Concept proposal comparison

N1899 = 05-0159

Matt Austern

2005-10-06

Papers

- N1510 etc: *outline of design space*.
  Bjarne and Gaby, 2003
- N1849: *Indiana proposal*.
  Doug et al, August 2005.
- N1782 and N1886: *Texas proposal*.
  Bjarne and Gaby, 2005.

Common approach

Goals

- Better error detection
  - Separate error checking
  - Simpler and earlier instantiation errors
- Express documentation in code
- Concept-based overloading

Basic scheme

- Declare *concept*: requirement on type or types
- *Model assertion*: declare that a type, or family of types, satisfies a concept
- *Constrained template*: arguments must satisfy a concept

Declaring a concept

- Parameters: types it’s constraining
  - List of operations the parameters must provide
  - Associated types, and constraints on them.
  - Refinement
- Rejected approaches:
  - base class
  - exact signature

Model assertions

- Assert that a type (or family) models a concept
- Assertion failure is an error
- Can be used for syntax rewriting
- Concept author gets to say whether it’s mandatory or optional

Constrained templates

- `where` clause
  - Assert that argument(s) used for instantiation model(s) a concept
  - Multiple assertions allowed
  - Syntactic sugar: omit `where` for one-argument concepts
- Type check at definition time
- Concept matching at instantiation time

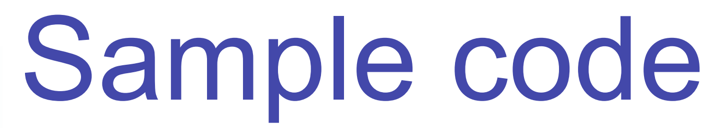
Separate type checking

- We check types in at least three places
- Soundness:

  "If a constrained template definition concept checks and if its uses both concept check and type check then its instantiations for those uses also type check."

Overloading on concepts

- Attempt concept matching on all overloads
- No matches ⇒ error
- One match ⇒ choose it
- Multiple matches
  - One best match ⇒ choose it
  - Otherwise error



Sample code

Defining a concept: TAMU

```
concept Input_iterator<Trivial_iterator Iter>
    where Equality_comparable<Iter>
        && Assignable<Iter>
        && Arrow<Iter> {
  Integer difference_type;
  Var<Iter> p;
  const Iter::value_type& v = *p;
  const Iter::value_type& v2 = *p++;
};
```

Defining a concept: IU

```
template <typename X>
concept InputIterator : IteratorAssociatedTypes<X>.
                        CopyConstructible<X>,
                        Assignable<X>,
                        EqualityComparable<X> {
  where SignedIntegral<difference_type>;
  where Convertible<reference, value_type>;
  where Arrowable<pointer, value_type>;

  typename postincrement_result = X;
  where Dereferenceable<postincrement_result, value_type>;

  pointer operator->(X);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
  reference operator*(const X&);
};
```

Defining a template: TAMU

```
template <Forward_iterator Iter, typename T>
  where Assignable<Iter::value_type, T>
void fill(Iter first, Iter last, const T& t) {
  ...
}
```

Defining a template: IU

```
template <Forward_iterator Iter, typename T>
  where { Assignable<value_type, T> }
void fill(Iter first, Iter last, const T& t) {
  ...
}
```

Model assertion: TAMU

```
concept MyConcept<typename T> {
  ...
};

// optional
static_assert template <typename T> !MyConcept<T>;
...

static_assert MyConcept<MyType>;
```

Model assertion: IU

```
template <typename T>
/* struct */ concept MyConcept {
  ...
};

...

template<> concept MyConcept<MyType>;
```

Differences

- Use patterns (valid expressions) vs pseudosignatures (abstract signatures)
- Implicit checking vs nominal conformance
- Concept composition: disjunction and negation?
- Builtin same-type requirement
- Model assertions for as-yet-undeclared types
- Default definitions in concepts
- Syntactic differences
  - Refinement: special support, or just `where` clause and conjunction?
  - Associated types in concepts vs concept parameters

Usage patterns vs pseudosignatures

- Believed to have equivalent expressive power
  - Can map a valid expression to pseudosignatures by introducing auxiliary associated types
  - Is there an algorithm for mapping the other way?
- Is this purely a syntactic difference?

Usage patterns vs pseudosignatures

Usage pattern warts	Pseudosignature warts
Can’t express ->	Exactly how pseudo is it?
Special-case syntax for variables	Less traditional for generic programming
No clean syntactic hook for extra stuff	Hard to express a*b+c
Less precise	Invites overspecification

Syntax remapping (both proposals)

```
template <typename T>
concept X {
   typename type;
   T foo(const T&);
};

...

template<> concept X<MyType> {
   typedef MyType::type type;
   T foo(const T& t) { return t.Foo(); }
};
```

Default definitions in concept (IU only)

```
template <typename T>
concept Comparable {
   bool operator<(const T&, const T&);
   bool operator>(const T& x, const T& y) {
      return y < x;
   }
};

// A type that models Comparable only needs to
// provide <.
```

Implicit checking vs nominal conformance

- Both proposals provide both forms
- Author of concept chooses which form is used
- Defaults differ
  - TAMU: use negative assertion to request nominal conformance
  - IU: use `struct concept` to request implicit checking

Why nominal conformance might just be workable

```
template<typename OldIter>
where {Convertible<typename std::iterator_traits<OldIter>::category*,
                                std::random_access_iterator_tag*> }
model RandomAccessIterator<OldIter> { };
```

- Very broad model declaration
- Applies even to types we haven't seen yet

Why implicit checking might just be workable

Use negative assertions to distinguish between concepts that differ only in semantics

```
concept InputIterator<typename Iter> { ... };
concept ForwardIterator<InputIter Iter>  { };

static_assert
template <ValueType T>
  !ForwardIterator<std::istream_iterator<T> >;
```

- IU: conjunction only
- TAMU: conjunction, disjunction, negation
  - Negation: probably not necessary except to choose nominal conformance
  - Disjunction: harder call

Disjunction

- Argument for:

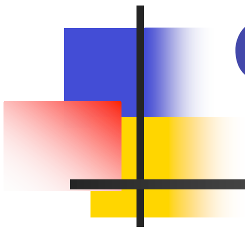
```
template<C1 T> void helper(T x);
template<C2 T> void helper(T x);

template<C0 T> void foo(T x) where C1<T> || C2<T> {
   ...
   helper(x);
}
```

- Argument against:
  - Aesthetic: should factor out into a base concept
  - Essentially splits template into duplicates
  - Unclear what to do in case of multiple matches

Philosophy: reification of concepts and models

- IU: model is “the realization of a concept.”
- TAMU: assert “explicit checking of conformance of a type with respect to a concept.” Model not mentioned as a noun.

- Is a *model* a thing? A *concept*?



Technical challenges and open issues

Soundness and name lookup

- Soundness: type-check a constrained template at definition time, not instantiation time
- $\Rightarrow$ All name lookup at definition time
- Tension with areas where we might want later lookup

Name lookup in templates

- Today’s templates: two-phase name lookup
- Constrained templates
  - Dependent name found in concept: use it
  - Dependent name not mentioned in concept: what do we do?

The helper function problem

```
template <ForwardIterator FI>
bool binary_search(FI first, FI last) {
  ...
  advance(first, n);
}
```

- We want the random access version of advance when appropriate
- What type lookup rules will ensure that, and also ensure soundness?

The ambiguity problem

```
template <class T> where TrivialIterator<T> void foo(T& x) { … }
template <class T> where InputIterator<T> void foo(T& x) { … }
template <class T> where OutputIterator<T> void foo(T& x) { … }

template <class T> where TrivialIterator<T> void bar(T& x) { foo(x); }
```

- bar seems to pass concept check, but fails if we call it with a forward iterator.
- How can we modify type checking rules so that bar won’t concept check?

The specialization problem

```
template<typename T> where { CopyConstructible<T> }
void foo(T x)
{
   std::vector<T> vec(1, x);
   T& f = vec.front();
}
```

- Foo appears to pass concept check, but might fail type check at instantiation time
- Possible solution (TAMU): forbid specialization that changes template's conformance to requirements

Can we prove a soundness theorem?

- IU: yes for System $F^{\mathcal{G}}$, no for C++ as it stands now
- TAMU: yes, but
  - Proof isn’t yet complete
  - May require restrictions on specialization

Other open questions

- Implications for expression templates
- Is concept-safe template metaprogramming possible?

References

- Bjarne Stroustrup, "Concept checking - A more abstract complement to type checking", N1510, October 2003.
- Bjarne Stroustrup and Gabriel Dos Reis, "Concepts - Design choices for template argument checking", N1522, October 2003.
- B. Stroustrup, G. Dos Reis, "Concepts - syntax and composition", N1536, October 2003.
- Robert Klarer, John Maddock, Beman Dawes, and Howard Hinnant, "Proposal to Add Static Assertions to the Core Language (Revision 1)", N1604, February 2004.
- Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Jarvi, Andrew Lumsdaine, "Concepts for C++0x", N1758, January 2005.
- Doug Gregor, Jeremy Siek, "Explicit model definitions are necessary", N1798, April 2005.
- Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine, "C++ Language Support for Generic Programming", N1799, April 2005.
- Douglas Gregor, Jeremy Siek, "Implementing Concepts", N1848, August 2005.
- Douglas Gregor, Jeremy Siek, Jeremiah Willcock, Jaakko Jarvi, Ronald Garcia, Andrew Lumsdaine, "Concepts for C++0x, Revision 1", N1849, August 2005. (Supersedes N1758.)
- Lawrence Crowl and Thorsten Ottosen, "Synergies between Contract Programming, Concepts and Static Assertions", N1867, August 2005.
- Bjarne Stroustrup and Gabriel Dos Reis, "A concept design (Rev. 1)", N1782, 2005. (Supersedes unnumbered paper sent out on the reflector.)
- Gabriel dos Reis, Bjarne Stroustrup, "A Formalism for C++", N1885, September 2005.

References (cont)

- Gabriel dos Reis, Bjarne Stroustrup, "Specifying C++ concepts", N1886, September 2005.
- Alex Stepanov and Meng Lee, "The Standard Template", Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories http://www.hpl.hp.com/techreports, 1994.
- SGI (Matt Austern, Hans Boehm, Jeremy Siek, Alexander Stepanov, John Wilkinson), SGI Standard Template Library Programmers Guide, http://www.sgi.com/tech/stl, 1998.
- Jeremy Siek, Andrew Lumsdaine, "Concept Checking: Binding Parametric Polymorphism in C++", Proceedings of the First Workshop on C++ Template Programming, Erfurt, Germany, 2000.
- Jaakko Jarvi, "Concept based overloading", from the Lillehammer concepts wiki, 2005.
- Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock, "A Comparative Study of Language Support for Generic Programming", Proceedings of the 2003 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'03), October 2003.
- Gabriel Dos Reis, "Generic Programming in C++: The next level", ACCU Spring Conference 2002.
- Jaakko Jarvi, Jeremiah Willcock, and Andrew Lumsdaine, "Algorithm specialization and concept-constrained genericity", Adobe talk, April 2004.
- Jeremy Siek and Andrew Lumsdaine. Essential Language Support for Generic Programming. In PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, NY, USA, pages 73--84, June 2005. ACM Press