Document number P0717R1

Date 2017-07-24

Authors Richard Smith <richard@metafoo.co.uk>
Andrew Sutton <asutton@uakron.edu>

Audience None (for posterity)

Semantic constraint matching for concepts

Changes since R0

- Acknowledgement added

Problem

In the Concepts TS, various situations emerge where we would like to compare two sets of constraints (perhaps from two different templates in an overload set, or when determining whether one declaration is a redeclaration of another). We would like to view constraints semantically as predicates, but that approach is incomplete: we need a mechanism to identify whether an atomic portion of one constraint is the same as an atomic portion of another constraint.

The approach taken to this problem in the Concepts TS is to use C++'s "equivalence" rules for templates. These rules can be summarized as follows, when applied to two dependent expressions in constraints:

- Compare the token sequences of the two expressions (where differences in the token sequences due to different identifiers being used to name a template (or function) parameter are ignored)
- If the token sequences are identical, the expressions are equivalent.
- If the token sequences are not identical but have the same meaning for all possible template arguments, the expressions are "functionally equivalent but not equivalent", and any program that results in such a comparison being attempted is ill-formed, with no diagnostic required.
- Otherwise (the token sequences are not identical and there are some actual template arguments for which they mean different things), the expressions are non-equivalent.

The third bullet exists in order to avoid unreasonably burdening implementations with tracking the original token sequences: semantics-preserving canonicalization is permitted prior to performing the comparison.

There are several issues with the approach taken by the Concepts TS:

- The Concepts TS requires an implementation to treat the “functionally equivalent but not equivalent” cases as simply being non-equivalent (thus using a token comparison in all cases). This burdens implementations with tracking exact token sequences. The prototype implementation of the Concepts TS does not do this, and the implementation cost is considerable for some implementations.
- When ordering independently-defined concepts, comparison of the token sequence is brittle and burdensome to programmers: not only are the required semantics of a concept part of an API, the exact token sequence used to describe that concept is also part of the API, and any refactoring that modifies the token sequence is a source compatibility break.
- The rules support -- and to some extent encourage -- a programming model of concept refinement by textual coincidence, not semantic equivalence

Examples:

```
namespace X {
  template<C1 T> void foo(T);
  template<typename T> concept Fooable = requires (T t) { foo(t); };
}
namespace Y {
  template<C2 T> void foo(T);
  template<typename T> concept Fooable = requires (T t) { foo(t); };
}
```

`X::Fooable` is equivalent to `Y::Fooable` despite them meaning completely different things (by virtue of being defined in different namespace). This kind of incidental equivalence is problematic: an overload set with functions constrained by these two concepts would be ambiguous.

That problem is exacerbated when one concept incidentally refines the others.

```
namespace Z {
  template<C3 T> void foo(T);
  template<C3 T> void bar(T);
  template<typename T> concept Fooable = requires (T t) {
    foo(t);
    bar(t);
  };
}
```

An overload set containing distinct viable candidates constrained by `X::Fooable`, `Y::Fooable`, and `Z::Fooable` respectively will always select the candidate constrained by `Z::Fooable`. This is almost certainly not what a programmer wants.

The existing rules can also be confusing in other cases.

```
template<typename T> concept A1 = requires (int n) { T{n}; };
template<typename T> concept A2 = requires { T{declval<int&>()}; };
```

Neither of A1 and A2 subsume the other despite expressing the same semantic constraint. The non-relation of these concepts an artifact of the token equivalence rule and not because of some more principled reason.

Many similar examples exist: for example, two token-equivalent constraints could differ due to having different unqualified lookup results or different levels of access to class members.

Approach

Equivalence during partial ordering

We propose to use a different model for determining whether two atomic constraints are equivalent. Tersely, we can describe this as follows: two atomic constraints are equivalent only if they originate from the same source-level construct.

In order to understand this, it is important to understand that constraints undergo a normalization process before being compared. This normalization process expands references to concepts into their definitions, identifies the atomic constraints, and produces a boolean expression specified in terms of those atomic constraints. For example:

```
template<typename T> concept A = requires { T(); };
template<typename T> concept B = requires (T t) { t.~T(); };
template<typename U> concept X = A<U> && B<U>;
template<typename V> concept Y = B<V> && A<V>;
```

If we wish to compare the *constraint-expression*s `X<W>` and `Y<W>`, we first normalize both. This reduces `X<W>` to the boolean expression $r \wedge s$, and reduces `Y<W>` to the boolean expression $s \wedge r$, where r is the constraint “`requires { T(); }`, with `T = W`” and s is the constraint “`requires (T t) { t.~T(); }`, with `T = W`”. These boolean expressions are obviously equivalent, so the resulting normalized constraints are equivalent.

Under our proposal, the above normalization process would still be performed, so constraints like `X<W>` and `Y<W>` would still be equivalent (they both require default construction and destruction). However, if someone textually duplicated some portion of the constraints:

```
template<typename T> concept Z = requires { T(); } && B<T>;
```

… then `Z<W>` would produce a new concept that is neither more nor less specialized than `X<W>` and `Y<W>`. That is, `Z<W>`’s additional requirement on default construction is not equivalent to that of `A<W>`’s, even though its spelling is identical.

This approach was discussed at Kona, with the following direction guidance:

Identity of atomic constraints doesn't depend on token sequence? tons | dozen | 1 | 0 | 0

Equivalence of redeclarations

The Concepts TS allows different syntaxes to be used in multiple declarations of the same function template, relying on a textual rewrite rule to determine whether the declarations are equivalent. For example, the following two declarations can appear in the same program and declare the same function template:

```
void f(ConceptA a, ConceptB b, ConceptC c);

template<ConceptA A, ConceptB B, ConceptC C>
  void f(A a, B b, C c);
```

This violates the normal rule that dependent portions of a template are required to be written with the same syntax, as described above, and in so doing restricts the freedom of an implementation to model these different declarations differently -- an implementation is required to act as if it rewrites the compound constraints as a conjunction of the specified and inferred constraints to support this rule, potentially adding cost and complexity.

This rule does not address any common user need. There may occasionally be a desire to declare a function with one syntax for readability to their users and define it with a different syntax, but doing so requires understanding the details of an arcane token rewrite rule so we would not recommend it even to an expert user. As a consequence, we recommend removing this deviation from normal template rules and that the regular rule for C++ templates be used: all declarations of a function template must use the same syntax for dependent portions of a template, including in the specification of constraints on constrained function templates.

This approach was discussed at Kona, with the following direction guidance:

Restrict redeclarations to identical forms? 15 | 14 | 13 | 2 | 3

Implementation experience

The change to partial ordering rules has been experimentally implemented in a private fork of GCC. At the time of writing there are still some regressions when compiling the concepts-based implementation of the Ranges TS.

Beyond eliminating the conceptual and overloading problems discussed above, the proposed change makes atomic constraint comparison more efficient. Atomic constraints can be uniquely identified as a triple comprised of: the concept in which the constraint appears, the position of the constraint within the concept, and the template arguments used for normalization.

Acknowledgements

The core idea in this proposal, of using expression identity rather than token equivalence to determine equivalence of constraints, was originally suggested by Hubert Tong in reflector posting core-2016/10/1033.