input Iterator concept and that the latter function also requires a model of the Predicate concept. Additionally, as stated in the "requires" clause, the type parameter T of find() must be a model of Equality Comparable. The naïve translation of these requirements into concepts is shown on the right-hand side of Figure 7, but it is incorrect. Attempting to compile this constrained function template with ConceptGCC produces the error message illustrated in Figure 8. The problem, in this case, is that two T values are comparable via == (since EqualityComparable<T> is required), but the code attempts to compare the result of dereferencing first (a value_type) to T. The type value_type need not be the same as T.

There are two immediate ways to resolve the error in find(). We could either require that the value_type of the iterator be equivalent to T (using a Same Type constraint) or we could use a type-asymmetric Equality Comparable. While both options are viable, it is unclear which is dictated by the specification on the left-hand side of Figure 7. Therefore, we decided to err on the side of leniency, because it allows more uses of the find() algorithm,

Forward Iterator requirements [25]

operation	type
X u;	
X()	
X(a)	
X u(a); X u = a;	
a == b	convertible to bool
a != b	convertible to bool
r = a	X&
*a	T& if X& is mutable, otherwise const T&
a→ m	U& if X is mutable, otherwise const U&
r→ m	U&
++r	X&
r++	convertible to const X&
*r++	T& if X is mutable, otherwise const T&

Type X is a model of Forward Iterator, u, a, and b are values of type X, type T is a value type of iterator X, m (with type U) is the name of a member of type T, and r is a reference to a non-constant X object.

```
concept InputIterator<typename Iter>
  : CopyConstructible<Iter>, EqualityComparable<Iter>, Assignable<Iter> {
  typename value_type = Iter::value_type;
  typename reference = Iter::reference;
  typename pointer = Iter::pointer;
  SignedIntegral difference_type = Iter::difference_type
  where Arrowable<pointer, value_type> && Convertible<reference, value_type>;
  reference operator*(Iter);
  pointer operator→ (Iter);
  Iter& operator++(Iter&);
  typename postincrement_result;
  postincrement_result operator++(Iter&, int);
  where Dereferenceable<postincrement_result, value_type>;
};

concept ForwardIterator<typename Iter>
  : InputIterator<Iter>, DefaultConstructible<Iter> {
  where Convertible<reference, const value_type&>;
  where Convertible<pointer, const value_type*>;
};

concept MutableForwardIterator<Iter>
  : ForwardIterator<Iter>, OutputIterator<Iter> {
  where SameType<reference, value_type&> &&
        SameType<pointer, value_type*>;
};
```

Figure 6. Forward Iterator and its mutable variant, expressed using documentation conventions, and defined with concepts. For the concept description we also show the definition of Input Iterator, which encapsulates many of the Forward Iterator requirements.

```
template<class InputIterator, class T>
  InputIterator find(InputIterator first, InputIterator last,
                     const T& value);
template<class InputIterator, class Predicate>
  InputIterator find_if(InputIterator first, InputIterator last,
                        Predicate pred);
```

1. **Requires:** Type T is Equality Comparable.
2. **Returns:** The first iterator i in the range [first,last) for which the following corresponding conditions hold: *i == value, pred(*i) != false. Returns last if no such iterator is found.
3. **Complexity:** At most last−first applications of the corresponding predicate.

```
// Literal translation: will not type−check
template<InputIterator Iter, EqualityComparable T>
Iter find(Iter first, Iter last, const T& value) {
  while (first != last && !(*first == value))
    ++first;
  return first;
}

// Literal translation: complete and correct
template<InputIterator Iter, Predicate<Iter::reference> Pred>
Iter find_if(Iter first, Iter last, Pred pred) {
  while (first != last && !pred(*first))
    ++first;
  return first;
}
```

Figure 7. Specification of the STL find() and find_if() algorithms as described in the ISO C++ standard [25, §25.1.2] (left) and its literal translation into ConceptC++.

including many uses that will work with the existing STL. The definition of find() contained in ConceptC++ reads:

```
template<InputIterator Iter, typename T>
where EqualityComparable<Iter::value_type, T>
Iter find(Iter first, Iter last, const T& value) {
  while (first != last && !(*first == value))
    ++first;
  return first;
}
```

Our exercise in stating STL algorithm requirements with concepts has uncovered several bugs and ambiguities due to the informal specification style used by the C++ standard. In many cases, the intended requirements are clear in the C++ standard but the libstdc++ implementation was incorrect. Most often, these errors were due to assumptions made about the relationships between types that were not specified in the requirements. For instance, the GNU C++ library contained the following implementation of replace_copy() (with the direct translation from requirements to where clause):

```
template<InputIterator InIter,
         OutputIterator<InIter::value_type> OutIter,
         CopyConstructible T>
where Assignable<OutIter::reference, T> &&
      EqualityComparable<InIter::value_type, T>
OutIter replace_copy(InIter first, InIter last, OutIter out,
                     const T& old_val, const T& new_val) {
  for ( ; first != last; ++first, ++result)
    *result = *first == old_val ? new_val : *first;
  return result;
}
```

```
find.cpp: In function 'Iter std::find(Iter, Iter, const T&)':
find.cpp:8: error: no match for 'operator==' in '*first == value'
.../concepts.h:170: note: candidates are: bool std::InputIterator<_Iter>::operator==(const _Iter&, const _Iter&)
.../concepts.h:170: note:                 bool std::EqualityComparable<T, T>::operator==(const T&, const T&)
.../concepts.h:170: note:                 bool std::SignedIntegral<difference_type>::operator==(const difference_type&,
                                                                                                const difference_type&)
```

Figure 8. ConceptGCC error message produced when attempting to compile the naïvely translated version of find() shown on the right-hand side of Figure 7.

The where clause requires that one can assign both values from the input sequence and values of type T to the output sequence, even though the associated type value_type of the Input Iterator may be different from the type T. However, when type-checking this routine, ConceptGCC produces the following error message:

```
replace_copy.cpp:15: error: operands to ?:
have different types
```

The problem, originally noted by Siek and Lumsdaine [58], is that the requirements to replace_copy() do not say that T and the value type of the input iterator are equivalent or coercible. Since the vast majority of uses of replace_copy() have T and the iterator's value type equivalent, it is easy to see how this bug could have gone undetected. The fix is easy: replace the conditional assignment via ? : with an if-else statement.

We found several other errors within libstdc++ where the implementation assumed type equivalence that is not guaranteed by the C++ standard. One particularly vexing example occurred in the sort_heap() function, which had the following definition once augmented with a where clause:

```
template<MutableRandomAccessIterator RAIter>
where { LessThanComparable<Iter::value_type> }
void pop_heap(RAIter first, RAIter last);

template<MutableRandomAccessIterator Iter>
where LessThanComparable<Iter::value_type>
void sort_heap(Iter first, Iter last) {
  while (last − first > 1)
    pop_heap(first, last−−);
}
```

This algorithm is subtly incorrect. Attempting to compile with ConceptGCC produces the following error message:

```
sort_heap.cpp:8: error: no matching function for call to
                 'pop_heap(Iter&, postdecrement_result&)'
```

The problem in this case is that the postdecrement operator, used in last−−, is specified to return a value of associated type postdecrement_result, which is convertible to—but distinct from—the type Iter of last. In the call to pop_heap(), this difference causes a failure in template argument deduction: RAIter is bound to Iter by the first parameter to pop_heap(), but RAIter is bound to postdecrement_result by the second parameter. Again, fixing the problem once it has been detected is trivial: one can insert a cast from last−− to Iter.

The last major source of errors we found when defining the requirements of concepts in the STL is due to the infamous vector<bool> iterator (Section 4.4.2). The problems with the vector<bool> iterator arise because the C++ standard incorrectly states that the vector<bool> iterator is a Random Access Iterator [25, §23.2.5]. This error cannot occur when using concepts: an attempt to write the concept map definition will result in a compile-time failure, because the reference type of a vector<bool> iterator does not meet the requirements of the Forward Iterator concept.

5.3 Backward Compatibility

Backward compatibility of the ConceptC++ STL with programs written for the existing STL is of paramount importance for two reasons. First, by providing all of the benefits of ConceptC++ (better error messages, improved library implementations, syntax remapping) without requiring users to port their code, backward compatibility improves the chances for adoption of concepts into real-world use and C++0x. Second, if we are able to apply concepts to existing STL without significant changes, then it is likely that they can be similarly applied to many other generic libraries.

We can report that the ConceptC++ STL provides excellent backward compatibility. The entirety of the libstdc++ test suite compiles and produces identical results with the ConceptC++ STL and existing STL, and required only three changes:

1. The ConceptC++ STL defined a name in use by one of the tests (Integral), requiring the test to use a different identifier.
2. The ConceptC++ STL does not pretend that vector<bool> iterators model Random Access Iterator.[1] One test case that relied on this idiosyncrasy had to be modified.
3. One type that was used as an output iterator in the example required the addition of a single concept map to be used with the ConceptC++ STL algorithms.

We also ran the test suites for several libraries in the C++ Boost library collection [6], including the graph and iterator adaptor libraries that make heavy use of STL constructs, using ConceptC++. Again, there was a case where a type used as an output iterator required a concept map to be used with the ConceptC++ STL.

Assuming that we can express the requirements of an existing generic library using concepts, the major impediment to backward compatibility is the possibility that users will need to add many concept maps. We used a combination of implicit concepts (for trivial concepts such as Copy Constructible or Less Than Comparable) and concept map templates to minimize (to nearly zero) the number of concept maps that users will be required to write.

In the ConceptC++ STL, the only widely-used concepts that require explicit concept maps are the iterator concepts. For nearly all of the iterator concepts (Output Iterator is the only exception), however, we can write concept map templates that adapt iterators written for the existing STL into the new iterator concepts. This process is completely transparent to the user, otherwise porting the libstdc++ and C++ Boost [6] test suites to the ConceptC++ STL would have required many concept maps for various user-defined iterator types.

The key to adapting old-style iterators to take advantage of concepts is in the observation that existing STL iterators already provide explicit declarations of the concepts they model in the form of traits [47]: we need only query these traits. Figure 9 illustrates how old-style forward iterators can be seamlessly mapped into the iterators of the ConceptC++ STL. The Iterator Traits concept is a structural concept that extracts all of the types from the existing

[1] In fact, the ConceptC++ *can not* pretend that vector<bool> iterators are Random Access Iterators: the compiler will reject the definition of the concept map.

```
auto concept IteratorTraits<typename Iter> {
  typename iterator_category = Iter::iterator_category;
  typename value_type = Iter::value_type;
  typename difference_type = Iter::difference_type;
  typename pointer = Iter::pointer;
  typename reference = Iter::reference;
};
template<typename !Iter>
where { IteratorTraits<Iter>,
        Convertible<iterator_category, input_iterator_tag>,
        Convertible<iterator_category, forward_iterator_tag>}
concept_map ForwardIterator<Iter> {
  typedef Iter::value_type value_type;
  typedef Iter::difference_type difference_type;
  typedef Iter::pointer pointer;
  typedef Iter::reference reference;
};
```

Figure 9. Concept map templates that seamlessly map old-style STL iterators into ConceptC++ STL iterators, to enable backward compatibility.

STL iterator_traits trait class. Most important of all of these is the iterator_category type, which states which concept—Input Iterator, Forward Iterator, etc.—the type models.

Implicit generation of the concept map IteratorTraits<Iter>, implies that Iter is an iterator of some form. Therefore, we can declare concept map templates that take any type Iter with an implicit-generated concept map IteratorTraits<Iter> and query its iterator_category to determine which concepts it models. In Figure 9, we generate a concept map of Forward Iterator so long as there is a concept map of Iterator Traits and its iterator_category is convertible to forward_iterator_tag. The mapping is valid because the where clause of the Forward Iterator concept map template matches precisely the requirements needed to identify a type as a Forward Iterator in the existing STL. Thus, with the exception of Output Iterator, types that meet the iterator requirements of the existing STL will automatically meet the requirements of the ConceptC++ STL with no porting required. We believe that upcoming extensions to C++ [26] will make it possible to provide the same automatic mapping for Output Iterator concept maps as well, providing nearly perfect backward compatibility for existing iterators.

Complete backward compatibility also requires that pre-concept generic algorithms written using the old iterator interfaces (based on iterator_traits) will continue to function, even with new iterators that expose only the concepts-based interface. For instance, the following algorithm counts all of the elements in the sequence that meet some specific criteria:

```
template<typename InputIterator, typename Predicate>
typename std::iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred) {
  typename std::iterator_traits<InputIterator>::difference_type
    n = 0;
  for (; first != last; ++first)
    if (pred(*first)) ++n;
  return n;
}
```

One could require that all new iterators expose both a concepts-based interface (e.g., provide concept maps) and a pre-concept interface (e.g., specialize iterator_traits). However, this places the burden of backward compatibility on the authors of iterators. Aside from the problems of maintaining two similar (but incompatible) interfaces, this approach would never allow the concepts-based interface to completely replace the existing interface.

```
template<InputIterator Iter>
struct iterator_traits<Iter> {
  typedef input_iterator_tag iterator_category;
  typedef Iter::value_type value_type;
  typedef Iter::reference reference;
  typedef Iter::pointer pointer;
  typedef Iter::difference_type difference_type;
};
```

Figure 10. Class template partial specializations that seamlessly map new-style ConceptC++ iterators into the old-style C++ iterator interface.

Concepts allow us to define a set of class template partial specializations for iterator_traits that provide new, concept-based iterators with the old-style interface. Figure 10 illustrates one of the partial specializations, which provides a suitable iterator_traits definition for any model of the Input Iterator concept. Like the mapping from existing iterators into the concept system, this mapping is seamless and invisible: existing generic algorithms will be able to access the associated types of new-style iterators through iterator_traits.

With a series of concept maps (illustrated in Figure 9) and class template partial specializations (illustrated in Figure 10), we are able to ensure that existing, pre-concept iterators seamlessly interoperate with concept-constrained algorithms and that pre-concept algorithms interoperate with new, concept-based iterators. Moreover, the use of these techniques can simplify the development of "dual-mode" generic libraries, which provide the same functionality with and without concepts. The ConceptGCC implementation of the Standard Library is one such library, allowing users to "turn off" concepts at the library level to provide a pre-concept, C++03-compliant library. The ability to develop "dual-mode" generic libraries is crucial: even if vendors could coordinate the release of compilers and libraries supporting concepts (which they can't), users are still likely to rely on a mix of compilers and libraries, some of which support concepts and others that do not. The burden of maintaining the same library for C++ with and without concepts would be a significant barrier to adoption.

5.4 Summary

We evaluated the design of concepts by translating the ideas and implementation of the Standard Template Library into ConceptC++. Defining the requirements in the STL using concepts uncovered errors and ambiguities in both the specification and the implementation of the STL. In the end, we were able to produce a concept-enhanced STL that is demonstrably better than the original, both from the user's point of view (better error messages, clearer abstractions) and from the implementer's point of view (improved type checking, much less reliance on arcane template techniques). Moreover, the concept-enhanced STL provided near-perfect backward compatibility with programs written using the existing STL.

6. Related Work

Concepts extend C++ with constraints on type parameters, as well as with modular type checking and overloading based on these constraints. Various mechanisms for constraining polymorphic types and for overloading are part of many widely-used languages. Our design has been influenced by prior work in this area. Here, we present concepts with respect to these mechanisms and languages.

Several object-oriented languages, including Eiffel [39], Java [18], and C# [40], support constrained generics with subtyping constraints. In such languages, interfaces or abstract classes describe a

set of requirements, as method signatures, that types declared to be subtypes of such classes must satisfy. The constraints on generic methods and classes are then mutually recursive systems of subtyping requirements on type parameters, essentially a generalization of F-bounded polymorphism [8]. There are significant differences between the above kind of constrained polymorphism and concepts, including the treatment of associated types, use of subtyping as the basis of constraints, use of constraints in overloading, and the compilation model of generic definitions. We discuss compilation model in Section 4.1, and address the other differences below.

Associated types, as well as requirements on them, are an essential part of concepts. Representing associated types in Java or C# is less direct: interfaces do not provide type members that could serve for this purpose. It is possible, however, to express mappings between types (which is what associated types essentially are), as well as constraints on associated types, without type members. This is accomplished with a distinct type parameter, and a set of constraints, for each associated type. This approach, however, does not encapsulate associated types and their constraints into the interfaces; we report how both associated types and constraints on them must be repeated in all sites where generic interfaces are used as type parameter constraints [15]. Concepts directly support associated type requirements in concepts. Note that we have developed language features to allow associated types in generic interfaces [29] in C# or Java-like languages. Moreover, type members in objects and classes have been extensively studied since the early work on virtual types [38]. We review this work in [29].

An interesting recent language regarding generic programming is Scala [50,51]. In particular, Scala's type system provides abstract type members, which could serve as associated types of concepts. Moreover, the latest version of Scala includes a feature referred to as *implicit parameters* [49, §7], which can be used to pass models (in the generic programming sense) to generic functions either explicitly or implicitly. We plan to do a more comprehensive analysis of Scala with respect to generic programming in the future.

A type (a set of types, values, operations, etc.) can model a concept that is defined after the definition of the type. Such *retroactive modeling* is important when composing separately developed libraries as discussed in [15]. In constrained generics based on subtyping, the ability of retroactive modeling is tied to the ability of retroactive subtyping, which is typically not provided in mainstream object-oriented languages where a subtype relation is nominal and established via subclassing. Of well-known object-oriented languages, Cecil [37] supports retroactive subtyping, and the feature has been suggested for Java as well [4]. Also, mechanisms for specifying structural subtyping relations have been proposed for C++ [5] and for Java [33].

Concepts do not build on C++'s subtype relation. In that respect, concepts are similar to constraint mechanisms such as signatures in ML [41], and type classes in Haskell [70]. Comparing concepts with ML signatures, we note that ML signatures encompass equivalents of associated types, in particular we can observe the correspondence of same-type constraints to ML's sharing constraints (see manifest types [35] and translucent sums [36]). A major difference between concepts and ML signatures is the granularity of parametrization. In ML, signatures are used to constrain *functors*, i.e., parametrized modules, not parametrized functions. Generic functions very seldom share the same parametrization and requirements, so each function would have to be wrapped in a module of its own. Furthermore, functors must be explicitly instantiated with type arguments, making their use quite heavy-weight, whereas C++ provide implicit instantiation, with the type arguments deduced from the types of the actual arguments.

In ML, checking that the requirements of a signature are satisfied is based purely on structural conformance. Our concept extensions support both structural and nominal modeling relations (corresponding to implicit and explicit concepts, respectively). While implicit concepts are not (strictly speaking) necessary, this feature has been found to reduce the number of "trivial" concept maps significantly. However, explicit concepts are crucial for concepts that differ only semantically, as illustrated in Section 3.4.1. In particular, often a concept refinement (e.g. Forward Iterator refining Input Iterator) only adds more semantic requirements, having no syntactic difference between the concepts. Thus the structural properties of types do not suffice to uniquely determine which concepts the types model. Further details of the use of ML for generic programming can be found in [15].

Similar to concepts, Haskell type classes define a set of required functions that instances (models) of the type class must provide. Subtyping does not enter into the picture. Furthermore, type classes constrain individual functions, rather than modules. In their standard form, type classes do not support associated types, but recent research [9, 10] addresses this issue. The most notable difference is Haskell's support for type inference, in particular inferring the constraints of a generic function from its body. To ensure that the constraints of a generic function can be uniquely determined, a significant restriction is placed on type classes: each function name can occur only once in all type classes visible in the program. This requires foresight, and is problematic in importing type class definitions from separately defined libraries.

Concepts participate in overload resolution. This is contrary to Java and C# where constraints are ignored when selecting the best matching overload: two overloads of a function that only differ in their constraints are considered ambiguous. Overloading in ConceptC++ differs from ML's or Haskell's behavior too, since polymorphic functions in these languages cannot be overloaded. In Haskell, all overloading occurs by providing different instance declarations for a type class, and thus only functions defined within some type class are overloadable. In ConceptC++ one overloaded function is considered to be a specialization of another if its concept constraints imply the constraints of the other one, as described in Section 3.3. This is to provide direct support for algorithm specialization, which is an essential part of the generic programming approach to software library development. We analyze concept-based overloading and specialization in detail in [27].

7. Conclusion

We propose new language features for C++, based on *concept*s, that provide (nearly) modular type checking for templates and directly support Generic Programming. The addition of concepts to C++ libraries brings an immediate benefit to library users by drastically shortening and simplifying template error messages. In the longer term, concepts make it easier to design and implement template libraries, replacing a grab-bag of template tricks with a single, coherent language feature.

ConceptGCC [19], built on the GNU C++ compiler, implements concepts and associated features as described here. Using ConceptGCC, we have reimplemented the C++ Standard Template Library using concepts. This process of formalizing the requirements of STL using concepts uncovered several deficiencies in the semi-formal specification of C++ [25] and detected several new bugs in the GNU implementation. The resulting STL using concepts provides the same functionality as the existing STL, with nearly-perfect backward compatibility, but greatly improves the user experience. ConceptGCC is available online at `http://www.generic-programming.org/software/ConceptGCC`.

8. Future Work

The current design and implementation of concepts draw from a larger on-going effort led by the authors [12,22,23,53,64]. The aim of this work is the inclusion of concepts—in a form very similar to what is presented here—into C++0x, the next ISO C++ standard.

To make that happen, we will complete ConceptGCC, implementing the remaining few features of ConceptC++. Furthermore, we must document our design to the extent that it can be (re)implemented in commercial C++ compilers and used by practicing programmers. The final word on the design of concepts in C++0x will be determined by the committee and the ISO national representatives. Their criteria include completeness, simplicity, and stability of the design as well as performance and smooth integration with the rest of the language and the standard library.

To further evaluate expressiveness and usability, we will port additional generic C++ libraries to use concepts and tune their performance and that of ConceptGCC. Concepts must not introduce any run-time overhead compared to unconstrained templates. We will evaluate the trade-offs between use patterns and signatures. In general, we will continue to try to simplify our concept mechanisms to make it easier for programmers to use concepts effectively.

To increase the scope of analysis and optimization, we will explore the notion of attaching semantic properties to concepts, to permit semantic descriptions of abstractions and aid compilers in concept-based optimization [52].

Acknowledgments

The effort to introduce language support for Generic Programming into C++ in the form of concepts has been shaped by many, including David Abrahams, Matthew Austern, Ronald Garcia, Mat Marcus, David Musser, Sean Parent, Sibylle Schupp, Alexander Stepanov, and Jeremiah Willcock. This work was supported by NSF grants EIA-0131354, CCF-0541014, CCF-0541335, and by a grant from the Lilly Endowment.

References

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

[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A standard template adaptive parallel C++ library. In *Int. Wkshp on Adv. Compiler Technology for High Perf. and Embedded Processors*, page 10, July 2001.

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

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

[5] Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice and Experience*, 25(8):863–889, August 1995.

[6] Boost. *Boost C++ Libraries*. `http://www.boost.org/`.

[7] Alex Breuer, Peter Gottschling, Douglas Gregor, and Andrew Lumsdaine. Effecting parallel graph eigensolvers through library composition. In *Performance Optimization for High-Level Languages and Libraries (POHLL)*, April 2006.

[8] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280. ACM Press, 1989.

[9] Manuel M. T. Chakravarty, Gabrielle Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the International Conference on Functional Programming*, pages 241–253. ACM Press, September 2005.

[10] Manuel M. T. Chakravarty, Gabrielle Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2005.

[11] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 2000.

[12] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308. ACM Press, 2006.

[13] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM (JACM)*, 27(4):758–771, 1980.

[14] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, a computational geometry algorithms library. *Software – Practice and Experience*, 30(11):1167–1202, 2000. Special Issue on Discrete Algorithm Engineering.

[15] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 115–134. ACM Press, 2003.

[16] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 2005. Accepted.

[17] GNU compiler collection. `http://www.gnu.org/software/gcc/`, 2005.

[18] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

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

[20] Douglas Gregor and Andrew Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the 2005 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*, pages 423–437, October 2005.

[21] Douglas Gregor and Jeremy Siek. Implementing concepts. Technical Report N1848=05-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2005.

[22] Douglas Gregor, Jeremy Siek, Jeremiah Willcock, Jaakko Järvi, Ronald Garcia, and Andrew Lumsdaine. Concepts for C++0x (revision 1). Technical Report N1849=05-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2005.

[23] Douglas Gregor and Bjarne Stroustrup. Concepts. Technical Report N2042=06-0112, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.

[24] Michi Henning. The rise and fall of CORBA. *ACM Queue*, 4(5):28–34, June 2006.

[25] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. Geneva, Switzerland, September 1998.

[26] J. Järvi, B. Stroustrup, and G. Dos Reis. Decltype and auto (revision 4). Technical Report N1705=04-0145, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, September 2004. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1705.pdf`.

[27] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy Siek. Algorithm specialization in generic programming: Challenges of constrained generics in C++. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming*

language design and implementation, pages 272–282. ACM Press, 2006.

[28] Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, and Andrew Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, 21(6):25–32, June 2003.

[29] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 1–19. ACM Press, 2005.

[30] Mehdi Jazayeri, Rüdiger Loos, David Musser, and Alexander Stepanov. Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, Schloss Dagstuhl, Germany, April 1998.

[31] D. Kapur and D. Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report RPI–92–20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180, July 1992.

[32] Steve Karmesin, James Crotinger, Julian Cummings, Scott Haney, William Humphrey, John Reynders, Stephen Smith, and Timothy J. Williams. Array design and expression evaluation in POOMA II. In Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn, editors, *ISCOPE*. Advanced Computing Laboratory, LANL, 1998.

[33] K. Läufer, G. Baumgartner, and V. F. Russo. Safe structural conformance for Java. *The Computer Journal*, 43(6):469–481, 2000.

[34] Lie-Quan Lee, Jeremy Siek, and Andrew Lumsdaine. Generic graph algorithms for sparse matrix ordering. In *ISCOPE'99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

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

[36] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Pittsburgh, PA, May 1997.

[37] Vassily Litvinov. Contraint-based polymorphism in Cecil: towards a practical and static type system. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 388–411. ACM Press, 1998.

[38] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406. ACM Press, 1989.

[39] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall, first edition, 1992.

[40] Microsoft Corporation. Generics in C#, September 2002. Part of the Gyro distribution of generics for .NET available at `http://research.microsoft.com/projects/clrgen/`.

[41] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[42] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 2nd edition, 2001.

[43] David R. Musser and Alexander A. Stepanov. A library of generic algorithms in Ada. In *Using Ada (1987 International Ada Conference)*, pages 216–225. ACM SIGAda, December 1987.

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

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

[46] Nathan Myers. A new and useful technique: "traits". *C++ Report*, 7(5):32–35, June 1995.

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

[48] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[49] Martin Odersky. The Scala language specification: Version 2.0, draft march 17, 2006. `http://scala.epfl.ch/docu/files/ScalaReference.pdf`, 2006.

[50] Martin Odersky and al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[51] Martin Odersky and Matthias Zenger. Scalable component abstractions. *SIGPLAN Not.*, 40(10):41–57, 2005.

[52] Sibylle Schupp, Douglas Gregor, David R. Musser, and Shin-Ming Liu. User-extensible simplification: Type-based optimizer generators. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 86–101, London, UK, 2001. Springer-Verlag.

[53] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concepts for C++0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, January 2005.

[54] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[55] Jeremy Siek and Andrew Lumsdaine. The Matrix Template Library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, 1(6):70–78, Nov/Dec 1999.

[56] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.

[57] Jeremy Siek and Andrew Lumsdaine. Essential language support for generic programming. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, pages 73–84. ACM Press, June 2005.

[58] Jeremy Siek and Andrew Lumsdaine. Language requirements for large-scale generic libraries. In *GPCE '05: Proceedings of the fourth international conference on Generative Programming and Component Engineering*, September 2005. To appear.

[59] Jeremy G. Siek. *A Language for Generic Programming*. PhD thesis, Indiana University, August 2005.

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

[61] Bjarne Stroustrup. *Design and Evolution of C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[62] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[63] Bjarne Stroustrup and Gabriel 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++, October 2003. `http://www.open-std.org/jtc1/sc22/wg21`.

[64] Bjarne Stroustrup and Gabriel Dos Reis. A concept design (rev. 1). Technical Report N1782=05-0042, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2005.

[65] Mathias Troyer and Prakash Dayal. The Iterative Eigensolver Template Library. `http://www.comp-phys.org:16080/software/ietl/`.

[66] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, May 1995.

[67] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[68] Todd L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, volume 1505 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[69] Todd L. Veldhuizen. Five compilation models for C++ templates. In *First Workshop on C++ Template Programming*, October 10 2000.

[70] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.