Future of Abstraction

Alexander Stepanov
Outline of the Talk

- What is abstraction?
- Abstraction in programming
- OO vs. Templates
- Concepts
- A new programming language?
Abstraction

- The fundamental way of organizing knowledge
- Grouping of similar facts together
- Specific to a scientific discipline
Abstraction in Mathematics

Vector space

\{V: \text{Group}; F: \text{Field}; \times: F, V \rightarrow V;\}

distributivity; distributivity of scalars; associativity; identity

Algebraic structures (Bourbaki)
Abstraction in Programming

for (i = 0; i < n; i++)
    sum += A[i];

- Abstracting +
  - associativity; commutativity; identity
  - parallelizability; permutability; initial value
- Abstracting i
  - constant time access; value extraction
Abstraction in Programming

1. Take a piece of code
2. Write specifications
3. Replace actual types with formal types
4. Derive requirements for the formal types that imply these specifications
Abstraction Mechanisms in C++

- Object Oriented Programming
  - Inheritance
  - Virtual functions
- Generic Programming
  - Overloading
  - Templates

Both use classes, but in a rather different way
Object Oriented Programming

- Separation of interface and implementation
- Late or early binding
- Slow
- Limited expressability
  - Single variable type
  - Variance only in the first position
class reducer {
    public:
        virtual void initialize(int value) = 0;
        virtual void add_values(int* first, int* last) = 0;
        virtual int get_value() = 0;
};

class sequential_reducer : public reducer { … };

class parallel_reducer : public reducer { … };
Generic Programming

- Implementation is the interface
  - Terrible error messages
  - Syntax errors could survive for years
- Early binding only
- Could be very fast
  - But potential abstraction penalty
- Unlimited expressability
template <class InputIterator, class BinaryOperation>
typename iterator_traits<InputIterator>::value_type
reduce(InputIterator first,
       InputIterator last,
       BinaryOperation op) {
    if (first == last) return identity_element(op);
    typename iterator_traits<InputIterator>::value_type
    result = *first;
    while (++first != last) result = op(result, *first);
    return result;
}
template <class InputIterator, class BinaryOperation>
typename iterator_traits<InputIterator>::value_type
reduce(InputIterator first,
        InputIterator last,
        BinaryOperation op) {
    if (first == last) return identity_element(op);
    typename iterator_traits<InputIterator>::value_type
    result = *first;
    while (++first < last) result = op(result, *first);
    return result;
}
We need to be able to define what `InputIterator` is in the language in which we program, not in English.
Concepts

concept SemiRegular : Assignable, DefaultConstructible{};
concept Regular : SemiRegular, EqualityComparable {};
concept InputIterator : Regular, Incrementable {
    SemiRegular value_type;
    Integral distance_type;
    const value_type& operator*();
};
Reduction done with Concepts

```c++
value_type(InputIterator) reduce(InputIterator first,
      InputIterator last,
      BinaryOperation op )
(value_type(InputIterator) == argument_type(BinaryOperation))
 {
   if (first == last) return identity_element(op);
   value_type(InputIterator) result = *first;
   while (++first != last) result = op(result, *first);
   return result;
 }
```
Signature of merge

OutputIterator merge(InputIterator[1] first1, 
        InputIterator[1] last1, 
        InputIterator[2] first2, 
        InputIterator[2] last2, 
        OutputIterator result)

(bool operator<(value_type(InputIterator[1]), 
        value_type(InputIterator[2])),
        output_type(OutputIterator) ==
        value_type(InputIterator[1]),
        output_type(OutputIterator) ==
        value_type(InputIterator[2]));
Virtual Table for InputIterator

- type of the iterator
  - copy constructor
  - default constructor
  - destructor
  - operator=
  - operator==
  - operator++
- value type
- distance type
- operator*
Unifying OOP and GP

- Pointers to concepts
- Late or early binding
- Well defined interfaces
- Simple core language
Other Language Problems

- Semantic information:
  - assertions, complexity
- Multiple memory types:
  - pointers, references, parameter passing
- Compilation model:
  - cpp, includes, header files
- Design approach:
  - evolution vs. revolution
Conclusion

We have to create a language that expresses everything we want to say about computations:

If it is worth saying, it is worth saying formally.