Specifying C++ Concepts

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

Bjarne Stroustrup
Texas A&M University
and AT&T Labs — Research
bs@cs.tamu.edu

Abstract

C++ templates are key to the design of current successful mainstream libraries and systems. They are the basis of programming techniques in diverse areas ranging from conventional general-purpose programming to software for safety-critical embedded systems. Current work on improving templates focuses on the notion of *concepts* (a type system for templates), which promises significantly improved error diagnostics and increased expressive power such as concept-based overloading and function template partial specialization. This paper presents C++ templates with an emphasis on problems related to separate compilation. We consider the problem of how to express concepts in a precise way that is simple enough to be usable by ordinary programmers. In doing so, we expose a few weakness of the current specification of the C++ standard library and suggest a far more precise and complete specification. We also present a systematic way of translating our proposed concept definitions, based on use-patterns rather than function signatures, into constraint sets that can serve as convenient basis for concept checking in a compiler.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.3 [*Logic and Meanings of Programs*]: Studies of Program Constructs

General Terms Languages, Design, Performance

Keywords C++ Templates, C++ Concepts, Type Systems, Separate Compilation, Generic Programming

1. Introduction and overview

ISO Standard C++ [ISO03, Str00] directly supports generic programming through the notion of *template*. Templates are essential in C++ for capturing commonalities of abstractions while retaining optimal performance. Those properties are key to the success and the wide acceptance of the Standard Template Library [SL94]. Templates have also been used to reduce abstraction penalties and code bloat in embedded systems to an extent that is impractical in conventional object-oriented systems [Str04]. There are two key reasons for that. First, template instantiation combine information from both definition and use contexts (§2.4). This means that full information from both calling and called contexts (including full type information) is made available to the code generator. Current code generators are adept at using this information to minimize run time and/or code space. This contrasts to the usual case in object-oriented programming languages where the caller and the callee are completely separated through an interface relying on indirect function calls. Second, a C++ template is typically implicitly instantiated if and only if it is used in a way that is essential to the program semantics, automatically minimizing the footprint of an application (§2.4). This contrasts to systems that require the programmer to explicitly manage instantiation, such as Ada [TDBP00] or System F [Gir72, Rey74].

The work described here is part of an effort to design a type system — called *concepts* — for C++ types and values that can be used for template arguments as currently successfully used. It is a critical design concern for "concepts" that they should be usable by programmers who currently successfully use templates [SDR05a]. In some form or other, concepts will be part of the revised ISO C++ standard, C++0x [Str05]. This paper focuses on a specific problem: How to express concept definitions in a way that is sufficiently simple and flexible to be used, yet precise enough to be implementable in current C++ compilers [DRS05a]. Our notation for concept definition is based on "use patterns" that can be translated into a set of operation signatures suitable for type checking. Basically, concepts are compile-time predicates on types and values (e.g., integral constant values). They can be combined with the usual logical operators (*and*, *or*, *not*.)

This paper discusses concepts for C++ templates. However, the fundamental ideas generalize to a type system that supports parametric polymorphism, supports some forms of local type inference, and extends the notion of dependent names (§2.2). Our contributions include the development of a formal framework for specifying concepts, clarifications of the C++ standard iterator library requirements, and a precise formulation of iterator concepts. Concepts differ from type classes [WB89, HHPJW96] which also act as predicates over types. In particular, concepts cope with general overloading, including use of the same name for functions of different number of arguments and operations with argument types that are not generic instances of each other (§6).

The remaining of the paper is structured as follows. We

1. examine the fundamental properties of templates, the problems caused by those (§2) and the requirements on possible solutions (§2.5);

2. introduce concepts (§3) based on "use patterns" and an algorithm for generating constraints sets for template arguments; apply it to simplified examples (§4);
3. apply our concepts to a known hard problem of significant practical importance: The specification of the lowest levels of the C++ standard iterator library, exposing some weakness (§5);
4. survey recent related works (§6) and briefly explain the differences between type classes and concepts;
5. outline several directions for future work and conclude (§7).

2. The problem

The near-optimal performance offered by ISO C++ templates comes at the price of very weak separation between template definitions and their uses. In C++ source code, the complete template definition is the only expression of the assumptions it makes about its parameters. However, it is clearly desirable to check a template definition independently of its uses, and to check the uses independently of the definition (see §2.3). To do that, we must concisely specify the assumptions separately from the code in the template definition. In short, we need a type system for template parameters. The holy grail of concept design for C++ is a system that allows for perfect separate checking of template definitions and uses, without loss of expressive power or performance. That is, if the definition of a template is sucessfully checked against the concepts specified for its parameters and if arguments specified in its uses are also successfully checked against those concepts, then the resulting instantiation will type-check and all information available will be used to generate optimal code. Please note that separate checking without optimal code generation is trivial: Just use some form of abstract class, as is often done explicitly in C++ and is the basis of the "generic" language facilities of Java and C#. However, the cost of doing so is to over-constrain solutions and to require unreasonable foresight of designers. In the absence of whole-program optimization (which is infeasible for many important C++ application domains), the abstract class approach also implies the use of indirect function calls as opposed to the inlining provided by templates.

2.1 Template basics

A template is a recipe from which a C++ translator generates declarations. For example, the program fragment

```
template<typename T>
T square(T x) { return x * x; }
```

declares a family of functions indexed by a type parameter. Like System F, a particular member of that family can be referred to by applying the template name `square` to a template-argument as in `square<int>`. We say that a template specialization is requested for `square` with template-argument list `<int>`. The process of creating a specialization is called template instantiation, so a specialization is also colloquially known as an instantiation. A C++ compiler would internally generate the moral equivalent of the function definition

```
int square(int x) { return x * x; }
```

where the type argument `int` is substituted for the type parameter `T`. The resulting code is type-checked to ensure that no type errors resulted from that substitution. Template instantiation is done only once for a given specialization even if the program contains multiple requests for that specialization.

Unlike programming languages like Ada or System F, template-argument list can be omitted from a function template instantiation request. Usually, the values for template-parameters are deduced. For example, given the definition above, the following is idiomatic C++:

```
double d = square(2.0);
```

The type-argument is deduced [Str00, ISO03] to be `double`. It worths noting that, unlike programming languages like Haskell [PJ03] or System F, C++ template parameters are not restricted to types. For example, the program fragment

```
template<typename T, size_t N>
struct buffer {
  T data[N];
};
```

declares a data type parameterized by an element type and a buffer-size, which can be any positive compile-time integral value.

2.2 Parameterization

Conceptually, the parameters of a template are specified in two ways:

1. *template-parameters* — explicitly mentioned as parameters in the template declaration; and
2. *dependent names* — inferred from the use of parameters in the definition of the template.

In C++, a name cannot be used without a prior declaration. This requires careful treatment of template definitions. For example, in the context of the definition of the template `square`, the symbol `*` has no visible declaration. However, in the instantiation context of `square<int>`, the compiler can resolve `*` to the (built-in) multiplication operator for `int` values. For a call `square(complex(2.0))`, `*` would be resolved to the (user-defined) multiplication operator for `complex` values. We say that the symbol `*` is a *dependent name* in the definition of `square`. That is, it is an implicit parameter of the template definition. Had we wanted to, we could have made the multiplication operations an explicit parameter like this:

```
template<typename Mul, typename T>
T square(T x) { return Mul()(x,x); }
```

Here, the sub-expression `Mul()` constructs a function object (see [Str00, Chapter 18]) that implements the multiplication operation over values of type `T`. The notion of dependent names helps keep the number of explicit parameters manageable.

2.3 Instantiations and checking

Only minimal semantics processing is done when a template definition or a template use is first seen. Full semantics processing is postponed to instantiation time (just before link time), on a *per instantiation* basis. That implies that assumptions made on template arguments are not checked before instantiation time. Consider

```
string hello = "Hello World";
square(hello);
```

The nonsensical use of a `string` as function-argument for `square` is not caught at the point of use. Only at instantiation time will the compiler discover that there is no suitable declaration for `*`. This is a huge practical problem because that instantiation may be caused by code written by a user

who wrote neither the definition of `square` nor the definition of `string`. A programmer who did not know the definition of `square` and `string` would have great difficulty understanding error messages relating to their interaction (such as "*illegal operand for* `*`").

The existence of an operator symbol `*` is not sufficient to ensure successful compilation for `square`. There must exist a `*` that takes arguments of the appropriate types and that `*` must be the unique best match under the C++ overload resolution rules [ISO03, Clause 13]. Furthermore, `square` takes its argument by-value and returns its result by-value. That is, it must be possible to copy objects of the deduced type. We need a rigorous framework for describing a template definition's requirements on its arguments. For space limitations and various practical reasons, such a formalism is described elsewhere [DRS05a].

Experience shows that successful compilation and linking may not be the end to our problems. A successful build only shows that the template instantiations were type correct given the arguments we passed it. What about template-argument types and values with which we didn't try to use it? The template definition may contain assumptions about its arguments that held for the arguments we gave it, but would fail for other, apparently reasonable, arguments. A simplified version of a classical example is

```
template<typename FwdIter>
bool palindrome(FwdIter first, FwdIter last)
{
  if (last <= first) return true;     // the middle
  if (*first != *last) return false;  // a difference
  return palindrome(++first, --last);
}
```

Here, we test whether a sequence, designated by a pair of iterators to its first and last elements, is a palindrome. The iterators are assumed to be of the *forward iterator category* (see §2.6), *i.e.*, they are assumed to support at least the operations `*`, `!=` and `++`. It is a widely used convention in the C++ community to use suggestive names such as `FowardIterator` (or a briefer `FwdIter`) for the template-parameters to convey assumptions. The definition of `palindrome` examines the sequence elements going from each end towards the middle. We can test it with a `vector`, with a C-style array, and with a `string`. In each case, our `palindrome` function template will instantiate and run correctly. Unfortunately, putting that `palindrome` into a library would be a time bomb: Not all sequences support `--` and `<=`. For example, singly-linked lists do not. Experts use elaborate systematic techniques to avoid such problems. However, the fundamental problem is that a template definition is not (by itself) a good specification of its requirements on its parameters. We need to make those requirements explicit and less ad hoc than the expression of an algorithm. "Concepts" are such requirements.

2.4 Performance

"Preserving the performance of the current template techniques" is a major requirement for templates with concepts. Users of concepts should not only get better ways of expressing ideas and better error messages when they make mistakes. They should also get performance at least equal to what is currently achieved using templates. After all, templates play a key role in C++ programming for performance critical applications. This performance has three sources:

- elimination of function calls in favor of inlining;
- combining information from several contexts for better optimization;
- avoiding generating code for unused functions.

The first point is not particular to templates but a general feature of C++ inline functions. However, inlining is crucial for fine-grained parameterization as is commonly used in the STL and other libraries relying on generic programming techniques. The performance in question is both run-time and memory footprint. When used well, as described here and used in major libraries, templates can simultaneously reduce both. The reduction in code size is especially important because on modern processors a reduction in code size implies a reduction in memory traffic and improved cache performance.

As an example, let's consider the function template `accumulate` from the Standard Library:

```
template<typename FwdIter, typename T>
T accumulate(FwdIter first, FwdIter last, T init)
{
   for (FwdIter cur = first; cur != last; ++cur)
      init = init + *cur;
   return init;
}
```

In other words, `accumulate` returns the sum of the elements in its input sequence with its third argument ("the accumulator") as the initial value. We might call it like this:

```
vector<complex<double>> v;
// ...
complex<double> z = 0;
z = accumulate(v.begin(),v.end(),z);
```

To do its job, `accumulate` will use addition and assignment operators on elements of type `complex<double>` and dereference `vector<complex<double>>` iterators. Adding `complex<double>` values involves addition of `double` values. To do this efficiently, all of these operations must be inlined. Both `vector` and `complex` are user-defined types; that is, they and their operations are defined somewhere in the C++ source code. Current C++ compilers handle this example so that the only function call generated is the one of `accumulate`. Access to `vector` members become simple load machine operation, addition of `complex` values becomes two floating-point add machine instructions, etc. To accomplish this, the compiler needs access to the complete definition of `vector` and `complex`. However, the result is a vast improvement (arguably optimal) over the naive approach of generating a function call for each use of an operation on a template parameter. Obviously, an add instruction executes much faster than a function call containing an addition. Furthermore, there is no function call preamble, passing of arguments, etc., so the resulting code is also much smaller.

Further reduction in generated code size is achieved by not emitting code for unused functions. The `vector` template class has many member functions that are not used by this example. Similarly, the `complex` class template has many member function and non-member supporting functions that are not used by this example. The C++ standard guarantees that no code is emitted for those unused functions.

To contrast, consider the more conventional case where arguments are accessed through interfaces defined as indirect function calls [Aug93, Str94]. Each operation then becomes a function call in the executable generated for user code such as `accumulate`. Furthermore, it would be distinctly non-trivial to avoid laying down code for unused (virtual)

member functions. It is beyond the ability of current C++ compilers and is likely to remain so for mainstream C++ programs where separate compilation and dynamic linking is the norm. This problem is not unique to C++; it is rooted in the fundamental difficulty of assessing which part of the source code is used and which is not when any form of run-time dispatch is in effect. Templates do not suffer this problem because their specializations are resolved at compile time.

This `accumulate` example is not quite perfect for illustrating the subtleties of generating object code from source code found in different parts of a program: It does not rely on implicit conversions or non-trivial overloading. However, consider a variant where `int` values are accumulated in a `complex<double>` object:

```
vector<int> v;
// ...
complex<double> s = 0;
s = accumulate(v.begin(),v.end(),s);
```

Here, the addition is done by converting the `int` value to a `double` value and then adding that to the accumulator, `s`, using a `+` operator of a `complex<double>` and a `double`. That's basically a simple floating-point addition. The point is that the `+` operator in `accumulate` depends on two template parameters and that it is the compiler's job to pick the most appropriate `+` based on all information about those two arguments. It would have been possible to maintain a better separation between the different contexts by always converting the element type to that of the accumulator. In this case, doing so would have resulted in the creation of an additional `complex<double>` for each element and an addition of two complex values. The code size and run time would have more than doubled.

We would not expect to see this last example directly in source code. If we saw it, we would consider it poorly written. However, equivalent code is common as the result of nested abstractions. It is especially important to generate good code in such cases because not doing so would be to discourage abstraction.

Please note that these optimizations are common place. Large amounts of real-world software depend on them. Consequently, improved type checking, as promised through the use of concepts, must not come at the expense of these optimizations.

2.5 A more realistic example

The `square` and `palindrome` examples are very simplistic cases where the assumptions can be stated on individual template-parameters in isolation. However, reality is more complicated than that. Consider `fill` from the Standard Library:

```
template<typename FwdIter, typename T>
void fill(FwdIter first, FwdIter last, const T& t)
{
   for (FwdIter cur = first; cur != last; ++cur)
     *cur = t;
}
```

In this definition, the symbols `!=`, `++`, `*` and `=` are dependent names. A call `fill(p, q, v)` will assign `v` to each element of the sequence defined by the interval $[p,q)$.

The ISO C++ rules for successful instantiation of that template require that values ι and τ for the type parameters `FwdIter` and `T` must fulfill the following assumptions:

1. Objects of type ι must be copy-initializable, so that they can be used as function arguments in calls to `fill`.
2. Two such objects must be equality comparable in the sense that the expression `cur != last` must be valid and its value convertible to the boolean type `bool`.
3. An expression of type ι must support the pre-increment operation.
4. The expression `*cur = t` must be valid (which implies that every sub-expression must also be valid).

For example, the following program fragment

```
vector<double> v(42);
fill(v.begin(), v.end(), 7);
```

constitutes a valid use of `fill` and the corresponding instantiation is type correct: The (deduced) template-arguments are iterator type for `vector<double>` and `int`, respectively, and all the constraints are satisfied. On the other hand, the function call in the fragment

```
int i = 0;
int j = 39;
fill(i, j, 43);
```

passes type checking, but produces errors during instantiation: The deduced type for the first template-argument, `int`, does not support the dereference operation (unary `*`). To diagnose that error, we need both the argument types (here, the built-in type `int`) and the body of the template definition (not just its declaration). This is the kind of error that we want immediately caught and reported at the point of call.

Finally consider, this fragment:

```
struct Generator {
   Generator(int);
   operator double();
   // ...
};

vector<int> v(42);
fill(v.begin(), v.end(), Generator(25));
```

This is valid. The result of `Generator(25)` can be converted to a `double` value (user-defined conversion) which can be converted to the `int` value (built-in conversion).

It follows that our type system must include a way to state assumptions on combinations of template-parameters. Such combinations often involve implicit user-defined and built-in conversions. Here is our current best bet for a convenient and compatible syntax:

```
template<Forward_iterator Iter, class T>
      where Assignable<Iter::value_type,T>
void fill(Iter first, Iter last, const T& t);
```

`Forward_iterator` and `Assignable` are concepts, *i.e.* compile-time predicates on types. For example, the boolean-valued expression `Assignable<double,int>` is true as we can assign an `int` value to a `double` object but `Assignable<double,string>` is false as we cannot assign a `string` value to a `double` object.

The checking of the use of `fill` proceeds as follows:

1. Deduce values ι and τ for the type parameters `Iter` and `T` from a call `fill(p, q, v)`.
2. Concept check; that is, evaluate `Forward_iterator<`ι`>` and `Assignable<`ι`::value_type, `τ`>`.
3. If the concept check succeeds, then type check the call.

Our problem now becomes how to provide a way to define such predicates. Everybody's first idea for that is to specify a concept as a set of operations with signatures. However, we found that specifying such complete sets of operations was feasible only for small examples or given incredible amounts of time and patience [SDR03a, SDR03b, DRS05a, DRS05b]. Producing the complete list of operations — complete with conversions, overloads, etc. — is distinctly non-trivial for real-world examples. For example, consider a few ways that an addition operator for a type `X` might be defined:

```
+    // built-in operation for X
X operator+(X,X);
X operator+(const X&, const X&);
X X::operator+(const X&) const;
X X::operator+(X);
const X& X::operator+(const X&);
const N& operator+(const& N, const N&); // X converts to N
X operator+(X, N); // N converts to X
```

In real-world code, conventions vary for operators, such as `+`, and often the variation reflects reasonable design choices. Conventions vary even more for named functions. Nevertheless, an algorithm written as a simple template function can just use `+` and rely on optimal code (without spurious conversions and indirections) being generated for each kind of definition of `+`.

When several operations are used in combination, a combinatorial explosion of possibilities can occur. For example, the resolution of the simple and common expression `*p++ = v` involves 3 explicit operations, each of which can be specified with a variety of signatures. In addition, it's easy to imagine auxiliary type being introduced for the subexpressions `p++`, and `*p++`. Furthermore, each of the operations `++`, `*`, and `=` may require a conversion operator to bring its argument(s) to the required type(s). In all, we may have to use 2 auxiliary types and $3+4$ operations (or $3+3*4$ operations if we count built-in conversions and user-defined conversions separately in each possible conversion), each of which we would have to explicitly specify if we used a primitive signature-based approach.

2.6 An iterator concept for `fill`

We must distinguish between the *internal form* and *external form* of a language. The internal language is usually biased toward implementations whereas the external language is directed towards programmers for use in source program. In this context, the external form of the concept system must be simple and flexible enough to cope with millions of lines of existing code in the hands of hundreds of thousands of programmers. On the other hand, the internal form must be precise and straightforward enough for use in compilers (including being retrofitted into existing compilers). One solution is to define a special language for defining "abstract signatures" [DR02, SDR03a, SGG+05]. Another solution, the one we describe here, is to generate sets of "primitive operations" with signatures from a notationally simpler and more abstract language, that we call "use patterns". In the use pattern approach, `*p++=v` is the user's notation for requiring the caller to (somehow, usually indirectly) supply a set of types and operations to type-safely compile expressions similar to `*p++=v`.

Before digging into technical difficulties of precise and concise concept definitions, let us illustrate the general idea with the concept `Mutable_fwd` defined below. This is a "first attempt" that will only serve `fill`, but it will get us started and provide a basis for refinements (see §4 and §5). The concept definition must express assumptions needed to separately check the definition of the template `fill`.

```
concept Mutable_fwd<typename Iter, typename T> {
  Var<Iter> p;       // a variable of type Iter.
  Var<const T> v;    // a variable of type  const T.

  Iter q = p;        // an Iter must be copy-able

  bool b = (p != q); // must support ``!='' operation,
                     // and the resulting expression
                     // must be convertible to ``bool''

  ++p;               // must support pre-increment, no
                     // requirements on the result type

  *p = v;            // must be able to dereference p,
                     // and assign a ``const T'' to the
                     // result of that dereference; no
                     // requirements on the result type
};
```

A concept definition is introduced with the keyword `concept`, followed by the concept-name, the declarations of the parameters for the concept (in template-parameter list notation, to emphasize the compile-time nature of concepts.) The set of assumptions, represented by a concept, are expressed as ordinary — if somewhat stylized — C++.

`Mutable_fwd` is a binary predicate that expects types as arguments, corresponding to the type parameters `Iter` and `T`. The notation "`Var<Iter> p;`" introduces the name `p` for a variable of type `Iter`. We could not just write "`Iter p;`" because that would imply default initialization — and we need not require that an `Iter` supports default initialization. The declaration of `q` states that it must be possible to copy-initialize a variable of type `Iter`. The declaration of `b` states that the symbol `!=` is required and the resulting expression must be implicitly convertible to `bool`. We also require pre-increment, but impose no requirement on the type of the result (except that it must be valid C++ type). The last line is interesting because it states a requirement involving both parameter types: It must be possible to dereference an expression of type `Iter` and to assign a `const T` to the result.

During checking of a *template definition* guarded by the concept `Mutable_fwd`, the compiler makes sure that all syntax trees it creates respect the type assumptions listed above. We can use this to check the definition of `fill`:

```
template<typename Iter, typename T>
   where Mutable_fwd<Iter, T>
void fill(Iter first, Iter last, const T& t)
{
   for (Iter cur; cur != last; ++cur)
     *cur = t;
}
```

As expected, this definition concept checks. The reasons are:

1. An `Iter` is copy-initializable, therefore it can serve as a function parameter — in other words, the declarations for `first` and `last` are well-formed.
2. The declaration for `t` is well-formed, because the type parameter `T` is assumed to be an object type according to the concept `Mutable_fwd`.
3. Since two `Iter`s are comparable and the result is implicitly convertible to a `bool`, it is valid to write the expression `cur!=last` in the second part of the `for`-loop header.
4. Pre-increment is part of the assumptions.

5. Since a reference of type `const T&` is indistinguishable from a variable of type `const T` in expression contexts, it follows that it is legitimate to use the reference `t` in the assignment expression `*cur=t`.

It can be argued that this long justification is unnecessary since we have abstracted the expressions so that the concept `Mutable_fwd` gives us exactly the code we want to write. It is nevertheless instructive to go through those trivialities before everything gets obscured by the technicalities. In particular, please note the use of implicit conversion, copy-initialization and substitution of reference for variables.

Using the concept `Mutable_fwd` (only and not also the definition of `fill`), we can check *uses* of `fill`. For example:

```
vector<double> v(42);
fill(v.begin(), v.end(), 7);
```

First, the compiler deduces `vector<double>::iterator` for the type parameter `Iter` and `int` for `T`. Then it goes on checking for the satisfiability of the predicate `Mutable_fwd` with the argument list `<vector<double>::iterator,int>`, which succeeds with the appropriate operations.

Specifying requirements as use patterns has a long story in C++ programming [Str94] and adopted in the ISO C++ standard. However, such usage has been informal and conventional, we promote it to a formally defined and automated mechanism supported by language constructs. From use patterns, we derive sets of primitive, easy to use in checking, constraints that template arguments must fulfill. Note that this difficult step is already done by C++ compilers as part of template instantiations.

2.7 Iterator concepts

Throughout this paper, we draw examples from the theory of iterators [SL94, Aus98, Str00, ISO03]. We emphasize that the iterator classification is just one example (albeit important one for programming in C++ using the Standard Library). Other sources of inspiration include the theory of mathematical structures in computer algebra [JS92].

The C++ standard library contains a classification of iterators, which are divided into five major categories: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*. See [Str00, Chapter 19] and [ISO03, Clause 24] for detailed exposition. We base our discussion on a particularly simple, yet difficult example from that classification. However, to follow the discussion here, the reader needs only a few key observations: An input iterator is a data-source abstraction and an output iterator is a data-sink abstraction. Each provides an operation called `++` to *advance* to the next element of a sequence. A forward iterator supports the notion of multi-pass algorithms (in particular, it is copyable) and an input iterator does not. A forward iterator that is mutable fulfills both input and output iterator assumptions. Iterators are pervasive in performance-critical code and optimal performance is expected. This implies that we can't impose significant overheads, such as a function call per operation, on iterators.

3. A concept system

The "use pattern"-based concept system presented here is based on the observation that a C++ expression, such as `*p++`, as it appears in a template is far more abstract, general, and readable than the set of operations and auxiliary types needed to implement it. Similarly, we can represent type predicates (concepts) as C++ code. C++ compilers typically represent template definitions as parse trees. Using identical compiler techniques, we can convert concepts to parse trees as well. Given that, we can implement concept checking as abstract tree matching. A convenient way of implementing this matching is to generate and compare sets of required functions and types (called "constraint sets", see §3.4) from templates and concepts definitions.

A concept definition is a set of abstract syntax tree equations with type assumptions. Concepts serves two purposes:

1. In *template definitions*, concepts act as typing judgment rules. If an abstract syntax tree depends on template parameters and cannot be resolved by the surrounding typing environment, then it must appear in the guarding concept bodies. Such dependent abstract syntax trees are implicit parameters of the concepts and will be resolved by concept checking at use sites.
2. In *template uses*, concepts act as set of predicates that the template-arguments must satisfy. Concept checking resolves implicit parameters at instantiation points.

So, if the set of concepts for a template definition specifies too few operations, the compilation of the template will fail concept checking: The template is under-constrained. Conversely, if the set of concepts for a template definition specifies more operations than needed, some otherwise legitimate uses may also fail concept checking: The template is over-constrained. By "otherwise legitimate" here, we mean that type checking would have succeeded in the absence of concept checking.

3.1 Concept definition

The concrete syntax for a concept definition is

```
concept ConceptName<P> where G { B };
```

It is a triple $\langle \mathsf{P}, \mathsf{G}, \mathsf{B} \rangle$ where:

- P is a list of explicit concept-parameters, with exactly the same declaration syntax as same for template-parameters. In particular, non-type concept-parameters are considered compile-time values (as the case in the current template system) in the scope of a concept definition.
- G is the "guard." It is a logical formula, made of compile-time expressions combined with the usual logical operators. The `where`-clause is optional. It usually expresses additional assumptions on combinations of the parameters P.
- B is the body of the concept. It is a sequence of simple declarations and expression-statements that enunciate syntax and type equations between the concept-parameters.

For example, here is a concept that captures the notion of *small object type* relative to some maximum size (also a parameter to the concept):

```
concept Small<typename T, int N>
   where sizeof (T) <= N
{ };
```

`Small`'s body is empty: the only "interesting" information is in the `where`-clause. More complicated concepts will involve constraints on individual arguments and in the body. A guard involves only compile-time expressions. In particular, a `sizeof`-expression is a compile-time value, and the non-type concept-parameter `N` is a compile-time expression.

A concept is a compile-time predicate. Therefore concepts can be combined with the logical operators and used in

`where`-clauses. As a short cut, concepts usable as unary predicates can also be used as the type of template-parameter. For example:

```
concept C<typename X>
   where C1<X> && C2<X>
{ };
```

is equivalent to

```
concept C<C1 X>
   where C2<X>
{ };
```

which again is equivalent to

```
concept C<C1&&C2 X>
{ };
```

A template type parameter introduced with the keyword `typename` is unconstrained. That is, any type can be used as an argument. Code involving such parameters cannot be concept checked. This "loophole" leaves existing code using templates valid, with its meaning unchanged. This compatibility feature is essential for the adoption of concepts into the C++ standard. There also appears to be uses for such completely unconstrained types in template meta-programming where properties of an argument are sometimes not needed until deep in a sequence of instantiations

3.2 Explicit check request

Programmers can explicitly ask the compiler to check conformance of a specific combination of types and values with respect to a concept. The syntax for that is either

```
assert Concept<argument-list>;
```

or

```
assert Concept<argument-list> with {
     declaration sequence
};
```

The first form is primarily used for checking whether a concept-argument list satisfies a concept. The program is considered in error if concept checking fails with those arguments. Otherwise, the translation of the program is carried on. For example, the explicit check

```
assert Mutable_fwd<double*, int>;
```

succeeds and the following values are deduced for the implicit parameters:

1. *CopyInitialize* is the built-in copy operator for values of type `double*`, and built-in copy operator for `bool` values;
2. *Convert* is the identity conversion on `double*`, and on the built-in conversion operator from `int` and `double`;
3. `!=` is built-in inequality comparison operator for `double*` values;
4. prefix `++` is the usual built-in pre-increment operator on pointers to `double`;
5. unary `*` is built-in dereference operator on `double*`;
6. and `=` is built-in assignment operator on `double`.

The second form of explicit check request is used in situations where it is necessary to "rewrite" syntax for the check to succeed. For example, the pointer type `int*` does not have members so it is necessary to map the abstract requirement of member syntax to something appropriate when checking for random access iterator properties:

```
assert Random_access<int*> with {
  int*::value_type = int;   // init*'s value_type is int
};
```

More generally, an explicit concept conformance assertion can be written for a family of types and values using one of

```
assert template<parameter-list> where G
          Concept<argument-list>;

assert template<parameter-list> where G
          Concept<argument-list> with {
              declaration sequence
          };
```

In either case, the `where`-clause is optional, just like in a template or concept declaration. For instance, instead of trying to request explicit check `Random_access<T*>` for pointer type `T*` individually for each type `T`, it is better to write the assertion in a template form

```
assert template<typename T> Random_access<T*> with {
  T*::value_type = T;  // T*'s value type is T
};
```

With that assertion, when the compiler needs to evaluate `Random_access<A*>` for a given type argument `A` it first (internally) generates the equivalent of

```
assert Random_access<A*> with {
  A*::value_type = A;
};
```

then evaluate that assertion, like in a non-template case. In particular, the program is erroneous if the generated assertion fails.

3.3 Implicit check request

Implicit checking of concepts typically happens in situations where an implicit instantiation is requested for a function template and the declaration is guarded by concepts like in

```
fill(v.begin(), v.end(), 42);
```

assuming the declaration of `fill` from §2.6. If concept checking fails for the deduced template-arguments then the function is disregarded; this failure is considered an error only if no declaration of the function can match the use.

For example, consider a situation where a function `f` needs to allocate a temporary object for its internal work. If the object is small we can allocate it in `f`'s activation record (e.g., execution stack). That is usually far more efficient than allocation from free store. What constitutes "small" is determined by the program's need, execution environment, and other implementation considerations. We can define `f` as overloaded on the size of its argument type:

```
const int max_size = 256;   // implementation-defined
// declare two variants of f().
template<typename T> where Small<T, max_size>
  void f(const T&);         // #1
template<typename T> where !Small<T, max_size>
  void f(const T&);         // #2
 int main()
 {
    buffer<char, 32> sbuf;
    f(small_buf);           // call #1
    buffer<char, 1024> lbuf;
    f(lbuf);                // call #2
 }
```

The function `f` is overloaded because `where`-clauses are part of function template signatures. The construction of overload set is very similar to the way implicit instantiation of function

template works in C++. The difference here is that we have added an additional step for concept check, before overload resolution proceeds.

3.4 From concepts to constraints sets

How is a concept definition — assumptions written in very abstract forms — turned into notations or requirements suitable for checking with conventional compiler technology?

A concept definition $\langle \mathsf{P}, \mathsf{G}, \mathsf{B} \rangle$ is further processed and refined into a quadruple $\langle \mathsf{P}_{\text{exp}}, \mathsf{P}_{\text{impl}}, \mathsf{G}, \mathsf{C} \rangle$ for the purpose of concept checking. The components are determined as follows:

1. A set of explicit parameter declarations P_{exp} as P in the definition, where properties implied by each nominated concept are assumed to hold for the corresponding parameter.
2. A list P_{impl} of dependent names.
3. A guard G, as in the definition. This predicate is assumed to be true during the definition of the concept body and checking of template definition.
4. A sequence of constraint equations C derived from the body B of the concept definition. The constraints are constructed reading "backwards" the static type rules of ISO C++.

Let's illustrate the general idea with the definition of `Mutable_fwd` from §2.6. The set of explicit parameters and implicit parameters are $\mathsf{P}_{\text{exp}} = \{\texttt{Iter}, \texttt{T}\}$ and

$$\mathsf{P}_{\text{impl}} = \{\textit{CopyInitialize}, \textit{Convert}, \texttt{!=}, \text{prefix } \texttt{++}, \text{unary } \texttt{*}, \texttt{=}\},$$

respectively. The symbol *CopyInitialize* and *Convert* are not explicitly mentioned; they are implied and overloaded at several places in the concept definition as detailed below. The implicit parameters are resolved (through name lookup) when deducing the values for the explicit parameters. In the definition of `fill`, the compiler assumes that the set of parameters is $\mathsf{P}_{\text{exp}} \cup \mathsf{P}_{\text{impl}}$. The constraint set generated from the body of `Mutable_fwd` was surveyed in §2.6 and is now detailed as follows:

1. The declaration "`Var<Iter> p`", introducing `p` as the name of a variable, requires `Iter` to be an object type.
2. Similarly, the declaration "`Var<const T> v`" generates the constraint that `const T`, hence `T`, is an object type.
3. The declaration `Iter q = p;` uses the C++ syntax of copy-initialization; consequently it generates the copy-initialization constraint. We denote that operation by *CopyInitialize*. It is dependent on the concept parameter `Iter`, therefore it is added to the list of dependent names.
4. The declaration `bool b = (p != q)` also generates a copy-initialization constraint. The type of the initializer in that declaration depends on a concept-parameter; therefore the *CopyInitialize* symbol required is also added to the dependent name list.

   The initializer requires the existence of a symbol `!=`, which in this case is also a dependent name. Furthermore, the argument types of `!=` need not be `Iter`. We only need that both operands be convertible to the types expected by `!=`. Consequently, we generate the corresponding implicit conversion constraints. Implicit conversion is denoted by the symbol *Convert*. The implicit conversions needed for both operands depend on a concept-parameter, so *Convert* is added to the list of dependent names.
5. The pre-increment expression `++p` requires the existence of a prefix `++` operator, added to the dependent name list. That expression also generates an implicit conversion constraint, that converts an expression of type `Iter` to the argument type of the pre-increment operator.
6. Finally, the expression `*p = v` requires the dereference operator and the assignment operators (added to the dependent name list). It also generates three implicit conversion constraints: One to convert unary `p` to the argument type of `*`, and two for converting `*p` and `v` to the argument types of `=`.

Please note that checking for copy-initialization and implicit conversion are simple operations done in C++ compilers.

3.5 Concept checking

When checking for the satisfiability of predicates at the template use site with typing environment Γ, the program translator recursively applies the following steps:

1. Substitute the concept-arguments for the concept-parameters P in the environment Γ, the concept guard G and body B.
2. If the guard evaluates to false, then concept check fails.
3. Look up the dependent names in the environment Γ. If lookup fails for a name, then concept check fails.
4. The result of name lookup for dependent names add additional equations for constraints variables introduced for symbols in function calls. Solve those equations through overload resolution. If overload resolution fails, then concept check fails.

When concept check succeeds, it should produce

1. a new typing environment;
2. a substitution, mapping dependent names to actual declarations;
3. and set of rewrite rules necessary for implicit conversions as required.

That triple is then used to produce instantiations from template. Only after concept checking succeeds, is type checking carried on. A nontrivial step here is to ensure that if the definition of the template is concept-correct, then the substitutions and rewrites resulting from concept and type checking of its uses will be well-formed in the new typing environment.

3.6 Associated types and values

An associated type or value is the value of a constraint variable, or an implicit parameter. They need not be named. For example, in the concept `Mutable_fwd` the type of the expression `*p` is an associated type. It is colloquially known as the *value type* of the iterator `Iter`. It is associated to `Mutable_fwd`, but it is not an explicit parameter. Associated types and values are essential in composing independently developed concepts. They help bridge the gap between different concepts. For example, in the theory of iterators (as briefly outlined in §5) the value type of an iterator plays a key rôle in the expression of `where`-clauses of function templates. For mathematical concepts like *group*, *ring* or *field*, the units of the respective structures are associated values. Uses of associated

types are presented in §5, where standard iterator concepts are discussed.

4. `fill` and associates revisited

Having outlined a framework for discussing concept definition and template checking; we now turn to the examples considered in §2.5 and §2.6.

4.1 The Assignable and Movable puzzle

The notion of iterator expressed as `Mutable_fwd` is a simplified version of the notion of *forward iterator* used by the C++ Standard Library. Unfortunately, it is oversimplified so that we can't use it for other algorithms where the C++ standard requires a forward iterator. Consider another (slightly simplified version) of the standard function `copy`:

```
template<typename Iter>
   where Mutable_fwd<Iter, Iter::value_type>
void copy(Iter first, Iter last, Iter out)
{
   for (Iter cur = first; cur != last; ++cur, ++out)
      *out = *cur;
}
```

The `value_type` is `Iter`'s associated type, the type of the element that the iterator refers to. However, `copy` fails to concept-check because we did not include the assumption that we could *read* from an iterator that meets the `Mutable_fwd` properties. We can compensate by adding a read operation:

```
concept Mutable_fwd<typename Iter, typename T> {
  Var<Iter> p;  // placeholder for variable of type Iter.
  Var<const T> v;
  Var<T> v2;
  // ... as before ...
  *p = v;      // we can write to *p
  v2 = *p;     // we can read from *p
};
```

With this version of `Mutable_fwd`, we slightly overconstrained `fill` (because `fill` never reads from elements of its sequence) but that is probably acceptable as the C++ Standard Library does the same. Unfortunately, we also overconstrained `copy` in a way that is unacceptable. Consider:

```
auto_ptr<Resource> v[10];
auto_ptr<Resource> w[10];
// ...
copy(v, v + 10, w);
```

An `auto_ptr` holds a pointer to a value and implements ownership semantics for that value. That is, instead of making a duplicate `auto_ptr`, assignment makes the target of the assignment the owner and invalidates the source. To do that invalidation, the assignment operation on `auto_ptr` writes to its source. Looking at `Mutable_fwd` (§2.6) we see that this won't work. `Mutable_fwd` requires only read access to the source of an assignment (`v` is `const`). That may be reasonable, but that it is not what the standard requires and not what is needed to cope with `auto_ptr`. We can try to fix that by requiring write access to the source of an assign:

```
concept Mutable_fwd<typename Iter, typename T> {
  Var<Iter> p;  // a variable of type Iter.
  Var<T> v;     // note, no ``const'' here.
  Var<T> v2;
  // ... as before ...
  *p = v;       // we can write plain ``T'' to *p
  v2 = *p;      // we can read from *p
};
```

Unfortunately, this is even worse: With that definition of `Mutable_fwd`, every type `T` that defines only an assignment taking a `const T&` now fail concept checking as an element accessed through a `Mutable_fwd` — and that is most types. Fundamentally, what we see here is that it is hard to precisely specify concepts so that we get both perfect separate checking of template arguments and the flexibility we are used to given the semi-formal specification of the standard iterators, containers and algorithms. Rather than patching, we will start by making the fundamental distinction between destructive assignments and non-destructive assignments and then build upon those.

4.1.1 Unary iterator predicates

To match the Standard Library requirements, we need a unary predicate to define a forward iterator. We do that by making the second template-parameter, the element type, implicit:

```
concept Forward_iterator<typename Iter> {
  Var<Iter> p;       // a variable of type Iter.
  typename Iter::value_type // must have a named member
                     // associated type value_type.

  Iter q = p;        // an Iter must be copy-able

  bool b = (p != q); // must support ``=='' and ``!=''
  b = (p == q);      // operations, and the resulting
                     // expressions must be convertible
                     // to ``bool''.

  ++p;               // must support pre- and
  p++;               // post-increment operations, no
                     // assumption on the result type
};
```

Here we have eliminated any requirements on the element type beyond the fact that it must exist and we can refer to it as `value_type`. That solves our problem deciding what kind of access we need to the `value_type` object by leaving that to the `where`-clause. In general, the use of *associated types*, such as `value_type` simplify the expression of generic programs [GJL+03].

4.1.2 Assignable

We can define what we mean for a value of type `U` to be *assignable* to an object of type `T`:

```
concept Assignable<class T, class U = T> {
  Var<T> a;
  Var<const U> b;
  a = b;         // non-destructive assignment
};
```

The assignment operator just reads its right hand side without modifying it — it can't modify because it takes a `const` operand. Usually, we also need the semantics invariant that, after assignment, the values `a` and `b` are equivalent (in some sense). That is what the C++ Standard Library requires. However, even though concepts could be designed to express semantics notions we haven't (yet) defined a syntax for expressing semantics for our concepts. The C++ standard makes the assumption that the type of an assignment to `T` is `T&`. We do not need that extra assumption, so we don't include it in `Assignable`.

To contrast and complement, we define destructive assignment (often called "a move") like this:

```
concept Movable<class T, class U = T> {
  Var<T> a;
```

```
    Var<U> b;
    a = b;       // potentially-destructive assignment
};
```

Given the concepts `Forward_iterator`, `Assignable` and `Movable`, we can declare the templates `fill` and `copy` as

```
template<Forward_iterator Iter, class T>
   where Assignable<Iter::value_type, T>
void fill(Iter first, Iter last, const T& t);

template<Forward_iterator Iter>
   where Assignable<Iter::value_type>
         || Movable<Iter::value_type>
Out copy(Iter first, Iter last, Iter out);
```

With these declarations, both the definitions of `fill` and `copy` will concept check. Their uses will succeed in all valid cases and a `fill` with an `auto_ptr` as its third argument will fail. In other words, we have a system that allows us to specify key C++ standard library components. Please note how the `where`-clauses in the function declarations complement the general concepts `Asslignable` and `Movable` by tieing the template-parameters together for the purpose of enforcing specific requirements of each function.

5. Standard iterator concepts

In this section, we define concepts for the most difficult part of the C++ Standard Library iterator hierarchy: input iterator, output iterator, and forward iterator. The C++ standard says: "Forward iterators satisfy all the requirements of the input and output iterators and can be used wherever either kind is specified". The first half of that statement is not actually true, but it is close enough for the second half to hold in all reasonable cases given experienced Standard Library implementers with common sense. However, a type system cannot work for reasonable cases only, and compilers (enforcing a type system) are not noted for their common sense.

As part of a complete solution, we need to formally define

1. how one moves and compares iterators (++, +, ==, etc.);
2. whether one can write to the iterator (`*p = x`);
3. whether one can read from an iterator (`x = *p`);
4. whether the assignment (or copy-initialization) used is destructive (the `auto_ptr` mess);
5. whether one can use a multi-pass algorithm (visiting an element twice; impossible for an input iterator or an output iterator.)

The C++ standard, and countless successful applications, reduce that to the simple programmers' rules of thumb:

1. one can read from an input iterator;
2. one can write to an output iterator;
3. a forward iterator is both an input iterator and an output iterator
4. the rest is detail that one can look up if needed.

Users who have written successful — and type safe — applications based on this (over) simplification of the current rules would not accept an "improved" system that required them to understand significantly different rules and to write more code, just to do the same work. In other words, just parameterizing an iterator concept by all sources of variability would not do the job. Users cannot be asked to explicitly select their iterators from a set of more than a dozen iterator categories. The ideal solution would be one where what the programmers thought was true (but isn't) is, and where all reasonable code compiles. This is almost possible, but only almost. In particular, we cannot avoid `where`-clauses to express requirements on combinations of template parameters.

We will not explain the problems of the current iterator requirements in detail. That would be tedious and pointless as many of the weaknesses are well understood in the C++ implementor and language lawyer community, are being corrected, and don't actually affect applications builders. They include:

1. The lack of distinction between destructive and non-destructive assignments.
2. A failure to consistently require iterators to be copy constructible.
3. A failure to point out that a forward iterator to a `const` value type isn't an output iterator.
4. Problems with composability of requirements: Specifying that `*p = v` must work for an output iterator `p` and that and `v = *q` must work for an input iterator `q`, but failing to note that this does not imply that `*p = *q` must work even when `p` and `q` have the same value types. (The reason is that both `*p = v` and `v = *q` may require a user-defined conversion so that `*p = *q` could require two user-defined conversions, and that's disallowed by the C++ rules for implicit conversion.)

Some of these problems were first discovered as part of our effort to find a practical and formal specification of the Standard Library facilities.

First we define a few supporting concepts. Some deal with basic access issues:

```
concept Copy_constructible<typename T> {
    Var<T> a;
    T b = a;  // copy construction
    T c(a);   // direct copy construction
};

concept Assignable<typename T, typename U = T> {
    Var<T> a;
    Var<const U> b;
    a = b;    // copy (non-destructive read)
};

concept Movable<typename T, typename U = T> {
    Var<T> a;
    Var<U> b;
    a = b;    // potentially-destructive read
};

concept Equality_comparable<typename T, typename U = T> {
    Var<T> a;
    Var<U> b;
    bool eq = (a == b);
    bool neq = (a != b);
};

concept Arrow<typename T> {
             // built-in
};
```

`Arrow<typename T>` is a built-in predicate expressing the curious Standard Library requirement for iterators that if `(*p).m` is legal then `p->m` is legal. Note that `Arrow<P>` is true for all C++ built-in pointer types and for all standard conforming user-defined ("smart") pointer types. It has been ob-

served that, for a given `m`, it may be possible possible to express the "arrow assumption" using `!` and (short-circuit) `||`:

```
concept Just_star_dot_m<typename T> {
  Var<T> p;
  (*p).m;
};
concept Just_arrow_m<typename T> {
  Var<T> p;
  p->m;
};
concept Arrow_m<tpename T>
   where !Just_star_dot_m<T> || Just_arrow_m<T>
{ };
```

However, we have no way of abstracting over `m`; that is the main reason why `Arrow` is a built-in predicate.

The notion of a `Trivial_iterator` specifies what is common for all iterators (not much):

```
concept Trivial_iterator<Copy_constructible Iter> {
  typename Iter::value_type;

  Var<Iter>  p;
  Iter& q = ++p;          // usable as
  const Iter& q2 = p++;   // converts to
};
```

Initializing a `const` reference means "converts to" and initializing a `non-const` reference means "usable as" (according to the C++ standard). Note that this use of references ensures that inheritance is taken into account. We don't feel an urgent need to invent new notation for that.

We can now define an input iterator as a trivial iterator that we can increment, compare, assign to, use `->` on, and read from:

```
concept Input_iterator<Trivial_iterator Iter>
           where Equality_comparable<Iter>
                 && Assignable<Iter>
                 && Arrow<Iter> {
  Integer difference_type; // the type of distance between
                           // two input iterators
  Var<Iter> p;
  const Iter::value_type& v = *p;    // converts to
  const Iter::value_type& v2 = *p++; // converts to
};
```

This is much more succinct than the "input iterator requirements" in the C++ standard [ISO03, §24.1.1], more precise, and also more correct. For example, the standard forgot part of the `Copy_constructible` requirement, but fortunately, none of the implementations did or this example (from the standard) wouldn't have compiled:

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

We discovered this when defining the concepts.

Note that the `Input_iterator` concept does not say what type is returned by `*`. In particular, it does not say that the result type of `*p` is `p`'s `value_type`; it could be a proxy type that implicitly converted to `value_type`. Eliminating the possibility of a proxy here would not only over-constrain the problem, it would also break real optimized code. One of the beauties of using use patterns compared to signatures is that we don't have to be explicit about possible intermediate types.

We do not assume that the result of dereferencing an input iterator is an lvalue; thus, we do not require (or allow) destructive reads from an input iterator. For example, `auto_ptr` is not an acceptable value type for an input iterator. If we want to read an `auto_ptr` from an input iterator, we need to say so in some `where`-clause.

The `difference_type` is the type used to express the number of elements between two iterators. It is defined using the concept `Integer` to require it to be a signed integer type.

For output iterator, we had to explicitly cope with the possibility of destructive assignment, so first we define concepts to express that:

```
concept Output_assign<Trivial_iterator Out, typename T> {
  Var<Out> p;
  Var<const T> v;
  *p = v;
  *p++ = v;
};

concept Output_move<Trivial_iterator Out, typename T> {
  Var<Out> p;
  Var<T> v;
  *p = v;
  *p++ = v;
};
```

Given those we can define `Output_iterator` as a `Trivial_iterator` that one can write to:

```
concept Output_iterator<Trivial_iterator Out>
          where Output_assign<Out,Out::value_type>
                || Output_move<Out,Out::value_type>
{ };
```

This does not in itself solve all problems with using output iterators. The reason is that our analysis shows that most problems with specifying iterators and the iterator hierarchy relates to specifying exactly when and how one can write to an output iterator. Furthermore, most of the troublesome variations and alternatives directly reflect algorithms and relationship between the value written and the iterator. Such issues are best dealt with in the algorithms' `where`-clauses. This definition of `Output_iterator` simply takes care of the minimal case where an algorithm simply assigns a value to an output iterator.

Note that an output iterator isn't `Equality_comparable` or `Assignable`. That's not our interpretation but a requirement that the Standard Library imposes for good reasons. Output iterators are an oddity, but a useful oddity that directly reflects the nature of output.

To help writing `where`-clauses for output iterators, we provide a helper concept reflecting the most common use involving another type, copying from another iterator:

```
concept Output_from_input<Output_iterator Out,
                          Input_iterator In> {
   Var<Out> p;
   Var<In> q;
   *p = *q;
   *p++ = *q++;
};
```

The C++ standard explicitly defines a forward iterator as something that meets all the requirements of an input iterator and an output iterator. In addition, it adds assumptions needed to support multi-pass algorithms:

```
concept Forward_iterator<Input_iterator Iter>
          where Output_iterator<Iter> {
  Iter p;                      // default constructible
  Iter::value_type& t = *p;    // usable as
  Iter::value_type& t2 = *p++; // usable as
};
```

So, how does this specification of the Standard Library requirements fare vis-à-vis our Assignable-and-Movable puz-

zle (§4.1)? For simple assignments (`*p = v`) an output iterator simply works for both ordinary and destructive assignments. The real (not simplified) `copy` copies from a sequence defined by a pair of input iterators to a sequence defined by an output iterator. To define that, we need to deal with the relationship between the value types of the two iterator types:

```
concept Move_from_input<Output_iterator Out,
                        Input_iterator In>
          where Output_from_input<Out, In> {
  Var<In> q;
  In::value_type& v = *q;
};

template<Input_iterator In, Output_iterator Out>
    where Output_from_input<Out, In>
          || Move_from_input<Out, In>
Out copy(In first, In last, Out out)
{
   for (In cur = first; cur != last; ++cur, ++out)
      *out = *cur;
}
```

The `Move_from_input` differs from `Output_from_input` only in requiring that the input iterator refer to a value that one could possible modify.

The implementation for `copy` remains the same as ever. The code for `copy` remains as good as ever. All we have done is to get perfect separate checking and some new opportunities for overloading.

We value the iterator requirements as a pretty extreme real-world challenge to any system aimed at specifying requirements for template arguments.

6. Related work

6.1 Siek's proposal

Jeremy Siek *et al.* [SGG+05] proposed a somewhat different concept system for C++. After discussions in the C++ standards committee, that system was modified [GSW+05] to handle many key issues discussed in the "use pattern based concepts" proposal [SDR05a]. In that system, a concept definition itself consists of sequence of operations with so-called pseudo-signatures. When concept checking template definitions, the pseudo-signatures act like the exact type of the operations. When concept checking a template use, wrapper functions are implicitly generated to implement conversions between the exact type of the declarations as found in the use context and the signature of the operations as assumed by the concept definition. That constitutes a huge departure from C++ semantics, which implies that parts of a program can be led to believe that some functions exist when, in fact, they do not. It is uncertain whether such forwarding functions can be consistently eliminated to allow optimal and consistent use of overloaded versions of a function. There are other differences between this system and ours, such as an absence of the *or* and *not* operators for concepts.

A similar, but slightly more abstract variant of the pseudo-signature idea called "abstract signatures", was presented and analyzed in [Str03] and [SDR05a]. We rejected that in favor of the use pattern notation because it was too verbose and because it would introduce a whole new declaration syntax with associated special semantics into an already crowded syntactic universe.

There is a close relationship between concepts, as described here, and constraints classes [Str]. This allows us to test concepts by transcribing them into constraints classes and explicitly insert them into code.

6.2 Type classes

It has been claimed that concepts as envisioned for C++0x are just Haskell's type classes. In fact, "overloading" in Haskell is limited compared to C++'s notion of overloading. In C++, overloading is exclusively a compile-time notion: Two functions are said overloaded if they have different signatures; there is no requirement of generic instance relationship between them. For example, C++ allows programs like

```
int min(int, int);                  // #1
template<typename T> T min(T, T);   // #2
int min(int, int, int);             // #3
// ...
int x = min(1, 2);                  // call #1
int y = min('a', 'b');              // call #2
int z = min(1, 2, 3);               // call #3
```

An overload set can contain function templates as well as "ordinary" functions. Haskell overloading is expressible in C++ as a combination of overriding and template specialization. Note that in C++, selection among overridden virtual functions takes place at run-time whereas overloading takes place at compile time. When all Haskell instance declarations for a given type class are available in a module, the type class construct and instance declarations correspond closely to C++'s notion of template partial or full specialization — this morally is what happens when the dictionaries, in the dictionary passing style translation [WB89], are optimized away. Given this close correspondence between Haskell type classes and C++ template specialization, it would not be surprising at all that a large body of C++ *type traits* techniques can be translated to Haskell type class techniques, and vice versa. Indeed, there are recent proposals [CKPJM05, CKPJ05] to add associated types to Haskell.

When instance declarations for a given type class are spread over several modules, information about the class members usually cross modules through indirect function calls (as do C++ virtual function calls). In that case, type classes closely correspond to parameterized abstract classes and virtual function overriding. A more detailed account is provided in the "use pattern based concept" proposal [SDR05a, Appendix B]. The problems addressed by type classes are more limited than those addressed by concepts.

The contexts of Haskell instance declarations are additional features absent from current C++. These are among the simplest contructs expressible with concepts where assumptions can be expressed individually on template-parameters in isolation.

As general predicates, concepts can be combined with the logical operators (*and*, *or*, *not*). An obvious place where the ability to write such logical formula is useful is in the `where`-clauses of template declarations as extensively illustrated.

6.3 Qualified types

In his PhD thesis [Jon94], Mark Jones introduced the notion of *qualified types* as a general framework to approach constrained type systems as studied by Stefan Kaes [Kae88], and Philip Wadler and Stephen Blott [WB89] that form the basis of Haskell's type classes. Jones' framework is general enough to account for Haskell's type classes, sub-typing and extensible records. It was later generalized to constructor classes and type classes with functional dependencies. However, Jones' system strives at describing systems where constraints are expressed purely at the type level. As we have seen, that is not accurate enough for C++ templates. Furthermore, Jones' framework seems to be more appropriate for type systems with *overriding* or *specialization* semantics than

with general overloading and type scheme as found in C++. Finally, while Jones' qualified types are formally type with predicates, the predicates cannot be directly used in formula involving logical connectors as in the concept system presented in this paper; and use of compile-time integer values is not included in the theory of qualified types.

7. Conclusion and future work

In this paper, we have defined a framework for specifying a concept system for checking C++ templates. This system, unlike conventional signature-based or object-oriented style type system, is powerful enough to express simply, concisely and accurately the C++ Standard Library notions and requirements. This formulation of concepts enables perfect checking of template definitions and uses in isolation without adverse effects on the performance of generated code. In the process, we uncovered several weaknesses in the current informal formulation of the Standard Library requirements. We have a complete typed abstract syntax tree representation for C++, including concepts, that will become a testbed for further work [SDR05b]. Future directions for work include:

- Complete our formalism for ISO C++. It is capable of handling proposed extensions that affect its type system including concepts.
- Explore various assumptions [SDR05a] on the overload resolution operator to provide a complete proof of the "fundamental theorem":

  Theorem 1 (Soundness) *If a template definition concept checks and if its uses both concept check and type check then its instantiations for those uses also type check.*

  We have a draft of that proof, but it is too long to fit in the margin here. That might require restrictions on template partial specializations as discussed in [SDR05a].
- Complete a concept checker based on abstract syntax trees.
- Experiment with the use of concepts to specify the STL and other domains, such as the theory of mathematical structures in Computer Algebra [JS92].
- The work reported in this paper focuses on the static semantics of concepts, but concepts also have dynamic semantics components that will be subject of future work.

Together with our colleagues in the ISO C++ standards committee, we will analyze the various ideas for concepts and synthesize a concept system for C++0x.

Acknowledgments

We would like to thank the anonymous reviewers. We are grateful to Phil Wadler for his helpful comments and suggestions for improving on earlier versions of this paper. Feedbacks from Sophia Drossoupoulou were also useful in improving on the introductory material to C++ templates. We appreciate Jaakko Järvi's comments on an early draft.

References

[Aug93] Lennart Augustsson. Implementing Haskell Overloading. In *Functional Programming Languages and Computer Architecture*, pages 65–73, 1993.

[Aus98] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, October 1998.

[CKPJ05] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated Type Synonyms. In *ACM SIGPLAN International Conference on Functional Programming*, Tallinn, Estonia, 2005. ACM Press.

[CKPJM05] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated Types with Class. *ACM SIGPLAN Notices*, 40(1):1–13, January 2005.

[DR02] Gabriel Dos Reis. Generic Programming in C++: The next level. *The Association of C and C++ Users Spring Conference*, April 2002.

[DRS05a] Gabriel Dos Reis and Bjarne Stroustrup. A Formalism for C++. Technical Report N1885=05-0145, ISO/IEC SC22/JTC1/WG21, October 2005.

[DRS05b] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ Concepts. Technical Report N1886=05-0146, ISO/IEC SC22/JTC1/WG21, October 2005.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'État, Université Paris VII, 1972.

[GJL+03] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A Comparative Study of Language Support for Generic Programming. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 115–134. ACM Press, 2003.

[GSW+05] Douglas Gregor, Jeremy Siek, Jeremiah Willcock, Jaakko Järvi, Ronald Garcia, and Andrew Lumsdaine. Concept for C++0x (Revision 1). Technical Report N1849=05-00109, ISO/IEC SC22/JTC1/WG21, August 2005.

[HHPJW96] Cordelia V. Hall, Kevin Hammond, Simon Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[ISO03] International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.

[Jon94] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[JS92] Richard D. Jenks and Robert S. Sutor. *AXIOM: The Scientific Computation System*. Springer-Verlag, 1992.

[Kae88] Stefan Kaes. Parametric Overloading in Polymorphic Programming Languages. In *Proceedings of the 2nd European Symposium on Programming*, volume 300 of *Lecture Notes In Computer Science;*, pages 131–144. Springer-Verlag, 1988.

[PJ03] Simon Peyton Jones. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[PT98] Benjamin C. Pierce and David N. Turner. Local Type Inference. In *Symposium on Principles of Programming Languages*, pages 252–265, San Diego CA, USA, 1998. ACM.

[Rey74] John Reynolds. Towards a theory of type structure. In *Proceedings of Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, New York, 1974. Springer-Verlag.

[SDR03a] Bjarne Stroustrup and Gabriel Dos Reis. Concepts — Design choices for template argument checking. Technical Report N1522, ISO/IEC SC22/JTC1/WG21, September 2003.

[SDR03b] Bjarne Stroustrup and Gabriel Dos Reis. Concepts — syntax and composition. Technical Report N1536, ISO/IEC SC22/JTC1/WG21, September 2003.

[SDR05a] Bjarne Stroustrup and Gabriel Dos Reis. A Concept Design (rev.1). Technical Report N1782=05-0042, ISO/IEC SC22/JTC1/WG21, April 2005.

[SDR05b] Bjarne Stroustrup and Gabriel Dos Reis. Supporting SELL for High-Performance Computing. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.

[SGG+05] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concept for C++0x. Technical Report N1758=05-0018, ISO/IEC SC22/JTC1/WG21, January 2005.

[SL94] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report N0482=94-0095, ISO/IEC SC22/JTC1/WG21, May 1994.

[Str] Bjarne Stroustrup. Technical FAQ: Why can't I define constraints for my template parameters? `http://www.research.att.com/~bs/bs_faq2.html#constraints`.

[Str88] Bjarne Stroustrup. Parameterized Types for C++. In *Proceedings of USENIX C++ Conference*, Denver, CO., October 1988.

[Str89] Bjarne Stroustrup. The Evolution of C++: 1985–1989. *USENIX Computer Systems*, 2(3), 1989.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[Str03] Bjarne Stroustrup. Concept checking — A more abstract complement to type checking. Technical Report N1510, ISO/IEC SC22/JTC1/WG21, September 2003.

[Str04] Bjarne Stroustrup. Abstraction and the C++ model. In *Proceedings of ICESS'04*, December 2004.

[Str05] Bjarne Stroustrup. The design of C++0x. *C/C++ Users Journal*, May 2005.

[TDBP00] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, and Erhard Ploederer, editors. *Consolidated Ada Reference Manual*, volume 2219 of *Lecture Notes in Computer Science*. Springer, 2000.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, Austin, Texas, USA, 1989.