

Implementing Concepts

Douglas Gregor¹

Jeremy Siek²

1. Indiana University, Open Systems Lab, Bloomington, IN 47405

2. Rice University, Department of Computer Science, Houston, TX 77005

Document number: N1848=05-0108

Date: 2005-08-26

Project: Programming Language C++, Evolution Working Group

Reply-to: Douglas Gregor <dgregor@cs.indiana.edu>

Abstract

This document describes the implementation of the “Indiana” concepts proposal [GSW⁺05]. We relate the challenges we faced in the development of *ConceptGCC*, our prototype compiler for C++ with concepts. ConceptGCC, based on the GNU C++ compiler [GCC05], provides support for all major features of the concept proposal [GSW⁺05] and includes an updated Standard Library implementation that uses concepts extensively. We also discuss the compiler for \mathcal{G} , a language designed specifically for Generic Programming, to illustrate how one can implement concepts cleanly in a new compiler.

The intent of this document is to inform implementors of the interesting and difficult details of implementing concepts and advise implementors how to approach the task. Additionally, we hope to provide users and implementors alike a clearer picture of the compilation model required to fully support concepts and give a feel of how users will program with concepts. It is strongly recommended that the reader be familiar with the Indiana concepts proposal, N1849, prior to reading this document.

Contents

1	Introduction	2
2	Concepts and models	2
2.1	C++ analogy to templates	4
2.2	Verifying model correctness	4
2.2.1	Associated types	4
2.2.2	Operations	5
2.2.3	Nested requirements	7
2.2.4	Refinements	7
3	Using and declaring constrained templates	8
3.1	Where clauses	8
3.2	Binding models to requirements	8
3.3	Structural concepts	9
3.4	Partial ordering with where clauses	10
3.5	Diagnostics	11
3.6	Backing out of failed instantiations	12
4	Type-checking templates	12
4.1	Non-dependent templates	13
4.2	Name lookup	13
4.3	Type equality and same-type constraints	14
4.4	Archetypes	18

5	Compilation models	19
5.1	Instantiating constrained templates	19
5.2	Inlined templates	21
5.3	Separate instantiation	21
5.4	Separate compilation	21
6	Conclusion	22
7	Acknowledgments	22

1 Introduction

The introduction of support for generic programming—mainly, the addition of concepts—into C++ is a significant change to the language, particularly to templates, which are already quite complicated. With any such change, the question of implementability is foremost in our minds. Can the ideas presented in the proposal be solidified into rules that can be implemented in an existing compiler? Do these rules conflict with or complicate other features of the language? Finally, is the benefit to users worth the cost of specifying and implementing the extension?

This paper demonstrates that we have a solid implementation model backing the ideas presenting in our concepts proposal [GSW⁺05] and that this model can be introduced into existing C++ compilers with a reasonable amount of effort. We will make ConceptGCC available so that the final question—whether this extension provides real benefit to users—can be answered.

Concepts add an interface layer between the implementors of templates (functions or classes) and the clients of templates. Templates can be augmented by **where** clauses, which state the requirements that clients must meet before the template can be instantiated. On the client side of templates, it is the compiler’s responsibility to verify that these requirements are met before instantiating the template. On the implementor side of templates, it is the compiler’s responsibility to verify that the requirements are sufficient to ensure proper instantiation of the template. In essence, support for generic programming in C++ is an extension to the type system of C++ permitting additional type-checking for templates.

We present our experiences implementing both sides of type-checking with concepts and **where** clauses, with a dual focus on the user experience (what kind of diagnostics should the compiler be able to produce?) and on lower-level details of the compiler implementation (how should models be represented?). The implementation details are presented primary for ConceptGCC. However, where the ideal implementation of concepts differs greatly from what we were able to implement in ConceptGCC, we also describe the implementation in the \mathcal{G} compiler and note how an existing compiler could evolve in that direction.

We will make the two concept compilers, ConceptGCC and the \mathcal{G} compiler, and their source code, available to committee members.

2 Concepts and models

The first and most obvious decision when implementing concepts is how to represent the concepts and models within the compiler. Concepts are namespace-level entities that bundle a set of requirements together under a single name. The representation of concepts must include:

- The set of parameters to the concept, which will be a template parameter list.
- The set of refinements of the concept. For instance, the `ForwardIterator` concept refines the `InputIterator` concept. However, refinements are more specific because they actually name a *model-id*. For instance, `ForwardIterator` might be defined as:

```
template<typename Iter>
concept ForwardIterator : InputIterator<Iter> { };
```

The refinement is specifically for `InputIterator<Iter>`, not just `InputIterator`.

- The set of requirements for the concept, including requirements for associated types, operations, and so-called “nested” requirements. The following code snippet illustrates some such requirements:

```
template<typename Iter>
concept InputIterator
{
    typename ptrdiff_t; // associated type requirement
    typename reference; // associated type requirement
    where SignedIntegral<ptrdiff_t>; // nested requirement
    reference operator*(const Iter&); // operation requirement
};
```

- The set of models of this concept, including model templates (also called parameterized models). This set is needed when searching for a model to fulfill a certain concept requirement.

Note that, although many concepts have a single type parameter, concepts should not be represented as the type of a type: this formulation fails for multi-type concepts and model templates that rely only on concept information. A better analogy is that concepts are predicates on a set of template arguments. However, keep in that there are two sides to these predicates: on the client side, we are checking that these predicates are true; on the implementation side, we are assuming them to be true to enable type-checking of templates.

Models are the realization of concepts, which contain implementations and definitions for each of the requirements within their concept. Like concepts, a model is declared at namespace scope, but it should be represented only within the set of models for its corresponding concept. The representation of models must contain:

- The list of parameters to the model, which will be a template parameter list. This list may be empty.
- A set of requirements on the parameters in the template parameter list, which comes from the **where** clause of the model.
- The set of arguments to the concept, which gives the pattern of concept arguments that this model (template) will match. For example, consider the following model template:

```
template<typename Iter>
    where { RandomAccessIterator<Iter> }
concept RandomAccessIterator<reverse_iterator<Iter> > { ... };
```

In this model template, the set of parameters to the model contains only `Iter`, the set of requirements on the parameters contains `RandomAccessIterator<Iter>`, and the set of arguments to the concept contains `reverse_iterator<Iter>`.

- Declarations for and implementations of all of the requirements in the concept, either provided explicitly or generated implicitly by the compiler from “matching” definitions in the surrounding scope. For instance, there should be type definitions for all of the associated types of the concept and functions for all of the required operations in the concept. The signatures of the functions will match precisely the function requirements in the concept, once the concept arguments have been substituted. For instance, consider the following concept and its model template:

```
template<typename Iter>
concept RandomAccessIterator : BidirectionalIterator<Iter>
{
    typename difference_type; // associated type requirement
    difference_type operator-(Iter const&, Iter const&); // operation requirement
};

template<typename T>
concept RandomAccessIterator<const T*>
{
    typedef ptrdiff_t difference_type;
    ptrdiff_t operator-(const T* const& x, const T* const& y) { return x-y; }
};
```

Note that the model defines the type `difference_type` as `ptrdiff_t`, to meet the requirement for the associated type `difference_type`. Similarly, it defines an `operator-` with a signature that is identical to the requirement in the `RandomAccessIterator` concept, once `Iter` has been replaced with `const T*`. These signatures must match *exactly*.

2.1 C++ analogy to templates

Concepts and models are analogous to class templates and their (partial or full) specializations. The analogy is close enough that concepts and models are implemented in this manner in ConceptGCC and the relationship is closely represented by the syntax of the Indiana concepts proposal.

A concept is like the primary template, through which all lookups occur. The requirements of the concept can be represented as typedefs and (mainly static) member functions in this primary template. A model is like a partial or full specialization of its concept. Model templates are like partial specializations, and may have `where` clauses (just as class templates and class template partial specializations can). The declarations and implementations in a model that meet the requirements of the concept are implemented as typedefs or (mainly static) member functions.

The analogy to class templates and specializations breaks down where consistency is concerned. Class template (partial) specializations may be completely different from their primary templates, with different member types, functions, base classes, etc. Concepts and models, however, are much more closely related: each model must provide precisely the same types and functions as specified in the concept. This is how concepts enable type-checking: when one looks into a concept and finds a type, that type is sure to exist in any model of that concept.

2.2 Verifying model correctness

When a model is parsed or instantiated, the compiler must verify that all of the requirements of the concept are met by the model. In many cases, these requirements have default values that will be used if no declaration is provided to satisfy the requirement.

2.2.1 Associated types

Associated type requirements mandate that a type of a certain name occur within all models of that concept. Associated type requirements may have default values, in the form of an arbitrary type expression. For instance:

```
template<typename Iter>
concept MutableForwardIterator
{
    typename value_type = Iter::value_type;
    typename reference = value_type&;
};

concept MutableForwardIterator<int*>
{
    typedef int value_type;
};

template<typename Iter>
where { MutableForwardIterator<Iter> }
concept MutableForwardIterator<reverse_iterator<Iter> > { };
```

Consider how the associated types are checked for these two models. With the first model, the `typedef` of `value_type` satisfies the requirement for the associated type `value_type`. Since there is no `reference` type defined in the model, the default value of `value_type&` (which becomes `int&` after substitution) is used. With the model template, neither associated type is assigned a type, so `value_type` becomes the type `reverse_iterator<Iter>::value_type` and `reference` becomes `reverse_iterator<Iter>::value_type&`.

Note that the default for an associated type does not imply any constraints. A model of `MutableForwardIterator` could declare the `reference` associated type to be, e.g., `read_write_proxy<value_type>`. If the `reference`

type must actually be a reference to the `value_type` (as in a true `MutableForwardIterator`), the user would have to write a same-type constraint `value_type& == reference`.

2.2.2 Operations

Checking the existence and correctness of the declarations for required operations is relatively simple, requiring only the ability to perform template instantiation and compare the types of declarations. For instance, consider the following concept and model:

```
template<typename T>
concept Swappable
{
    void swap(T&, T&);
};

concept Swappable<int>
{
    void swap(int&, int&);
};
```

To verify that the model meets the requirements of the concept, we substitute `int` for `T` in the `swap` operation of the concept. The resulting declaration—`void swap(int&, int&)`—matches precisely the declaration in the model, so the model is correct and the client will define `Swappable<int>::swap` elsewhere in the program.

Synthesizing operations. It is often the case that users will omit declarations within the definition of a model. In this case, the compiler must synthesize the operations and their implementations. Consider, for instance, the definition of an `Addable` concept and several models:

```
template<typename T>
concept Addable
{
    T operator+(const T& x, const T& y);
};

concept Addable<int> {};

struct MyInt
{
    MyInt operator+(MyInt) const;
};
concept Addable<MyInt> {};

template<typename T>
concept Addable<valarray<T> > {};
```

Each of the models presented is correct. The required operations in a concept are referred to as *pseudo-signatures*, because while they are written as concrete syntactic entities, they permit a much more loose matching of syntax, allowing conversions and templates. This loose matching comes into play when the implementations of an operation are generated for a particular model. The implementations for various kinds of pseudo-signatures are:

- **Functions:** The implementation of a function will be a call to a free function of the same name, initiated at the point of definition of the model. For instance, given a pseudo-signature `void swap(T& x, T& y)` in concept `Swappable` and a model where `T = int`, the implementation will generate:

```
void Swappable<int>::swap(int& x, int& y)
{
    swap(x, y); // ignores Swappable<int>::swap!
}
```

- **Operations:** The implementation of an operator pseudo-signature invokes the operator with the appropriate syntax. For instance, the `operator+` in `Addable<int>` will be implemented as:

```

int Addable<int>::operator+(const int& x, const int& y)
{
    return x + y; // ignores Addable<int>::operator+!
}

```

Thus, this operator would match the built-in `+` operation for integers. The `operator+` for `Addable<MyInt>` would be written in the same way, but would end up matching the member function `MyInt::operator+`.

Implicit conversion operators are written merely as return statements, and can be satisfied either by converting constructors or user-defined conversion operations. For instance, here is the `Convertible` concept, a model converting `int` to `float`, and the synthesized conversion operator implementation:

```

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

concept Convertible<int, float> {};

// Synthesized...
Convertible<int, float>::operator float(const int& t)
{
    return t;
}

```

- **Member functions:** Member function requirements are implemented by performing a call to the member function. The transformation is similar to the transformation for free functions. Constructors and destructors work in the same way.

Duplicate signatures Identical signatures in a concept or model will be coalesced into a single requirement. For instance, consider a concept that says that two types `T` and `U` can be compared for equality but the order does not matter:

```

template<typename T, typename U>
concept MutuallyEqualityComparable
{
    bool operator==(const T&, const U&);
    bool operator==(const U&, const T&);
};

```

What happens if we create a model `MutuallyEqualityComparable<int, int>?` In this case, both `operator==` signatures become identical after substitution, so the compiler must eliminate one of them. Within `ConceptGCC`, this is trivially implemented by looping over all of the requirements of a concept, instantiating the requirements with the model arguments, then verifying that the same signature does not yet exist before introducing it into the model.

Default implementations Default implementations of concept operations add another layer of complexity when synthesizing the implementations of required operations. Default implementations provide a fall back should the original synthesized implementation fail to type-check. For instance, consider the `EqualityComparable` concept:

```

template<typename T, typename U = T>
concept EqualityComparable
{
    bool operator==(const T& t, const U& u);
    bool operator!=(const T& t, const U& u) { return !(t == u); }
};

template<>
concept EqualityComparable<int> {};

struct X {};

```

```

bool operator==(X, X);

template<>
concept EqualityComparable<X> {};

```

For `EqualityComparable<int>`, the compiler synthesizes the implementations for `operator==` and `operator!=` as described above, and the defaults are unnecessary. For `EqualityComparable<X>`, the synthesized implementation for `==` works, but consider the synthesized implementation of `!=`:

```

bool EqualityComparable<X>::operator!=(const X& t, const X& u)
{
    return t != u;
}

```

This implementation will fail to type-check; however, the compiler is *not* permitted to produce a diagnostic or fail. The default implementation of `!=` provided by the `EqualityComparable` concept will be used instead. This implementation was type-checked against the concept itself, so the `==` operator of the concept is employed. Thus, the second synthesized implementation—which does type-check properly—is:

```

bool EqualityComparable<X>::operator!=(const X& t, const X& u)
{
    return !(t == u); // == is found in EqualityComparable<X>
}

```

Dealing with default implementations therefore requires the ability to determine if an expression type-checks without emitting diagnostics when the expression fails to type-check. We discuss this issue further in Section 3.6. The synthesis of implementations without defaults is, however, relatively simple and straightforward.

2.2.3 Nested requirements

The nested requirements of a concept are easily verified once the model has been completely defined. The requirements can be checked in the same way that `where` clauses are checked (see Section 3.1) by verifying the same-type constraints and the existence of models for concept requirements either in the `where` clause (for model templates) or via other declared models.

2.2.4 Refinements

In addition to checking all of the requirements of the concept for a model, the compiler must check all of the requirements for the refinements of the concept. For instance, consider type checking the `ForwardIterator` model below:

```

template<typename lter>
concept InputIterator
{
    typename difference_type;
    require SignedIntegral<difference_type>;
};

template<typename lter>
concept ForwardIterator : InputIterator<lter> { };

concept ForwardIterator<int*>
{
    typedef std::ptrdiff_t difference_type;
};

```

The compiler must verify that the requirements from `InputIterator<lter>` are met. For instance, the `difference_type`—in this case, `std::ptrdiff_t`—is a model of the `SignedIntegral` concept. ConceptGCC achieves this by coalescing the requirements from all refined concepts in a concept post-processing step. For instance, the associated types from all refined concepts are re-introduced into the concept when it is declared, as are the required operations and nested requirements. This approach made name lookup within ConceptGCC simpler, but does result in a larger memory footprint; one could also opt to perform lookups in refined concepts as one would perform lookups into base classes.

3 Using and declaring constrained templates

This section discusses the implementation of the “client side” of constrained templates, including the issues of finding models to meet concept requirements and producing reasonable diagnostics. The client side of constrained templates is by far the most visible change for users, because it adds new syntax to the language and will provide users with greatly improved error messages for template libraries such as the C++ standard library. Interestingly, it is also by far the simplest part of the concepts proposal to implement: ConceptGCC required only about a month’s worth of hacking (from someone more accustomed to developing C++ libraries, not compilers) to be almost fully functional from the client’s perspective.

3.1 Where clauses

Where clauses contain the requirements that are placed on the parameters of a template. These requirements consist of:

- **Model requirements:** These requirements, such as `RandomAccessIterator<Iter>`, indicate that a model must be present before the template can be instantiated. For instance, when we try to match the template with `Iter = int*`, there must exist a model `RandomAccessIterator<int*>` or a model template that can be instantiated to it.
- **Same-type requirements:** These requirements, such as `value_type& == reference`, indicate that two types must be exactly equivalent.
- **Integral constant expressions:** These requirements, such as `is_empty<T>::value`, indicate that the given integral constant expression must evaluate true.

When the user names a template, either directly (`std::vector<int>`) or indirectly (`std::sort(first, last)`), the compiler first attempts to unify the template parameters with the arguments passed to the template or implied by the call. When this unification succeeds, it produces a set of bindings from template parameters to actual types and values. These bindings are then substituted into the requirements of the **where** clause, which are checked. Same-type requirements and integral constant expressions are trivial to evaluate, and if they fail then the template fails to match; model requirements require more effort to evaluate, detailed in the next section, but the effect is the same. For a class template this means that the compiler should emit an error or choose another specialization; for a function template this means that the template cannot enter the overload set. The effect is similar to what happens with the SFINAE rule: in fact, one can implement concepts in this way with C++03, but the resulting code is unwieldy and not extensible.

Within the GNU C++ compiler, checking if a particular template-id matched a template proceeded in two steps. First, the compiler performed unification to get bindings for each template parameter. If that succeeded, it would substitute these bindings into the template declaration to determine if any other language rules were violated. ConceptGCC adds a final pass that verifies that all of the requirements in the **where** clause are met.

3.2 Binding models to requirements

The search for models that match a particular model requirement (such as `RandomAccessIterator<int*>`) is analogous to the search for a (partial) specialization of a class template. The compiler checks each model for the concept, performing unification, consistency checks, and checking the **where** clause to determine which models may apply. If several such models apply, they are ordered as described in Section 3.4. If no models are found, then the model requirement cannot be met and the template will not be instantiated. This process can be recursive, because checking the **where** clause of a model template may require the compiler to search for additional model templates. Model lookup may sometimes result in tautologies (`Model<T>` exists if `Model<T>` exists if...), which will create model lookup loops: to avoid the problems associated with these loops, ConceptGCC marks a model-id as “not a model” while it is verifying if that model-id is truly a model. Consider the following example, which requires multiple stages of model lookup:

```

template<typename T>
concept LessThanComparable
{
    bool operator<(const T&, const T&);
};

template<typename Iter>
concept MutableRandomAccessIterator { typename value_type; };

template<typename T>
concept MutableRandomAccessIterator<T*> { typedef T value_type; }; // #1

template<typename Iter>
where { MutableRandomAccessIterator<Iter> }
concept MutableRandomAccessIterator<reverse_iterator<Iter> > // #2
{
    typedef MutableRandomAccessIterator<Iter>::value_type value_type;
    // ...
};

template<typename Iter>
where { MutableRandomAccessIterator<Iter>, LessThanComparable<value_type> }
void sort(Iter first, Iter last);

// ...
struct int {};
int array[10];
sort(reverse_iterator<int*>(array + 10), reverse_iterator<int*>(array));

```

The compiler will perform the following steps to determine if `sort` can be called:

1. Deduce `Iter = reverse_iterator<int*>` from the call to `sort`.
2. Substitute `reverse_iterator<int*>` for `Iter` in the **where** clause of `sort`.
3. Search for a model `MutableRandomAccessIterator<reverse_iterator<int*> >`.
 - (a) Try to unify model #1: Fails, because `reverse_iterator<int*>` does not match `T*`.
 - (b) Try to unify model #2: Passes, with `Iter = int*`.
 - (c) Substitute `int*` for `Iter` in the **where** clause of model #2.
 - (d) Search for a model `MutableRandomAccessIterator<int*>`.
 - i. Try to unify model #1: Passes, with `T = int`.
 - ii. Try to unify model #2: Fails, because we don't have a match with `reverse_iterator<Iter>`.
 - iii. Instantiate model #1 with `T = int`. It succeeds, giving us `MutableRandomAccessIterator<int*>`.
 - (e) Since we have the model `MutableRandomAccessIterator<int*>`, the **where** clause of model #2 with `Iter = int*` is satisfied, so we instantiate the model.
4. Search for a model `LessThanComparable<int>` (`int` is the `value_type` once substitutions have been performed). There are no models present, so this search fails and `sort` cannot be called.

ConceptGCC implements model lookup in the same way as it implements the lookup for class template partial specializations. The only difference is when the result of model lookup returns the primary template (the concept) instead of a specialization (a model): in this case, model lookup is considered to have failed, because no model exists.

3.3 Structural concepts

Structural concepts are a form of concepts that are identical to normal (“nominal” or “explicit”) concepts except that they may be matched implicitly based on structure. We mentioned previously that model lookup fails when no matching model is found (i.e., the “primary template” corresponding to the concept is the

only template returned). If a concept is structural, there is one last opportunity for the model to exist if it can be synthesized. We reconsider the `sort` example from the previous section, but change the definition of `LessThanComparable` to:

```
template<typename T>
struct concept LessThanComparable
{
    bool operator<(const T&, const T&);
};
```

Now, we re-examine the search for a model `LessThanComparable<int>`:

4. Search for a model of `LessThanComparable<int>`:
 - (a) There are no models or model templates.
 - (b) Since the concept is structural, tentatively introduce a model and type-check it:


```
template<> concept LessThanComparable<int> {};
```
 - (c) Type-checking succeeds (because of the existence of the built-in `<` operator), so this model is introduced.
5. Since we have a model `LessThanComparable<int>`, `sort`'s **where** clause is satisfied.
6. Instantiate `sort`.

What happens if the “tentative” model `LessThanComparable<int>` had failed to type-check? In this case, the compiler would have to back out of the model definition, eliminate the declaration, and conclude that there does not exist a model `LessThanComparable<int>`.

Support for structural concepts requires that the compiler be able to “back out” of an arbitrary chain of instantiations. The implementation of this capability is discussed further in Section 3.6.

3.4 Partial ordering with **where** clauses

The concepts proposal introduces a partial ordering for templates that contain **where** clauses. Two templates T_1 and T_2 are partially ordered by:

1. Determine if T_1 and T_2 can be ordered by the existing C++ rules for partial ordering of function and class templates. If so, return that result.
2. Determine if T_1 and T_2 are identical (e.g., same template parameters, template arguments, function parameters, etc.) *ignoring the requirements of the **where** clause*, again with the existing C++ rules used to match a declaration to a definition. If they are not identical, then T_1 and T_2 cannot be ordered.
3. Now, consider the **where** clauses to perform a partial ordering:
 - (a) Introduce the requirements from the **where** clause of T_1 into a new environment.
 - (b) Check each of the requirements in the **where** clause of T_2 to determine if they are satisfied in the new environment. If so, T_1 is at least as specialized as T_2 .
 - (c) Repeat the process with a new environment, to determine if T_2 is at least as specialized as T_1 .
 - (d) If T_1 is at least as specialized as T_2 , but T_2 is not at least as specialized as T_1 , then T_1 is the more specialized template. Similarly, we can determine if T_2 is more specialized than T_1 .

Consider the partial ordering of these three function templates:

```
template<typename Iter>
  where {InputIterator<Iter>}
  difference_type distance(Iter first, Iter last); // Function #1
```

```
template<typename Iter>
  where {RandomAccessIterator<Iter>}
  difference_type distance(Iter first, Iter last); // Function #2
```

```
template<typename T>
  ptrdiff_t distance(T* first, T* last); // Function #3
```

Function template #3 is more specialized than both #1 and #2 by the existing rules. However, #1 and #2 are identical by existing rules, so we compare based on the **where** clauses:

1. First, introduce the **where** clause of #1 (`InputIterator<Iter>`) into a new environment. Then check the **where** clause of #2 (`RandomAccessIterator<Iter>`) in that environment: since `RandomAccessIterator<Iter>` is not implied by `InputIterator<Iter>`, #1 is *not* at least as specialized as #2.
2. Create a new environment and introduce the **where** clause of #2 (`RandomAccessIterator<Iter>`) into it. Then, check if the **where** clause of #1 (`InputIterator<Iter>`) is satisfied. Since `RandomAccessIterator<Iter>` refines `InputIterator<Iter>` (i.e., all random access iterators are also input iterators), the requirements are satisfied, so #2 is at least as specialized as #1.
3. From the first two steps, we conclude that #2 is more specialized than #1.

Both the \mathcal{G} compiler and ConceptGCC implement partial ordering of templates with **where** clauses in this way.

3.5 Diagnostics

One of the most obvious initial benefits provided by the introduction of concepts is the chance to improve template error messages. Errors in the use of templates can be caught at the call site, when the compiler determines if the requirements in the **where** clause are met. Thus, common mistakes that typically generated syntactic or type errors deep in the body of a function template will instead be detected at the call site.

However, achieving good error messages will require some work. When a function template does not match due to unsatisfied requirements in the **where** clause, that function will never make it into the overload set. Thus, the compiler may only produce an unhelpful error message that does not say *why* the failure occurred. To see the effect for a given compiler, create a function template that fails to match because of an unsatisfied `enable_if` and try to call it. For instance, consider the following two formulations of a function that requires its template parameter to be a **float**, and calls to those two functions with an **int** parameter:

```
template<typename T>
  typename enable_if<(is_same<T, float>::value)>::type foo1(T);

template<typename T> where { T == float } void foo2(T);

// ...
foo1(17);
foo2(42);
```

For the call to `foo1`, GCC gives a particularly unhelpful error message:

```
sfnac.C:31: error: no matching function for call to 'foo1(int)'
```

To reach this conclusion, the compiler had to determine that the declaration of `foo1` was not suitable, but why not? With the addition of **where** clauses as explicit language features, supported by the compiler, the compiler can collect the reasons that functions fail to meet **where** clause requirements of functions that otherwise might match. If no match for a function is found, it prints this list in the diagnostic and the list of requirements that failed. For instance, ConceptGCC produces the following error message for the call to `foo2`:

```
sfnac.C:32: error: no matching function for call to 'foo2(int)'
sfnac.C:27: note: candidates are: void foo2(T) [with T = int] <where clause>
sfnac.C:32: note: same-type constraint 'T' == 'float' is not satisfied ('int' is not 'float')
```

Likewise, we can produce much improved error messages when concept requirements are not met. For instance, here is the error message produced by ConceptGCC when one attempts to call `std::sort` with `std::list` iterators:¹

```
sort.C:7: error: no matching function for call to 'sort(std::List_iterator<int>, std::List_iterator<int>)'
<path>: note: candidates are: void std::sort(_Iter, _Iter) [with _Iter = std::List_iterator<int>] <where clause>
sort.C:7: note: unsatisfied concept requirement 'std::MutableRandomAccessIterator<std::List_iterator<int> >'
```

Note that we explain again which requirement failed to be met, so the user knows what needs to be changed to get that particular function template to match. In the potentially common case where the syntax of the data types matches the concepts but no model declaration exists (e.g., because we've just upgraded to a new generic library that uses concepts), the compiler can check for the syntactic match and give a better error message. Here, we've omitted the `RandomAccessIterator` model for the `deque` iterator. Since it syntactically matches, the compiler can inform the user how to write the model:

```
sort2.C: In function 'int main()':
sort2.C:32: error: no matching function for call to 'sort(std::deque_iterator<int>, std::deque_iterator<int>)'
sort2.C:26: note: candidates are: void std::sort(_Iter, _Iter) [with _Iter = std::deque_iterator<int>] <where clause>
sort2.C:32: note: unsatisfied concept requirement 'std::RandomAccessIterator<std::deque_iterator<int> >'
sort2.C:9: note: model 'std::RandomAccessIterator<std::deque_iterator<int> >' syntactically matches concept
sort2.C:32: note: if the concept semantics are met, write a model definition:
    namespace std {
        template<> concept RandomAccessIterator<std::deque_iterator<int> > {};
    }
```

While we have not done user studies to verify that this is the information that will be most useful, our experience with generic programming indicates that it should be adequate. Regardless of the particular formulation, any of these results is far superior to the pages and pages of error messages one would receive from current C++ compilers when making the same error.

3.6 Backing out of failed instantiations

One particularly contentious issue with implementors has been with language features that require the compiler to be able to attempt an arbitrarily-complicated series of instantiations but “back out” without emitting a diagnostic if any of these fail. This issue has come up in various forms: for instance, should SFINAE rules apply to expressions inside `sizeof`? At present they do not, but some users have asked for the extension, to allow them to determine whether a given expression will type-check or not without making the program ill-formed.

There are two features of the Indiana concepts proposal that require the ability to silently back out of a failed instantiation: structural concepts and default implementations of required operations. However, limiting the number of features that use this capability does not make it any easier to implement.

In ConceptGCC, implementing this capability turned out to be quite simple. GCC itself is very tolerant of errors, and most routines just fall back by returning the sentinel `error_mark_node` when they uncover an error. We modified the diagnostics reporting code so that we could turn off output of warnings and errors. When turned off, the implementation records the presence of errors but does not report them or abort. Then, to implement this “tentative instantiation,” we turn off diagnostics and attempt to perform the instantiation. If the result of the instantiation is an error (`error_mark_node` or `NULL`), or if any errors were produced in the process, then the instantiation failed and we can take appropriate action. The handling of structural concepts in ConceptGCC required only 50 lines of C code, with another 50-line change to the GCC diagnostics-handling code. Clearly, the experiences of other compiler vendors may vary, but for this particular compiler the contentious issue of tentative instantiation was a non-issue.

4 Type-checking templates

The concepts proposal permits complete type checking of function and class templates. This type-checking is the more complicated part of the proposal to implement, and primarily improves the “implementor side”

¹Note that we removed a long path to fit the width of the page.

of templates. Type checking of templates makes it possible to write template code that is guaranteed to instantiate properly, because type checking is performed when the template is parsed, not when it is instantiated.

4.1 Non-dependent templates

The most important idea to keep in mind when understanding and implementing type-checking for templates with concepts is that it changes the notion of dependent and non-dependent types. In C++ without concepts, a type is dependent if its definition somehow depends on a template parameter (although there is a tighter definition). When concepts are added, types become dependent only if their definition somehow depends on a *dependent* template parameter. Thus, the presence of non-dependent (or “constrained”) template parameters in the definition of a type does not make that type dependent.

Expressions using only non-dependent types should be type-checked when a template is first parsed. For instance, the following code should produce a diagnostic both in C++03 and C++0X with concepts:

```
struct X {};

template<typename T> void f(const X& x, const X& y)
{
    x + y;
}
```

Since the types in the expression `x + y` are all non-dependent, the compiler checks the validity of the expression when the template is parsed. Had the type of either `x` or `y` been dependent (e.g., used a `T`), `x + y` would not have been type-checked. Thus, this code is valid both in C++03 and C++0X with concepts:

```
template<typename T> void f(const T& x, const T& y)
{
    x + y;
}
```

With concepts, template parameter types become non-dependent, thereby introducing more type-checking at parse time. Thus, the following example should produce a diagnostic similar to the first example, because the types involved in `x + y` are non-dependent (due to the existence of the **where** clause):

```
template<typename T> where {} void f(const T& x, const T& y)
{
    x + y;
}
```

Compilers that are better at checking non-dependent expressions when initially parsing templates will be more easily adapted to concepts. GCC, for instance, is rather poor at checking non-dependent expressions (the initial example does not produce an error with GCC 3.3!). The EDG front end, on the other hand, seems to type-check non-dependent expressions thoroughly. We therefore had to take two steps to introduce concepts: first, implement checking of non-dependent expressions in templates and, second, improve that checking for non-dependent expressions that do involve template parameters.

While implementing the first step in ConceptGCC, the greatest number of problems came from the use of a global variable `processing_template_decl`, which indicates when the compiler is processing a template. Many routines in GCC would check this flag and, if it is set, immediately return some unchecked representation for a template. Removing these checks typically involved checking for dependent types instead, then disabling any operations that cannot be performed for a template, such as emitting actual low-level code.

The implementation of the second step consisted mainly of removing the assumption that expressions involving template parameters could not show up in certain places in the compiler, e.g., when looking up members in a class or finding user-defined conversions and converting constructors. Part of the effort required detangling the notion of “uses template parameters” from “is or has a dependent type.”

4.2 Name lookup

The concept proposal introduces another scope in which name lookup can find declarations. The **where** clause scope resides just inside the template parameter scope (in which the names of template parameters reside),

and includes the names of associated types and operations from model requirements. For instance, consider the following concept and function template:

```
template<typename F, typename T1>
concept Callable1
{
    typename result_type;
    result_type operator()(F&, const T1&);
};

template<typename Op, typename X>
where { Callable1<Op, X> }
result_type transform(Op op, const X& x)
{
    return op(x);
}
```

The `Callable1<F, T1>` requirement introduces the type name `result_type` and the function

```
result_type operator()(Op&, const X&);
```

into the `where` clause scope, allowing the body of the function to be fully type-checked: the function return type `result_type` resolves to `Callable1<Op, X>::result_type` and the call to `op` resolves to `Callable1<Op, X>::operator()`.

If two disjoint concepts have the same associated type name, then an unqualified use of that name is ambiguous (and should result in a diagnostic) *unless* the types can be proved equivalent. Please refer to Section 4.3 for more information on proving the equivalence of types. If two disjoint concepts have operations of the same name, they are overloaded and overload resolution will decide amongst them in the normal way.

Any names in a template that are not resolved by the `where` clause will fall back to the lexical scope of the function, as would occur with non-templates or expressions of non-dependent types in a template.

Member functions, types, constructors, and conversion operators often require special treatment. In C++ without concepts, a template type cannot have any of these operations. However, since one can write requirements for members, the compiler must be able to do name lookup within template type parameters and even uninstantiated templates. To address this problem, ConceptGCC employs the notion of *archetypes*, which are discussed further in Section 4.4.

4.3 Type equality and same-type constraints

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

```
template<typename InputIterator1, typename InputIterator2>
where { InputIterator<InputIterator1>::value_type == InputIterator<InputIterator2>::value_type,
        LessThanComparable<InputIterator<InputIterator1>::value_type> }
bool includes(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2) {
    ...
    if (*_first2 < *_first1)
    ...
}
```

In the body of `includes`, the type

```
InputIterator<InputIterator1>::value_type
```

is considered the same type as

```
InputIterator<InputIterator2>::value_type
```

This is important, for example, in type checking the expression `*_first2 < *_first1`.

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

```

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

```

Type equality is also congruence relation, which just means that, for example, if $S == T$ then we have $\text{vector}\langle S \rangle == \text{vector}\langle T \rangle$. Conversely, if we know that $\text{vector}\langle S \rangle == \text{vector}\langle T \rangle$ then this would imply $S == T$.

The compiler must also ensure that the same-type constraints appearing in a **where** clause are not invalid. For example, a constraint such as $\text{int} == \text{char}$ is ill formed.

The problem of determining whether two types are equal given a set of same-type constraints in an instance of the congruence closure problem. The congruence closure problem already shows up in modern compilers, for example, in the common subexpression elimination optimization. There are efficient algorithms for the congruence closure problem: the algorithm by Nelson and Oppen [NO80], which we describe here, is $O(n \log n)$ time complexity on average, where n is the number of type nodes. It has $O(n^2)$ time complexity in the worst case. This can be improved by instead using the slightly more complicated Downey-Sethi-Tarjan algorithm which is $O(n \log n)$ in the worst case [DST80].

The Nelson Oppen algorithm itself is fairly simple, most of the smarts is in the data structures that are used. To prevent the duplication of work, types are stored in a directed acyclic graph (DAG) with a node for each type and edges connecting a type to its parts. Each type is represented by a unique node, and a type may be shared by several larger types. Figure 1 illustrates the DAG of the types T , U , int , $\text{vector}\langle T \rangle$, U_* , int_* , $\text{pair}\langle \text{vector}\langle T \rangle, U_* \rangle$, U_* , $\text{vector}\langle U_* \rangle$, and $\text{vector}\langle \text{int}_* \rangle$. The \bullet placeholders indicate pointers to other nodes in the type DAG.

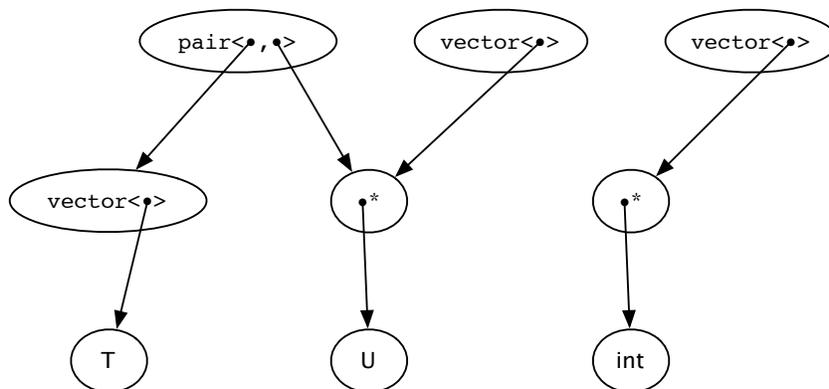
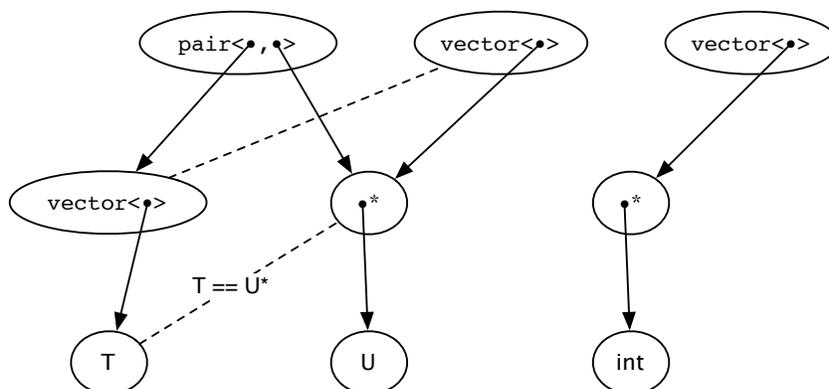
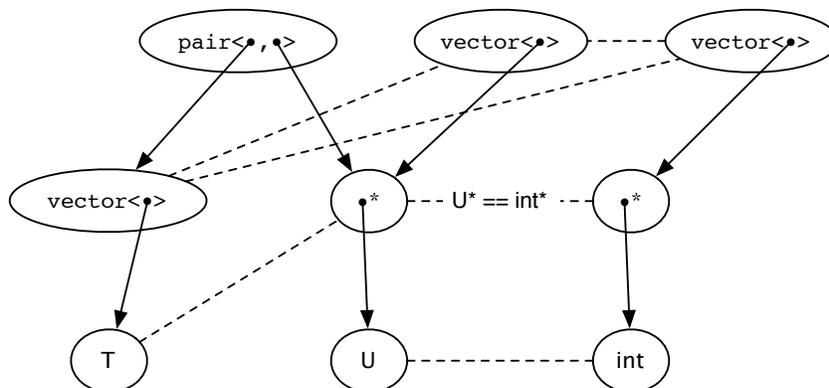


Figure 1: Directed acyclic graph representation of types in a compiler.

When a same-type constraint is encountered, conceptually we need to merge two nodes to become a single node. However, merging nodes is expensive because all the in-edges and out-edges must be rewired. Instead of merging the nodes we record that the two nodes are equivalent using a union-find data structure [Tar83, CLR90] (also known as disjoint sets). Disjoint sets are implemented by providing each type with a `parent` field. By recursively following the `parent` field of a type (and its parent, and its parent's parent, etc.), we arrive at the *representative* for an equivalence class. If two types have the same representative, they are in the same equivalence class and therefore are equivalent. Using the path compression and union-by-rank heuristics, disjoint sets can be implemented in almost linear time.

Figure 2 illustrates the same types in the DAG, but now we have added the same-type constraint $T == U_*$. We represent equivalence classes with dashed lines. T and U_* are put into the same equivalence class, therefore we also need to put $\text{vector}\langle T \rangle$ and $\text{vector}\langle U_* \rangle$ in an equivalence class (since they've just become equivalent). Note that we end up propagating equivalence class information up the graph, from T and U_* to vectors of those types.

Figure 3 evolves Figure 2 further, by introducing the same-type constraint $U_* == \text{int}_*$. This constraint is propagated in two directions: first, we let $U == \text{int}$, then we note that $\text{vector}\langle U_* \rangle$ and $\text{vector}\langle \text{int}_* \rangle$ are in the same equivalence class (which also contains $\text{vector}\langle T \rangle$).

Figure 2: DAG representation of types in a compiler under the same-type constraint $T == U^*$.Figure 3: DAG representation of types in a compiler under the same-type constraints $T == U^*$ and $U^* == \text{int}^*$.

The pseudo-code for the `merge` and `congruent` functions, which make up the congruence closure algorithm, is shown in Figure 4. The auxiliary `find` function maps a type node to its representative and the `@union@` function records in the disjoint sets data structure that two equivalence classes must be joined into one. The notation $u[i]$ denotes the target of the i th out-edge. The notation P_u denotes the set of all predecessors of the vertices equivalent to u in the DAG. To compute this set efficiently, the DAG must contain bidirectional links for each edge.

For our purposes, the `@union@` operation needs to be biased towards selecting more-specific representatives. For instance, when placing a template type parameter T and `int` in the same equivalence class, `int` should become the parent of T . Thus for values of type T we will find the built-in arithmetic operations for integers.

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

```

template<typename Iiter>
concept InputIterator
{
  typename difference_type;
  require SignedIntegral<difference_type>;
  // ...
};

```

```

merge(u,v) {
  if (find(u) == find(v))
    return;
  union(u,v);
  k = outdegree(u);
  if (label(u) != c<...>:t) // skip associated types
    for i=1...k
      merge(u[i], v[i])
  for each (x,y) such that x in P_u(G) and y in P_v(G)
    if (find(x) != find(y) and congruent(x,y))
      merge(x, y);
}
congruent(u,v) {
  return label(u) == label(v)
  && for i=1...outdegree(u). find(u[i]) == find(v[i])
}

```

Figure 4: Nelson Oppen congruence closure algorithm.

```

template<typename Iiter1, typename Iiter2>
  where { InputIterator<Iiter1>, InputIterator<Iiter2>,
          InputIterator<Iiter1>::difference_type == InputIterator<Iiter2>::difference_type }
  void wibble(Iiter1 first1, Iiter1 last1, Iiter2 first2, Iiter2 last2);

```

The compiler for \mathcal{G} stores types in a DAG and determines type equality using the Nelson Oppen congruence closure algorithm, with quite satisfactory results. Of course, with this compiler we had the luxury of designing it from the beginning with same-type constraints in mind.

Implementing same-type constraints in the context of GCC, on the other hand, was a considerable challenge. The main reason is that GCC’s data structures for representing types do not match the above described DAG. Changing such a low-level and pervasive data structure is quite difficult, for it requires widespread changes, so our first approach to implementing same-type constraints sacrificed much of the efficiency of the congruence closure algorithm in favor of requiring fewer changes to the data structure for types.

Several decisions in the design of GCC made the introduction of same-type constraints more complicated than we would have hoped. In particular, the sharing of type nodes in GCC is inconsistent. Sometimes GCC duplicates type nodes so that, e.g., there may be several type nodes for `int`. Thus, comparison of types may require deep structural comparisons, e.g., comparing two pointers to `int` nodes requires comparing the pointers themselves, the cv-qualifiers, and (recursively) comparing the pointed-to types.

On the other hand, GCC sometimes shares type nodes across different templates in ways that make using a `parent` field unsafe, requiring that the `parent` field be stored separately, e.g., in a hash table specific to the template being parsed. Comparison of template parameters T and U in GCC is based on their kind (type, template, non-type) and position in the template parameter list. This rule is used to match template definitions to declarations (where parameter names aren’t significant). However, using this as a general type-equality rule leads to some interesting behavior. In the following example the type nodes for `vector<T>` and `vector<U>` will be identical because T and U are the same by the above criteria.²

```

template<typename T> void foo(vector<T> x);
template<typename U> void bar(vector<U> x);

```

The sharing of template types from one template to the next means that we could not extend types in GCC with the necessary `parent` field. Instead, we create a hash table containing the `parent` of each type. It also means that our hash function and equality function have to take into account that there may exist distinct nodes that are actually equivalent. For instance:

- Associated types and `typename` types, e.g., `typename Foo<T>::bar_type`, are hashed based on the name `bar_type` and we perform a deep comparison on `Foo<T>` (since there may be a `Foo<U>` that is equivalent to `Foo<T>`).

²This also results in some very confusing diagnostics, because one can mention U in the source code but have the compiler refer to T !

- Template parameters are matched based on index (into the template parameter list), level, and opacity. This is because `Foo<T>` may be shared with `Foo<U>` (from a different template!), and `T` is a different node from `U` that is in a sense equivalent.

Once we have the hash table, we need to decide how to propagate same-type constraints around the DAG. This illustrates the second problem with GCC's representation: there are no backlinks from, e.g., `T` to `vector<T>`. Thus, we only propagate downward in the type DAG, e.g., from `T* == U*` to `T == U`. All other comparisons rely on the deep comparisons already implemented by GCC (the `comptypes` routine in particular). The downward-propagation logic additionally verifies that the same-type constraints are valid (e.g., one cannot make a reference the same as a pointer, or make an `int` the same as a `float`).

The implementation of same-type constraints in the ConceptGCC is neither elegant nor efficient, but it is serviceable. The ConceptGCC compiler is slow due to same-type constraint checking, a problem that we will address in the near future and will report back. We expect that a more aggressive scheme for caching same-type comparisons and for limited upward propagation of constraints will improve performance drastically. However, it should not be assumed that because ConceptGCC is inefficient in its handling of same-type constraints, that same-type constraints cannot be implemented efficiently. The congruence closure algorithm does permit efficient implementation, and could be achieved in an existing compiler by carefully examining type-equality issues.

4.4 Archetypes

Concepts may include requirements that a particular type contain certain constructors, member functions, or types. These types—which may be template parameters or associated types—typically don't have fields for members within their data structures. Even if they did, we wouldn't necessarily want to use them: only the representative of an equivalence class should have these members defined, otherwise they would be replicated many times for many different names of the same type.

To solve this problem, ConceptGCC uses *archetypes*. An archetype is a type that embodies the minimal requirements of a concept, i.e., it contains only the members that are mandated by the concepts it models. For instance, consider the following function template `bar()`:

```
template<typename T>
concept Fooable
{
    int T::foo();
};

template<typename T, typename U > where {Fooable<T>, T == U}
bool bar(T& t, U& u)
{
    return t.foo() == u.foo();
}
```

Here, the type `T` is required to have a member function `foo()` that takes no arguments but returns an `int`. Since `T == U`, `U` should be the same as `T` and therefore also have this member function. Thus, the body of function `foo()` can be type-checked properly. ConceptGCC does this by routing member lookups (e.g., for the function `foo()`) to the archetype itself. So, ConceptGCC builds an archetype `T'` that is equivalent to `T` (and, thus, `U`), but is a full-fledged class with a member function `foo`. A lookup in `T` becomes a lookup in `T'`, which can be resolved immediately. Those readers familiar with concept checking may note that archetypes as described here are equivalent to the archetypes used to verify the correctness of generic functions in C++03 [SL00].

Archetypes have an interesting place in ConceptGCC. For each equivalence class there exists a single archetype. The archetype is reachable from the representative of that class, and the parent of the archetype is the representative. However, the archetype is *not* the representative of the equivalence class, because archetypes do not depend on any template parameters. In essence, the archetype is always available but is only accessed when its members need to be searched. At all other times archetypes are ignored by the compiler.

The archetype mechanism has a second purpose in ConceptGCC: when an instantiation of a template is needed for type-checking a generic function, we instantiate using archetypes instead of actual types. For instance, consider the `mismatch()` algorithm below:

```

template<InputIterator Iter1, InputIterator Iter2>
  where { EqualityComparable<InputIterator<Iter1>::reference,
                               InputIterator<Iter2>::reference> }

  pair<Iter1, Iter2>
  mismatch(Iter1 first1, Iter1 last1, Iter2 first2)
  {
    while (first1 != last1 && *first1 == *first2)
    {
      ++first1;
      ++first2;
    }
    return pair<Iter1, Iter2>(first1, first2);
  }

```

How can the compiler verify that `pair<Iter1, Iter2>` has a constructor accepting values of types `Iter1` and `Iter2`? ConceptGCC implements this by replacing the template arguments to `pair` with their archetypes, then instantiates `pair<Iter1', Iter2'>`. Once instantiated, `pair<Iter1', Iter2'>` becomes the archetype for the equivalence class containing `pair<Iter1, Iter2>`, so that the search for a constructor finds an appropriate constructor.

Could archetypes be avoided in a concept-enabled compiler? It is possible they could, if any type node could have a list of members attached to it. Additionally, one would need to be able to instantiate a template whose arguments are themselves template parameters, e.g., `pair<Iter1, Iter2>`. Based on our evaluation of GCC, the use of archetypes was less upsetting to the structure of GCC than would be the ability to “instantiate” templates with template parameters in the argument list.

5 Compilation models

One of the first questions users are likely to ask when concepts are introduced is “Do we get separate compilation of templates?”. The answer to this is a bit fuzzy: concepts can make separate compilation of templates happen, with more success and less burden on implementors than **export**, but one must really tie down what “separate compilation” means. This section describes how instantiating constrained, type-safe templates can be implemented and discusses three different models for template instantiation that are well-supported by concepts. No single model of instantiation is best for all purposes: some favor executable performance at all costs, including higher compilation time, whereas others enable much more rapid development at the cost of slower, larger executables.

5.1 Instantiating constrained templates

Instantiating constrained templates is very different from instantiating unconstrained templates. When constrained templates are parsed, all name lookups are performed immediately and the template’s abstract syntax tree is complete except for substituting actual template arguments for template parameters and selecting specializations. For instance, consider the following type-safe function template `sum()`:

```

template<ForwardIterator Iter>
  where { Addable<value_type>, Regular<value_type> }
  value_type sum(Iter first, Iter last, value_type init)
  {
    while (first != last) {
      init = init + *first;
      ++first;
    }
    return init;
  }

```

Since all name lookups are performed when the template is first parsed, the compiler can resolve these calls as calls into the model. Thus, `sum()` can be rewritten as:

```

template<ForwardIterator Iter>
  where { Addable<value_type>, Regular<value_type> }
  ForwardIterator<Iter>::value_type
  sum(Iter first, Iter last, ForwardIterator<Iter>::value_type init)
  {

```

```

// typedefs for brevity of presentation, only
typedef ForwardIterator<Iter>::value_type value_type;
typedef ForwardIterator<Iter>::reference reference;
while (ForwardIterator<Iter>::operator!=(first, last)) {
    // Was: init = init + *first
    Regular<value_type>::operator=
        (iter,
         Addable<value_type>::operator+
            (init,
             Convertible<reference, value_type>::operator value_type
                (ForwardIterator<Iter>::operator*(first))));

    // Was: ++first
    ForwardIterator<Iter>::operator++(first);
}
// Calls the copy constructor to return the value
return Regular<value_type>::value_type(init);
}

```

In this expanded representation of `sum()`, every single operation performed on values that use template parameters (e.g., `first`, `last`, `init`) is now performed through model operations. This translation permits type-checking of the template, because models must provide precisely the same signatures as the concepts they implement. The translation also ensures that instantiation of the template produces calls *through the model* for each operation, so that models can adapt syntax. One can also instantiate a template for a certain set of template parameters given only the model definitions needed by the template. For instance, in the following code we can instantiate `sum<int*>`:

```

concept ForwardIterator<int*>
{
    typedef int value_type;
    int*& operator++(int*&);
    int& operator*(int* const&);
    // ==, !=, constructors, etc...
};

concept Addable<int>
{
    int operator+(const int&, const int&);
};

```

Note that we have omitted the definitions for operations in models `ForwardIterator<int*>` and `Addable<int>`: they could be provided by another translation unit, then `sum<int*>` would call them to perform the dereference, advance, or addition operations. If available within the current translation unit, the model operations could be inlined, permitting the same performance as unconstrained, unchecked templates or non-template code.

Although name lookup does not need to be performed during instantiation, we will need to select among specializations for function calls and templates. For instance, `binary_search()` calls `advance()` within its body:

```

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

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

template<BidirectionalIterator Iter, typename T>
where { LessThanComparable<reference, T> }
bool binary_search(Iter first, Iter last, T value)
{
    // ...
    advance(first, n);
}

```

```

    // ...
}

```

Note that we consider the second `advance()` a specialization of the first. In this case, how do we instantiate `binary_search<int*, int>?` When we reach the call to `advance()`, we perform partial ordering of function templates and determine that the second `advance()` applies and is more specialized. Note that this selection requires that we know of a model `RandomAccessIterator<int*>` even though it is not part of the requirements of `binary_search()`. Thus, while concepts have reduced the amount of “global knowledge” we need to perform instantiation (from “the entire translation unit” to “the set of model definitions”), we have not completely isolated the task of instantiation from the environment.

5.2 Inlined templates

Most C++ compilers support the “inclusion model” of templates, where the definitions of templates need to occur wherever they are used. There is little to said about the inclusion model: it works unchanged with concepts, although instantiations now call through models for each of their operations. If the models are also inlined, the result is essentially equivalent to compilation of templates without concepts.

5.3 Separate instantiation

Concepts make separate instantiation simpler and more predictable. What we refer to as “separate instantiation” is similar in spirit to the prelinking steps used by some C++ compilers. When a compiler sees a call to a constrained function template, it builds and emits the models needed to satisfy the requirements of the template. It then leaves a marker noting that it should (at some point later in the compilation process) instantiate the generic function using those models, which can be done in a separate translation unit. Since name lookup need not be performed during instantiation, the complexities of merging two translation units together for late instantiation are eliminated. With name lookup out of the equation, `export` becomes less complicated.

5.4 Separate compilation

Concepts make true separate compilation possible, but it remains nontrivial to implement. The \mathcal{G} compiler implements true separate compilation (it does not even have the notion of instantiation). Although not all aspects of the implementation are directly applicable to C++ with concepts— \mathcal{G} does not support specialization, for instance, because it breaks type safety—it illustrates some of the design decisions and problems associated with separate compilation.

The basic model for implementing separate compilation is *dictionary passing*. Each concept is essentially turned into an abstract base class, with a virtual function for each operation requirement. A model is an implementation of that abstract base class, which overrides the virtual functions with implementations of each required operation. A separately-compiled generic function will accept parameters that are pointers to the abstract base classes corresponding to each concept requirement. When invoking a separately-compiled generic function, the implementation will generate instances to the implementations of that base class for each model, and will pass those “dictionaries” along with the other parameters to the function.

One important caveat for a stack-based language such as C++ is that the size of template parameters is unknown when the generic function is compiled. The immediate effect is that one cannot get a constant from `sizeof`, but there is a more pressing concern: values of this type cannot be allocated on the stack, because their size is unknown. The \mathcal{G} compiler, which compiles to C++, employs the `boost::any` class to turn stack-allocated values into heap-allocated values. It is hard—and may be impossible—to keep certain exception-safety guarantees within separately-compiled generic functions.

The existence of specializations in C++ provides another source of complication. For instance, separately compiling the `binary_search()` function above means that the compiler must generate code for run-time partial ordering of function templates based on the existence of certain models. Class template specializations offer yet another problem, because specializations need not have the same form as their primary templates. This issue is not fully resolved at this point in time, either for type-checking or separate compilation. \mathcal{G} avoids the issue by disallowing specializations.

6 Conclusion

The Indiana concepts proposal [GSW⁺05] has been prototyped in ConceptGCC with a very similar concept system implemented in the compiler for the \mathcal{G} language. Although not trivial, both implementations were developed within a matter of months by researchers familiar with concepts and Generic Programming, neither of which is a compiler implementor by trade. From our experience with these two compilers, we can confidently conclude that the Indiana concepts proposal can be implemented in real, existing C++ compilers in a reasonable amount of time. Moreover, we believe that benefits of implementing concepts in C++ far outweigh the cost, because concepts make it easier to write correct, safe templates and will make the Generic Programming paradigm accessible to all C++ programmers.

The ConceptGCC and \mathcal{G} compilers will be made available to members of the C++ committee and eventually as open source. The authors welcome comments, questions, and criticisms. ConceptGCC-specific questions should be directed to Douglas Gregor (dgregor@cs.indiana.edu); \mathcal{G} -specific questions to Jeremy Siek (jgs3847@cs.rice.edu).

7 Acknowledgments

Jeremiah Willcock annotated a large portion of the GNU C++ standard library implementation in ConceptGCC with **where** clauses and introduced many of the concepts used in ConceptGCC. Most impressive is that he did so without the aid of type checking for templates, which was not implemented in the compiler until much later. This work was supported by a grant from the Lilly Endowment and NSF grant EIA-0131354.

References

- [CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.
- [DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM (JACM)*, 27(4):758–771, 1980.
- [GCC05] GNU compiler collection. <http://www.gnu.org/software/gcc/>, 2005.
- [GSW⁺05] Douglas Gregor, Jeremy Siek, Jeremiah Willcock, Jaakko Järvi, Ronald Garcia, and Andrew Lumsdaine. Concepts for C++0x (revision 1). Technical Report N1849=05-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2005.
- [NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.
- [SL00] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.
- [Tar83] Robert Endre Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.