Generic Programming and First-Class Libraries

Jaakko Järvi
jarvi@cs.tamu.edu

September 15, 2004
Overview

- About software libraries
- Generic programming as a paradigm for constructing reusable libraries
- Research agenda for generic programming and first-class libraries.
  - Theory of generic programming, language support
  - Library-specific optimizations
  - Library-specific analyses
  - Compiler support to enable the above
- Conclusion
Others behind this material...

- Andrew Lumsdaine, Douglas Gregor (IU)
- Dave R. Musser (RPI)
- Bjarne Stroustrup, Jaakko Järvi (Texas A&M)
- Sibylle Schupp (Chalmers, Göteborg, Sweden)
- Alex Stepanov (Adobe, no picture)
Software libraries

- Static collections of routines/functions/classes?
Software libraries

- Static collections of routines/functions/classes?
- Libraries (also) encapsulate domain specific knowledge
  - Syntax appropriate for a domain
  - Optimizations specific to a domain
  - Safety checks specific to a domain
- Standardization and layering of libraries ⇒ reuse
  - Containers/algorithms in C++ standard library
Software libraries

- Static collections of routines/functions/classes?
- Libraries (also) encapsulate domain specific knowledge
  - Syntax appropriate for a domain
  - Optimizations specific to a domain
  - Safety checks specific to a domain
- Standardization and layering of libraries ⇒ reuse
  - Containers/algorithms in C++ standard library

Libraries do not get much attention in CS
Generic programming: discipline for library design

- Conceptual categorization of a computational domain
  - Description as interfaces between algorithms and types
- Decoupling of algorithms and problem domain
  - Reducing algorithms to their minimal conceptual requirements
- Including non-functional requirements in the abstraction process, design, and implementation

~ methodology of *generic programming*
Generic programming: an emerging discipline

- Successful in C++
  - The Standard Template Library (STL) 1994
    - Musser, Stepanov
  - The Standard Template Adaptive Template Library (STAPL) 1998
    - Rauchwerger et al
  - The Matrix Template Library (MTL) 1998, The Boost Graph Library (BGL) 2000
    - Siek, Lumsdaine, Lee
  - C++ Boost library collection (MultiArray, MPL, $\mu$BLAS, ...)

Tens of thousands of users.

- Growing interest both in industry and academia.
Generic programming: lifting process

Minimal requirements: works with maximal family of types

Generic algorithm

Remove an unneeded requirement on the type

Less specialized: works with more than one type

Concrete algorithm: requires specific data type

Start here
Maximal family of types?

Maintain usefulness of the algorithm – make efficiency part of the requirements

Generic algorithm

Concrete algorithm

Lift

When instantiated, the generic algorithm is as efficient as the original concrete algorithm.
Key ideas of generic programming

- More than minimal capabilities may enable a faster/better implementation
  - Automatic dispatching to most efficient implementation
- Algorithms must be as efficient as if written for concrete types
  - No abstraction penalty
- High-performance regardless of abstraction
- High-performance because of abstraction
- Serious approach to libraries
Generic programming: example

define function to sum elements

```c
double sum(double* array, int n) {
    double s = 0;
    for (int i = 0; i < n; ++i)
        s = s + array[i];
    return s;
}
```

- Requirements
  - Element type must be of type `double`
  - Elements stored in an array
Generic programming: example

```c
double sum(list_node* first, list_node* last) {
    double s = 0;
    while (first != last) {
        s = s + first->data; first = first->next;
    }
    return s;
}
```

- **Requirements**
  - Element type must be of type **double**
  - Elements stored in a linked list
Generic programming: example

```c
double sum(double* array, int first, int last) {
    double s = 0;
    while (first != last)
        s = s + array[first++];
    return s;
}
```

- Requirements
  - Element type must be of type `double`
  - Elements stored in an array
Generic programming: example

```cpp
template <class T>
T sum(T* first, T* last) {
    T s = 0;
    while (first != last)
        s = s + *first++;
    return s;
}
```

- Requirements
  - Element type must support +
  - Elements addressed by pointers
Generic programming: example

```cpp
template <class T>
T sum(T* first, T* last, T s) {
    while (first != last) {
        s = s + *first++;
    }
    return s;
}
```

- Requirements
  - Element type must support +
  - Elements addressed by pointers
Generic programming: example

```cpp
template <class InputIterator>
InputIterator::value_type
sum(InputIterator first, InputIterator last,
    InputIterator::value_type s) {
    while (first != last)
        s = s + ∗first++;
    return s;
}
```

- **Requirements**
  - Element type must support +
  - InputIterator must support !=, ++, *
  - Must be able to access the associated type value_type
    - 'minimal requirements' ≠ 'fewest requirements'
Example usage: programming with concepts

- `sum` can be used with any type that satisfies the concept of `Input Iterator`

```cpp
double x[10];
vector<string> y(42);
list<complex<double>> z(100);

double a = sum(x, x + 10, 0.0);
string b = sum(y.begin(), y.end(), "");
complex<double> c = sum(z.begin(), z.end(), 0);
```
Concepts

- Group requirements on abstractions (usually types) into reusable entities
  - Valid expressions
  - Semantic constraints
  - Complexity guarantees
  - Associated types
- Allow concise specification of constraints of parameters to generic algorithms
- Types can *model* a concept
- Concepts can *refine* other concepts \( \Rightarrow \) concept taxonomies
- Design tool: manifestations in programming languages vary
Example: conjugate gradient

```cpp
int cg(const LinearOperator& A, HilbertSpaceX& x,
    const HilbertSpaceB& b) {
    typedef typename itl_traits<HilbertSpaceX>::
        value_type T;
    T rho, rho_previous, alpha, beta;
    HilbertSpaceX p(size(x)), q(size(x)), r(size(x)), z(size (x));

    itl::mult(A, itl::scaled(x, -1.0), b, z);
    itl::solve(L, z, r);

    while (! iter.finished(r)) {
        rho = itl::dot_conj(r, r);
        if (!iter.first()) {
            beta = rho / rho_previous;
            itl::add(z, itl::scaled(p, beta), p);
        }
        itl::mult(A, p, q);
        alpha = rho / itl::dot_conj(p, q);
        itl::add(x, itl::scaled(p, alpha), x);
        itl::add(r, itl::scaled(z, -alpha), r);
        rho_previous = rho;
        ++iter;
    }
    return iter.error_code();
}
```

- Example from ITL [Lumsdaine, Lee, Siek]
- Natural high-level (domain specific) description of the algorithm
- Can operate on any types that meet the concept requirements
- Abstract description of an algorithm is also its implementation
- No penalty for abstractions
Towards first class libraries

- The above is being used, and working (though not perfect) today.
- Concepts provide a mechanism for organizing libraries and describing their interfaces.
- Concepts provide an abstraction barrier.
- What is missing?

```cpp
template <class InputIterator>
InputIterator::value_type
sum(InputIterator first, InputIterator last,
    InputIterator::value_type s);
```

- Compiler is not aware of InputIterators
Languages vs. Libraries

- Libraries are *domain specific embedded languages* — yet not “real” languages
- Second-class citizens in compilation process
- Compilers cannot understand abstractions in libraries, and cannot thus optimize or analyze libraries
Towards first class libraries

- Understanding concepts
- Improved support of *syntactic concepts* in programming languages
- Compiler frameworks that can support
  - Concept-based optimizations described in libraries
  - Static analyses/checks described in libraries
Concepts and improved support for syntactic concepts

- Several representations for concepts
  - Many in C++: STL’s semi-formal concept descriptions, Tecton’s descriptions as algebras, Caramel, built-in language support in the works.
  - Manifestations in other programming languages: Type classes in Haskell, ML module system, OO and F-bounded polymorphism

- Unified view to type parameter constraints in different programming languages

- Unified representation to capture all features of modern generic programming (and constraint systems)

- Basis for designing new language features to support generic programming
Compiler frameworks that can support ...
Simplicissimus: Extensible simplifier

- Traditional simplifier
  - Fixed set of rewrite rules applied to built-in types

- Extensible simplifier
  - Rewrite rules for user-defined types
  - Extensible on the set of rewrite rules
  - Extensible on the set of operations

- Descriptions of simplifications *belong in the library*

- Described using concepts
Concept-based rules

- Traditional rules are redundant:
  \[ (+ \ x \ 0) \rightarrow x \]
  \[ (* \ x \ 1) \rightarrow x \]
  \[ (\text{concat} \ x \ "") \rightarrow x \]
  \[ (\text{and} \ x \ \text{true}) \rightarrow x \]

- Only one concept-based rule required to cover all cases:
  \[ (\text{op} \ x \ \text{id}) \rightarrow x \]
  \text{op} \ an \ operator, \ \text{id} \ identity \ element

- Simplification enabled for a user-defined type by modeling the Right Identity concept

- Simplicissimus allows rewriting any C++ expression
  - All rules in a library to be applied by the compiler
Library-specific static analyses

▶ STLlint: static checker for C++ code, that knows about STL
  ▶ Checks iterator invalidation, past-the-end dereferencing
  ▶ Checks proper use of sorting, searching, etc.
  ▶ Extensible, all knowledge is in a library
▶ Enforces semantic constraints on the use of STL algorithms
▶ Decouples semantic checking from algorithm specification
▶ Generic algorithms involve high-level semantic properties
  ▶ introduce, remove, preserve
  ▶ E.g. sort() introduces sorted property
  ▶ Statically keep track of these properties
Library-specific static analysis

- STL: 10 container classes, 2 iterators per container class, \( \sim 70 \) algorithms
- Algorithms and container operations affect high-level properties
- New property \( \rightarrow \) revisit every algorithm (and vice versa)
- Redundant work hampers extensibility \( \rightarrow \) algorithmic concepts
  - Categorize behavior of algorithms
  - No changes needed if new algorithm can be modeled using existing concepts
```
vector<int> v;

// Sort v: after this, v should be ‘sorted’ with predicate ‘less<int>’
sort(v.begin(), v.end(), less<int>());

// Find out where we should insert ‘42’
// This is an error: v was sorted with less<int>, not greater<int>!
vector<int>::iterator i = lower_bound(v.begin(), v.end(), 42, greater<int>());
```

`tryit.cpp`, line 21, warning: sequence may have been sorted with a
different predicate than the one given

```
vector<int>::iterator i =
    lower_bound(v.begin(), v.end(), 42, greater<int>());
```

in call to function lower_bound_pred at "tryit.cpp", line 21
Importance of standardization

- STL was a boon to C++ programmers
- Big reason for this: standardization!
- Never again must I write a string/array/vector class!
- But also, efforts to build tools (like STLlint) become easier to justify, or layer libraries on top of STL.
Related approaches

- Different forms of constrained polymorphism
  - Type classes, module systems, predicate dispatch
- Data-type generic programming
- Generative programming
- Metaprogramming and staged languages
Future work

- Language support for concepts
- Theoretical foundations of concepts, libraries and generic programming
- Compiler support for first-class libraries
- Bigger, better, faster, *optimized*, *checked* libraries
Conclusion

- Software advances with libraries
- Libraries are more than collections of reusable static software elements
- Libraries must integrate more closely with compilers
- Generic programming and concepts provide a discipline for library construction