Automatically Identifying C++0x concepts in Function Templates

Andrew Sutton
Jonathan I. Maletic
C++03 Function Template

template <typename Iter, typename T>
void fill(Iter f, Iter l, const T& x)
{
    for( ; f != l; ++f) {
        *f = x;
    }
}

What types can I use here?

What abstraction does this represent?
C++0x Function Template

template <typename Iter, typename T>
    requires
        MutableForwardIterator<Iter> &&
        SameType<Iter::value_type, T>
void fill(Iter f, Iter l, const T& x)
{
    for( ; f != l; ++f) {
        *f = x;
    }
}
Our Goals

- Facilitate migration to C++0x (this paper)

- Support evolution of C++0x generic libraries (future work)
  - Evolve requirements with implementations
  - Assist in specification of one algorithm with different implementations
  - requirement coverage of algorithms
Concepts (concept)

- Represents abstraction of a family of types
- Defines syntactic and semantic requirements of those types
- A type models a concept if it satisfies the requirements
- Organized into hierarchies by refinement
- Used by templates to express constraints on substitutable types
A concept Definition

```cpp
class InputIterator<
  typename T> 
  : IteratorBase<T>
{
  typename value_type;
  typename reference;

  reference operator*(
      T&); 
  T& operator++(
      T&); 
  T operator++(
      T&, int);
};
```
Our Approach

- Use static analysis to identify concepts in function templates

Inputs

- AST of a function template
- Concept hierarchy (repository)

Output

- Boolean requirements expression
Process

- Extract type requirements from AST
- Match/Instantiate requirements against concept hierarchy
- Create refinement hierarchy from instances and search it for candidate
- Use FCA to order candidates
Type Requirements

template <typename Iter, typename T>
void fill(Iter f, Iter l, const T& x)
{
    for( ; f != l; ++f) {
        *f = x;
    }
}

Type Requirement
Iter::ctor(Iter)
Type Requirement
preinc(Iter)
Type Requirement
deref(Iter)->*Iter
Type Requirement
neq(Iter, Iter)->bool
Concept Matching

- Match type requirements to concept definitions in concept repository
- Match name of type requirement to name in concept definition
- Discard obvious mismatches
  - Wrong types
  - Wrong number of parameters
Concept Matching

```cpp
template <typename Iter, typename T>
void fill(Iter f, Iter l, const T& x)
{
    for( ; f != l; ++f) {
        *f = x;
    }
}
```

**Matched Concept**
- CopyConstructible
- Dereferenceable
- EqualityComparable
- PreIncrementable
Concept Instantiation

- Generate concept instances from previously matched concepts
- Try to substitute concept parameters with template parameters
- Pairwise comparison of parameters in requirement to definition
- Reject mismatches
Concept Instantiation

template <typename Iter, typename T>
void fill(Iter f, Iter l, const T& x)
{
    for( ; f != l; ++f) {
        *f = x;
    }
}

Matched Concept
EqualityComparable<Iter>

Matched Concept
Dereferenceable<Iter>

Matched Concept
CopyConstructible<Iter>

Matched Concept
PreIncrementable<Iter>
Refinement Search

- Consider refinements of previously instantiated concepts
- Use refinement relation to create lattice over concept instances
- Search lattice for concepts with non-redundant expression of requirements
  - Meet of all subsets of powerset of initial candidates
Refinement Lattice Example

- Sublattice over instances of Iter
- Initial instances in gray
- Algorithm yields
  - InputIterator
  - OutputIterator
  - MutFwdIterator
Finding Solutions

- Apply FCA to concept instances & type requirements to generate lattice
- Each template parameter found in disjoint, bounded sublattice
- Best abstractions found in extent of least element of sublattices
- Can find multiple instances per extent
Lattice

Diagram of lattice with nodes labeled with attributes and objects.
template <typename Iter, typename T>
requires
    MutableForwardIterator<Iter>
void fill(Iter f, Iter l, const T& x) {
    for( ; f != l; ++f) {
        *f = x;
    }
}
Implementation

- Use srcML as parsing base
- Approach implemented as pipe and filter application
  - srcexpr - built using srcTools framework, extracts requirements from templates
  - srccon - takes requirements, identifies concept requirements
Evaluation

- Tested against STL generic algorithms in two ways:
  - Applicability - ensure that the approach is valid on a range of algorithms
  - Stability - ensure that the approach generates similar requirements for variations of the same algorithm
Results

- Works well for most algorithms
- Compared results against proposed requirements in C++ standard

- Found problems in STL0x proposals
  - Ambiguities in STL iterator hierarchy
  - Missing concepts for function objects
Conclusions

- Developed methods to analyze and infer constraints on template parameters
- Supports migration of current template functions to C++0x
- Could be used to assist in comprehension of both generic algorithms and concept hierarchies
Late Breaking News

- We talked with C++0x authors about problems with concept hierarchy proposal in STL
- They have modified proposal to address errors we uncovered in the most recent version of the proposal