Concept Implication and Requirement Propagation

Document number: N2646 = 08-0156
Authors: Peter Gottschling, Technische Universität Dresden
Date: 2008-05-16
Project: Programming Language C++, Evolution/Core Working Group
Reply to: Peter.Gottschling@tu-dresden.de

1 Introduction

Concept constraints give the programmer the opportunity to express all type requirements of a generic function. Only if all these requirements are fulfilled the function can be called. In addition, the compiler verifies whether or not the required concepts cover all structural requirements of the template function and all called function. The completeness of semantic requirements on the other hand cannot be verified by the compiler and the responsibility for it lies entirely in the hands of the programmers.

A considerable burden in programming with concepts is that each function must specify its own requirements and the union of all called functions' requirements. The worst case could be an exponential grow in constraint definition complexity with respect to the depth of the call tree. So far, only relatively simple functions were implemented with concepts. It is questionable if large-scale software, like finite element packages, can be realized with concepts in the context of accumulating function constraints.

The completeness of structural prerequisites in the template constraints has the huge advantage that erroneous calls are caught directly with the function call and that readable error messages can be generated. However, while the programmer is very busy to provide completeness of structural requirements, *the risk increases that semantic requirements will be forgotten!*

The necessity of defining the structural requirements in its completeness is problematic for three reasons:

1. The source becomes increasingly redundant. The fact that a certain type must be assignable or copy-constructable is already expressed in the function body. Requesting this in the function constraints is entirely redundant and does not improve the program source. *Thus, we tell the compiler a second time what it already knows.*

2. Every constraint of a generic function is repeated in all functions that call it. *Thus, we repeat the compiler arbitrarily often what it already knows.*

3. Which semantic requirements are needed by a generic function cannot be determined by the compiler. This *must* be specified by the programmer. *Thus, after all the repetition of known facts we risk to forget telling the compiler the really important issues that in turn it does not know.*

We therefore propose to automate the requirement specification of structural conditions, i.e.

Let the compiler do what it knows better than we do.

The programmer shall focus on requiring semantic conditions.

The programmer writes only semantic requirements but those must be given completely.

2 Examples

The following very simple function (see [lib.power] in Nxxxx) computes the power with respect to a functor. It uses an implementation for non-negative coefficients. If the coefficient is negative then the second function is called with the magnitude of the exponent and the inverse of the basis:

```
template <typename Op, std::Semiregular Element, Integral Exponent>
    requires Group<Op, Element>
          && std::Convertible<std::Callable2<Op, Element, Element>::result_type, Element>
inline Element power(const Element& a, Exponent n, Op op)
{
    return n < 0 ? multiply_and_square(Element(inverse(op, a)), Exponent(-n), op)
                 : multiply_and_square(a, n, op);
}
```

A slight modification of the source, i.e. removing the explicit conversion to Element and Exponent forces us to add a large number of additional requirements:

```
template <typename Op, std::Semiregular Element, Integral Exponent>
    requires Group<Op, Element>
          && std::Convertible<std::Callable2<Op, Element, Element>::result_type, Element>
          && std::Semiregular<math::Inversion<Op, Element>::result_type>
          && std::HasNegate<Exponent>
          && math::Monoid<Op, math::Inversion<Op, Element>::result_type>
          && Integral< std::HasNegate<Exponent>::result_type>
          && std::Callable2<Op, math::Inversion<Op, Element>::result_type,
                                math::Inversion<Op, Element>::result_type>
          && std::Convertible<std::Callable2<Op, math::Inversion<Op, Element>::result_type,
                                                 math::Inversion<Op, Element>::result_type>::result_type,
                                 math::Inversion<Op, Element>::result_type>
inline Element power(const Element& a, Exponent n, Op op)
{
    return n < 0 ? multiply_and_square(inverse(op, a), -n, op)
                 : multiply_and_square(a, n, op);
}
```

For two lines of code we need 11 lines of requirements. Most of the constraints only repeat what is already written in the function body.

Another example worth looking at is sort from STL. The STL documentation requires several structural and two semantic concepts to be modeled. If we look at the implementation in ConceptGCC:

```
template<MutableRandomAccessIterator _Iter>
  requires LessThanComparable<_Iter::value_type>
        && CopyAssignable<_Iter::reference, _Iter::reference>
        && Swappable<_Iter::value_type>
```

```
        && CopyConstructible<_Iter::value_type>
inline void sort(_Iter __first, _Iter __last) { /* ... */ }
```

we find all structural requirements but only one semantic condition (MutableRandomAccessIterator). The point here is not to criticize this implementation — on the contrary, the enrichment of STL with concepts is a tremendous improvement. We rather want to illustrate the statement from the introduction that we, the programmers, will put a lot of efforts in the structural requirements and risk to forget the semantic ones. It should be exactly the opposite, the structural conditions can be generated automatically and the program source only needs the semantic conditions.

In the following section we show how to automate the structural constraints.

3 Concept Implication [concept.impl]

We propose to add the new keyword **imply_concept** in order to automatically generate constraints for structural requirements. The extended list of concepts from N2501 is therefore:

asm	**continue**	**goto**	**register**	**throw**
auto	**default**	**if**	**reinterpret_cast**	**true**
axiom	**delete**	**imply_concept**	**requires**	**try**
bool	**do**	**inline**	**return**	**typedef**
break	**double**	**int**	**short**	**typeid**
case	**dynamic_cast**	**late_check**	**signed**	**typename**
catch	**else**	**long**	**sizeof**	**union**
char	**enum**	**mutable**	**static**	**unsigned**
char16_t	**explicit**	**namespace**	static_assert	**using**
char32_t	**export**	**new**	**static_cast**	**virtual**
class	**extern**	**operator**	**struct**	**void**
concept	**false**	**private**	**switch**	**volatile**
concept_map	**float**	**protected**	**template**	**wchar_t**
const	**for**	**public**	**this**	**while**
const_cast	**friend**			

Table 1: Keywords

3.1 Starting Point: Unsatisfied Structural Requirements [impl.unsat]

As mentioned before, the compiler knows already which structural requirements are needed from the analysis of the function body. We exemplify this statement with a simplistic scenario:

```
auto concept C<typename T> {}
template <typename T> requires C<T> T f(T x) { T y(x); return y; }
f(3);
```

The concept C is not really needed; it only serves to enable concept checking. Compiling this code with ConceptGCC yields the following error message:

```
imply_test.cpp: In function 'T f(T)' :
imply_test.cpp:15: error: constructor  'T::T(const T&)' is inaccessible
imply_test.cpp:15: error: constructor  'T::T(const T&)' is inaccessible
```

Apparently, the compiler recognizes correctly the missing structural requirement. The only remaining task is to associate this requirement with a suitable concept.

3.2 Association Between Structural Requirements and Concepts [impl.assoc]

In order to associate an unsatisfied structural requirement with an appropriate concept, we propose declaration similar to the **friend** declaration. For the example above, the solution can be to associate the templated signature with the concept CopyAssignable:

```
template <typename T>
  T::T(const T&)
imply_concept std::CopyConstructable<T>;
```

Discussion: The addition of any concept that contains copy construction to the list of constraints will cover the structural requirements. For the sake of genericity, the added constraints shall increase the number of requirements only minimally.

imply_concept declarations are in principle the dual entity of concept declarations. While the latter relate a concept to requirements, the former conversely relate a requirement in terms of a signature to a concept.

3.3 Explicit Usage of Requirement Implication [impl.expl]

Programmers shall have control when constraint template functions use implicit requirement generation and when all requirements are forced to be declared explicitly. Analogously to implicit **concept_map**s for structural concepts, we suggest to use the keyword **auto** to enable the requirement implication.

The example above will read:

```
template <typename T>
  auto requires
T f(T x)
{ T y(x); return y; }
```

With the **imply_concept** declaration above this is equivalent to:

```
template <typename T>
  requires std::CopyConstructable<T>
T f(T x)
{ T y(x); return y; }
```

3.4 Template Requirements with Implication [temp.req.impl]

The inclusion of the keyword **auto** in the requirement clause modifies the the specification from [temp.req] in N2501 to:

requires-clause:
`auto`$_{opt}$ `requires` *requirement-list*$_{opt}$
`auto`$_{opt}$ `requires (` *requirement-list* `)`

requirement-list:
requirement `...`$_{opt}$ `&&` *requirement-list*
requirement `...`$_{opt}$

requirement:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*
`!` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*

For cases of entirely generated requirements, the *requirement-list* is optional in our proposal.

To do: BNF for **imply_concept**.

4 More Declarations of Concept Implications [concept.impl.decl]

Disclaimer: The declaration in this section are not meant to be complete or sufficient; it is rather intended as starting point. Defining a sound set of declarations might need a separate proposal.

The need for a less-than operator implies:

```
template <typename T, typename U>
  bool operator<(const T&, const U&)
imply_concept LessThanComparable<T, U>;
```

The following three definitions are arguable because the operators' default implementations in LessThanComparable are wrong for partial ordering.[1] For now we refer only to the requirement that the operators exist, not to their default implementations.

```
template <typename T, typename U>
  bool operator>(const T&, const U&)
imply_concept LessThanComparable<T, U>;

template <typename T, typename U>
  bool operator<=(const T&, const U&)
imply_concept LessThanComparable<T, U>;

template <typename T, typename U>
  bool operator>=(const T&, const U&)
imply_concept LessThanComparable<T, U>;
```

Similarly we imply EqualityComparable:

[1] The concept should either remove the **auto** attribute or the default implementations because strict weak order cannot be guaranteed in general. On the other hand, one can argue that partial ordering are rarely used and that in such a rare case the operators still can be defined explicitly. At the very least, the wording of this standard concepts needs a warning regarding partial orders.

```
template <typename T, typename U>
  bool operator==(const T&, const U&)
imply_concept EqualityComparable<T, U>;

template <typename T, typename U>
  bool operator!=(const T&, const U&)
imply_concept EqualityComparable<T, U>;
```

Types that can be default-constructed imply:

```
template <typename T>
  T::T()
imply_concept DefaultConstructible<T>;
```

Types that can be move-constructed imply:

```
template <typename T>
  T::T(const T&)
imply_concept MoveConstructible<T>;
```

Types that can be copy-constructed imply:

```
template <typename T>
  T::T(const T&)
imply_concept std::CopyConstructable<T>;
```

Types that are move-assignable imply:

```
template <typename T, typename U>
  auto T::operator=(U&&)
imply_concept MoveAssignable<T, U>;
```

Types that are copy-assignable imply:

```
template <typename T, typename U>
  auto T::operator=(U&)
imply_concept CopyAssignable<T, U>;
```

Types that are swappable imply:

```
template <typename T>
  void swap(T&, T&)
imply_concept Swappable<T>;
```

Types with new operator imply:

```
template <typename T>
  void* T::operator new(size_t size)
imply_concept Newable<T>;
```

Types with delete operator imply:

```
template <typename T>
  void T::operator delete(void*)
imply_concept Deletable<T>;
```

Types with array-new operator imply:

```
template <typename T>
  void* T::operator new[](size_t size)
imply_concept ArrayNewable<T>;
```

Types with array-delete operator imply:

```
template <typename T>
  void T::operator delete[](void*)
imply_concept ArrayDeletable<T>;
```

Converting type T in type U implies:

```
template <typename T, typename U>
  operator U(const T&)
imply_concept Convertible<T, U>;
```

Types that need addition of l-values — e.g., x + y or x + (y*= z) — imply:

```
template <typename T, typename U>
  auto operator+(const T&, const U&)
imply_concept HasPlus<T, U>;
```

The addition of an l-value and an r-value — e.g., x + f(y), x + y*z or the first addition in x + (y + z) — implies:

```
template <typename T, typename U>
  auto operator+(const T&, U&&)
imply_concept HasPlus<T, U>;
```

Types that need addition of an r-value and an l-value — e.g., f(x) + y, x*y + z, or the second addition in x + y + z — imply:

```
template <typename T, typename U>
  auto operator+(T&&, const U&)
imply_concept HasPlus<T, U>;
```

Types that need addition of r-values — e.g., f(x) + g(y) — imply:

```
template <typename T, typename U>
  auto operator+(T&&, U&&)
imply_concept HasPlus<T, U>;
```

Automatically generating constraints for structural requirements eliminates the major part of declaration redundancy in constrained template functions. In the next section, we will show how the remainder of redundant requirements in the constraining clauses can be removed.

5 Propagation of Semantic Requirements [req.prop]

Semantic requirements cannot be generated automatically like structural ones. Constraining functions with concepts that contain semantic properties is per se not redundant. Redundancy is created when a function, say f, with semantic conditions is called by other constrained functions. Then, all callers of f must include f's semantic concepts in their requirement list for the sake of completeness.

We propose that semantic requirements are automatically propagated to calling template functions when the latter have an **auto requires** clause. For instance:

```
template <typename T>
  requires Semantic<T> // some semantic concept
T f(T x) { /* ... */ }

template <typename T>
  auto requires // Semantic<T> is propagated from f
T f2(T x) { f(x); /* ... */ }
```

The type specification is adapted to the context of the calling function, e.g.:

```
template <typename T>
  requires Semantic<T> // some semantic concept
T f(T x) { /* ... */ }

template <ForwardIterator Iter>
  auto requires // Semantic<Iter::value_type> is propagated from f
T f2(Iter p) { f(*p++); /* ... */ }
```

Resuming, every semantic requirement *must* be defined. But only once! The repetition of these requirements in all calling template functions can be generated automatically.

6 Impact on the Example Functions

Applying the new techniques to the introductory examples reduces their code complexity dramatically. The power function for Groups now reads:

```
template <typename Op, typename Element, Integral Exponent>
    auto requires Group<Op, Element>
inline Element power(const Element& a, Exponent n, Op op)
{
    return n < 0 ? multiply_and_square(inverse(op, a), -n, op)
                 : multiply_and_square(a, n, op);
}
```

The semantic prerequisite math::Monoid<Op, math::Inversion<Op, Element>::result_type> is propagated from the calling function and all other requirements were purely structural and can be generated by concept implication.

In case of the sort function we add the semantic requirement that **operator**< constitutes a strict weak ordering. On the other hand, we remove all purely structural conditions:

```
template<MutableRandomAccessIterator _Iter>
  auto requires LessThanThanStrictWeakOrdering<_Iter::value_type>
inline void sort(_Iter __first, _Iter __last) { /* ... */ }
```

In fact, a short look at the implementation reveals that even these two requirements can be propagated from the called functions so that we can omit all explicit concept requirements:

```
template<typename _Iter> auto requires
inline void sort(_Iter __first, _Iter __last) { /* ... */ }
```

For the sake of better source code documentation, we nevertheless favor the first implementation with explicit declaration of semantic prerequisites.

7 Open Questions

Are similar techniques useful for constraining template classes?

With all the automatic constraint generation, introspection is desirable. Is this feasible?

8 Conclusions

We proposed two mechanism in this document: concept implication from unsatisfied structural requirements and requirement propagation to calling functions. These two techniques allow us to use the compiler to generate all purely structural requests and to require each semantic condition only once. The entire redundancy of function constraints can be eliminated. Stating important deductible requirements in the **requires** clause can be still desirable for the sake of source code documentation.

9 Acknowledgments

The author thanks Andrew Lumsdaine from Indiana University and Axel Voigt from Technische Universtität Dresden for supporting this work on concepts for the sake of the scientific computing community.

A Ordering Concepts [concept.order]

The most general ordering is a partial order. It is rarely used directly but its requirements belong to all order relations.

```
concept PartialOrdering<typename Comparison, typename T>
{
    axiom Irreflexivity(Comparison cmp, T x) {
        !cmp(x, x);
    }
    axiom AntiSymmetry(Comparison cmp, T x, T y) {
        !(cmp(x, y) && cmp(y, x));
    }
    axiom Transitivity(Comparison cmp, T x, T y, T z) {
        if (cmp(x, y) && cmp(y, z))
            cmp(x, z);
    }
}
```

Strict weak ordering additionally requires that the complementary relation is transitive. This allows for introducing an equivalence relation where two elements x and y belong to the same equivalence class when neither cmp(x, y) nor cmp(y, x):

```
concept StrictWeakOrdering<typename Comparison, typename T>
  : PartialOrdering<Comparison, T>
{
    axiom ComplementaryTransitivity(Comparison cmp, T x, T y, T z) {
        if (!cmp(x, y) && !cmp(y, z))
            !cmp(x, z);
    }

    // Implies the following definition of equivalence classes
    bool equivalent(T x, T y) {
        return !cmp(x, y) && !cmp(y, x);
    }
}
```

Alternatively one can first define the relation of incomparable elements and then require the transitivity of this relation (the STL documentation is written in this form). StrictWeakOrdering is required by generic sorting methods in STL.

The strictest form of ordering is total ordering that requires for two elements that they equal when they are not comparable:

```
concept TotalOrdering<typename Comparison, typename T>
  : StrictWeakOrdering<Comparison, T>
{
    axiom Trichotomy(Comparison cmp, T x, T y) {
        // Can we request that x == y is defined appropriately?
        cmp(x, y) || cmp(y, x) || x == y;
    }
}
```

The concept LessThanPartialOrdering defines the same requirements as PartialOrdering with respect to the operator <:

```
concept LessThanPartialOrdering<typename T>
{
    axiom Irreflexivity(T x) {
        !(x < x);
    }
    axiom AntiSymmetry(T x, T y) {
        !(x < y && y < x);
    }
    axiom Transitivity(T x, T y, T z) {
        if (x < y && y < z)
            x < z;
    }
}
```

Adding the requirement of complementary transitivity yields:

```
concept LessThanStrictWeakOrdering<typename T>
  : LessThanPartialOrdering<T>
{
    axiom ComplementaryTransitivity(T x, T y, T z) {
        if (!(x < y) && !(y < x))
            !(x < z);
    }

    // Implies the following definition of equivalence classes
    bool equivalent(T x, T y) {
        return !(x < y) && !(y < x);
    }
}
```

Analogously defined is the total order:

```
concept LessThanTotalOrdering<typename T>
  : LessThanStrictWeakOrdering<T>
{
    axiom Trichotomy(T x, T y) {
        // Can we request that x == y is defined appropriately?
        x < y || y < x || x == y;
    }
}
```

All intrisic types (see [concept.intrinsic]) with an **operator**< are totaly ordered:

```
template <Intrinsic T>
  requires LessThanComparable<T>
concept_map LessThanTotalOrdering<T> {}
```

Accordingly, intrinsic types establish a total order with the less functor from STL:

```
template <Intrinsic T>
  requires LessThanComparable<T>
concept_map TotalOrdering<less<T>, T> {}
```

Thus, adding the semantic requirement of strict weak ordering to STL's generic sort functions does not require additional declaration effort from the user for sorting intrinsics.