Simplifying C++0x Concepts

Author: Doug Gregor
Document number: N3629
Date: 2013-04-09
Project: Programming Language C++, Evolution Working Group
Reply-to: Doug Gregor <doug.gregor@gmail.com>

Introduction

C++0x concepts provided a comprehensive solution for Generic Programming in C++, with an expressive system for expressing the requirements of templates, complete type checking for both template definitions and template uses, concept-based overloading, and syntactic remapping via concept maps. However, concepts failed as a language feature due to its overwhelming complexity, both in the language and in its use in the library. This paper dissects the reasons for this complexity and makes concrete suggestions that drastically reduce it while maintaining the essence of concepts. Each suggestion is accompanied by a discussion of the trade offs involved, for example, what kinds of programs could be written elegantly with C++0x concepts that become less elegant or impractical after that simplification.

Motivation

Concepts lite is an alternative, simplified form of concepts intended to provide many of the benefits of concepts without the complexity of the full C++0x design. Unfortunately, concepts lite misses the mark in several important ways:

- It only attempts to solve the easy half of the type checking problem: concepts lite only provides checking of template uses; template definitions remain completely unchecked.
- It doesn't solve the problem of spectacularly poor error messages: template instantiation error messages are poor because the compiler does not know whom to blame when the implicit contract between template user and author breaks down. Concepts lite lets one specify the contract, but because only the use side is verified mechanically, instantiation-time errors will still persist with alarming frequency.
- It does not provide language support for Generic Programming: concepts lite doesn't provide a constraint language for describing concepts, so the core vocabulary of Generic Programming is absent from this feature. Programmers must still use the vocabulary of template tricks (traits, specialization, etc.) to express their ideas.
- It purports to provide a partial solution that will pave the way to full concepts in the future. However, there is no clear path to that solution, and the authors admit that such a solution will likely require completely new syntax for describing constraints.

Type-Checking C++0x Concepts

Much of the implementation complexity of C++0x concepts comes from the type checking model. Given a constrained template such as the following

```
template<typename T>
requires LessThanComparable<T>
const T& min(const T &x, const T &y) {
  return y < x? y : x;
}
```

Where `LessThanComparable` is defined as the following concept:

```
concept LessThanComparable<T> {
  bool operator<(const T &x, const T &y);
}
```

The implementation performs a complete type check of the template definition. The only operations the template definition may use are those specified by the concept (e.g., `operator<` on `T`s), generic operations in the context whose parameters are a subset of the template's requirements, and built-in operations (e.g., `?:`). Anything else is a violation of the stated requirements (`LessThanComparable<T>`), and will be diagnosed as an error. The result of a successful type check transforms the function body to use calls into the concept itself, e.g.,

```
template<typename T>
requires LessThanComparable<T>
const T& min(const T &x, const T &y) {
  return LessThanComparable<T>::operator<(y, x)? y : x;
}
```

On the caller side, the user must provide a concept map that specifies how the template argument (say, `int`) provides each of the operations of each required concept.[1] For example, `LessThanComparable<int>`:

```
concept_map LessThanComparable<int> {
  bool operator<(const int &x, const int &y) {
    return x < y; // uses built-in operator <
  }
}
```

The compiler checks both the presence of the required concept maps and that each concept map satisfies all of the requirements of the corresponding concept. Any errors here cause either the use of the template or the definition of the concept map, respectively, to be rejected.

The final piece of the type checking puzzle is the template instantiation process. During instantiation, references into a concept requirement (e.g., `LessThanComparable<T>::operator<`) are replaced with references to a specific function in the corresponding concept map (`LessThanComparable<T>::operator<`). Because the only operations allowed in the template definition are those from the concept requirements, and concept maps are guaranteed to provide the same function signatures as the concepts they satisfy, template instantiation cannot fail.[2]

The guarantee that template instantiations will succeed is key to the usability of concepts: errors are diagnosed in the code that erred, because both sides of the template user/author contract are enforced by the compiler. Moreover, writing a correct template definition actually becomes *easier*, because the compiler helps you make sure you get all of the syntax, types, and requirements correct.

---

[1] That most concept maps are implicitly generated by the compiler, or partially generated, is irrelevant to the type checking model. It is, however, extremely important for the usability of concepts.

[2] There are a few intentional holes in this model for class template (partial) specializations, instantiation-time overload resolution, and the generally-disliked `late_check`, but the core model provides the instantiation-safety guarantee.

The Simplified Model

The proposed simplified C++0x concepts model relies on the following five simplifications.

#1: Eliminate concept maps

Concept maps are a powerful feature, allowing one to make a type satisfy the requirements of a concept without changing the concept at all. For example, consider a simple `Stack` concept:

```
concept Stack {
  typename value_type;

  void Stack::push(const value_type &value);
  void Stack::pop();
  void Stack::empty() const;
  const value_type &Stack::top() const;
}
```

One can fairly easily write a type that conforms to the `Stack` concept (e.g., `std::stack`). However, concept maps allow one to make an existing container (here we use `std::vector`) into a `Stack` without having to introduce an adaptor type like `std::stack`:

```
template<typename T>
concept_map Stack<std::vector<T>> {
  typedef T value_type;

  void std::vector<T>::push(const T &value) { push_back(value); }
  void std::vector<T>::pop() { pop_back(); }
  // std::vector<T>::empty() is already suitable
  const T &std::vector<T>::top() const { return back(); }
}
```

The interface of `std::vector` is unchanged by the addition of this concept map, except that a `std::vector` can now be used with any template that expects a `Stack`: no adaptor required. If we had a BackInsertionConcept concept, as follows, this concept map could be further generalized:

```
concept BackInsertionContainer<C> {
  typename value_type;

  void C::push_back(const value_type &);
  void C::pop_back(const value_type &);

  value_type &C::back();
  const value_type &C::back() const;

  bool C::empty() const;
}

template<BackInsertionContainer C>
concept_map Stack<C> {
  typedef BackInsertionContainer<T>::value_type value_type;

  void C::push(const T &value) { push_back(value); }
  void C::pop() { pop_back(); }
  // C::empty() is already suitable
  const T &C::top() const { return back(); }
```

```
}
```

While this is a powerful feature for composing generic libraries, it also greatly increases the surface complexity of concepts: it is no longer possible to think of concepts as "just" being more type checking for templates. Instead, one must always consider that concept maps could have some non-trivial mapping, which might make code harder to comprehend. Moreover, the presence of concept maps complicates the interaction with unconstrained templates, because the syntactic mapping in a concept map would get "dropped" when a constrained template called into an unconstrained template.

Eliminating the syntactic remapping ability of concept maps simplifies the user model for concepts considerably, especially reducing its surface complexity. It also simplifies the interaction with unconstrained templates. However, it does not (by itself) make implementing concepts all that much simpler, because removing concept maps does not actually change the type checking model.

#2: Type checking with archetype instantiation

The type checking model described in the third section, while fairly easy to reason about, is particularly hard to implement in today's C++ compilers. The basic problem is that C++ compilers tend not to type-check template definitions very thoroughly: some skip validation of the template definition entirely, opting to cache the preprocessed tokens to be re-parsed at template instantiation time, while others perform a partial type-check to produce an Abstract Syntax Tree for the template that will be realized with each instantiation. Either way, transitioning existing compilers from their existing template-parsing model to one that fully type-checks every aspect of a constrained template definition (while tracking how each operation maps back to a concept requirement) is a *huge* undertaking. Experience with the ConceptGCC prototype showed that while most concepts features are fairly straightforward (including checking concept maps, checking the use of constrained templates, concept-based overloading, refinement, etc.), the type-checking of template definitions was an order of magnitude more complicated than all of the others combined.

Instead, I propose a simpler model of type checking, based on David Abrahams' use of archetypes to check polymorphic lambdas. While this model does not guarantee that instantiations will not fail, it covers nearly all cases programmers are likely to encounter. In this model, the type check of a template definition simply performs an instantiation with a special set of template arguments that are *archetypes*. An archetype is a type that exhibits the minimal set of operations required of a type parameter. Returning to our `min` example, the archetype `T'` for `T` might look like this:

```
class T' {
   T'() = delete;
   T'(const T' &) = delete;
   T'(T' &&) = delete;
   T' &operator=(const T' &) = delete;
   T' &operator=(T' &&) = delete;
   ~T'() = delete;
  void operator&() const = delete;

public:
  // From LessThanComparable<T>:
  friend bool operator<(const T' &x, const T' &y);
};
```

Note that all of the implicitly-generated special member functions have been suppressed, `operator&` has been deleted, etc. This archetype is the "worst case" template argument for `min`, and if `min<T'>` is a well-formed instantiation, the `min` definition type-checks. Let's consider a slightly modified min:

```
template<LessThanComparable T>
T min2(T x, T y) {
  return y < x? y : x;
}
```

The archetype is the same, but instantiating `min2<T'>` will result in an error because `T'` has a deleted copy constructor. Adding `CopyConstructible` and `Destructible` requirements changes the archetype (call it `T''`) to the following:

```
class T'' {
   T''() = delete;
   T'' &operator=(const T'' &) = delete;
   T'' &operator=(T'' &&) = delete;
  void operator&() const = delete;

public:
  // From CopyConstructible<T>:
  T''(const T' &);
  // From Destructible<T>:
  ~T''();
  // From LessThanComparable<T>:
  friend bool operator<(const T'' &x, const T'' &y);
};
```

Instantiating `min2<T''>` would then succeed.

The main advantage of this approach is simplicity: constructing a set of archetypes based on the requirements of a constrained template is straightforward, and instantiation with an archetype is identical to instantiation with any other type (except perhaps for some customization of diagnostics).[3] Specifically, implementing this model of type checking requires no changes to the way in which a compiler implements parsing of template definitions, nor any other major structural changes.

The downside of this simplified model is that admits errors into the template instantiation process. For example, consider a simple `find_if` algorithm:

```
template<InputIterator Iter, typename P>
requires Predicate<P, InputIterator<Iter>::value_type>
Iter find_if(Iter first, Iter last, P pred) {
  while(first != last && !pred(*first))
    ++first;
  return first;
}
```

Next, let's consider an "evil" predicate with an evil bool-like type that has an overloaded operator:

```
struct EvilBool {
  operator bool() const;
  EvilBool operator!() const;
  friend void operator &&(bool, EvilBool);
};
```

---

[3] The Boost Concept Check library provides hand-written archetypes that one can use to test one's own templates using similar techniques. However, writing archetypes by hand is extremely tedious and quite error-prone.

```
template<typename T>
struct EvilPred {
  EvilBool operator()(const T &value) const;
};
```

Instantiating an unconstrained `find_if` with `EvilPred<some_value_type>` will obviously cause an instantiation-time error, because `EvilBool`'s `operator==` has a `void` return type. It will also break the simplified type checking model described here, because the archetypes don't account for this behavior, and template instantiation remains the same as in today's unconstrained templates.

The C++0x concepts type checking model makes this code work predictably, because every operation is rewritten into an operation on a concept requirement:

```
template<InputIterator Iter, typename P>
requires Predicate<P, InputIterator<Iter>::value_type>
Iter find_if(Iter first, Iter last, P pred) {
  while(InputIterator<Iter>::operator!=(first, last) &&
        !Predicate<P, InputIterator<Iter>::value_type>
           ::operator()(InputIterator<Iter>::operator*(first)))
    InputIterator<Iter>::operator++(first);
  return first;
}
```

Because `Predicate`'s `operator()` is specified to return a `bool`, the rewritten call to `operator()` is guaranteed to return a `bool`, so `EvilBool`'s `operator&&` will not be picked at instantiation time, eliminating the problem.

The extent to which potential problems of this nature are a problems in practice is unknown. Some standard library implementations have changed their `find_if` definitions (as well as other templates) to avoid these problems, although it is unclear whether such changes were motivated by actual, reasonable user code.

#3: Eliminate concept refinement semantics

C++0x concepts provided both *refinement* and *nested requirements* to compose concepts together. Refinement is meant for hierarchical relationships, where one concept "is-a" another concept. For example, a random access iterator is-a bidirectional iterator:

```
concept RandomAccessIterator<typename Iter>
  : BidirectionalIterator<Iter> { }
```

Nested requirements are meant more for requirements on associated types. For example, the `iterator` type of a container must be a `ForwardIterator` with the same `value_type` as the container.

```
concept Container<typename C> {
  typename iterator;
  typename value_type;
  requires ForwardIterator<iterator>;
  requires SameType<ForwardIterator<iterator>::value_type,
                    value_type>;
}
```

Despite the intended differences in usage, the two features are quite similar semantically... but not identical. Specifically, refinement requires that identical requirements that occur in multiple concepts within the refinement hierarchy have identical implementations in their concept maps (to support is-a).

However, this distinction disappears when concept maps can no longer remap syntax, per simplification #1. As such, there is no cost to eliminating the semantic differences between concept refinement and semantics. One could also eliminate the concept refinement syntax (because nested requirements are more general), which would reduce surface complexity somewhat but would not allow concept authors to express the notion of a concept hierarchy directly.

#4: Eliminate axioms

Axioms in C++0x concepts are a way to describe the semantics of concepts. For example, the `CopyConstructible` requirement would have an axiom specifying that the result of copying a value is the same as the original value:

```
concept CopyConstructible<typename T> {
  T::T(const T&);

  axiom CopyResult(T x) {
    T(x) == x;
  }
}
```

Theoretically, one could use axioms to implement various compiler optimizations (e.g., copy propagation for class types) and automated testing tools (e.g., randomly test whether values of a given type meet the axioms). However, such tools are a long way off, and without having those tools, it is likely that the axiom language provided by C++0x concepts will not be sufficient to support them. Given that, the downside of removing axioms seems fairly small.

#5: Make concepts implicit by default

By default, C++0x concepts are explicit, requiring one to (by default) write a concept map to state that a particular type meets the (syntactic and semantic) requirements of concepts. By removing syntactic remapping in concept maps (simplification #1) and removing any way to express semantics in concepts (simplification #4), we eliminate much of the motivation for this choice. Swapping the defaults (so `concept` allows implicit conformance and `explicit concept` requires explicit conformance) reduces the surface complexity, while still retaining the ability to deal with problematic cases (such as the syntactic collision between `InputIterator` and `ForwardIterator`).

Conclusion

C++0x concepts failed because of their complexity, but inside that large feature is a kernel of functionality that can make templates easier to write and use, and provide first-class support for the Generic Programming paradigm in C++. This paper proposes to strip out the inessential parts of C++0x concepts and replace the type-checking model with one that is less comprehensive but easily implementable in today's C++ compilers, drastically simplifying both the surface complexity and the implementation complexity of concepts. The resulting concept system maintains the core functionality of C++0x concepts, including great support for the Generic Programming paradigm, type checking for both template uses and definitions, and greatly improved error messages.