Programming Language Support for Generic Libraries

Jeremy Siek and Walid Taha

RICE

Abstract

The generic programming methodology is revolutionizing the way we develop software libraries, drastically increasing their adaptability while retaining high performance. Examples of successful generic libraries include:

- Standard Template Library (Stepanov, Lee),
- Boost Graph Library (Siek, Lee, Lumsdaine),
- Blitz++ (Veldhuizen).

Current programming languages provide only partial support for generic programming, making the development and use of generic libraries more difficult than necessary. The goal of our research is to improve language support for generic programming.

Generic libraries rely on two key language technologies:

- **Type parameters constrained with concepts,**
- **Metaprogramming**.

What is generic programming?

- Parameterize algorithms on the data-structure type.
- Capture the essential properties of the data-structures needed to implement the algorithm.
- Group these properties into "concepts".
- Perform dispatching at compile time, ensuring maximum run-time performance.

sort<S>
merge<S1,S2>
transform<S1,S2>
partition<S>

max_flow<G>
shortest_paths<G>
isomorphic<G1,G2>

What is a "concept"?

Sequence
array
list
deque

- A special kind of interface.
- Consists of requirements such as:
  - function signatures,
  - helper types,
  - other concepts (think inheritance),
  - efficiency requirements.

What is a constraint?

- Algorithms must make assumptions about what operations are available on a data-structure.
- Express these assumptions by requiring a type parameter to satisfy a concept.

sort<S> where
S satisfies *Sequence*

max_flow<G> where
G satisfies *Graph*

What is wrong with C++ templates?

- Error messages are indecipherable.
- Bugs lurk in generic libraries.
- Compilation time is slow.

```
// buggy program
int main() {
  list<int> l;
  stable_sort(l.begin(), l.end());
}
```

```
stl_algo.h: In function `void std::__inplace_stable_sort(_RandomAccessIter, _RandomAccessIter)    [with _RandomAccessIter =
std::_List_iterator<int, int&, int*>]':stl_algo.h:2565:   instantiated from `void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
[with _RandomAccessIter = std::_List_iterator<int, int&, int*>]'stable_sort_error.cpp:5:   instantiated from herestl_algo.h:2345: error:
no match for `std::_List_iterator<int, int&, int*>& - std::_List_iterator<int, int&, int*>&' operatorstl_algo.h:2565:   instantiated from
`void std::stable_sort(_RandomAccessIter, _RandomAccessIter)    [with _RandomAccessIter = std::_List_iterator<int, int&,
int*>]'stable_sort_error.cpp:5:   instantiated from herestl_algo.h:2349: error: no match for `std::_List_iterator<int, int&, int*>& -
std::_List_iterator<int, int&, int*>&' operatorstl_algo.h:2352: error: no match for `std::_List_iterator<int, int&, int*>& -
std::_List_iterator<int, int&, int*>&' operatorstl_algo.h:2352: error: no match for `std::_List_iterator<int, int&, int*>& -
std::_List_iterator<int, int&, int*>&' operatorstl_algo.h: In function `void std::__stable_sort_adaptive(_RandomAccessIter,
_RandomAccessIter, _Pointer, _Distance)    [with _RandomAccessIter = std::_List_iterator<int, int&, int*>, _Pointer = int*, _Distance =
int]':stl_algo.h:2567:   instantiated from `void std::stable_sort(_RandomAccessIter, _RandomAccessIter)    [with _RandomAccessIter =
std::_List_iterator<int, int&, int*>]'stable_sort_error.cpp:5:   instantiated from herestl_algo.h:2497: error: no match for
`std::_List_iterator<int, int&, int*>& - std::_List_iterator<int, int&, int*>&' operatorstl_algo.h:2498: error: no match for
`std::_List_iterator<int, int&, int*>& + int&' operatorstl_algo.h:2567:   instantiated from `void std::stable_sort(_RandomAccessIter,
_RandomAccessIter)
```

What is the root of the problem?

- Templates are type checked after instantiation, for each use site.
- Template libraries are not separately compiled, but included as header files.
- There are no constraints on template parameters.
- Templates are just a better behaved form of macro

```
template<typename T>
T min(T a, T b) {
  if (b < a) return b;
  else return a;
}
```

A prototype language: *G*

- Concepts and constraints are part of the language
- Templates are type checked separately from their use
- Templates are separately compiled to object files
- Much better error messages
- Uncovers bugs in generic algorithms

```
fun main() -> int {
  let v = list<int>();
  stable_sort(begin(v), end(v));
  return 0;
}
```

```
concept Comparable<X> {
  fun operator<(X, X) -> bool;
};

// Ok, implementation is valid
fun min<T> where { Comparable<T> }
(T a, T b) -> T {
  if (b < a) return b;
  else return a;
}
// Ok, int satisfies Comparable
model Comparable<int> { };

min(1,2); // Ok, constraint satisfied
```

```
Error: In application stable_sort(begin(v), end(v)),
Model RandomAccessIterator<list_iter<int>> needed to satisfy requirement,
but it is not defined.
```

Work on the C++ Standards Committee to add Concepts

- Wrote a proposal to add concepts to C++: paper N1849.
  - http://www.open-std.org/jtc1/sc22/wg21/
- Implemented the proposal in the GNU C++ compiler: paper N1848.
- Look for concepts in C++200X!

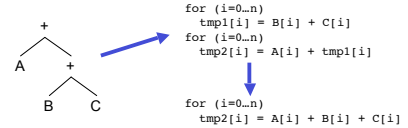
What is metaprogramming?

- Compile-time computation, often performing code generation.
- Metaprogramming can be done in C++ using templates.

```
template<int n>
struct fact {
  static const int result = n * fact<n-1>::result;
};
template<>
struct fact<0> {
  static const int result = 1;
};
// compute factorial of 5 at compile time
// and use as size for array.
int array[fact<5>::result];
```

What is metaprogramming used for?

- Self-optimizing libraries, such as Blitz++, perform loop fusion and loop unrolling to speed computation on arrays.
- User-configurable data-structures, such as a graph class that can be optimized for fast traversal or for vertex and edge insertion and removal.



How are we improving metaprogramming?

- Create better type systems to catch bugs in the metaprograms and to catch bugs in the generated programs.
- Simplify language constructs for metaprogramming. (Metaprogramming with templates is baroque!)

Acknowledgments

- This project is funded by the National Science Foundation.
- Thanks to our collaborators Andrew Lumsdaine and members of the Open Systems Laboratory at Indiana University.
- Part of this work were also supported by NSF grant EIA-0131354 and by a grant from the Lilly Endowment.

Further reading

- *Essential language support for generic programming.* Jeremy Siek and Andrew Lumsdaine. In *Programming Language Design and Implementation 2005.*
- *Environment Classifiers.* Walid Taha and Michael Nielsen. In *Principles of Programming Languages 2003.*
- *A Comparative Study of Language Support for Generic Programming.* Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock. In *OOPSLA'03.*