Concepts Lite

Constraining Template Arguments with Predicates

Andrew Sutton,
Bjarne Stroustrup, Gabriel Dos Reis
Texas A&M University

© 2013 Andrew Sutton
Overview

Introduction

Constraining templates

Defining constraints

Language mechanics

Implementation and examples

© 2013 Andrew Sutton
Templates: An Ideal

Abstract expression of algorithms, data structures

Integers, Reals, Sequences, Sets, Graphs, etc.

Generality

Not limited to a single model

Fast code

No abstraction penalty
Type-based optimizations

© 2013 Andrew Sutton
Templates: The Reality

template<
typename T>

typename enable_if<is_integral<T>::value, T>::type

gcd(T a, T b)
{
    return do_gcd(a, b,
                   typename is_unsigned<T>::type{});
}

© 2013 Andrew Sutton
Templates: Reality Bites

gcd(16.0, 2.0); // Error!

error: In the instantiation of ‘gcd(T, T)’
    where T = double
error: In the instantiation of ‘do_gcd(T, T, X)’
    where T = double, X = integral_constant<bool, false>
error: In the instantiation of ‘euclid_gcd(T, T)’
    where T = double
error: no match for operator ‘%’ in ‘a % b’
note: candidates are:
note:    operator%(int, int)
note:    operator%(long, long)
note:    ...

© 2013 Andrew Sutton
Concepts Lite: Template Constraints

Improve language support for generic programming

Directly state requirements on template arguments
Support overloading and specialization based on constraints
Improved interfaces and greatly enhanced diagnostics
Usable by “ordinary programmers”

Without runtime overhead or long compilation times

Almost completely implemented (modulo bugs)
Handles the Standard Library algorithms and their uses

© 2013 Andrew Sutton
Constraints Are Not Concepts

Only check requirements at the point of use

Does not check template definitions
Late-caught errors are still possible

Approach allows incremental adoption/use of concepts in generic libraries

There is a (language) migration path to concepts

© 2013 Andrew Sutton
Concepts Lite: The Source

Concepts Lite: Constraining Templates with Predicates (N3580)

http://concepts.axiomatics.org/~ans/

https://github.com/asutton/origin
Constraining Template Arguments

Constrain template arguments with predicates

template<typename Sortable C>
void sort(C& container);

Equivalently:

template<typename C>
   requires Sortable<C>()
void sort(C& container);
Constraints

Are just constexpr function templates

template<typename T>
constexpr bool Sortable()
{
    return ...; // Returns true when T is a permutable container whose elements can be totally ordered
}
Constraint Checking

Constraints are checked at the point of use

```cpp
forward_list<int> l { ... };
sort(l);
```

error: no matching call to ‘sort(forward_list<int>&)’
note: candidate is ‘sort(C& container)’
note: where C = forward_list<int>

note: template constraints not satisfied
note: ‘C’ is not a/an ‘Sortable’ type since
note: ‘c[n]’ is not valid syntax
Constraints on Class Templates

Just like function templates

```cpp
template<Object T, Allocator A>
class vector;
```

Equivalently:

```cpp
template<typename T, typename A>
    requires Object<T>() && Allocator<A>()
class vector;
```
Constraints on Member Functions

Member functions and constructors

template<Object T, Allocator A>
class vector
{
    requires Copyable<T>()
    vector(const vector& x);

    requires Movable<T>()
    void push_back(T&& x);
};
Multi-type Constraints

Constraints can be applied to multiple types

template<Sequence S, 
  Equality_comparable<Value_type<S>> T>
Iterator_type<S> find(S&& s, const T& value);

Equivalently with a requires:

template<typename S, typename T>
  requires Sequence<S>() 
    && Equality_comparable<T, Value_type<S>>()
Iterator_type<S> find(S&& s, const T& value); 

© 2013 Andrew Sutton
Overloading

Function overloading is extended to include constraints

template<\texttt{Input\_iterator} \ I> 
void advance(I\& iter, \texttt{int} n);

template<\texttt{Bidirectional\_iterator} \ I> 
void advance(I\& iter, \texttt{int} n);

template<\texttt{Random\_access\_iterator} \ I> 
void advance(I\& iter, \texttt{int} n);
Overloading

Compiler selects the *most constrained* overload

```cpp
istream_iterator<int> iter(cin);
advance(iter, 1); // *Input* overload

list<T>::iterator first = l.begin();
advance(first, 1); // *Bidirectional* overload
```

The most constrained is automatically determined by comparing template constraints
An More Interesting Example

template<Object T, std::size_t N>
class array
{
  T& operator[](std::size_t n)
  { return data[n]; }

  requires N == 0
  T& operator[](std::size_t n) = delete;

  T[N] data;
};
Class Template Specialization

Also extended to support constraints

template<
typename T>
    class complex; // Undefined primary template

template<Real T>
    class complex<T> { ... }; // Complex number

template<Integer T>
    class complex<T> { ... }; // Gaussian integer

Specialization arguments

© 2013 Andrew Sutton
More About Constraints

Discussed in n3580:

- Alias templates
- Template template parameters
- Variadic constraints
Defining Constraints

A *constraint* is a `constexpr` function template

Can use type traits, call other `constexpr` functions

Constraints check *syntactic requirements*

Is this expression valid for objects of type `T`?
Is the result type of an expression convertible to `U`?
Constraints: First Pass

Use type traits

template<typename T>
constexpr bool Equality_comparable()
{
  return has_eq<T>::value // a == b
      && is_convertible<eq_type<T>, bool>::value
      && has_ne<T>::value // a != b
      && is_convertible<ne_type<T>, bool>::value;
}

Many, many downsides
Constraints: Current Design

Invent new syntax for requirements

template<typename T>
constexpr bool Equality_comparable()
{
    return requires (T a, T b) {
        {a == b} -> bool;
        {a != b} -> bool;
    };
}

Work in progress

© 2013 Andrew Sutton
Constraints: Longhand

Can be equivalently written as

```cpp
template<typename T>
constexpr bool Equality_comparable() {
    return requires (T a, T b) {
        a == b;
        requiresConvertible<decltype(a == b), bool>();
        a != b;
        requiresConvertible<decltype(a != b), bool>();
    };
}
```
Constraints: Type Requirements

We can also write type requirements

```cpp
template<typename I>
constexpr bool User_defined_iterator()
{
    return requires (I i) {
        typename I::iterator_category;
        {*i} -> const Value_type<I>&;
    };
}
```
Constraints: The Language

Constraints: how do they work?

- Language primitives
- Reduction
- Decomposition
- Overload resolution

© 2013 Andrew Sutton
Constraint Language

Formally, constraints are defined over a set of atomic propositions, connected by && and ||

is_lvalue_reference<T>::value && is_const<T>::value

is_integral<T>::value || is_floating_point<T>::value

Equality_comparable<T>() && Weakly_ordered<T>()

© 2013 Andrew Sutton
Atomic Propositions

For the most part, any C++ expression that is not an && or || expression

is_integral<T>::value
!is_void<T>::value
N == 2
0 < M
is_prime(N)
ture
false

Calls to constraints are not atomic
Constraint Reduction

Function calls to constraints are *reduced* by inlining them into a requires clause

template<typename T>
constexpr bool Arithmetic()
{
    return is_integral<T>::value  
        || is_floating_point<T>::value;
}
Constraint Reduction

Before:

```cpp
template<typename T>
    requires Arithmetic<T>()
T do_math(T a, T b);
```

After:

```cpp
template<typename T>
    requires is_integral<T>::value
    || is_floating_point<T>::value
T do_math(T a, T b);
```
Constraint Decomposition

Reduced constraints are decomposed into sets of propositions through the application of sequent calculus for first order logic.

\[
\frac{\Gamma, A \vdash \Delta}{\Gamma, A \land B \vdash \Delta} \quad (\land L_1) \quad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \lor B, \Delta} \quad (\lor R_1)
\]

\[
\frac{\Gamma, B \vdash \Delta}{\Gamma, A \land B \vdash \Delta} \quad (\land L_2) \quad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \lor B, \Delta} \quad (\lor R_2)
\]

\[
\frac{\Gamma, A \vdash \Delta \quad \Sigma, B \vdash \Pi}{\Gamma, \Sigma, A \lor B \vdash \Delta, \Pi} \quad (\lor L) \quad \frac{\Gamma \vdash A, \Delta \quad \Sigma \vdash B, \Pi}{\Gamma, \Sigma \vdash A \land B, \Delta, \Pi} \quad (\land R)
\]
Overload Resolution

Find candidates, instantiate templates

- Deduce template arguments
- Instantiate and check constraints
- Instantiate the declaration

Choose the best candidate

- Most specialized
- Most constrained

© 2013 Andrew Sutton
Constraint Satisfaction

Constraints are just constant expressions

Evaluate it!
Most Constrained

Given two sets of propositions \( P \) and \( Q \), \( P \) subsumes more than \( Q \) iff \( P \) contains all of \( Q \)’s propositions.

Solved as an application of first order logic.

Easily thought of as a subset problem.

Given two declarations \( A \) and \( B \) with the same type, \( A \) is more constrained than \( B \) iff \( A \)’s requirements subsume \( B \)’s.

Unconstrained templates are the least constrained.

© 2013 Andrew Sutton
Overloads

template<Forward_iterator I>
void advance(I&, int);

template<Bidirectional_iterator I>
void advance(I&, int);
Conjunction and Refinement

Constraints that subsume others are *refinements*
Disjunction and Overlap

The disjunction of overlapping constraints

Integral

Floating_point

Arithmetic

© 2013 Andrew Sutton
Implementation

Experimental prototype (GCC-4.8, from September)

http://concepts.axiomatics.org/~ans/

Official GCC branch (GCC-4.9, follows trunk)

http://gcc.gnu.org/svn/gcc/branches/c++-concepts/

Library support (Origin)

https://github.com/asutton/origin

© 2013 Andrew Sutton
Library Support

Supplemental library: Origin

All constraints for all concepts in Palo Alto TR (n3351)

Equality_comparable, Totally_ordered, Regular, Function, Predicate, Relation
Input_iterator, Forward_iterator,
Bidirectional_iterator, Sortable

Etc.
template<Random_access_range R>
  requires Sortable_range<R>()
void sort(R& range)
{
  std::sort(std::begin(range), std::end(range));
}
Sortable Range

template<typename R>
constexpr bool Sortable_range()
{
    return Range<R>()
        && Sortable<Iterator_type<R>>;
}
Iterator type

template<typename T>
using Iterator_type =
  decltype(std::begin(std::declval<T>()));
Sortable Iterator

template<
typename I>
const expr bool Sortable ()
{
    return Forward_iterator<I> ()
        && Permutable<I> ()
        && Totally_ordered<Value_type<I>> ();
}
Value Type

template<typename T>
using Value_type =
    typename get_value_type<T>::type;
Value Type

template<typename T>
struct get_value_type;

template<typename T>
struct get_value_type<T*>& { using type = T; };

template<typename T>
  requires has_nested_value_type<T>()
struct get_value_type<T> {
  using type = typename T::value_type;
};

© 2013 Andrew Sutton
Nested Value Type

template<typename T>
constexpr bool has_nested_value_type()
{
    return requires () {
        typename T::value_type;
    };
}
Forward Iterators

template<typename I>
constexpr bool Forward_iterator()
{
    return Regular<I>() && requires (I i) {
        Difference_type<I>;
        {++i} -> I&;
        {i++} -> I;
        {*i} -> const Value_type<I>&;
    };
}
Conclusions

Just scratched the surface

  Small language extension
  Does many, many cool things

Go get the compiler. Tell me how it works
Questions