Algorithm Specialization in Generic Programming

Challenges of Constrained Generics in C++

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

Douglas Gregor
Jeremiah Willcock
Andrew Lumsdaine
Indiana University
{dgregor, jewillco, lums}@osl.iu.edu

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

Abstract

Generic programming has recently emerged as a paradigm for developing highly reusable software libraries, most notably in C++. We have designed and implemented a constrained generics extension for C++ to support modular type checking of generic algorithms and to address other issues associated with unconstrained generics. To be as broadly applicable as possible, generic algorithms are defined with minimal requirements on their inputs. At the same time, to achieve a high degree of efficiency, generic algorithms may have multiple implementations that exploit features of specific classes of inputs. This process of algorithm specialization relies on non-local type information and conflicts directly with the local nature of modular type checking. In this paper, we review the design and implementation of our extensions for generic programming in C++, describe the issues of algorithm specialization and modular type checking in detail, and discuss the important design tradeoffs in trying to accomplish both. We present the particular design that we chose for our implementation, with the goal of hitting the sweet spot in this interesting design space.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; D.3.2 [*Programming Languages*]: Language Classifications—C++

General Terms Languages, Theory

Keywords Generic programming, parametric polymorphism, constrained generics, concepts, specialization

1. Introduction

Templates are the cornerstone of modern C++ libraries. The Standard Template Library (STL) [42], Boost Graph Library [37], the μBLAS Library [49], and the Computational Geometry Algorithms Library (CGAL) [12], are all examples of widely-used, heavily-templated libraries. Ironically, the same template system that makes these libraries so flexible is also a significant source of problems. In the unconstrained genericity model of C++, generic functions cannot be compiled, or even type checked, in isolation of their uses. Constrained generics for C++, in the form of type checking for C++ templates, can remedy many of the problems faced by users and designers of generic libraries. Early discussions on constrained generics for C++ can be found in [43]. A more recent and serious effort was initiated by Stroustrup and Dos Reis [45], laying out design goals for C++ constrained generics. This effort draws motivation from *generic programming*, a paradigm for designing and implementing efficient and highly reusable algorithm libraries. Pioneered by Musser and Stepanov [32], and introduced to C++ with the STL [41], this approach is the foundation of the design principles of numerous C++ libraries.

Based on the goals in [45], an extensive survey of language support for generic programming [13], and experience in developing several generic libraries, we have specified a constrained generics feature for C++ [16] and implemented the specification as an extension to the GNU C++ compiler [15]. (In this paper, we refer to C++ with our proposed extensions as ConceptC++, and the extended compiler as ConceptGCC.) The constrained genericity mechanism we propose is designed to directly support generic programming. Section 2 explains generic programming in more detail. We include a general overview of ConceptC++ in Section 2.1, but the focus of the paper is on features enabling algorithm specialization. We refer to a component as *specialized* when its interface describes the minimal requirements on its type parameters, but internally the component takes advantage of additional capabilities not required from those type parameters. For example, C++ standard library implementations specialize the `copy()` algorithm for iterator types with random access, for pointer types, for "segmented" iterator types, etc. Such specialization is straightforward in today's C++, which lacks enforced interface specifications for template parameters altogether, and is frequently used to simultaneously guarantee good performance and generality.

Unfortunately, accommodating specialization and modular type checking together for constrained generics presents a conflict. With modular type checking, a component is type checked in isolation of other components (only relying on the declarations but not the definitions of components it refers to), guaranteeing that no use of that component conforming to its interface will produce type errors. Without specialization, our model of constrained generics for C++ reduces to System F^G, which has provably modular type checking [40]. Adding unrestricted specialization, however, creates variations in the body of a generic component, thereby compromising the guarantees that modular type checking provides. For instance, the addition of new specializations and overloads into a generic component can produce ambiguities in the selection process. This paper reports on the fundamental tension between specialization and modular type checking and explores the design space of con-

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

strained generics for C++ in search of a "sweet spot" between unrestricted specialization and modular type checking.
Our contributions are:

- We illustrate how and to what extent specializations compromise modular type checking for constrained generics in C++.
- We relate these problems to similar problems studied in the context of multimethods [8, 29] and Haskell type classes with overlapping instance declarations [35] [47, §7.4.4.1].
- We consider and evaluate solutions proposed for multimethods in the context of constrained generics for C++.
- We justify our decisions about specialization and modular type checking for constrained generics in C++.

2. Constrained generics for C++

Generic programming is a paradigm for designing and implementing reusable software libraries. A key principle of generic programming is that abstraction does not compromise efficiency. The generic programming discipline is essentially a process of gradually making an algorithm more general by lifting unnecessary requirements, such as dependencies on particular data-structures. Starting with an efficient and practical algorithm, the author of a generic library abstracts over details that are not essential for the operation of the algorithm. This is repeated as long as it is still possible to instantiate the generic algorithm and arrive at a concrete algorithm that has the same performance and semantics as the original one [32]. Lifting an algorithm that adds the elements of an integer array, into an algorithm that can *fold* the elements of any sequence using an arbitrary binary operator (this is the accumulate() function of the C++ standard library), serves as an example. Specialization allows the lifting process to be taken further than would otherwise be possible: the binary_search() function in the C++ standard library, for example, can operate on sorted sequences with or without random access. The function takes a logarithmic number of steps in traversing the sequence with random access and a linear number of steps without.

Integral to the generic programming paradigm is the systematic organization of constraints on type parameters into *concepts*, which describe the commonly recurring abstractions in a particular domain. The archetypical example of the results of this activity is the STL and its documentation [1], which describes several abstractions, chiefly different kinds of iterators and containers, as concepts. The requirements on the input types of the wide selection of generic algorithms in STL are then concisely described in terms of these concepts. In current C++, however, such constraints are mere documentation and are not enforced, or even accessed, by the C++ compiler. Library techniques exist to simulate the checking of a limited class of constraints [25, 39], but their use cannot be enforced. These library techniques have only gained modest use and are not an adequate substitute for constrained generics.

A brief description of the generic programming terminology established by Stepanov and Austern [1] is as follows: A *concept* is a collection of requirements on a type, or types. We say that a type (or a tuple of types) *models* a concept whenever it satisfies all the requirements of that concept. A concept is said to *refine* another concept if its set of requirements includes all requirements of the other concept. The kinds of requirements in a concept are *valid expressions* (or function signatures), *associated types*, *semantic constraints*, and *complexity guarantees*. Function signatures specify the operations that must be defined for the modeling type(s). The *associated types* of a concept specify mappings from the modeling type(s) to other collaborating types (such as the mapping from a container to the type of its elements). Semantic constraints and complexity guarantees describe constraints on the required operations, but as non-syntactic requirements they fall outside of the capabilities of ConceptC++.

2.1 ConceptC++

Following the terminology and practice of generic programming, ConceptC++ introduces three new language features: concepts, which contain the requirements on a set of type parameters; models, which state conformance of a set of types to the requirements of a concept; and where clauses, which describe the constraints placed on template parameters.

Our extensions to C++ allow the definition of concepts, refinement hierarchies, modeling relationships between types and concepts, and constraints on template parameters using concepts. Our concept system draws ideas from several languages and their particular features. Parallels exist to Haskell type classes and instance declarations; signatures, structures, and sharing constraints in ML; interfaces and subtype constraints in object-oriented generics, etc. We point out these parallels below while discussing the features of ConceptC++. A thorough account of the relationship of generic programming to features of several mainstream programming languages can be found in [13].

2.1.1 Concepts

A concept in ConceptC++ is an entity that describes the syntactic requirements on a set of type parameters. Concepts can require the existence of certain operations (functions), the existence of certain associated types that are related to the concept's type parameters, and nested requirements.

Operations. Concepts may require any number of operations with signatures using the concepts' type parameters or associated types. The following example defines a concept EqualityComparable that requires its type parameter T to support the equality operation == and inequality operation !=. The two operators are written as C++ declarations, with argument and return types specified precisely in terms of the type parameter of the concept.

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

Associated types. The types used for the argument and return types of operations within a concept need not be limited to built-in types, user-defined concrete types, or type parameters. There are often other types associated with the computation, such as the types stored in a container or pointed to by an iterator. Figure 1 defines the InputIterator concept, which contains an associated type value_type (introduced by the **typename** keyword) that indicates the type that the iterator Iter points to. It is used as the result type of the dereference operator $*$.

Nested requirements. An associated type defined in a concept may refer to any type, and as such we cannot assume any functionality, even the ability to copy an instance of that type. Requirements on the associated type can be introduced in two ways. First, one may include the associated type in concept operations, such as the **operator**$*$ in Figure 1. Second, one may include *nested requirements* that state that the associated type (or any other type available in the concept) must meet certain other requirements, that is, model a particular concept. The **where** statement in Figure 1 is a nested requirement stating that value_type must model the CopyConstructible concept, which allows it to be copied and returned from a function call. Nested requirements are similar to where clauses, described in Section 2.1.3.

```
template<typename Iter>
concept InputIterator : EqualityComparable<Iter> {
  typename value_type;
  where CopyConstructible<value_type>;
  value_type operator*(const Iter&);
  Iter& operator++(Iter& iter);
};
```

Figure 1. A simplified definition of the InputIterator concept from our prototype implementation [15] of parts of the C++ standard library in ConceptC++. The concept includes the associated type value_type, a nested requirement, and two required operations.

Refinement. When the requirements of a concept B subsume the requirements of a concept A, we say that concept B refines concept A. Refinement in ConceptC++ is written explicitly in the concept definition, by listing all refined concepts in the concept header, reusing the C++ inheritance syntax. In Figure 1, the InputIterator concept refines EqualityComparable<Iter>, indicating that all of the requirements of EqualityComparable are also requirements of InputIterator. Refinement introduces an ordering between the two concepts: InputIterator is *more specialized than* EqualityComparable, and every model of InputIterator is also a model of EqualityComparable.

A concept collects a set of requirements into a single entity, and can describe the interface of a software element, such as a function or a class. Several mainstream object-oriented languages, including Eiffel [26], Java [14], and C# [27], support constrained generics with subtyping constraints, using interfaces or abstract classes to describe a set of requirements. Constraints in ConceptC++ do not build on the C++ subtype relation. This enables concepts to describe constraints on out-of-class functions, which are prevalent in generic programming in today's C++. Mainstream object-oriented languages have no equivalent feature to associated types in concepts, such as type members in interfaces, requiring associated types to be pushed into the type parameter list for the interface [13, 19]. Scala is a new, prominent object-oriented language that does include type members in interfaces [33, 34].

Haskell type classes [48] and ML signatures [31] are language mechanisms that express requirements on a set of types without relying on subtyping. ML signatures can contain type members that play the same role as associated types, and there is work towards incorporating associated types into Haskell [4, 5].

2.1.2 Models

Models establish the crucial link between a set of (concrete) types and a concept. Each model is a declaration that states that the specified types model a particular concept, providing bindings for the associated types and operations required by that concept. Models themselves are syntactically similar to concepts. The following model states that the type **char*** is a model of the InputIterator concept shown in Figure 1:[1]

```
model InputIterator<char*> {
  typedef char value_type;
  char operator*(char* const& p) { return *p; }
};
```

The model provides definitions that meet each of the requirements in the InputIterator concept, some explicitly and others implicitly. The ConceptGCC compiler verifies that the model meets all syntactic requirements of the corresponding concept. The **typedef** of value_type meets the requirement for an associated type of the same name. Likewise, the definition of **operator*** meets the requirement for an **operator*** with precisely the same signature, and provides a concrete implementation. Conspicuously absent is a definition of **operator**++: the compiler will implicitly generate a definition from the InputIterator concept. To do so, the compiler synthesizes the following function for **operator**++:

```
char*& operator++(char* & p) { return ++p; }
```

Since ++p is an expression, any suitable ++ operator (built-in or user-defined) can be found and normal C++ name lookup, overloading, and conversions apply. Thus, even though the argument and return types for concept and model operations are specified with exact types (enabling type checking in templates), the implementation in the model allows "loose" matching to actual functions and operators (allowing existing C++ code that relies on conversions to work flawlessly with constrained templates).

Models themselves can be generic, thereby making them applicable to a wide range of types. Generic models are written as model templates, which describe a family of models and can be instantiated with a particular set of types when necessary. The following example generalizes the model InputIterator<**char***> to apply to all C++ pointer types:

```
template<typename T>
model InputIterator<T*> {
  typedef T value_type;
  T operator*(T* const& p) { return *p; }
};
```

Model definitions are akin to Haskell's instance declarations. Whereas instance declarations must contain a definition for each requirement (that does not have a default definition) in the corresponding type class, model definitions need not provide definitions where a suitable function is in the scope of the model. Defining a class that implements an object-oriented interface is also comparable to defining a model. Model definitions in ConceptC++, however, can be added retroactively: a model definition can apply to a set of existing types after they have been defined. In mainstream object-oriented languages, retroactive subtyping is not possible.

2.1.3 Where clauses

Concepts define requirements on type parameters, models specify how concrete types meet those requirements. The third major language construct in ConceptC++ is *where clauses*, which specify the requirements on the type parameters of a function or class template. Where clauses have a dual role in ConceptC++, ensuring both that the arguments to a template meet the requirements of that template and that the template itself does not rely on any template-dependent constructs not explicitly stated as requirements. Requirements in where clauses take on two forms:

Model requirements. The most common form of requirement in a where clause is a *model requirement*, which indicates that there must be a model for a particular concept and set of types. Model requirements are written using the same form as models, e.g., a where clause containing InputIterator<Iter> requires that there exist a model InputIterator<X> for any concrete type X bound to the type parameter Iter.

Figure 2 contains a complete, fully-constrained implementation of the C++ standard find() function. The model requirement InputIterator<Iter> introduces the !=, *, and ++ operations for the type Iter (shown in Figure 1), along with the associated type value_type. Similarly, EqualityComparable<value_type>, introduces the != operator on values of type value_type. Together,

[1] We have slightly altered the ConceptGCC model syntax for presentation purposes, eliminating the specialization header **template**<> and using the keyword **model**.

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

Figure 2. A simplified implementation of the STL find() algorithm as a constrained template.

these requirements cover all operations used in the find() function template, allowing it to type check; see Section 2.2 for more information about the type checking process for templates.

Same-type requirements Same-type requirements indicate that two types are equivalent within the template. These requirements correspond to ML's sharing constraints between types in signatures (see manifest types [23] and translucent sums [24] for detailed discussions on sharing constraints). As an example of the use of same-type constraints, consider the STL merge() algorithm, which has two template type parameters that model the InputIterator concept. The value_types of the iterators must be equivalent, even though the iterator types themselves do not need to be the same.

```
template<typename InIter1, typename InIter2, typename OutIter>
where { InputIterator<InIter1>, InputIterator<InIter2>,
        InputIterator<InIter1>::value_type
          == InputIterator<InIter2>::value_type }
OutIter merge(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2,
              OutIter result);
```

2.1.4 Concept-based function overloading

ConceptC++ extends the notion of function overloading in C++ by allowing two functions with the same name to differ only in their where clause, and overload resolution will choose the most specific function given the modeling relationships of the argument types. Figure 3 shows the standard advance() function templates, written in ConceptC++, overloaded for three standard concepts. The RandomAccessIterator concept is a refinement of BidirectionalIterator, which in turn refines InputIterator. The constraints of the third overload thus imply the constraints of the second and the constraints of the second imply the constraints of the first, making the three overloads ordered from least to most specialized. In general, constraints can induce an overload set that is not totally ordered.

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

template<typename T> where { BidirectionalIterator<T> }
void advance(T& i, difference_type n)
{ if (n > 0) while (n--) ++i; else while (n++) --i; }

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

Figure 3. The advance() function overloaded on several concepts.

The basic model of resolving a call to an overloaded function is as follows: (1) select the set of *candidate functions*, that is, all functions for which the argument types match the parameter types and satisfy where clauses; (2) in the case of more than one candidate function, perform overload resolution based on types only; (3) in the case of more than one function still left, use constraints in the where clauses for determining the best matching function. This differs from, e.g., C#, where all type parameter constraints are ignored for overloading purposes [28, §20.9.7]. The ordering determined from where clauses is based on implication: one overload is more specialized than the other if its concept constraints imply the constraints of the other but not vice versa. In ConceptC++, a set of constraints A implies another set B if for every constraint in B, there is some constraint in A which is either the same or a refinement of the constraint from B. This is a direct extension of the current C++ rules for partial ordering of function templates [17, §14.5.5.2]. Section 4.1 discusses the relation induced by implication between constraints in more detail.

2.2 Type checking constrained templates

ConceptC++ introduces improved type checking for constrained templates. It is a common misconception that C++ offers no type checking for templates at the time of their definition. In fact, type checking of templates is divided into two phases. The first phase occurs at the time of template definition, during which the compiler type checks expressions whose types do not depend on any template parameter (directly or indirectly); these expressions are called *non-dependent*. The second phase of type checking occurs at template instantiation time, when the template parameters are substituted for their concrete arguments; then the compiler type checks expressions whose types depend on one or more template parameters, that is, *dependent* expressions.

Constrained templates in ConceptC++ perform more extensive type checking in the first phase then C++, by considering every expression to be non-dependent. Each expression is type checked in the lexical scope of the template definition augmented with the requirements stated in the where clause. These requirements are introduced into the scope of the template. For example, the two concept requirements in the where clause of Figure 2 introduce the associated type name value_type into the scope of find() along with several function declarations:

```
// From InputIterator<Iter>
value_type operator*(const Iter&);
Iter& operator++(Iter&);

// From EqualityComparable<Iter>
bool operator==(const Iter& x, const Iter& y);
bool operator!=(const Iter& x, const Iter& y);

// From EqualityComparable<value_type>
bool operator==(const value_type& x, const value_type& y);
bool operator!=(const value_type& x, const value_type& y);
```

Type checking of the template then uses a similar procedure to the type checking of non-template code in current C++. To type check the expression first != last, for instance, the compiler initially gathers all operators with the name !=, including the two declarations above from the where clause in addition to built-in operations and any in the scope of the template. C++ overload resolution then selects the **operator**!= that pertains to values of type Iter, and the call is resolved to EqualityComparable<Iter>::**operator**!=(first, last).

If ConceptC++ provided perfect modular type checking, the first phase of type checking would be sufficient to ensure that any instantiation that meets the requirements of the template would be valid. However, support for specialization implies that during the second phase of type checking, at instantiation time, the compiler will select more specialized functions and data types. The following section describes the effect of specialization on the type system.

3. Type checking specialized generic components

Modern C++ libraries use specialization frequently to attain the ideal of generic programming: combining generality and efficiency.

Several idiomatic implementation techniques are in use to implement specialization using current C++. This section discusses these idioms, and explains how ConceptC++ provides direct language support for them. The latter part of this section demonstrates the difficulties that supporting specialization causes for modular type checking of generic components.

3.1 Specialization in C++

In the compilation model of C++ templates, different code is generated for every different instance of a template. This *instantiation model*, combined with the ability to specialize templates and the fact that no constraints are necessary for type parameters, invites generic component authors to take advantage of the properties of concrete types bound to type parameters. For example, algorithm specialization is used to allow the best implementation of an algorithm to be selected for particular data types, or particular categories of data types described using concepts. This is a prevalent idiom, e.g., in the C++ standard library.

Because overload resolution is delayed until instantiation time, specialization works for calls inside other generic functions as well; a single non-specialized definition of a generic algorithm may invoke other generic algorithms that are specialized. For example, the inplace_merge() function in the standard library implements its own functionality without specialization, but calls directly or indirectly several other functions, such as advance() and rotate(), that are specialized to obtain higher performance on data structures supporting random access, and copy(), that can additionally be specialized to use an architecture-specific routine for built-in pointer types.

To deliver the expected performance, C++ generic programming libraries often rely on the compiler's ability to inline generic functions as well as it can inline non-generic functions. For example, the reverse_iterator adaptor in the standard library, given a sequence, provides a view on the reverse of the sequence. This adaptor has minimal performance overhead, as its functions contain hardly any code, and thus are inlined into whichever component uses reverse_iterator. Similarly, the instantiation model can enable the inlining of the arguments to higher-order functions. For example, the transform() function is roughly equivalent to map() in functional programming languages. In the C++ implementation, however, the binary operation passed to transform() is supplied both as a template argument and as a value, and thus its code can be inlined into the body of an instance of transform(). Note that inlining calls to generic functions does not strictly require an instantiation model, and in fact has been done in Haskell, which usually implements generic functions using dictionary passing [20].

Finally, the combination of instantiation model, lack of template parameter constraints, and specialization allow a form of metaprogramming with templates. For example, version 2 of the Matrix Template Library (MTL) [38] includes a set of metaprograms to generate unrolled code for some simple linear algebra operations on fixed-size vectors and matrices. These metaprograms are written using only the C++ template system, without any additional tools being required. Due to the instantiation model, each use of the metaprograms produces a separate piece of code, which can then be inlined and optimized by the compiler.

Specialization is essentially about overloading on concepts. Several implementation techniques are in use to emulate it in current C++. Tag dispatching is based on tag types that correspond to certain concepts; each type that models one of these concepts defines an associated type that gives the tag type for the concept the type models. Specialized generic algorithms are then implemented as functions that detect the tag and then dispatch to a specialization, which is defined as a separate function. A more refined dispatching mechanism is presented in [25]. We reported on a library technique for direct form of concept-based overloading [18] that does not require a dispatching function. Concept-based overloading was later proposed as a language feature by Stroustrup [44].

```
template<typename T> class vector {
public:
    typedef T& reference;
    reference operator[](std::size_t);
    T* data();
};
template<> class vector<bool> {
public:
  class reference;
  reference operator[](std::size_t);
  // no data() member: it doesn't make sense
};
template<typename T> void foo(vector<T>& values) {
  // type checks against vector<T>, but fails when T=bool
  T* first = values.data();
  T& front = values[0];
}
```

Figure 4. Specializations of a class template do not necessarily define member functions as specializations of those of the primary template, and may even omit member functions.

3.2 Challenging overloading situations

This section describes two problematic scenarios in type checking generic definitions in ConceptC++: mismatch in the members of class template specializations and ambiguous specializations.

The first problem arises because C++ class templates may be specialized, but the definitions of these specializations can be arbitrary. Figure 4 illustrates a portion of a generic, dynamically-sized array class vector, its vector<**bool**> specialization, and a generic function foo() that uses vector. The vector<**bool**> specialization, as in the C++ standard library, optimizes for space by packing boolean values as individual bits into an array of integers. This optimization changes the interface to vector<**bool**> in two ways. First, since there is no longer an array of **bool** values, a data() member function that returns a pointer to **bool** cannot be implemented. Second, and more subtly, the type returned from the subscripting operator [] is no longer an actual reference to a **bool** value in memory; instead, it is proxy class that one can use to read and write values of type **bool**, but it translates these operations into bitwise manipulations.

The generic function foo() in Figure 4 proves challenging for type checking. When the template is initially type checked, the primary (most general) definition of vector will be used for type checking, and the type check succeeds. However, when foo() is instantiated with T = **bool**, the vector<**bool**> specialization is found and both statements in foo() fail to type check at instantiation time.

Although it is possible to remedy these problems by restricting the way in which specializations can be defined (i.e., a specialization of a class template must provide members that are specializations of the members of the primary template), we do not opt for this solution in ConceptC++. To do so would make illegal many existing, correct programs and make the template system largely unusable for template metaprogramming.

The second problem, the possibility of ambiguous specializations, is an inherent conflict of specialization and modular type checking. Even though all specializations are known to be correct to use, an error occurs if no single specialization can be determined to be the best. Consider the overloads in Figure 5. We assume three concepts (loosely based on those from the SGI STL documentation [41]): Sequence provides the ability to traverse a sequence lin-

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

Figure 5. Three sort() overloads. We assume here that the concepts SortedSequence and RandomAccessSequence are not in a refinement relationship with each other, but that both are refinements of the Sequence concept. The definitions to establish these assumptions are not shown. Note that a sequence type modeling SortedSequence implies that its value_type models LessThanComparable.

early, SortedSequence additionally guarantees that the sequence is sorted, and RandomAccessSequence enables constant time access using an index (but does not guarantee sortedness). The Sequence concept captures the minimal requirements for sorting, assuming the element types can be compared, enabling the implementation in the first overload. The second and third overloads both provide faster implementations, to be used when more is known about the input type. For the purposes of this example, we assumed that only one comparison function is allowed for each type, and so any model of SortedSequence is known to be sorted by the same key as sort() uses.

Assume the following model definitions and a call to sort():

```
class my_ra_priority_queue { ... };
model SortedSequence<my_ra_priority_queue> { ... };
model RandomAccessSequence<my_ra_priority_queue> { ... };

my_ra_priority_queue pq; ...
sort(pq);
```

The call to sort() is ambiguous. Overload hierarchies like this one compromise modular type checking: a non-overloaded library component, whose first-phase type checking succeeds, can call an overloaded function internally, which may be ambiguous in overload resolution performed at instantiation time. To see this, assume that the sort() functions in Figure 5 are internal to a module that publishes the following function in its interface:

```
template<typename S>
where { Sequence<S>, LessThanComparable<value_type> }
S select_n_first(S s, int n) { ...; sort(s); ...; }
```

Type checking select_n_first() type checks the call to sort() with the information that S models the concept Sequence, matching the first sort() function in Figure 5. Then, all overloads in the current lexical scope that are token-wise equivalent to the selected function, except for stronger constraints, are considered specializations of the selected function. Here, the second and third overloads are specializations of the first one. No ambiguity is detected at this point. Later, invoking select_n_first() with an object of type my_ra_priority_queue that satisfies the constraints of both specializations realizes the ambiguity between them. Here, sort() might not be part of the public interface of the library, and thus potentially not known to the caller of select_n_first(). We note that it is common practice in generic programming to dispatch to overloaded functions internal to the library, as discussed in Section 3.1. Moreover, the performance of libraries such as MTL and μBLAS is crucially dependent on specializing calls to internal library functions.

```
template object A;
method m(x@A, y@A) { ... }

template object B isa A;
method m(x@A, y@B) { ... }
method m(x@B, y@A) { ... }

object b isa B; m(b, b);
```

Figure 6. Ambiguous multimethod definitions.

The ambiguity problem is not specific to ConceptC++ and has been explored in the context of other languages and there are many possible solutions. The following section analyzes these solutions and their applicability to ConceptC++.

4. Design space of concept-based overloading

The notion of multimethods arose from the limitations of single argument dispatch, which was, and still is, the only mechanism in many object-oriented languages. With multimethods, dispatching is based on the types of all of the arguments of a method. Multimethods were pioneered in CLOS and its predecessors [2, 3], followed by a host of research and implementations in other languages, such as Cecil and Java extensions [6, 8, 29, 30]. The expressive power gained over single-dispatch came with a price, however, compromising modular type checking and opening up the possibility of "message not understood" and "ambiguous message" errors. For multimethods, the goal is to detect calls that could fail at run-time whereas in ConceptC++ we are interested in detecting calls that could fail at later phases of compilation, that is, during the instantiation of generic components. The "message not understood" errors were illustrated by the missing data() method in Figure 4. The "ambiguous message" errors correspond directly to the ambiguity problem described in Section 3.

The Cecil code in Figure 6 shows a classic example of an ambiguous method call in multimethods. The call m(b, b) cannot be resolved to either of the last two definitions of method m(). Here, calling m() will trigger the ambiguity at run-time; ConceptC++ detects ambiguities at instantiation time of generic components.

The research on multimethods has focused on type checkers and analyses to efficiently detect ambiguities, and, to make this possible, suitably restricting the kinds of allowed multimethods. See [29] for an analysis of the design space of such restrictions. More recent work has focused on frameworks that can tolerate potential ambiguities, and only deal with them if they actually appear [30]. In what follows, we describe overloading across various languages, the issue of potential ambiguities and possible solutions to it, and how these relate to ConceptC++.

4.1 Detecting and resolving ambiguities

Whether based on concepts, types, or values, the mechanism of function overloading has a common core: an overloaded function describes a set of constraints on its arguments. For example, the constraint set of the first advance() function in Figure 3 could be loosely described as:

- first argument is of type T
- second argument is of type difference_type
- T models InputIterator

Constraint sets thus arise from the combination of concept constraints, traditional C++ type constraints, and same-type constraints (which the example does not demonstrate). The constraints on a

group of functions induce a specialization ordering between the functions; overload resolution then tries to find the function with the most specialized set of constraints that matches a particular set of argument types. Note that a constraint set can include constraints involving more than one argument type, making it impossible to define the specialization ordering by looking at each argument position in isolation, as is done by C++ overloading. All constraints in the constraint set are considered equal in weight; there is no preference ordering between them.

Jones [21] studied extensively the properties of constraint sets with specialization ordering based on implication, and their use as bounds for type parameters of software components. The specialization ordering induced by the implication relation of constraint sets is a preorder, a reflexive and transitive (but not antisymmetric) relation on functions. Note that although we discuss functions here, same principles apply to other kinds of overloadable entities, such as class templates in C++. Under this preorder, which we represent by $\precsim$, $A \precsim B$ means that A is *at least as specialized as* B. For overloading purposes, functions can be identified with their associated constraint sets, and so we can think of A and B as sets of constraints. Note in this definition that the wording is "at least as specialized," and so it is possible that two overloads are equally specialized (e.g., they have exactly the same constraint set). It is also possible for neither overload of a pair to be at least as specialized as the other, as with overloads on distinct concrete types.

From the preorder, we define an irreflexive, antisymmetric relation $\prec$ in the standard way: $A \prec B$ iff $A \precsim B$ and $B \not\precsim A$. The meaning of $A \prec B$ is that A is *strictly more specialized than* B. This removes the possibility of two overloads being mutually related by the order $\prec$, as was allowed by $\precsim$.

Given an overload set we resolve overloading by finding the set of its minimal elements under $\prec$. A single minimal element means that overload resolution was successful, an empty set indicates that no overloads were found (if there was at least one overload before overload resolution, there will be at least one which is minimal), and two or more minimal elements indicates that several overloads were equally good, and thus ambiguous.

We can now describe the ambiguity in the sort() overloads in Figure 5 using this framework. Assume the three sort() overloads are internal to a library L1, and let their constraint sets be A, B, and C, in the order of their definition. The relation of the constraint sets is described as $B \prec A$, $C \prec A$, $B \not\prec C$, and $C \not\prec B$. Further, assume a public function f() (this corresponds to select_n_first() in Section 3.2), whose constraint set is A, and makes a call to sort(). We are interested in type checking library L1 in isolation of any uses of f(). Furthermore, we consider a call to f(), from an external library, with types that satisfy some constraint set D, such that $D \precsim B$, and $D \precsim C$. In this case, all overloads of sort() match, as D includes enough requirements to satisfy all of their constraints. The overload with constraint set A cannot be the best match, as A is strictly dominated by the overloads B and C. Which of B and C is a better match cannot be determined.

In general, there are three main approaches to deal with ambiguities of the above kind: (1) type-systems that reject programs with *potentially ambiguous* calls, (2) defining the rules of comparing the constraint sets so that a total order is guaranteed, or (3) allowing ambiguities to be detected at a later phase and providing ways to deal with them. By "potentially ambiguous" we mean that it is possible to define types, somewhere in the program, that would lead to an ambiguous call. We discuss these approaches in more detail below, and analyze their suitability for C++.

4.1.1 Static overloading

The most straightforward approach is to forgo specialization altogether inside a generic function. In our example, this would mean that overload resolution performed during type checking the definition of f() based on the constraint set A stays in effect, and is not redone at instantiation time. Non-generic functions, on the other hand, will still use overloading to select the most specialized versions of the functions they call. This approach provides fully modular type checking. The $\mathcal{G}$ language for generic programming [36, 40] follows this model. The approach is also used for overloaded methods in object-oriented languages without multi-methods, such as C++, Java, and C#.

One can design around the lack of specialization by making each function that has specializations into a requirement of a concept. For example, the advance(), distance(), etc. functions could be function requirements in the InputIterator concept. Alternatively, separate concepts, such as Advanceable, could be defined. This either leads to "fat" concepts with many redundant requirements, or where clauses with large numbers of model requirements, each of which captures a single function.

This design for specialization does remove the problem of ambiguities in internal overloads called by a generic function: all function calls are resolved when the function is first seen, and the overloads used never change based on the types of the arguments to the function. Therefore, the only ambiguities possible from calling a library function are ambiguities caused in finding models to satisfy the concept constraints in the function's interface. Moving internal overloaded functions from the library into extra concept constraints does not change this: ambiguities are found when resolving those extra constraints when calling the library function, and those new constraints become part of the function's interface.

As a variation of static overloading, overload resolution could be performed to find the most-specialized *unambiguous* specialization. In our example this approach would lead to the same result as static overloading. To exhibit a difference, assume a fourth overload of sort() with constraint set A_1, such that $B \prec A_1$, $C \prec A_1$, and $A_1 \prec A$. A_1 would now be the most-specialized unambiguous match. A result is guaranteed to exist by the existence of the overload A that was necessary for type checking the body of f(). Again, type checking is modular. This approach provides a limited form of specialization. The specializations of advance(), distance(), etc. in the STL would be found, since their constraint sets form a chain. The down-side is that the selected overload may be less efficient or use more resources than any of the more specialized overloads that were rejected due to ambiguity, which would be the case in Figure 5. Silently ignoring the ambiguous definitions may thus not be the desired outcome: the only observable effect of the ambiguity would be degraded efficiency.

4.1.2 Preference rules

Several additional criteria to define a total ordering between constraint sets, and thus overloads, are possible. One criterion is the order of definitions within the program text. This rule corresponds to pattern matching as commonly found in functional languages. In pattern matching constraints can freely overlap, but are tried in order. Pattern matching constructs typically require all the cases to be together. The cases in ConceptC++ are freely extensible, however, making the ordering vulnerable to subtle changes in code, such as reordering of include files.

CLOS [2] and CommonLoops [3] averted the ambiguity problems by using the argument positions as a tie-breaker, favoring earlier argument positions. Such rules have been criticized (e.g., [6]) for forcing programmers to consider the effects of a seemingly orthogonal detail, the argument order of functions, on method lookup. Furthermore, such automatic rules can hide ambiguities that are programming errors and should be detected.

Such rules would be even more subtle in ConceptC++, where constraints do not necessarily arise from a single function argu-

ment: multi-parameter concepts are possible and appear, e.g, in the standard library.

4.1.3 Exhaustive search

It is possible to detect that a generic call could, with suitable argument types, exhibit an ambiguity. In our example, this means examining the generic function f() only and proceeding as follows. Once the best matching sort() overload, based on constraint set A, has been identified, all other overloads that can potentially match (have stronger requirements than A) are collected. We make the open-world assumption: the type checking cannot use any knowledge on what types exist in the program. This means that type checker must assume that for any combination of concepts, there can be types that model those concepts and types that don't. We can thus view a requirement "type T models concept A" as a truth variable, consider each overload as a Boolean formula, and analyze the set of these Boolean formulas to determine whether ambiguities can arise.

Let us pick a Boolean variable for each constraint mentioned in the set of potentially matching overloads. Overload resolution then becomes the analysis of all possible truth assignments to the formulas derived from each overload. If for each truth assignment:

- exactly one formula is true, or
- more than one formula is true and the overloads these formulas originate from are totally ordered,

then a unique best-matching function is guaranteed to exist. This algorithm is exponential in worst case. Most likely the exponent will not be big, but there is no guarantee of this. Ernst et al. [11] discuss a unified framework for dispatching, where specialization ordering is defined via logical implication, and show how ensuring that a single most specific method exists reduces to tautology testing. They argue that with some tailoring of the algorithm, the test is fast in many practical situations.

If a potential ambiguity is detected, it can be resolved by adding a tie-breaking overload between all pairs of ambiguous overloads. The constraint sets of these new overloads must be the conjunction of the constraint sets of the ambiguous overloads. In our running example, this means providing an overload for sort() with the constraint set $B \wedge C$. Note that these new functions can be ambiguous, so the process may need to be repeated recursively: an overload may be required for every combination of a set of independent concepts, possibly requiring an exponential number of overloads. This approach could lead to several false positives, and force library developers to write meaningless tie-breaker functions. Note that in our example, if it was guaranteed that no type can satisfy both constraint sets B and C simultaneously, the tie-breaking sort() function would be unnecessary. The type system of ConceptC++ does not support the expression of such guarantees, as discussed in Section 4.2.

Millstein and Chambers provide a set of restrictions on multimethods (System M) to allow less expensive modular type checking [29]. One of the necessary restrictions (M1) of System M however, is not suitable for generic programming: this rule requires that one specified argument position must be specialized with only types local to the current module. Converted to concept-based specialization, this implies that each specialization of a function must be based on a locally defined concept.

4.1.4 Allow instantiation time errors

In this approach overload resolution is redone with full type information at instantiation time, and errors are produced if ambiguities are detected. Hence, no effort is made to detect potential ambiguities, or even whether the program contains types that can exhibit those ambiguities. This is the approach taken by C++ today. Similar ambiguities can appear in the context of Haskell type classes as overlapping instances of a type class with type patterns that unify, but neither of which subsumes the other [35, § 3.7].

Approaches for handling them have varied from detecting all patterns which unify before any conflicting instance is used, as is done by Hugs and Glasgow Haskell Compiler (GHC) versions prior to 6.4, to allowing ambiguities to exist, and only detecting them when an ambiguous instance lookup occurs, as is done by GHC version 6.4 [47, §7.4.4.1]. A similar trend can be seen in the development of MultiJava and its later extension Relaxed MultiJava [30].

Delaying the detection of ambiguities to instantiation time can be viewed as a whole-program analysis. Less accurate whole-program analyses are possible, such as ensuring that a program contains no types that would exhibit the ambiguity; such an approach is described in the context of multimethods, e.g., as *System G* in [29] and is used in Cecil [9].

The approach chosen for ConceptC++ is allowing instantiation-time errors to occur. In the first phase of type checking a single best matching overload for each function call is identified, as described in Section 2.1.4. Every function overload whose constraints are stronger than the constraints of this selected function is considered when performing the second phase of overload resolution with full type information during instantiation. This set of functions can contain ambiguous overloads; constrained generics in C++, in our design, do not deliver fully modular type checking. The crucial question then becomes if and how it is possible to resolve the ambiguity errors at the client-side of a generic library. The following section discusses possible mechanisms to resolve ambiguities both at the implementation-side and client-side.

4.2 Resolving ambiguities

The basic language mechanism for resolving ambiguities is the definition of tie-breaking overloads, with the consequences described in Section 4.1.3. We can envision mechanisms that reduce the number of necessary tie-breakers that merely forward a call to one of the ambiguous overloads, rather than adding any new code. First, negating a concept constraint can cause an overload to not match for a case that would otherwise be ambiguous. For example, using a negative constraint in the last sort() overload in Figure 5, we could remove the ambiguity in favor of the sort() with the empty body:

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

Negative constraints are allowed in the frameworks of predicate classes [7] and predicate dispatching [11], and also advocated in [46]. Negative constraints can reduce the number of tie-breaker functions required, but the effect is "shallow." From a type checking perspective, the only information that a negative constraint can propagate into the template body is the fact that a particular model does not exist, which presumably could resolve ambiguities in the template. However, ambiguities occur when specializations are found at instantiation time, during which the constraints in the where clause are not used (because concrete types are available). Moreover, negative constraints require knowledge of the entire overload set *a priori*. New overloads may necessitate adding additional negative constraints to existing functions to keep the overload set unambiguous.

Another approach is to remove the source of the ambiguity without changing the ambiguous functions by declaring certain sets of concepts to be disjoint: no single type can ever be a model of more that one concept in such sets. This is the concept-based analog to disjointness assertions for predicate classes [7], as appear in the Cecil language [9]. For example, the ambiguity in Figure 5 could be resolved by stating that it the concepts RandomAccessSequence

and SortedSequence are disjoint. This would, however, render sorted random access sequences illegal!

Defining a preference order between concepts, in addition to the order induced by the refinement relation, could also serve to resolve ambiguities. In the sort() example, we could state that SortedSequence is a more preferred concept than RandomAccessSequence. Such decisions seem arbitrary. Here, the ambiguity would be resolved in a desired way, selecting the more efficient overload. The same ordering would be then forced in the whole program and might lead an undesired, less efficient, function to be selected for another overload set elsewhere in the program. A variation of this approach is to specify the preference ordering for a single type at a time. For example, we could declare that my_ra_priority_queue models SortedSequence "more" than RandomAccessSequence. A type class-based overloading mechanism for Haskell with similar behavior is discussed in [22].

A direct mechanism to resolve an ambiguity is to explicitly qualify the overload to call. This form of ambiguity resolution appears in C++ when taking a pointer of an overloaded function, where the entire signature of the selected overload must be written out. This is verbose, but more fundamentally, cannot serve as a general mechanism for resolving ambiguities. An ambiguous call, both in current C++ and ConceptC++, can occur nested inside a library, and cannot be accessed by the programmer.

While there are many partial solutions to resolve ambiguities, none of the solutions presented is clearly superior to the others. The following section describes the design choices that informed ConceptC++ and details library- and client-side approaches to resolving ambiguities.

4.3 Resolving ambiguities in ConceptC++

We rejected static overloading as the solution for ConceptC++. Specialization is an integral part of achieving efficiency with generic programming and is widely used in generic C++ libraries. With static overloading, specialization requires more pre-planning and is achieved in a roundabout manner, which discourages its use. Hence, ConceptC++ is vulnerable to ambiguities that follow from specialization. Potentially ambiguous overloads are not statically detected. Our conjecture is that vast majority of the potentially ambiguous overloads will never exhibit a real ambiguity. Although we do not yet have a sufficiently large base of ConceptC++ code to thoroughly evaluate our decision, we have yet to encounter such ambiguities in practice. This conjecture is in line with recent design choices in Haskell and Relaxed MultiJava, as we discuss in Section 4.1.4. Specialization is commonplace in C++ and overloads for the same functions can be provided in several include files. We also do not require programmers to add tie-breaking overloads during library composition that do not resolve real ambiguities. Neither do we provide any of the features to reduce the number of necessary tie-breaker overloads. However, we note that the client-side effects of negative constraints can be emulated within our system using metaprogramming techniques [18], but constraints expressed in this way will be ignored when type checking templates.

To resolve the (hopefully rare) ambiguities on the client side, we can rely on standard C++ mechanisms. Thin adaptor classes can be created for types that are the sources of the ambiguities, exposing a subset of the functionality of the data types involved. For instance, we could solve the ambiguity of sort() functions in Section 3.2 by creating an adaptor for the my_ra_priority_queue class as follows:

```
class my_ra_priority_queue { ... };
model SortedSequence<my_ra_priority_queue> { ... };
model RandomAccessSequence<my_ra_priority_queue> { ... };

my_ra_priority_queue pq; ...

class my_adapted_priority_queue : public my_ra_priority_queue {
  // define forwarding constructors/functions...
};
model SortedSequence<my_adapted_priority_queue> { ... };

sort(my_adapted_priority_queue(pq));
```

This is somewhat cumbersome, and it remains to be evaluated whether a more direct mechanism is worth supporting. We note, however, that there are cases that cannot be solved with adaptors, or with any other client-side ambiguity resolution mechanisms discussed in Section 4.2:

```
template<typename T> concept A {};
template<typename T> concept B : A<T> {};

template<typename T, typename U> where { A<T>, A<U> }
void f(T, U);

template<typename T, typename U> where { B<T>, A<U> }
void f(T, U);

template<typename T, typename U> where { A<T>, B<U> }
void f(T, U);

template<typename T> where { A<T> }
void g(T t) { f(t, t); }
```

If g() in this example is invoked with a type that models B, no adaptor can be written to select either of the more specialized overloads. C++, however, does not have a real module system, so in the case that such persistent and difficult to resolve ambiguities would arise, it is always possible to add the necessary tie-breaking overloads. In summary, our design of ConceptC++ favors the expressiveness of unrestricted specialization even though it compromises modular type checking in some pathological situations. The set of specializations is always extensible, and the programmer is not forced to maintain the property that a unique most specialized definition exists for all possible calls. As in C++ today, ambiguities are only reported if they appear, not if they are a mere possibility. C++ offers mechanisms to resolve the ambiguities in all cases.

5. Related work

Stroustrup and Dos Reis propose an alternative formulation for concepts in C++ [10, 46]. Although the terminology differs, the basic structure of both proposals is the same, with concepts as entities that encapsulate a set of requirements, models that provide the mapping between a set of types and a concept, and where clauses that list constraints on templates. The major differences between the proposals occur in two places: how operations and refinements are specified within a concept and what kinds of requirements can occur within a where clause. The two proposals take different approaches to type checking templates. Stroustrup and Dos Reis stress convertibility between types as fundamental to type checking. For instance, a concept can specify a requirement for a $<$ operator with the *usage pattern* **bool** b = x $<$ y (where x and y are variables of some type T), which requires an operator $<$ such that values of type T can be converted to its parameter types and whose return type is convertible to **bool**. Our proposal, on the other hand, considers type identity to be fundamental to type checking: a signature **bool operator**<(**const** T&, **const** T&) provides precise types that are used in type checking. Conversions may occur when there exists a suitable conversion function (e.g., **operator** T(**const** U&)), but are not considered to be special. In addition, our system permits strong type aliasing in templates by way of same-type constraints, a feature that cannot be emulated. Despite the differences in approaches to type checking templates, the same fundamental tension exists between modular type checking and specialization. Although we have explored this tension within our own system for

constrained genericity, we believe that the results can translate to the system of Stroustrup and Dos Reis.

Problems similar to those we have encountered with specialization and modular type checking for templates have been studied in the context of multimethods [2, 3, 6, 8, 29, 30] and Haskell type classes with overlapping instance declarations [35] [47, §7.4.4.1]. Section 4 discusses these connections in detail. The qualified types framework [21] details properties of the implication-based specialization ordering we use in ConceptC++.

Concepts in ConceptC++ are used to express sets of constraints on type parameters of generic functions. Related mechanisms exist in Haskell (type classes), C# and Java (interfaces), ML (signatures), and many other languages, as briefly discussed in Section 2.1. An extensive study of how these language mechanisms relate to concepts, and of their effectiveness for generic programming, is given in [13].

6. Conclusions

Generic programming grew up in the unconstrained world of C++ templates, the flexibility of which made it possible to develop generic libraries that pay little or no performance penalty for their abstractions. However, the techniques required to realize generic programming in C++ make the construction and use of generic libraries overly complicated.

By introducing constrained generics into C++, we hope to capture the essential features required for generic programming while retaining the flexibility and efficiency that has made generic programming successful in C++. In particular, we wanted to introduce modular type checking for constrained C++ templates but retain C++ specialization and overloading. In this paper we have described the tension between these two design goals and evaluated the solutions provided for similar problems with multimethods.

Our proposed variant of C++ for generic programming, ConceptC++, is the result of many design tradeoffs in modular type safety and support for specialization. We have implemented our design for constrained genericity [16] in ConceptGCC [15] and performed an extensive evaluation on the compiler by extending the GNU C++ Standard Template Library (libstdc++) to use ConceptC++. Detailed reporting on this effort is future work, but we can summarize the main results here. We found that ConceptC++ is able to express the ideas of the STL cleanly and concisely, formalizing what has long been present only in informal documentation. The end result is a major improvement in usability of generic programming in C++: errors that previously generated hundreds or thousands of lines of error messages now generate short, direct error messages; generic algorithms that before required complex C++ template idioms were drastically simplified using the new features of ConceptC++; and by introducing type checking of templates we uncovered several bugs in the libstdc++ implementation.

Acknowledgments

This work was supported in part by NSF grants EIA-0131354, CCF-0541014, CCF-0541335, and a grant from the Lilly Endowment. The third author was supported by a Department of Energy High Performance Computer Science Fellowship. We thank Bjarne Stroustrup and Gabriel Dos Reis for initiating the effort to bring constrained generics to C++ and for many interesting discussions on the topic. The fifth author was supported by the grants: NSF ITR-0113569 Putting Multi-Stage Annotations to Work, Texas ATP 003604-0032-2003 Advanced Languages Techniques for Device Drivers, and NSF SOD-0439017 Synthesizing Device Drivers.

References

[1] 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.

[2] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kicsales, and D. A. Moon. Common LISP Object System specification X3J13 document 88-002R. *SIGPLAN Not.*, 23(SI):1–143, 1988.

[3] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In *OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 17–29, New York, NY, USA, 1986. ACM Press.

[4] 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, New York, NY, USA, September 2005. ACM Press.

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

[6] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[7] Craig Chambers. Predicate classes. In *ECOOP '93 – Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 268–296, 1993.

[8] Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *ACM Trans. Program. Lang. Syst.*, 17(6):805–843, 1995.

[9] Craig Chambers and the Cecil Group. *The Cecil Language: Specification and Rationale, Version 3.1*. University of Washington, Computer Science and Engineering, December 2002. `http://www.cs.washington.edu/research/projects/cecil/`.

[10] 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, New York, NY, USA, 2006. ACM Press.

[11] Michael D. Ernst, Crag Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211, New York, NY, 1998. Springer-Verlag.

[12] 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.

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

[14] 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.

[15] Douglas Gregor. ConceptGCC: Concept extensions for C++. `http://www.osl.iu.edu/~dgregor/ConceptGCC`, 2005.

[16] 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.

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

[18] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-controlled polymorphism. In Frank Pfennig and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *LNCS*, pages 228–244. Springer Verlag, September 2003.

[19] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *OOPSLA*, October 2005.

[20] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 107–117, 1994.

[21] Mark P. Jones. *Qualified Types: Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[22] Oleg Kiselyov. Dynamic dispatch on a class of a type. Haskell mailing list, March 2003. `http://okmij.org/ftp/Haskell/types.html#class-based-dispatch`.

[23] 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.

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

[25] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming*, October 2000.

[26] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall, New York, NY, first edition, 1992.

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

[28] Microsoft Corporation. *C# Version 2.0 Specification, March 2005 Draft*, March 2005. `http://msdn.microsoft.com/vcsharp/-programming/language`.

[29] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303, Lisbon, Portugal, June 1999. Springer Verlag.

[30] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed MultiJava: balancing extensibility and modular typechecking. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 224–240, New York, NY, USA, 2003. ACM Press.

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

[32] David A. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of International Symposium on Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Rome, Italy, 1989.

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

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

[35] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Proceedings of the Second Haskell Workshop*, June 1997.

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

[37] 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.

[38] Jeremy Siek and Andrew Lumsdaine. A modern framework for portable high performance numerical linear algebra. In *Modern Software Tools for Scientific Computing*. Birkhäuser, 1999.

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

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

[41] Silicon Graphics, Inc. *SGI Implementation of the Standard Template Library*, 2004. `http://www.sgi.com/tech/stl/`.

[42] A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, April 1994. `http://www.hpl.hp.com/techreports`.

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

[44] Bjarne Stroustrup. Concepts – a more abstract complement to type checking. Technical Report N1510=03-0093, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2003. `http://www.open-std.org/jtc1/sc22/wg21`.

[45] 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`.

[46] 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.

[47] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide*, version 6.4.1 edition. `http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html`.

[48] 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.

[49] Jörg Walter and Mathias Koch. *Boost Basic Linear Algebra*. Boost, 2002. `www.boost.org/libs/numeric`.