Algorithm specialization and concept-constrained genericity

Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine

Open Systems Lab, Indiana University
Bloomington IN, USA
{jajarvi|jewillco|lums}@osl.iu.edu

1 Introduction

Generic programming is an emerging programming paradigm for creating reusable domain-specific software libraries. It is a systematic approach to software reuse focusing on finding the most general (or abstract) formulations of algorithms and then implementing efficient generic representations of them. The generic programming paradigm has gained popularity in the C++ community; the Standard Template Library [1, 2], now part of the C++ Standard Library [3], was the starting point. Other widely used libraries have since been developed following the paradigm, such as the Boost Graph Library [4], the Matrix Template Library [5] and the μBLAS Library [6].

Much of the appeal of C++ for generic programming can be attributed to its versatile template system, which ironically is also a significant source of problems. It is common knowledge that the *unconstrained genericity* model of C++ does not provide the best support for modularity; generic functions cannot be compiled, or even type checked, in isolation of their uses. Lack of *separate type checking* delays the detection of a class of errors from the definition of generic components to their use; discrepancies between the documented requirements of the parameters to the generic algorithms and their true requirements can go undetected until a generic library is already in use. This makes error diagnosis difficult; error messages triggered from template instantiations tend to be verbose, and it is difficult to assign blame for an error either to an erroneous library component or to the call site invoking the library component [7–9]. As a remedy, Stroustrup has initiated work towards adding *constrained genericity* to C++ [10–12].

Performance lies at the very heart of generic programming. Ideally, using a generic algorithm with some concrete input types should be no less efficient than an equivalent non-generic algorithm that was written for those concrete types to begin with. Such a generic algorithm is said to have no *abstraction penalty* [13–15]. One key idea for achieving this is *algorithm specialization*: the existence of specialized forms of the same algorithms that are automatically selected based on the the capabilities of the inputs. If the inputs provide more capabilities, a more efficient implementation may be possible. For example, finding a particular element from an ordered sequence is faster than from an unordered one. The importance of algorithm specialization and automatic selection of the most specialized algorithms for reducing the complexity of generic designs is discussed in [16].

C++'s unconstrained genericity and its strategy of compiling each different instantiation of a template into a distinct piece of code support algorithm specialization in

a natural way. Algorithm specialization with constrained genericity, however, is more problematic. In particular, attaining separate type checking of generic algorithms is not possible without restricting C++'s overloading model.

This paper is an informal description of the fundamental limitations of constrained genericity, compared to the current unconstrained template system of today's C++. We focus on specific problems that arise with algorithm specialization. We hope that the paper contributes to the work towards adding a constrained genericity mechanism to C++. Section 2 describes algorithm specialization and how it is commonly implemented in generic libraries written in current standard C++. Section 3 discusses the means of expressing constraints on generic parameters, and reviews different strategies for compiling generic definitions. Further, this section discusses two approaches for overload resolution under constrained genericity and details the difficulties of achieving algorithm specialization and separate type checking together. Section 4 concludes the paper.

2 Background

This paper adopts the terminology of generic programming established by Stepanov and Austern [17]. In particular, the requirements on the type parameters to generic algorithms are described using *concepts*. A concept describes a set of abstractions (usually types) via a set of requirements that the abstractions must satisfy. The kinds of requirements in a concept are *valid expressions* (or function signatures), *associated types*, *semantic constraints*, and *complexity guarantees*. 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.

2.1 Algorithm specialization

Libraries of generic algorithms often include several implementations of the same algorithm, all providing the same functionality. Such designs emerge due to a key idea behind generic programming [18]:

> Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.
> When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.

The selection of the most efficient specialized form is based on the properties of the inputs to the algorithm. In the most general case, the selection can be based on values not known until at run-time. However, this paper discusses the case where the selection is based on the types of the inputs, that is, types bound to the type parameters of a generic algorithm. We are focusing on properties that can be expressed in the type system of the language and decided statically.

Intuitively, placing more requirements on the type parameters can allow for a more efficient implementation. Each algorithm specialization must, in its interface, express the requirements that it places on the input types. A generic library must then arrange that these requirements are checked for each invocation of a generic algorithm, and select the most specialized form that satisfies the requirements. This is the essence of *concept-based overloading*. C++ allows concept-based overloading to be emulated with (at least) two different techniques: *tag dispatching* and *enable_if* [19, 20], described in Sections 2.2 and 2.3.

In the rest of the paper we use a running example of specializing an algorithm that performs a simple operation on an *iterator*. The STL and C++ standard library specify several iterator concepts, such as Input Iterator, Bidirectional Iterator, and Random Access Iterator. The refinement relations induce a hierarchy of iterator concepts. In particular, any type that meets the requirements of Random Access Iterator also meets the requirements of Bidirectional Iterator, and any type that meets the requirements of Bidirectional Iterator also meets the requirements of Input Iterator [3].

2.2 Tag dispatching

In tag dispatching, a special *tag class*, which usually has no members, is associated with each concept. For example the C++ Standard Library defines tag classes for each iterator concept (*forward_iterator_tag*, *bidirectional_iterator_tag*, etc.). Each type that models one of the iterator concepts is required to have an associated type *iterator_category* that gives the tag class corresponding to the concept. The iterator category can be accessed with via the *iterator_traits* traits class [21]. The refinement hierarchy between iterator concepts is expressed as the inheritance relation of the tag classes. Concept-based overloading can thus be emulated by overloading functions on the tag classes.

Figure 1 shows an example of algorithm selection using tag dispatching. The example follows roughly the C++ standard library implementation of the GCC 3.3 release. The *advance* function takes two parameters, an iterator *i* and a number of steps *n*, and moves *i* forward by *n* steps (backward if *n* is negative and the iterator supports backward motion). The *advance* function in the C++ standard library parameterizes the type of the step parameter; for less clutter, we fix this type to be ***int*** here.

The body of *advance* examines the most refined concept that the iterator argument type models (by querying its tag class using *iterator_traits*), and dispatches the work of advancing the iterator to one of the overloads of an internal library function *__advance_impl*. For types that are models of the Input Iterator concepts, the operation takes time linear in the number of steps *n*; for types that model Random Access Iterator, on the other hand, *advance* is a constant time operation.

This particular example of algorithm specialization has a peculiar characteristic, or frankly, a design flaw. The interface specification of the *advance* function sets the minimal requirements that the arguments must satisfy for the algorithm to work: the iterator type must model the Input Iterator concept, and the number of steps must be non-negative. However, the *advance* function, deliberately, works correctly with negative values for the step argument, as long as the iterator argument type is a model of Bidirectional Iterator. Hence, a more specialized form of an algorithm places more requirements on one input argument, but loosens the requirements on another. We do

```
template <class InputIterator>
void __advance_impl(InputIterator& i, int n, input_iterator_tag) {
  while (n--) ++i;
}

template <class BidirectionalIterator>
void __advance_impl(BidirectionalIterator& i, int n, bidirectional_iterator_tag) {
  if (n > 0) while (n--) ++i; else while (n++) --i;
}

template <class RandomAccessIterator>
void __advance_impl(RandomAccessIterator& i, int n, random_access_iterator_tag) {
  i += n;
}

template <class InputIterator>
void advance(InputIterator& i, int n) {
  __advance_impl(i, n, iterator_traits<InputIterator>::iterator_category());
}
```

Fig. 1. Tag dispatching in the STL

not give more attention to this issue here, but mention it so as not to advocate such incomparable preconditions between specializations as good design.

2.3 enable_if

A technique known as *enable_if* [19, 20] is another mechanism to implement algorithm specialization in C++. The technique allows one to define predicates that enable or disable function template overloads based on the properties of their template arguments. The technique takes advantage of the fact that certain types of failures during template instantiation are not compilation errors, but rather lead to removal of the failing function from the set of candidate overloads. One of these types of failures is an attempt to refer to a member type that does not exist. This type of failure can be encapsulated in a template class, for which an implementation is readily available in the C++ Boost library collection [22]. In Boost, the template name is *enable_if_c*, and it is defined as follows:

```
template <bool Cond, class T> struct enable_if_c { typedef T type; };
template <class T> struct enable_if_c<false, T> {};
```

The *enable_if_c* class template is a partial identity function from types to types: the type denoted by the expression *enable_if*<*C*, *T*>::**type** is *T* if *C* is **true**. If *C* is **false**, the expression attempts to refer to a nonexistent member type and thus does not denote any type.

The enabling predicate is placed either in the return type expression of the function template, or as an extra defaulted argument. In the former case, a function template with the return type *T* is rewritten as **typename** *enable_if_c*<*C*, *T*>::**type**, where *C* is

a compile-time predicate that examines properties of the template arguments. We can harness the predicate to test whether a template argument type is a model of a particular concept, and thus attain a simple form of concept-based overloading. Note that, similar to tag dispatching, the test of whether a type models a concept is based on nominal conformance. C++ offers no direct means to express the models relation, hence, the relation is expressed with library mechanisms: traits classes and tag classes. Any kind of structural conformance checking to concepts needs to rely on techniques such as *concept checking* [8].

Figure 2 demonstrates how the *advance* functions can be overloaded for different iterator concepts. The traits classes *is_input_iterator*, *is_bidirectional_iterator*, and *is_random_access_iterator* are predicates that indicate whether a type models particular iterator concepts. For example, *is_input_iterator<T>::value* is ***true*** if the iterator category of *T* is *input_iterator_tag*, or a tag class that inherits from *input_iterator_tag*. Appendix A shows the definitions of these helper traits classes.

```
template <class T>
typename enable_if_c<is_input_iterator<T>::value &&
                     !is_bidirectional_iterator<T>::value, void>::type
advance(T& i, int n) {
  while (n--) ++i;
}

template <class T>
typename enable_if_c<is_bidirectional_iterator<T>::value &&
                     !is_random_access_iterator<T>::value, void>::type
advance(T& i, int n) {
  if (n > 0) while (n--) ++i; else while (n++) --i;
}

template <class T>
typename enable_if_c<is_random_access_iterator<T>::value, void>::type
advance(T& i, int n) {
  i += n;
}
```

Fig. 2. Algorithm specialization using *enable_if*

There are notable differences between the tag dispatching approach and the approach based on *enable_if_c*. In tag dispatching, a single function serves as the entry point and dispatcher, whereas with *enable_if_c* the dispatcher function is not necessary. Adding a new specialization in tag dispatching means adding a new, potentially internal, implementation function. However, the parameter passing modes and the way the return type of the specialization is expressed is fixed by the dispatcher function. In the *enable_if_c* approach, specializations are added to the algorithm itself, not its implementation function. The conditions do not define any specialization ordering; they only enable or disable function templates. Unless some argument, or arguments, defines an

ordering (for overloading purposes) between two functions, it is ambiguous (and thus an error) to have more than one overload enabled simultaneously. Thus, one must arrange for the conditions to be mutually exclusive.

2.4 Type information propagation

C++ templates are compiled by generating a different entity for each template use. This is not a strict requirement in the standard, but more of an assumption that underlies the language rules concerning templates [23, page 11]. What the standard requires, however, is that full type information is retained in instantiated templates. To explain what we mean by this, consider the template function in Figure 3. The calls to *advance* are to the functions in Figure 2. It is not, however, relevant to the discussion which set of *advance* functions is used, the one written using *enable_if_c* or the one in Figure 1 using tag dispatching.

```
template <class InputIterator1, class IntputIterator2>
void pair_advance(std::pair<InputIterator1, InputIterator2>& p, int n) {
  advance(p.first, n);
  advance(p.second, n);
}
...
std::pair<std::vector<int>::iterator>, std::list<int>::iterator> p = ...;
pair_advance(p, 4);
```

Fig. 3. A generic function calling another generic function.

The code in Figure 3 creates a pair *p* of iterators, where the first element of *p* is an iterator iterating over a vector, and the second is an iterator iterating over a list. The type of the former is a model of Random Access Iterator, whereas that of the latter only a model of an Input Iterator. This information is available while compiling *pair_advance*, and as a result, the *advance* call with the *p.first* argument is dispatched to the overload with the predicate that requires the iterator type *T* to model Random Access Iterator. The call with the *p.second* argument is dispatched to the overload requiring *T* to only model Input Iterator.

The minimal requirements of the *pair_advance* function are that the types of the elements of the pair argument are models of the Input Iterator concept. However, more than this information is clearly used to enable the dispatching described above. Exact type information is available in the C++ compilation model, where a separate entity is generated for each different use of a template. This makes algorithm specialization straightforward.

3 Concepts as constraints

Stroustrup and dos Reis outlined syntax for expressing constraints for template parameters of generic functions and classes in C++ [10–12]. This paper deviates from the proposed syntax in favor of a more explicit presentation of constraints. The keyword ***where*** attaches a constraint, or a set of constraints, to a generic function or class. A constraint of the form:

a_concept<T1, T2, ..., TN>

is interpreted to mean that type arguments bound to *T1, T2, . . ., TN*, must collectively be a model of *a_concept*. In all examples in this paper *N* is 1; *multi-parameter* concepts do not seem to add new insights to the aspects of algorithm specialization discussed. Using the above syntax to express constraints the *advance* function for, say, Input Iterators can be written as:

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

To describe the concepts themselves, such as Input Iterator, Stroustrup and dos Reis propose a new language construct for naming a group of valid expressions or, alternatively, function signature constraints. For Input Iterator, the *++* operator, dereferencing operator, and so forth would be listed as constraints.

All in all, this may seem like a minor syntactic deviation from constraints expressed using *enable_if*. The difference is, however, that constraints expressed as concepts are suggested to provide separate type checking. Furthermore, such a language feature easily steers us to think of exploiting constraints for separate compilation as well.

Several forms of constrained genericity are in use in mainstream programming languages. Object-oriented languages with support, or proposed support, for generics, such as Java, C#, and Eiffel, implement variations of *F-bounded polymorphism* [24]. Haskell, a modern functional language, uses *type classes* [25] as the constraint mechanism for polymorphic functions. ML has parameterized modules, called functors, whose parameters are constrained by *signatures*. Other approaches include *where clauses* in CLU [26].

The two goals of constrained genericity, separate type checking and separate compilation of generic functions, frequently go hand in hand. For example, Generic Java and Eiffel both compile each generic definition once into a single unit of byte code or executable code. Separate type checking and separate compilation are, however, two different issues.

Separate type checking. In C++, the interface of a template places no constraints on the types with which a template can be instantiated. After the instantiation has be performed, the compiler type checks the instantiated body of the function or class template. In constrained genericity, however, constraints on generic parameters define an interface against which the body of a generic function can be type checked. Separate type checking ensures that the body does not rely on any properties of the generic parameters that are not explicitly stated using the constraints. Thus no type error can occur in the

body of a generic algorithm if the arguments that are bound to the generic parameters satisfy the requirements expressed in the interface. Naturally, any call to a generic algorithm that leads to an instantiation not satisfying the constraints is rejected by the type checker. Separate type checking can thus help to detect specification and program errors in a generic library earlier, before any instantiations of the generic algorithms, or other generic components, have been made.

Separate compilation. Generic functions can be compiled in two essentially different ways, which we refer to as the *dictionary passing* and *instantiation* model. The dictionary passing model produces a single copy of a generic code. Thus, any variability of the code must be converted into run-time parameterization. The basic approach is for the compiler to generate a collection of pointers to all functions that depend on the generic parameters, and arrange that to be passed to the generic function as an extra parameter. The constraints on type parameters of generic algorithms determine the contents of such collections. In object-oriented languages a *virtual function table* is commonly the mechanism for representing the function pointer collection; work on Haskell type classes talks about *function dictionaries* [25]. We refer to the above described general mechanism as *dictionary passing*. The effect of dictionary passing is that calls to dependent functions are not statically resolved, but require some form of indirection. Another implication of translating each generic function into a single universal representation is that the size allocated for each local variable or parameter on the stack, cannot vary. *Boxing* is often used to make all local variables and arguments be of the same size. This topic is discussed more thoroughly in [16].

In instantiation model a different instance of code for each distinct use of a template is generated. After generation, compilation does not differ from compiling non-generic code; exact type information is available.

Compared to the instantiation model, separate compilation of generic entities is often believed to provide some or all of the following benefits: shorter incremental compile times, the ability to distribute generic libraries as binaries, the ability to use generic code in shared libraries, and smaller executable sizes. An often-cited benefit of the instantiation model is better performance, due to the ability to statically resolve calls to functions that depend on generic parameters. These tradeoffs are not straightforward. For example, Jones examined [27] several real-world Haskell programs and found that the instantiation model produced smaller executables in all of his test cases. As another example, just in time compilers and various runtime instantiation strategies, as applied for example in .NET generics [28], can even out the difference in performance.

Different types of modules Several different forces affect how software is, or should be, broken into smaller parts. A closer look at these forces clarifies why the distinction between separate type checking and separate compilation is important. Programmers divide code into subroutines, classes, modules, components etc. in order to manage complexity. Such divisions are also needed to manage complexity for tools. For example, compilation units allow local changes without the need to recompile the entire program, shared libraries and plug-in components enable incremental changes to entire systems. Essential for dividing software into smaller entities are interface specifications

between the entities. Interfaces do not, however, need to be the same for tools and programmers. For example, to better optimize programs, a compiler can break interfaces that the programmer must respect. Function inlining is a typical example of such interface breaking; the compiler is allowed to look through the interface of a subroutine abstraction, not replicate the interface in the executable code, and perform transformations on the body of the subroutine. Any interface breaking must obviously preserve the behavior of the program.

Regarding interfaces of generic functions, one can view C++ style unconstrained genericity as interface breaking by default, even when the interface should not be broken, whereas full separate compilation of constrained genericity implemented using dictionary passing prohibits all interface breaking. In constrained genericity, if the definition of a generic function is accessible to the compiler, it may be beneficial to allow the compiler to optimize away the dictionaries and generate a specialized version of the generic function; separate compilation is intentionally not used, but separate type checking is still desired.

3.1 Algorithm specialization and constrained genericity

Constraints on generic parameters can serve as the criteria for function overloading. The *enable_if* technique provides a simple form of such overloading: functions whose constraints are not satisfied are ruled out from overloading. This is roughly the behavior in C# generics too. Stroustrup suggests, however, that constraints affect the specialization ordering of overloads via *concept inheritance*, a form of concept refinement [10]. With such a feature, algorithm specialization becomes concept-based overloading. Figure 4 repeats the example of three *advance* specializations, now overloaded for three different concepts (in pseudo-code).

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

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

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

Fig. 4. Algorithm specialization using concept-based overloading

It should be easy to see that such overloading on constraints can provide algorithm specialization capabilities. For example, consider the following invocations of *advance*:

```
std::vector<int>::iterator r = ...;
std::list<int>::iterator i = ...;

advance(r, 4);
advance(i, 4);
```

The variable *r* is an iterator over a vector and thus its type is a model of Random Access Iterator. The type of *i* is a model of Input Iterator. The first *advance* call, with *r* as the iterator argument, is therefore resolved to the third, most specialized, overload in Figure 4. The second *advance* call with *i* as an argument is dispatched to the least specialized overload.

Regarding type checking, it is clear that the generic *advance* functions can be type checked separately. Furthermore, overload resolution of the calls to advance introduce no complication. The situation, however, changes when we expect algorithm specialization in calls from within generic functions, where the arguments whose types determine the specialization themselves have generic types; it is not instantly clear how much type information the compiler should use to select which *advance* overload is called. We use a constrained version of the *pair_advance* function in Figure 5 as our example to explain.

```
template <class T, class U>
void pair_advance(std::pair<T, U>& p, int n)
  where InputIterator<T>, InputIterator<U> {
    advance(p.first, n);
    advance(p.second, n);
}
...
std::pair<std::vector<int>::iterator>, std::list<int>::iterator> p = ...;
pair_advance(p, 4);
```

Fig. 5. Algorithm specialization with constrained genericity.

We observe first that the type parameter constraints of *pair_advance* are adequate to allow its body to be type checked — both types, *T* and *U*, are required to model Input Iterator, which is enough to guarantee that the calls to *advance* will succeed. The invocation to *pair_advance* type checks too; the type of the first element of the pair argument models Random Access Iterator (and thus also Input Iterator), and the second also models Input Iterator. There are, however, two different approaches to overload resolution for the *advance* calls.

1. Overload resolution is based on the type information defined by the constraints on generic parameters. The constraints state that the types of both the first and second arguments are models of Input Iterator, and thus both *advance* invocations call the least specialized overload. The knowledge that the type of the first element of the pair actually models Random Access Iterator is not used.

2. Overload resolution is based on the exact types. The first advance invocation selects the most specialized overload based on the fact that the type bound to *T* is actually a model of Random Access Iterator.

In what follows, we refer to the former approach as the *static overloading model* and the latter as the *dynamic overloading model*. The constrained genericity models of Generic Java and C# are examples of systems following the static overloading model, whereas C++ templates follow the dynamic model. Selection of the virtual function to call based on the ***this*** parameter's run-time type is an example of the dynamic overloading model applied at run-time. Note that relative to the run-time types of objects, ***this*** parameter is an exception: the static overloading model is applied to all other arguments that are references or pointers to class types.

3.2 Dynamic overloading and dictionary passing

Dynamic overloading and dictionary passing as an implementation technology do not mix well. The dictionaries that are passed into generic functions contain exactly the functions that are required in the descriptions of the required concepts. The only constraints of the *pair_advance* function in Figure 5 are that *T* and *U* model the *Input Iterator* concept. The *advance* function is not a requirement in any concept description, and thus not part of any dictionary passed into *pair_advance*. If the same *advance* function is not called with all instances of *pair_advance*, some kind of run-time overload resolution would be necessary. Note also that the *advance* function for Random Access Iterators relies on the existence of functions that are not in the Input Iterator dictionary. The compiler would need to arrange the additional dictionary elements to be forwarded to the the right overloads.

One can think of two ways to design around the above obstacle. First, it is possible to add *advance* into the iterator concepts as one of the requirements. We can identify the following problems with that approach:

- 'Fat' interfaces. Every algorithm that, now or in the future, is potentially specialized and can be called from generic functions that use a particular set of concept constraints must be lifted to be a constraint in one of the concepts. This applies even to functions that are merely optimizations and 'internal' to the library.
- An unanticipated need for algorithm specialization requires changes to concepts and thus constraints. Such changes change the interface, and the dictionaries, and likely invalidate the binary interfaces of functions and necessitates thus recompilation. Again, this is true even if the specialization is an internal library optimization.

Second, we may overload *pair_advance* for different iterator categories, that is, move constraints from *advance* up to *pair_advance*. It is easy to see why this approach does not scale. To cover all cases of *pair_advance*, nine overloads would be needed: one for each combination of requirements where both *T* and *U* are independently required to model one of Input Iterator, Bidirectional Iterator, or Random Access Iterator. Note that all nine overloads would have the exact same function body.

To avoid the combinatorial growth of overloads, one can think of two categories of constraints: ones that require properties that are necessary, and others that require

properties that are not necessary but can possibly be taken advantage of. We call such constraints *optional*. Figure 6 illustrates optional constraints in a pseudo-code definition of *pair_advance*. Siek et al. describe an implementation of statically checked optional constraints in their research language [16]. We are not aware of any established programming languages with this feature, though one can view run-time downcasts of OO languages as an approximation. We present optional constraints here as one path worth exploring in the design space of constrained generics. At the moment, we have no experimental evidence on real designs exploiting optional constraints. We, however, identify the following concerns:

- In a deep call chain, all optional constraints that are necessary to allow algorithm specialization on any level of the chain percolate to the outermost function of the call chain. Siek et al. [16] analyze the call graph of C++ standard library algorithm *inplace_merge*, demonstrating that constraints need to be propagated four levels up.
- Increased size of dictionaries.
- To enable new opportunities for algorithm specialization, optional constraints must be changed. It is not clear to what extent such changes would trigger recompilation of call sites to changed functions.

```
template <class T, class U>
void pair_advance(std::pair<T, U>& p, int n)
where InputIterator<T>, InputIterator<U>,
    optional BidirectionalIterator<T>, optional BidirectionalIterator<U>,
    optional RandomAccessIterator<T>, optional RandomAccessIterator<T> {
  advance(p.first, n);
  advance(p.second, n);
}
```

Fig. 6. Optional constraints.

3.3 Dynamic overloading and instantiation model

Compared to dictionary passing, the instantiation model is a more natural fit for implementing dynamic overloading. The use of algorithm specialization in C++ template libraries demonstrates its applicability. It is an open question, however, whether separate type checking can be combined with the dynamic overloading model. Type checking a generic function must assure that no instantiation with types that conform to the constraints of the generic parameters can trigger a type error in the body of the function. This section describes two particular problems in attaining this goal.

Incompatible return types Consider the modified versions of the *advance* specializations for Input Iterator and Random Access Iterator, and the *pair_advance* function, shown

in Figure 7. In the Input Iterator case, instead of a ***void*** return, the *advance* function now returns the advanced iterator. The *pair_advance* function then uses the returned value as the right hand side of the assignment. During type checking the implementation of *pair_advance*, the constraints guarantee that *T* and *U* both model the Input Iterator concept. Thus, the calls to *advance* type check, as its requirements are clearly met. The value returned by *advance* is used in an assignment, which is fine as the return type of *advance* is the same as the type of the variable to which the value is being assigned. However, at instantiation time, assuming the *pair_advance* call from our earlier example

```
std::pair<std::vector<int>::iterator, std::list<int>::iterator> p = ...;
pair_advance(p, 4);
```

the member *p.first* is a model of Random Access Iterator. Thus, with C++ overloading rules, the latter call to *advance* in Figure 7 would be resolved to the second *advance* definition. This would lead to a type error in the body of *pair_advance*: an attempt to assign ***void*** to a variable.

```
template <class T>
T& advance(T& i, int n) where InputIterator<T> {
  while (n--) ++i;
  return i;
}

template <class T>
void advance(T& i, int n) where RandomAccessIterator<T> {
  i += n;
}

template <class T, class U>
void pair_advance(std::pair<T, U>& p, int n)
  where InputIterator<T>, InputIterator<U> {
    p.first = advance(p.first, n);
    p.second = advance(p.second, n);
}
```

Fig. 7. Algorithm specializations with conflicting return types.

A necessary but not sufficient requirement for type checking a generic function within the dynamic overloading model is that the function type checks using the static overloading model. This guarantees that for each function call in a generic function there must exist a least specialized unambiguous matching overload. We call this the *primary overload*. In the case of *pair_advance*, the primary overload for both calls to *advance* is the *advance* function requiring its type argument to model Input Iterator. For any valid instantiation of a generic function, the requirements of any primary overload of a function that the generic function invokes are guaranteed to be satisfied; no overload with looser constraints than the primary overload can thus ever be the most specialized overload.

To guarantee that instantiation does not cause type errors due to incompatible return types, one can think of (at least) three approaches:

1. Require the return types to be the same between any two overloads where one is more specialized than another. The return type expression of a function can, however, be a traits class instantiation which is specialized differently for different types. For example, the return type expression of all *advance* specializations could be ***typename*** *advance_return*<*T*>*::**type***, where the *advance_return* class was defined as:

   template <***class*** *T*> ***struct*** *advance_return* { ***typedef*** *T* ***type***; };

   It is possible that a subsequent specialization defines the member typedef ***type*** differently (or omits it altogether), making it impossible to guarantee that the return types are conforming — unless all specializations are known.
   One can think of less strict requirements than sameness on the return types, for example requiring assignability from the result type of *advance* to the type of the first argument in the example of Figure 7. However, class template specializations can invalidate such requirements as well.
   Note that even if all specialization are known, in the absence of instantiation limits C++ templates are a Turing complete language [29]. The construction in [29] demonstrates that the problem of determining whether an arbitrary C++ type has some given non-trivial property is undecidable.
2. Only consider the overloads and template specializations that are visible at the point of definition of a generic algorithm and ignore others. In such an approach, a generic function is type checked not only against its minimal requirements, but also against constraints that the input types could potentially satisfy. This is comparable to optional constraints, with the distinction that the compiler determines what the useful optional constraints are. At this point we cannot provide an account of the feasibility of this approach.
3. Make incompatible returns behave according to the *SFINAE* principle, described in [23], where invalid function templates are silently ignored without producing errors. Type checking according to the static overloading model can guarantee the existence of all primary overloads. The expectations on the return type are based on this primary overload. At instantiation time, the compiler can remove a particular overload from the set of candidate functions if the return type of that overload is not compatible with the primary overload. Regarding implementation, we see no major complications.
   This approach could delay the detection of certain class of errors; an erroneous return type in a specialization of a generic algorithm would not trigger a compilation error. A function with such a return type would simply never be executed, postponing the detection of the error and leaving it to the responsibility of run-time testing. Even in the testing phase, the error would only be detectable as a lower than expected performance.

Ambiguous specializations Successfully type checking a generic function according to the static overloading model guarantees the existence of an unambiguous primary

overload for each function invocation in the generic function. However, there is no guarantee of an unambiguous best match when full type information available. Figure 8 gives an example of ambiguous specializations. To stay within the iterator hierarchy we resort to somewhat artificial example. Trivial Iterator is the least refined concept in the iterator concept hierarchy. Both Input Iterator and Output Iterator are refinements of Trivial Iterator. The *bar* function requires the type of its argument *x* to model Trivial Iterator. The *foo* invocation inside *bar* type checks with static overloading model, guaranteed by the least specialized overload of *foo* for Trivial Iterators. The problematic case is when *bar* is instantiated with a type that models both Input Iterator and Output Iterator; there are two overloads that match equally well.

```
template <class T> void foo(T& x) where TrivialIterator<T> { ... }
template <class T> void foo(T& x) where InputIterator<T> { ... }
template <class T> void foo(T& x) where OutputIterator<T> { ... }

template <class T>
void bar(T& x) where TrivialIterator<T> {
  foo(x);
}
```

Fig. 8. Ambiguous specializations.

There are again several possibilities to cope with this issue:

1. Forbid such overloading situations. It seems that it would be difficult for the compiler to detect ambiguities prior to instantiation. A potential ambiguity cannot be reported prior to it being guaranteed that no tie-breaking overload is available, which means that the error can at the earliest be detected at the end of compiling a module. Moreover, in the current C++ type system there are no means to express that a type that *does not* satisfy a certain set of constraints cannot be defined. Neither is anything of this sort part of the concept proposals [10–12], or in any type system we know. Thus, a new type that models two concepts that cause an ambiguity can be defined in a module not available when a generic function is type checked. As a consequence, every pair of overloads where one is not clearly more specialized than the other, and which do not have conflicting concrete parameter types to guarantee that they can never match the same argument list, would have to be reported as a possible ambiguity.
2. Resort to the primary overload whose existence is always guaranteed. This is a viable option, but may not be the most desirable one. Assuming the library user relies on the more specialized overloads providing better efficiency, no compile-time warning or error would be provided that this benefit was not be delivered.
3. Pick either overload arbitrarily. Either of the more specialized overloads is assumed to be an improvement over the primary overload, so it would always be beneficial to favor either of them over the primary overload. Again, however, the library author,

or user, may be relying on one in particular being called, and does not realize that in fact another overload is being invoked instead.

4 Conclusions

During the recent years, C++ has been the main vehicle for implementing generic programming libraries. This is despite some known shortcomings of the language, most prominently, the lack of constraints on template parameters. Improving support for generic programming is one major criterium in guiding the standards committee on evolving C++, and indeed, work is under way to add concept-based constraints on templates. One of the main motivations for the work is to allow generic functions and classes to be type checked separately, in isolation of their uses, or instantiations.

Algorithm specialization is an important idea within generic programming, frequently used, for example, in the C++ standard library. A generic library can contain several versions of the same algorithm. Depending on the capabilities of the inputs (usually input types) to a generic algorithm, a generic library is designed to dispatch the work to the most specialized, and most efficient or otherwise best, algorithm out of many possiblities. It seems that algorithm specialization is coupled with C++'s unconstrained genericity, or at least, unconstrained genericity places no restrictions on implementing algorithm specializations. Other programming languages, such as C#, Generic Java, Eiffel, and Haskell, that support constrained genericity, do not as directly support algorithm specialization.

This paper describes combining separate type checking of generic algorithms and algorithm specialization, which proves to be difficult. In particular, we examine the C++ overloading mechanisms, extended with concept-based overloading, and describe two cases where separate type checking of generic algorithms is not possible with the current overloading rules of C++. The paper outlines several design choices for a type system that would prevent such instantiation time errors, discusses their feasibility, and describes drawbacks for these approaches. We have not found a fully satisfactory solution. The simplest solution is to allow instantiation time errors to occur; constraints on generic parameters would thus not be bullet-proof and we would give up full separate type checking of generic algroithms. It remains to be evaluated how frequent such instantiation time errors would be, and how difficult they would be to diagnose and correct. The cure to instantiation time type errors, restrictions to the type systems, may prove to be more harmful than the illness itself.

5 Acknowledgements

This work was supported by NSF grant EIA-0131354 and a grant from the Lilly Endowment. The second author was supported by a Department of Energy High Performance Computer Science Fellowship. We thank Jeremy Siek for his constructive comments on this work, and the discussions that lead to some of the insights described in the paper. Our thanks also to Dave Abrahams for helpful discussions.

References

1. Stepanov, A.: The Standard Template Library — how do you build an algorithm that is both generic and efficient? Byte Magazine **20** (1995)
2. Stepanov, A., Lee, M.: The Standard Template Library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories (1994) `http://www.hpl.hp.com/techreports`.
3. International Standardization Organization (ISO): ANSI/ISO Standard 14882, Programming Language C++, 1 rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland (1998)
4. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley (2002)
5. Siek, J., Lumsdaine, A.: A modern framework for portable high performance numerical linear algebra. In: Modern Software Tools for Scientific Computing. Birkhäuser (1999)
6. Walter, J., Koch, M.: The Boost uBLAS Library. Boost. (2002) `www.boost.org/libs/numeric`.
7. McNamara, B., Smaragdakis, Y.: Static interfaces in C++. In: First Workshop on C++ Template Programming. (2000)
8. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Programming. (2000)
9. Zolman, L.: An STL error message decryptor for visual C++. C/C++ Users Journal (2001)
10. Stroustrup, B.: 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++ (2003) `http://anubis.dkuug.dk/jtc1/sc22/wg21`.
11. Stroustrup, B., Dos Reis, G.: Concepts – design choices for template argument checking. Technical Report N1522=03-0105, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003) `http://anubis.dkuug.dk/jtc1/sc22/wg21`.
12. Stroustrup, B., Dos Reis, G.: Concepts – syntax and composition. Technical Report N1536=03-0119, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003) `http://anubis.dkuug.dk/jtc1/sc22/wg21`.
13. Technical report on C++ performance. Technical Report N1487=03-0070, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003)
14. Müller, M.: Abstraction benchmarks and performance of C++ applications. In: Proceedings of the International Conference on Supercomputing. (2000)
15. Robison, A.D.: The abstraction penalty for small objects in C++. In: POOMA '96: The Parallel Object-Oriented Methods and Applicatio ns Conference. (1996)
16. Siek, J., Lumsdaine, A.: Modular generics. In: Concepts: a Linguistic Foundation of Generic Programming, Adobe Systems (2004)
17. Austern, M.H.: Generic Programming and the STL. Professional computing series. Addison-Wesley (1999)
18. Jazayeri, M., Loos, R., Musser, D., Stepanov, A.: Generic Programming. In: Report of the Dagstuhl Seminar on Generic Programming, Schloss Dagstuhl, Germany (1998)
19. Järvi, J., Willcock, J., Lumsdaine, A.: Concept-controlled polymorphism. In Pfennig, F., Smaragdakis, Y., eds.: Generative Programming and Component Engineering. Volume 2830 of LNCS., Springer Verlag (2003) 228–244
20. Järvi, J., Willcock, J., Hinnant, H., Lumsdaine, A.: Function overloading based on arbitrary properties of types. C/C++ Users Journal **21** (2003) 25–32
21. Myers, N.C.: Traits: a new and useful template technique. C++ Report (1995)
22. Boost: (Boost C++ Libraries) `http://www.boost.org/`.
23. Vandevoorde, D., Josuttis, N.M.: C++ Templates: The Complete Guide. Addison-Wesley (2002)

24. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: Proceedings of the fourth international conference on functional programming languages and computer architecture. (1989)
25. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: ACM Symposium on Principles of Programming Languages, ACM (1989) 60–76
26. Liskov, B., Snyder, A., Atkinson, R., Schaffert, C.: Abstraction mechanisms in CLU. Communications of the ACM **20** (1977) 564–576
27. Jones, M.P.: Partial evaluation for dictionary-free overloading. Technical Report YALEU/DCS/RR-959 (1993)
28. Kennedy, A., Syme, D.: Design and implementation of generics for the .NET Common Language Runtime. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah (2001) 1–12
29. Veldhuizen, T.L.: C++ templates are Turing complete. `www.osl.iu.edu/~tveldhui/papers/2003/turing.pdf` (2003)

A Definitions for overloading advance with `enable_if`

```
#include <iterator>
#include <boost/type_traits.hpp>

template <class B, class D> struct is_base_or_same {
  static const bool value =
    boost::is_same<B, D>::value || boost::is_base_and_derived<B, D>::value;
};

template <class T>
struct is_input_iterator {
  static const bool value =
    is_base_or_same<std::input_iterator_tag,
                    typename std::iterator_traits<T>::iterator_category>::value;
};

template <class T>
struct is_bidirectional_iterator {
  static const bool value =
    is_base_or_same<std::bidirectional_iterator_tag,
                    typename std::iterator_traits<T>::iterator_category>::value;
};

template <class T>
struct is_input_iterator {
  static const bool value =
    is_base_or_same<std::random_access_iterator_tag,
                    typename std::iterator_traits<T>::iterator_category>::value;
};
```