Doc no: N2221=07-0081
Date: 2007-03-12
Reply-To: Bjarne Stroustrup
bs@cs.tamu.edu

An analysis of concept intersection

Bjarne Stroustrup Gabriel Dos Reis

Texas A&M University

Abstract

We present the notion of intersection of concepts (as initially presented in [DRS06] and [GS06]), give some motivating examples, explain workarounds, and give an algorithm for dealing with intersection which approaches zero cost for the simplest examples and the cost of the most naive approach for the most complicated examples.

1 Introduction

We see concept map intersection `||` as an integral part of the concept proposal. However, N2161 [GL07] raised doubts about its usefulness and implementability, so we thought it better to discuss it separately. This paper is that discussion, partly based on discussions at the "Google concept meeting" and a late evening exploration of the issues by Bjarne Stroustrup, Doug Gregor, and Gaby Dos Reis. Our conclusion is that concept intersection is more useful than we suspected and more easily implemented (an implementation was part of Gaby's concept prototype). Thus, `||` should be "part of concepts."

If you look at concepts as predicates on types (which is part of what they are) it is obvious that you can combine concepts using `&&`, `||`, and `!`. For example

```
template<typename T>
requires C1<T> || C2<T>
void f(const T& t);
```

That is, `f()` requires its argument type, `T`, to meet either the requirements of `C1` or those of `C2`. As stated, `C1<T>` and `C2<T>` need not have anything in common, but if they have not, `f()` cannot use any operations on `T`. What `f()` does to a `T` must be in the intersection of `C1<T>` and `C2<T>` – for some definition of intersection.

Please note that in the discussion below, we do not try to exactly match every detail of specific concepts as presented in various proposals. We are trying to present just what is necessary to discuss concept intersection.

2 Motivating examples

Imagine that you want to use argument types coming from libraries Lib1 and Lib2. In particular, you want to write a template function that takes as arguments types from two libraries that has each developed a notion of "and ordinary value type" and represented them as concepts. One way to proceed is to say that you'll accept both:

```
template<typename T>
requires Lib1::value_type<T> || Lib2::Regular<T>
void my_algorithm(vector<T>& v);
```

If my notion of an ordinary value type fits nicely into the intersection of their notions I can just use their notions. I don't have to develop my own notion. I don't have to understand the details of their notions. And, I don't have to keep my notion in sync with theirs as their notions evolve. Obviously, the compiler will sharply remind me if my guess about their concepts were wrong (so that I used operations on T not supported by those concepts) or if those concepts evolved in ways that didn't suit me.

When concepts become popular, this scenario will become common as different groups develop concepts independently. This will happen in many application domains, including many that the standards community will never get around to addressing.

2.1 value type example

First consider the "simple value type" scenario. We might have

```
namespace Lib1 {
        concept value_type<T> {
                typedef T value_type;
                T();
                T(const T&);
                T& operator=(T&, T);
                int hash(T);
        };
}
```

and

```
namespace Lib2 {
        concept Regular<T> {
                T();
                T(const T&);
                T& operator=(T&, T);
                bool operator==(T,T);
                bool operator!=(T a, T b) { return !(a==b); }
        };
}
```

There is basically no problem. `Lib1::value_type<T> || Lib2::Regular<T>` is simply a shorthand for what the user would have written after reading `Lib1::value_type<T>` and `Lib2::Regular<T>`:

```
concept Ordinary_value_type<T> {
        T();
        T(const T&);
        T& operator=(T&, T);
};
```

Now consider a use

```
template<typename T>
requires Ordinary_value_type<T>
void copy(vector<T> a, vector<T>& v)
{
        a = b;
}
```

Actually, this `Ordinary_value_type` requirement is not quite minimal. We over-specified by requiring the copy constructor. Using `||`, we don't run the risk of limiting the utility of our algorithm through over-specification.

2.2 Numeric example

There are many kinds of numbers. They all tend to have similar operations (e.g. + and *) with similar signatures. However, they can differ dramatically in the detailed semantics. For example, floating point numbers are not commutative (i.e. x*y can be different from y*x) but for a specific algorithm or for a slightly different implementation of floating point numbers it can be very advantageous to treat them as commutative. For example:

```
concept Integral<Value_type T> {
};

concept Associative_FP<Value_type T> {
};
```

Now, when we know that a given FP type can be treated as associative (possibly because of an explicit concept_map), we can treat is as associative, just like am integer.

We can imagine uses of `||` that keeps track of which alternative is chosen (eventhough the use is in the intersection) and generates different code for the alternatives to take advantage of differences in axioms. However, the use of axioms is implementation specific, so we don't make any suggestions or assumptions about such cases.

2.3 copy example

Consider `std::copy()`. It will copy anything that supports an assignment. For example

```
int* a1[10];
int* b1[10];
auto_ptr<int> a2[10];
auto_ptr<int> b2[10];
...
template<class T>
void cpy(T& a, T& b, int n)
{
        copy(a,a+n, b);
}
...
cpy(a1,b1,10);
cpy(a2.b2,10);
```

How could we define requirements for `cpy()`? For `int*`, `=` means copy (that is, `a=b` leaves two valid objects behind); whereas for `auto_ptr<int>`,

= means move (that is, a=b leaves a single valid object behind). An obvious solution is:

```
template<Random_iterator T>
requires Copyable<T::value_type> || Movable<T::value_type>
void cpy(T& a, T& b, int n)
{
        copy(a,a+n, b);
}
```

Obviously, we have made a few assumptions here, most prominently that Random_iterator doesn't make any assumptions about whether we can copy its value_type.

This example raises a lot of questions, most prominently whether it is a good idea to define a single operation to have two meaning with such different semantics. What is interesting here is

1. that people do do it (and have done so for at least a decade), so we can't just ignore it
2. we actually can distinguish the two uses of = because they have different signatures the copy version of of = takes a const T& and the move version takes a "plain" T&.

Consider

```
concept Moveable<T> {
        T(T&);
        T& operator=(T&, T&);
};

concept Copyable<T> {
        T(const T&);
        T& operator=(T&, const T&);
};
```

The interesting thing here is that we cannot write a concept that is a simple intersection of Movable and Copyable: Their signatures differ even tough the code that uses them does not. There is a difference in the semantics of the two meanings of = that happens not to be significant for our code.

It has been suggested that "such code is just bad" and that "anything like auto_ptr should be ignored." However, such moralizing missed the point that here (using ||), the writer of the algorithm is expressing the minimal requirements of the algorithm and expressed them using existing concepts. Requiring rewriting of existing code to suit the needs of a new algorithm is rarely feasible and eliminating the need to do so is one of the aims of concepts.

3 Workarounds

What would we do if we did not have a concept intersection operator? basically, we have three choices

1. replicate the template bodies
2. provide (replicated) concept maps
3. simplify our problem by imposing a hierarchical order on our concepts

3.1 The naive approach (replication)

So how do we implement concept intersection? The simplest and most naive approach is simply to replicate template definitions for each alternative and then rely on concept-based overloading. For example, given

```
template<typename T>
        requires Lib1::value_type<T> || Lib2::Regular<T>
void my_alorithm(vector<T>& v)
{
        // ...
}
```

we just generate

```
template<typename T>
        requires Lib1::value_type<T>
void my_algorithm(vector<T>& v)
{
        // ...
}

template<typename T>
        requires Lib2::Regular<T>
void my_algorithm(vector<T>& v)
{
        // ...
}
```

This works. The resulting object code is identical. The resulting compile time is at least as bad (worse, see §4). The code is replicated. The replication is a nuisance and as the number of alternatives (and replications) increase, it becomes a maintenance hazard. Manual replication is at best a nasty workaround (inviting macro artistry).

Compilation costs can be high for a naive implementation f `||`. However, this simple workaround will impose exactly the same costs. For nasty caces where alternatives pile on alternatives the combinatorial explosion is the same in both cases – and is unacceptable and (hopefully) self limiting. We have to do better.

Replicating template functions is relatively easy. Concept-based overloading allows us to distinguish the copies. However, consider template casses:

```
template<typename T>
        requires Lib1::value_type<T> || Lib2::Regular<T>
class My {
        void my_algorithm(vector<T>& v)
        {
                // ...
        }
        // ...
};
```

Replication gives

```
template<typename T>
        requires Lib1::value_type<T>
class My {
        void my_algorithm(vector<T>& v)
        {
                // ...
        }
        // ...
};

template<typename T>
        requires Lib2::Regular<T>
class My {
        void my_algorithm(vector<T>& v)
        {
                // ...
        }
        // ...
};
```

Such overloading is not well supported and would lead to (otherwise unnecessary) template metaprogramming or macro artistry.

3.2 The concept map approach

Rather than replicating template definitions, we could replicate concept maps.

For example, given

```
template<typename T>
        requires Lib1::value_type<T> || Lib2::Regular<T>
void my_alorithm(vector<T>& v)
{
        // ...
}
```

we write

```
template<typename T>
        requires my_regular<T>
void my_algorithm(vector<T>& v)
{
        // ...
}

template<class T>
concept_map<Lib1::value_type<T>> {
        // ...
}

template<class T>
concept_map<Lib2::Regular<T>> {
        // ...
}
```

This seems more elegant and less brute force, but it turns one algorithm into an algorithm plus a concept (here, `my_regular`) plus N concept maps (where N is determined by the number of alternatives – here, it's 2). So we still have an explosion of written source code that needs maintenance. If instead of writing my own `my_regular` concept, I recycled one of the library ones, I get from N+1 pieces of code down to the minimal N pieces. However, the cost is that I have now not stated my intent clearly in the code; I have over-specified and become directly dependent on some library.

The compile–time burden of this replication can still be significant. It is similar to simple replication.

3.3 Impose a hierarchical order

The general problem was that we were presented with related notions of "ordinary type" `Lib1::value_type<T>` and `Lib2::Regular<T>`. If one had been the refinement of the other, we would not have had a problem. Unfortunately, we can't in general impose a hierarchical order on concepts and even for concepts that could be hierarchically ordered (through refinement), we can't always go back and chance the concepts. Retroactive mappings is the concept_map solution. Hierarchical ordering is ideal in many cases. However, it is only feasible within an organization that controls all the concepts, and even then it can be hard to do after that organization has acquired users.

Imposing a hierarchy solves the compile-time overhead problem by eliminating the combinatorical explosion.

4 Calculating intersection

The intersection of two concept maps CM1 and CM2 is defined as the collection of all symbols that are common to both CM1 and CM2, along with consistent type description. More specifically:

1. All symbols that are not declared in both CM1 and CM2 are ignored from the result of the intersection.
2. If n is a name declared in CM1 and CM2, then both CM1 and CM2 must declare it as either a type, or a function. It is an error for n to designates a type in one concept map, and a function in the other.
3. If n designates a type in both CM1 and CM2, recursively compute the intersection of the requirements on n from CM1 and CM2.
4. if n designates a function in both CM1 and CM2, then compute the intersection of the overload set of n from CM1 and the overload set of n from CM2 — see §4.1.

4.1 Intersection of the requirements for two overload sets

An overload set for a function f is partitioned into disjoint subsets, corresponding to the arity of contained functions. For example, the overload set

```
void foo();
int foo(int);
int foo(double);
int foo(double, int);
int foo(int, double);
```

is partitioned into three subsets:

- arity 0:

  ```
  void foo();
  ```

- arity 1:

  ```
  int foo(int);
  int foo(double);
  ```

- arity 2:

  ```
  int foo(double, int);
  int foo(int, double);
  ```

A subset corresponding to arity n of an overloaded function f is written f/n. Furthermore, when performing overloading resolution within a given arity subset of an overloaded function, one is making (the best) choice between several candidates. For the discussion below, it is handy to manipulate the collection of those candidate as a whole, so we will denote the type of the best candidate (when one exist) as

$$T_1 \vee \cdots \vee T_n$$

were each T_i is the (function) type of a candidate.

So given the overload set for a function symbol f from two concepts maps CM1 and CM2, we do the following:

1. for every arity n
   - either f@CM1/n or f@CM2/n is empty, then so is f@intersect(CM1,CM2)/n.
   - Otherwise, call $T_1^1 \vee \cdots \vee T_n^1$ the type of the best candidate from CM1, and $T_1^2 \vee \cdots \vee T_n^2$ the type of the best candidate from CM2. The type of the best candidate in the intersection of CM1 and CM2 is
     $$(T_1^1 \vee \cdots \vee T_n^1) \wedge (T_1^2 \vee \cdots \vee T_n^2).$$

What this notation is that the use of f must give a best candidate from $(T^1_1 \vee \cdots \vee T^1_n)$ *and* a best candidate from $(T^2_1 \vee \cdots \vee T^2_n)$. If both types are identical, then we just write one component.

the type of the intersection must be one that guarantees typecking from both CM1 and CM2, without being the same type in CM1 and CM2.

2. The result of the intersection of CM1 and CM2 is the collection of the all intersections of f@CM1 and f@CM2.

Example: Consider

```
namespace Lib1 {
        concept value_type<T> {
                typedef T value_type;
                T();
                T(const T&);
                T& operator=(T&, T);
                int hash(T);
        };
}

namespace Lib2 {
        concept Regular<T> {
                T();
                T(const T&);
                T& operator=(T&, T);
                bool operator==(T,T);
                bool operator!=(T a, T b) { return !(a==b); }
        };
}
```

We want to compute `value_type<T> || Regular<T>`

1. the symbol `value_type` is defined in `value_type<T>` but, not in `Regular<T>`, so it is not part of the intersection.

2. Similarly the symbols `hash`, `operator==`, and `operator!=` are not part of the intersection.

3. Next both `value_type<T>` and `Regular<T>` declare the same set of constructors (same arity, same type), the result of the intersection contains

$$() \mapsto T \quad \vee \quad (const \;\; T\&) \mapsto T$$

for T's constrcutor, which is just another notation for saying that the intersection contains both the default constructor and the copy constructor.

4. Finally, the symbol `operator=` has the same type in both concept maps, so is part of the intersection.

References

[DRS06] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ Concepts. In *Conference Record of POPL '06: The 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, Charleston, South Carolina, USA, 2006.

[GL07] Douoglas Gregor and Andrew Lumsdaine. Considering Concept Constraint Combinator. Technical Report N2161=07-0021, ISO/IEC SC22/JTC1/WG21, September 2007.

[GS06] Douoglas Gregor and Bjarne Stroustrup. Concepts (Revision 1). Technical Report N2081=06-00112, ISO/IEC SC22/JTC1/WG21, September 2006.