Specifying C++ concepts

N1886=05-0146

Gabriel Dos Reis
Department of Computer Science
Texas A&M University
College Station, TX-77843
gdr@cs.tamu.edu

Bjarne Stroustrup
Department of Computer Science
Texas A&M University
College Station, TX-77843
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 embedded systems and safety-critical software. Current work on improving templates focus on the notion of *concept* (a type system for templates), which promises significantly improved error diagnostics and increased expressive power. In this paper, we use a formalism to analyze the problem of how to express concepts in a practical way. 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. Relying on the formalism for C++, we 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 a convenient basis for concept checking in a compiler.

1 Introduction

ISO Standard C++ [ISO03, Str00] directly supports generic programming through the notion of *template*. Templates are essential in C++ for capturing commonalities in 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 instantiations combine information available in both definition and instantiation contexts. Second, a C++ templates is typically implicitly instantiated if and only if it is used in a way that is essential to the program semantics; that automatically minimizing the footprint of an application. This contrasts to systems that require the programmer to explicitly manage instantiation, such as Ada [TDBP00] and System F [Gir72, Rey74].

This work is part of an effort to design a type system for C++ types and values that can be used for template arguments, as currently successfully used and by the programmers that successfully use them [SDR05]. This paper describes our formalism only as it is needed to discuss 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 [DRS05]. Our notation for concept definition is based on "use patterns" and our aim is to translate those usage patterns into a set of operation signatures suitable for type checking.

While the present paper centers the discussion of concepts around C++ templates, the fundamental ideas generalize to a type system that supports parametric polymorphism, some form of local type inference and an extension of the notion of dependent name. Our contributions also include a development of a formal framework to specify concepts, clarifications of the C++ standard iterator library requirements, and better and clearer formulations of iterators concepts not found in previous works

The remaining of the paper is structured as follows. We first examine the fundamental problem with templates (§1.1) and the requirements of its solutions (§1.2). Then present a solution based on concepts (§2) expressed using “use patterns” and apply it to simplified examples (§3). Finally, we apply our concepts to a known hard problem of significant practical importance: the specification of the lowest levels of the C++ standard library iterator, exposing some weakness (§4). We survey recent related works in §5 and conclude in §6.

1.1 The problem

The near-optimal performance offered by ISO C++ templates comes at the price of a very weak separation between template definitions and their uses. In the current definition of C++, a complete template definition is itself the only expression of a template’s assumptions 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. To do that, we must find a way to 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 per-

formance; that is, if concept checking succeeds for a template definition and for a use of it, then the resulting instantiation will type-check and all information will be used to generate optimal code. Please note that separate checking without optimal code generation is trivial: just use some form of abstract classes, as is often done explicitly in C++ and is the basis of the "generic" language facilities of Java and C#.

A template is a recipe from which a C++ translator generates declarations. Conceptually, the parameters of such recipe are specified in two ways: *template-parameters* (explicitly mentioned in the template declaration) and *dependent names* inferred from the definition of the template (based on the fact that their meanings depend on template parameters). The notion of dependent name is a departure from the general principle in C++ that a name can be used only if a corresponding declaration is visible. That device is a language feature designed to keep the number of explicit parameters manageable. At an instantiation point, dependent names are resolved by considering both the definition and instantiation contexts.

We will examine the definition for the function template `fill()` from the standard library, some uses of `fill`, and some related standard library functions. This will expose most of the problems with checking templates and provide a context for our solution. Here is a definition of `fill`:

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

In this definition, `FwdIter` and `T` are type template-parameters and 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 the values ι and τ for the type parameters `FwdIter` and `T` must fulfill the following assumptions:

1. instances of ι must be copy-initializable, so that they can be used as function arguments in calls to `fill()`;
2. two such instances must be equality comparable in the sense that the expression `first != second` must be valid and its value convertible to the ISO C++ boolean type `bool`;
3. an expression of type ι must support the pre-incrementation operation;
4. the expression `*first = 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[0], &v[0] + v.size(), 7);
```

constitutes a valid use of `fill()` and the corresponding instantiation is type correct. This is because the (deduced) template-arguments are `double*` and `int` respectively, and all the enumerated constraints are satisfied. However, 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. 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 X {
   X(int);
   operator double();
   // ...
};

vector<int> v(42);
fill(&v[0], &v[0] + v.size(), X(25));
```

It is valid, and there are perfectly reasonable programs that are logically equivalent. The expression `X(42)` can be converted to a `double` which can be converted to the `int` required by the `fill()` that takes pointers to `int`s as its iterators.

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

`Forward_iterator` and `Assignable` are *concepts*, e.g. predicates on types (and, where needed, values). The checking of the use of `fill` proceeds as follows:

1. deduce values ι and τ for the type parameters `Fwd` and `T` from a call `fill(p, q, v);`
2. concept check: `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. Looking at that, we found that it was feasible only for small examples or given incredible amounts of time and patience [SDR03a, SDR03b]. Producing the complete list of operations — complete with conversions and overloads — is distinctly non-trivial for real-world examples. In fact, one should distinguish between the *internal* and *external* forms 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, to be successful, the external form must be simple and flexible enough to cope with millions of lines of existing code in the hands of hundred of thousands of programmers. On the other hand, the internal form must be precise and straightfor-

ward enough for use in compilers (including being retrofitted into existing compilers). Our solution is to generate sets of "primitive operations" with signatures from a notationally simpler and more abstract language, from what we call "use patterns".

1.1.1 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. Either provides the notion of *advance* of computation. A forward iterator supports the notion of multi-pass algorithms (in particular, it is copy-able) 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.

1.2 Generic programming and concepts

Many forms of increased support of generic programming in C++ have been proposed since the initial template design [Str88]. In particular, constraints based on variations of inheritance have been proposed but not adopted as they tend to focus on only part of the problem and to rely on non-scalable mechanisms [Str94, §15.4]. The design, implementation, and use of the Standard Template Library have contributed significantly to the appreciation of templates and generic programming techniques. Most recent proposals focus on the notion of *concept*, which can roughly be summarized as a type system for template arguments. The design space of concepts for C++0x is explored in [SDR03a, SDR03b, Str03]. Currently, two main approaches to the design of concepts [SGG+05, SDR05] are being debated in the C++ community and standards committee.

To be useful in real-world C++, simple concepts should be simple and easy to express [SDR05]. A system that poses both major conceptual challenges and significant notational burdens to programmers will not succeed in mainstream programming — however elegant it may be from a theoretical perspective. Unfortunately, the ISO C++ rules governing C++ expressions have evolved over more than three decades and are hard to simply and accurately express in a conventional system. In particular, the notion of *convertible* is far more used than the notion of *same type*. Expressing concepts purely at the type level seems to lead to either under-specification or over-specification. See [SDR03a, SDR03b, Str03] for further discussion. Consequently, we express required properties of template parameter as usage patterns — whereby archetype C++ expressions are used to denote the relations/equations that template arguments must satisfy. Specifying requirements as use patterns has a long story in C++ programming [Str94] and are used in the ISO 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. Our proposed mechanism relies on the programmer explicitly specifying requirements in the form of concept instead of just relying on template definitions. Importantly, these types are available at template use sites to be used for template argument checking and overloading.

To explain the generation of type constraints from concept definition and uses, we use a formalism that contains the ISO C++ type system appropriately [DRS05].

2 A concept system

The basic idea behind the concept system presented in [SDR05] is that of providing for typed abstract syntax tree language. A concept definition is a set of abstract syntax tree equations with type assumptions. Concepts serves two purposes:

1. in template definitions, they act as typing judgment rules. If an abstract syntax tree depends on template parameters and cannot not 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, they act as set of predicates that the template-arguments must satisfy. Concept checking resolves implicit parameters at instantiation points.

Thus, if the set of concepts for a template definition specifiers too few operations, its compilation fails 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 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.

2.1 Concept definition

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

- P is a list of explicit concept-parameters, whose syntactic constructs are same as those of template parameters.
- G, if present, is the guard of the concept body provided by the `where`-clause. It is a logical formula, usually expressing 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.

The concrete syntax is

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

If no constraints involve a combination of template arguments, we can omit the `where`-clause. If all constraints are expressed as parameter types or in a `where`-clause, B can be empty. For example, the notion of *Assignable* is one of the recurring concepts of the C++ standard library, and also related to the `fill` function template introduced in §1.1. Here is a definition that will be discussed in §3.2.2.

```
concept Assignable<typename T, typename U = T> {
  Var<T> a;
  Var<const U> b;
  a = b;
};
```

A concept is a predicate. If concept checking fails, it evaluates to false; if concept checking succeeds, it evaluates to true. Therefore concepts can be combined with the logical operators *and* (`&&`), *or* (`||`), *not* (`!`). 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 type parameter introduced with the keyword `typename` (or its equivalent in this context `class`) is unconstrained. That is any type can be used as an argument.

2.2 From concepts to constraints sets

The concept definition $\langle \mathsf{P}, \mathsf{G}, \mathsf{B} \rangle$ is further processed and turned into a quadruple $\langle \mathsf{P}_{\text{exp}}, \mathsf{P}_{\text{imp}}, \mathsf{G}, \mathsf{C} \rangle$ 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 of implicit symbols P_{imp}. This is the list 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. This is constructed by inferring constraints that the explicit parameters and the dependent names must satisfy in order to make the expression-statements and declarations in B well typed, according to C++ normal rules.

The generation of the constraint set is governed by the constraint rules of Figure 2, which is a fragment of the formalism developed in the companion paper [DRS05]. We will give a first illustration for the simple case of `Assignable`. More elaborated examples will be discussed later (see §3.1).

The formalism developed in [DRS05] uses the framework of *local type inference* [PT98], which is itself a formalization of type checking and type inference folklore. Local type inference has been part of C++ template since its design [Str88]. The idea had been previously used, in some limited form, in the design and implementation of function overloading [Str89]. Basically, the idea consists in two parts. On one hand, some expressions can have their types locally synthesized, considering only types from adjacent nodes in the abstract syntax tree. This process does not involve global information like unification variables. The synthesis is done to yield the most faithful information as possible. On the other hand, type information from function call argument list is propagated and check against parameter types during the process of overload resolution.

The constraint rules yield the following assumption set for Assignable:

$$\Gamma \overset{\text{type}}{\vdash} \mathtt{T} \in \mathscr{O} \tag{1}$$

$$\Gamma \overset{\text{type}}{\vdash} \mathtt{const\ U} \in \mathscr{O} \tag{2}$$

$$\mathit{Assign} \in \xi \tag{3}$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathit{Assign} \nearrow (\tau_1, \tau_2) \to \tau \quad \tau_1, \tau_2 \in \mathscr{O} \cup \mathscr{R}, \quad \tau \in \mathscr{T} \tag{4}$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathtt{a} \in \mathtt{T} \searrow a' \in \tau_1 \tag{5}$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathtt{b} \in \mathtt{const\ U} \searrow b' \in \tau_2 \tag{6}$$

The first constraint says that T is assumed an object type; this is generated from rule [Var-cstr]. Similarly, the second constraint says that `const U` is assumed an object type (which also implies that U is an object type). In fact, here the assumption is even stronger: it is assumed that both T and U are complete, non-abstract object types. The third constraint comes from the fact that the assignment operator is referenced, but no declaration is in scope and its operands' meaning depends on the concept-parameter. Consequently, it is assumed an implicit parameter to the concept `Assignable`. The fourth constraint synthesizes the type $(\tau_1, \tau_2) \to \tau$ for Assign with fresh constraints type variables τ_1, τ_2 and τ. Finally, the fifth and sixth constraints state that the object variables a and b are assumed to implicitly convert to τ_1 and τ_2 respectively.

2.3 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 (elements of ξ) 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 Γ';
2. a substitution mapping ξ to actual declarations;
3. and set of rewrite rules necessary for implicit conversion as required in function calls.

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.

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.*

A template type parameter introduced with the keyword `typename` (or its equivalent in this context `class`) 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 unchecked. This compatibility feature is essential for the adoption of concepts into the C++ standard.

2.3.1 Explicit check request

Programmers can ask for explicit checking of conformance of a type with respect to a concept. The syntax for that is one of

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

The first syntax is primarily used for checking whether a combination of given types and values fulfills the assumptions of a concept. If concept checking fails with those arguments, then the program containing that explicit concept check request is considered in error. If concept check succeeds, then the translation of the program is carried on.

For example the assertion

```
assert Assignable<vector<int> >;
```

passes concept check and yields the following values for the constraints variables

$$\mathit{Assign} = \texttt{vector<int>::operator=}, \quad \tau_1 = \texttt{vector<int>\&},$$
$$\tau_2 = \texttt{const vector<int>\&}, \quad \tau = \texttt{vector<int>\&}.$$

The second syntax for explicit assert is used for cases where it is necessary to "rewrite" syntax. 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;
};
```

2.3.2 Implicit check request

Implicit checking of concepts typically happens in situations where an implicit instantiation for a function template is requested and its declaration is guarded by concepts. If concept checking fails then the function is disregarded; this failure is considered an error only if no declaration of the function can match the use. This 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.

2.4 Associated types and values

An associated type or value is the value of a constraint variable, or an implicit parameter. It is associated to a concept, 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, the nested type `value_type` of an iterator is an associated type. For mathematical concepts like *group*, *ring* or *field*, the units of the respective structures are associated values. Uses of associated types are presented in §4, where standard iterator concepts are discussed.

2.5 Typing rules with concepts

Template definitions are generally represented by compilers in their most abstract form, rarely with type annotations (because they are generally not known). However, to check such definitions, we clearly need typing rules. We have retained C++ rules that are most general as possible, to be applicable in concept definitions, and in checking of template definitions. They are listed in Figure 1. Those are the rules relevant to the situations discussed in this paper. A more complete set is to be found in the companion paper [DRS05].

The symbol Γ is used to denote *scopes* or type environments. Judgments for declaration kind are written as $\Gamma \overset{\text{decl}}{\vdash}$, whereas judgments for expression types are stated with $\Gamma \overset{\text{exp}}{\vdash}$. If a declaration introduces a name with object type (resp. reference type), then it is a variable (resp. reference) declaration. Similarly, if a declaration introduces a name with function type, then it is a function declaration.

Expressions are typed in two modes: (i) synthesis mode, where types are inferred from local information only; and (ii) checking mode, where a target type is given and an expression is checked for implicit convertibility to that type. Literals have synthesized types as dictated by ISO C++ rules. An expression that has synthesized type τ can be implicitly converted to an expression of type $\mathit{const}(\tau)$. A use of a variable

$$\text{[Var-decl]}\ \frac{x:(\Gamma,\tau)\quad \tau\in\mathscr{O}}{\Gamma \overset{\text{decl}}{\vdash} x\in Var_\tau}\qquad \text{[Ref-decl]}\ \frac{x:(\Gamma,\tau)\quad \tau\in\mathscr{R}}{\Gamma \overset{\text{decl}}{\vdash} x\in Ref_\tau}$$

$$\text{[Call-decl]}\ \frac{f:(\Gamma,\tau)\quad \tau\in\mathscr{F}}{\Gamma \overset{\text{decl}}{\vdash} f\in Call_\tau}$$

$$\text{[Lit]}\ \frac{\Gamma\vdash l \text{ literal}}{\Gamma \overset{\text{exp}}{\vdash} l\nearrow\tau_l}\qquad \text{[Const]}\ \frac{\Gamma \overset{\text{exp}}{\vdash} e\nearrow\tau}{\Gamma \overset{\text{exp}}{\vdash} e\searrow e'\in const(\tau)}$$

$$\text{[Var-use]}\ \frac{\Gamma \overset{\text{decl}}{\vdash} x\in Var_\tau}{\Gamma \overset{\text{exp}}{\vdash} x\nearrow\tau\cap lvalue}\qquad \text{[Ref-use]}\ \frac{\Gamma \overset{\text{decl}}{\vdash} x\in Ref_\tau}{\Gamma \overset{\text{exp}}{\vdash} x\nearrow\check{\tau}\cap lvalue}$$

$$\text{[Call-use]}\ \frac{\Gamma \overset{\text{decl}}{\vdash} f\in Call_{(\pi_1,\ldots,\pi_n)\to\tau}\quad x_i\in\alpha_i\searrow y_i\in\pi_i}{\Gamma \overset{\text{exp}}{\vdash} f(x_1,\cdots,x_n)\rightsquigarrow f(y_1,\cdots,y_n)\nearrow\tau}$$

$$\text{[Var-subs]}\ \frac{\Gamma \overset{\text{decl}}{\vdash} x\in Var_\tau\quad x\in fv(e)\quad \Gamma \overset{\text{decl}}{\vdash} y\in Ref_\tau}{\Gamma\vdash e=[y/x]e}$$

$$\text{[Ref-subs]}\ \frac{\Gamma \overset{\text{decl}}{\vdash} x\in Ref_\tau\quad x\in fv(e)\quad \Gamma \overset{\text{decl}}{\vdash} y\in Var_{\check{\tau}}}{\Gamma\vdash e=[y/x]e}$$

Figure 1: Typing rules

of type τ as an expression has synthesized type $\tau\cap lvalue$. If a function f has parameter-type list $(\pi_1,\ldots,\pi_n)$ and is called with argument list such that each argument is (implicitly) convertible to the corresponding parameter type, then the call is well-formed, and its synthesized type is the return type of f. Although we mention only function, it should be clear that overloaded operators are functions therefore subject to the rules; and built-in operators are nothing but functions with built-in implementations, therefore the rules we mention covers all operators. Finally, a reference of type τ can be substituted for a variable of type $\check{\tau}$ (the referenced type) in any expression where the variable appears free. And vice versa.

$$\text{[Var-cstr]}\ \frac{\Gamma\vdash x\in Var_\tau}{\Gamma\vdash\tau\in\mathscr{O}}\qquad \text{[Ref-cstr]}\ \frac{\Gamma\vdash x\in Ref_\tau}{\Gamma\vdash\tau\in\mathscr{R}}$$

$$\text{[Call-cstr]}\ \frac{\Gamma\vdash f(x_1,\ldots,x_n)}{\Gamma\vdash f\nearrow(\tau_1,\ldots,\tau_n)\to\tau}\ \text{with } \tau_i\in\mathscr{P},\tau\in\mathscr{T}\text{ fresh and }\Gamma\vdash x_i\searrow\tau_i$$

Figure 2: Constraints rules

For concept checking purposes, the typing rules are read "backward" to generate constraints, based on the premise that they are well-formed, leading to the rules in Figure 2. A variable declaration implies that its type is an object type. A reference declaration requires that its type be a reference type, thus implying that the referred type is either a function type or an object type. In the synthesis mode, the number of arguments presented to a call, determine the minimum number of parameters required (the actual call may have more, if its declarations permits and it has sufficient default arguments). The type variables are fresh. The arguments are required to be convertible to the synthesized type. Note that the types of the arguments are not used to synthesize the type of the call.

3 `fill` and associates revisited

Having set a formal framework for discussing concept definition and template checking; we now turn to the example considered in the introduction.

3.1 An iterator concept for `fill`

Before digging deep into technical difficulties of precise and concise concept definitions, let us illustrate generation of constraint set from `Mutable_fwd`. This is a "first attempt" that will only serve `fill`, but it will get us started and provide a basis for refinements. 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;       // placeholder for variable of type Iter.
  Var<const T> v;    // placeholder for 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++;               // must support post-increment,
                     // no requirements on the result type

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

The requirements are expressed as ordinary — if somewhat stylized — C++. `Mutable_fwd` is the name of a concept. It is a binary predicate that expects types as arguments, corresponding to the type parameters `Iter` and `T`. The declaration `Var<Iter> p` introduces a variable placeholder of type `Iter` named `p` without requiring initialization; we couldn't just write `Iter p` because that would imply default initialization — and we need not require that `Iter` supports default initialization. The declaration of `q` state that it must be possible to copy-initialize a variable of type `Iter`. In another words, `Iter` must be copy-constructible. The declaration of `b` states that `!=` comparison is required and the resulting expression must be implicitly convertible to `bool`. We also require pre-increment and post-increment, but imposes no requirements on the type of the resulting expression (except that it must be a 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.

Please note that the requirements are not stated as declarations of member and non-member functions. Exactly how those operations will be provided by a template argument

type is immaterial. Only their existence for well-formed programs is essential. Importantly, the programmer need not explicitly specify built-in or user-defined conversions or resolve overloads; see [SDR05]. Note also that requiring the existence of such operators, such as `++` and `*`, makes them *implicit parameters* to the concept `Mutable_fwd`, in addition to its *explicit parameters* `Iter` and `T`.

Following steps similar to those of §2.2, the definition of `Mutable_fwd` is turned into the constraint set:

$$\xi = \{\mathit{CopyInitialize}, \mathit{Neq}, \mathit{PreInc}, \mathit{PostInc}, \mathit{Deref}, \mathit{Assign}\} \quad (7)$$

$$\Gamma \overset{\text{type}}{\vdash} \texttt{Iter} \in \mathscr{O} \quad (8)$$

$$\Gamma \overset{\text{type}}{\vdash} \texttt{const T} \in \mathscr{O} \quad (9)$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathit{CopyInitialize} \nearrow (\tau_1) \to \texttt{Iter} \quad (10)$$

$$\Gamma \overset{\text{exp}}{\vdash} \texttt{p} \in \texttt{Iter} \cap \mathit{lvalue} \setminus \tau_1 \quad (11)$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathit{Neq} \nearrow (\tau_2, \tau_3) \to \tau_4 \quad (12)$$

$$\Gamma \overset{\text{exp}}{\vdash} \texttt{p} \in \texttt{Iter} \cap \mathit{lvalue} \setminus p' \in \tau_2 \quad (13)$$

$$\Gamma \overset{\text{exp}}{\vdash} \texttt{q} \in \texttt{Iter} \cap \mathit{lvalue} \setminus q' \in \tau_3 \quad (14)$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathit{CopyIntialize} \nearrow (\tau_5) \to \texttt{bool} \quad (15)$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathit{Neq}\,(p', q') \in \tau_4 \setminus \tau_5 \quad (16)$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathit{PreInc} \nearrow (\tau_6) \to \tau_7 \quad (17)$$

$$\Gamma \overset{\text{exp}}{\vdash} \texttt{p} \in \texttt{Iter} \cap \mathit{lvalue} \setminus p'' \in \tau_6 \quad (18)$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathit{PostInc} \nearrow (\tau_8) \to \tau_9 \quad (19)$$

$$\Gamma \overset{\text{exp}}{\vdash} \texttt{p} \in \texttt{Iter} \cap \mathit{lvalue} \setminus p''' \in \tau_8 \quad (20)$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathit{Deref} \nearrow (\tau_{10}) \to \tau_{11} \quad (21)$$

$$\Gamma \overset{\text{exp}}{\vdash} \texttt{p} \in \texttt{Iter} \cap \mathit{lvalue} \setminus p^{(iv)} \in \tau_{10} \quad (22)$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathit{Assign} \nearrow (\tau_{12}, \tau_{13}) \to \tau_{14} \quad (23)$$

$$\Gamma \overset{\text{exp}}{\vdash} \mathit{Deref}\left(p^{(iv)}\right) \in \tau_{10} \setminus \tau_{12} \quad (24)$$

$$\texttt{v} \in \texttt{const T} \setminus v' \in \tau_{13} \quad (25)$$

where $\tau_1, \ldots, \tau_{14}$ are constraints type variables.

During checking of a template definition guarded by the concept `Mutable_fwd`, the compiler makes sure all syntax trees it creates respect the above listed type assumptions. 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& v)
{
   for (; first != last; ++first)
     *first = v;
}
```

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

1. an `Iter` is copy-able as per constraints (10) and (11), therefore it can serve as a function parameter — in other words, the declarations for `first` and `last` are well-formed.
2. The declaration for `v` is well-formed, because we can assign from a value of type `T` — constraints (23) and (25) — therefore it cannot be `void`, which is all needed in order to form `const T&`.
3. since two `Iter` are comparable and the result is implicitly convertible to a `bool` — according to the C++ standard, constraints (12) and (16) —, it is valid to write the expression `first != last` in the second part of the `for`-loop header.
4. Pre-increment is part of the assumptions, (17) and (18).
5. Since a reference of type `const T&` is indistinguishable from a variable of type `const T` in expression context — C++ rules [Var-subs] and [Ref-subs] —, it follows that it is legitimate to use the reference `v` in the assignment expression `*first = v` as per constraints (21), (22), (24) and (25).

It can be argued that this long justification is unnecessary since we have abstracted the expressions so that the concept `Mutable_fwd` exactly gives us the code we want to write. It is nevertheless instructive stage to go through those trivialities before everything gets obscured by the technicalities. In particular, it worths noting implicit conversion, copy-initialization and substitution of reference to 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[0], &v[0] + v.size(), 7);
```

First, the compiler deduces the template-arguments `double*` for `Iter` and `int` for `T`. Then it goes on checking for the satisfiability of the predicate `Mutable_fwd<double*, int>` which succeeds with the following solution for the constraint set:

$$\texttt{Iter} = \texttt{double*}, \quad \mathit{CopyIntialize} = \textit{built-in copy constructor},$$
$$\tau_1 = \texttt{double*}, \quad \mathit{Neq} = \textit{built-in } \texttt{operator!=},$$
$$\tau_2 = \tau_3 = \texttt{double*}, \quad \tau_4 = \texttt{bool},$$
$$\mathit{PreInc} = \textit{built-in pre-increment}, \quad \tau_6 = \tau_7 = \texttt{double*\&},$$
$$\mathit{PostInc} = \textit{built-in pre-increment}, \quad \tau_8 = \texttt{double*\&},$$
$$\tau_9 = \texttt{double*}, \quad \mathit{Deref} = \textit{built-in dereference}$$
$$\tau_{10} = \texttt{double*}, \quad \tau_{11} = \texttt{double\&},$$
$$\mathit{Assign} = \textit{built-in assignment}, \quad \tau_{12} = \texttt{double\&},$$
$$\tau_{13} = \texttt{double\&}.$$

3.2 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 (; first != last; ++first, ++out)
      *out = *first;
}
```

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

```
concept Mutable_fwd<class Iter, typename T> {
  Var<Iter> p;  // placeholder for variable of type Iter.
  Var<const T> v;
  Var<T> v2;
  typename Iter::value_type; // there must exist a nested
                // type named ''value_type''.

  // ... as before ...

  *p = v;      // we can write to *p
  v2 = *p;     // we can read from *p
};
```

With this version of `Mutable_fwd`, we slightly over-constrained `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 over-constrained `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, `auto_ptr` assignment writes to its source. Looking at `Mutable_fwd` 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;  // placeholder for 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.

3.2.1 *Unary iterator predicates*

To match the C++ standard library requirements, we need a unary predicate to define a forward iterator. We do that by making the second template-parameter, the value type, implicit:

```
concept Forward_iterator<typename Iter> {
  Var<Iter> p;       // placeholder for variable of type Iter.
  typename Iter::value_type // must have an associated
                     // member 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` 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].

3.2.2 *Assignable*

We can define what we mean for a type `U` to be *assignable* to a value 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. Note that since a `T` can always be implicitly converted to a `const T&`, this assignment will also accept non-`const` operands. 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) specified a syntax to express semantics for our concepts. Furthermore the C++ standard makes the assumptions that the strict type of the resulting expressions 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 class 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.

4 Standard iterator concepts

In this section, we define concepts for the most difficult part of the C++ standard library iterator hierarchy as concepts: input iterator, output iterator, and forward iterator. The 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 you move the iterator (++, +, ==, etc.)
2. whether you can write to the iterator (`*p=x`)
3. whether you can read from an iterator (`x=*p`)
4. whether the assignment (or copy initialization) used is destructive (the `auto_ptr` mess)
5. whether you can use a multi-pass algorithm (visiting an element twice; you can't for an input iterator or an output iterator)

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

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

Users who have written successful — and type safe — applications based on this (over) simplification of the rules would take very badly to an "improved" system that required them to understand significantly different rules and to express that understanding in added 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<class T>` is a built-in predicate expressing a curious requirement 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 standards conforming user-defined ("smart") pointer types.

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 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 (section 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 you 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. Our analysis shows that most problems with specifying iterators and the iterator hierarchy relates to specifying exactly when and how you 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 the an output iterator isn't `Equality_comparable` or `Assignable`. That's not our interpretation but a requirement that the standard 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 standard explicitly defines a forward iterator as something that has meets all the requirements of an input iterator and an output iterator. In addition, it adds requirements 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 do this specification of the standard library requirements fare vis a vis our Assignable-and-movable puzzle (§3.2). 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 iterator to a sequence defined by an output iterators. 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);
```

The `Move_from_input` differs from `Output_from_input` only in requiring that the input iterator really refer to a value that you 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 types.

5 Related work

Jeremy Siek *et al.* [SGG+05] proposed a different concept system in which the use of every function template (whose definition is guarded by concept) needs to be preceded by declarations explictly stating that a given collection of types and value model a specific concept. A concept definition itself consists in sequence 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 "forwarding" functions are implicitly generated to implement conversions between the exact type of the declarations are found in the use context and the signature of the operations are assumed by the concept definition, where they differ. That constitutes a huge departure from C++ semantics, which implies that part of programs are led to believe that some functions exist when, in fact, they do not. Such inconsistencies are unacceptable for C++, which has a bias toward system programming.

A similar, but slightly more abstract notation of "abstract signatures" was presented and analyzed in [Str03] and [SDR05]. We rejected it (in favor of the use pattern notation) as being more 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 and [Str]. This allows us to test concepts by transcribing them into constraints classes and explicitly insert them into code.

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], 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.

It has been claimed that concepts as envisioned for C++0x are just actually Haskell's type classes. In fact, "overloading" in Haskell is very limited if compared to C++'s notion of overloading. Overloading in Haskell is expressible in C++ as a combination of overriding and template specialization. The problems type classes address and their scope are much more limited than that of C++ concept. A more detailed account in provided in Appendix B of [SDR05]. We mention that the concept system presented in this paper does not require declaration before use of a particular combination of template-arguments, in contrary to type classes in Haskell and Siek's system.

6 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 in-

formal formulation of the standard library requirements. 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. We have a complete typed abstract syntax tree representation for C++, including concepts, that will become a testbed for further work. In addition, various concept mechanisms are being actively explored within the ISO C++ standards committee.

7 References

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

[DRS05] Gabriel Dos Reis and Bjarne Stroustrup. A Formalism for C++. Technical Report D1885=05-0145, ISO/IEC SC22/JTC1/WG21, July 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.

[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.

[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 *Proceeding 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.

[SDR05] Bjarne Stroustrup and Gabriel Dos Reis. A Concept Design (rev.1). Technical Report N1782=05-0042, ISO/IEC SC22/JTC1/WG21, April 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 *Proceeding 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 *Proceeding of ICESS'04*, December 2004.

[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.