Generic Programming

Akim Demaille akim@lrde.epita.fr
Roland Levillain roland@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

June 14, 2012

Generic Programming

Some definitions

A Definition of Generic Programming [4, 2]

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction.

A Definition of Generic Programming [4, 2] (cont.)

Key ideas include:

- *Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.*
- *Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.*

A Definition of Generic Programming [4, 2] (cont.)

- *When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.*
  Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.

CLU

Genericity in CLU

- First ideas of generic programming date back from CLU [5] (in 1974, before it was named like this).
- Some programming concepts present in CLU:
  - data abstraction (encapsulation)
  - iterators
  - parameterized modules
- In CLU, modules are implemented as *clusters* which are programming unit grouping a data type and its operations.
- Notion of parametric polymorphism.

Parameterized modules in CLU

- Initially: parameters checked at run time.
- Then: introduction of `where`-clauses (requirements on parameter(s)).
- Only operations of the type parameter(s) listed in the `where`-clause may be used.

→ Complete compile-time check of parameterized modules.

→ Generation of a single code.

An example of parameterized module in CLU

```
set = cluster [t: type] is
  create, member, size, insert, delete, elements
where t has equal: proctype (t, t) returns (bool)
```

- Note:
  Inside `set`, the only valid operation on `t` values is `equal`.

Implementation of parameterized modules in CLU

- Notion of *instantiation*:
  binding a module and its parameter(s) [1].
- Syntax: *module* [*parameter*]
- *Dynamic instantiation* of parameterized modules.
- For a given module, each distinct set of parameters is represented by a (run-time) object.
- Instantiated modules derived from a non-instantiated object module. Common code is shared.
- Pros and cons of run- or load-time binding:

  Pros No combinatorial explosion due to systematic code generation (as with C++ templates).

  Cons Lack of static instantiation context means less opportunities to optimize.

Ada 83

Genericity in Ada 83

Introduced with the `generic` keyword [6].

```
generic
  type T is private;
procedure swap (x, y : in out T) is
  t : T
begin
  t := x; x := y; y := t;
end swap;

-- Explicit instantiations.
procedure int_swap is new swap (INTEGER);
procedure str_swap is new swap (STRING);
```

- Example of unconstrained genericity.
- Instantiation of generic clauses is explicit (no implicit instantiation as in C++).

Generic packages in Ada 83

```
generic
  type T is private;
package STACKS is
  type STACK (size : POSITIVE) is
    record
      space : array (1..size) of T;
      index : NATURAL
    end record;
  function empty (s : in STACK) return BOOLEAN;
  procedure push (t : in T; s : in out STACK);
  procedure pop (s : in out STACK);
  function top (s : in STACK) return T;
end STACKS;

package INT_STACKS is new STACKS (INTEGER);
package STR_STACKS is new STACKS (STRING);
```

Constrained Genericity in Ada 83

- Constrained genericity imposes restrictions on generic types:

```
generic
  type T is private;
  with function "<=" (a, b : T) return BOOLEAN is <>;
function minimum (x, y : T) return T is begin
    if x <= y then
    return x;
  else
    return y;
  end if;
end minimum;
```

- Constraints are only of syntactic nature (no formal constraints expressing semantic assertions)

Constrained Genericity in Ada 83: Instantiation

- Instantiation can be fully qualified

```
function T1_minimum is new minimum (T1, T1_le);
```

- or take advantage of implicit names:

```
function int_minimum is new minimum (INTEGER);
```

  Here, the comparison function is already known as “<=”.

More Genericity Examples in Ada 83

Interface ("specification"):

```
-- matrices.ada
generic
  type T is private;
  zero : T;
  unity : T;
  with function "+" (a, b : T) return T is <>;
  with function "*" (a, b : T) return T is <>;
package MATRICES is
  type MATRIX (lines, columns: POSITIVE) is
    array (1..lines, 1..columns) of T;
  function "+" (m1, m2 : MATRIX) return MATRIX;
  function "*" (m1, m2 : MATRIX) return MATRIX;
end MATRICES;
```

More Genericity Examples in Ada 83

Instantiations:

```
package FLOAT_MATRICES is new MATRICES (FLOAT, 0.0, 1.0);
```

```
package BOOL_MATRICES is
  new MATRICES (BOOLEAN, false, true, "or", "and");
```

More Genericity Examples in Ada 83

Implementation ("body"):

```
-- matrices.adb
package body MATRICES is
  function "*" (m1, m2 : MATRIX) is
    result : MATRIX (m1'lines, m2'columns)
  begin
    if m1'columns /= m2'lines then
      raise INCOMPATIBLE_SIZES;
    end if;
    for i in m1'RANGE(1) loop
      for j in m2'RANGE(2) loop
        result (i, j) := zero;
        for k in m1'RANGE(2) loop
          result (i, j) := result (i, j) + m1 (i, k) * m2 (k, j);
        end loop;
      end loop;
    end loop;
  end "*";
  -- Other declarations...
end MATRICES;
```

A History of C++ Templates [7]

- Initial motivation: provide parameterized containers.
- Previously, *macros* were used to provide such containers (in C and C with classes).
- Many limitations, inherent to the nature of macros:
  - Poor error messages, referring to the code written by cpp, not by the programmer.
  - Need to instantiate templates once per compile unit, *manually*.
  - No support for recurrence.

Simulating parameterized types with macros

```
#define VECTOR(T) vector_ ## T

#define GEN_VECTOR(T)                                \
  class VECTOR(T) {                                  \
  public:                                            \
    typedef T value_type;                            \
    VECTOR(T)() { /* ... */ }                        \
    VECTOR(T)(int i) { /* ... */ }                   \
    value_type& operator[](int i) { /* ... */ }      \
    /* ... */                                        \
  }

// Explicit instantiations.
GEN_VECTOR(int);
GEN_VECTOR(long);

int main() {
  VECTOR(int) vi;
  VECTOR(long) vl;
}
```

A History of C++ Templates [7] (cont.)

- Introduction of a *template* mechanism around 1990, later refined (1993) before the standardization of C++ in 1998.
- Class templates.
- Function templates (and member function templates).
- Automatic deduction of parameters of template functions.
- Type and non-type template parameters.
- No explicit constraints on parameters.
- Implicit (automatic) template instantiation (though explicit instantiation is still possible).
- Full (classes, functions) and partial (classes) specializations of templates definitions.
- A powerful system allowing metaprogramming techniques (though not designed for that in the first place!)

Class Templates

```
template <typename T>
class vector {
public:
  typedef T value_type;
  vector() { /* ... */ }
  vector(int i) { /* ... */ }
  value_type& operator[](int i) { /* ... */ }
  /* ... */
};

// No need for explicit template instantiations.

int main() {
  vector<int> vi;
  vector<long> vl;
}
```

Function Templates

Natural in a language providing non-member functions (such as C++).

```
template <typename T>
void swap (T& a, T& b)
{
  T& tmp = a;
  a = b;
  b = tmp;
}
```

Simulating Function Templates

Class templates can make up for the lack of generic functions in most uses cases.

```
template <typename T>
struct swap
{
  static void go (T& a, T& b)
  {
    T& tmp = a;
    a = b;
    b = tmp;
  }
};
```

(Eiffel does not feature generic function at all. Java and C# provide only generic *member* functions.)

Automatic deduction of parameters

- Parameters do not need to be explicitly passed when the compiler can deduce them from the actual arguments.

```
int a = 42;
int b = 51;
swap (a, b);
```

- A limited form of *type inference*.
- Explicit specialization is still possible.

```
int a = 42;
int b = 51;
swap<long> (a, b);
```

Automatic deduction of parameters (cont.)

- This mechanism does not work for classes.
  E.g., if the '<<' stream operator is defined for `std::pair`, one cannot write

```
std::cout << std::pair (3.14f, 42) << std::endl;
```

  since `std::pair` is *not* a type!

- The right syntax for this statement is

```
std::cout << std::pair<float, int> (3.14f, 42) << std::endl;
```

  which is painfully long.

- But *Object Generators* [8] can make up for this lack.

```
std::cout << std::make_pair (3.14f, 42) << std::endl;
```

Specialization of Template Definitions

- Idea: provide another definition for a subset of the parameters.
- Motivation: provide (harder,) better, faster, stronger implementations for a given template class or function.
- Example: `std::vector<bool>` has its own definition, different from `std::vector<T>`.
- Mechanism close to *function overloading* in spirit, but distinct.

No Explicit Constraints on Template Parameters

Remember this piece of code from the course on OO Languages?

```
#include <iostream>
#include <list>

int
main ()
{
  std::list<int> list;
  list.push_back (1);
  list.push_back (2);
  list.push_back (3);
  const std::list<int> list2 = list;

  for (std::list<int>::iterator i = list2.begin ();
       i != list2.end (); ++i)
    std::cout << *i << std::endl;
}
```

Poor Error Messages

G++ 2.95:

```
bar.cc: In function ‘int main()’:
bar.cc:13: conversion from
   ‘_List_iterator<int,const int &, const int *>’
   to non-scalar type
   ‘_List_iterator<int,int &, int *>’ requested
bar.cc:14: no match for
   ‘_List_iterator<int,int &,int *> & !=
      _List_iterator<int,const int &,const int *>’
/usr/lib/gcc-lib/i386-linux/2.95.4/../../../../include/g++-3/stl_list.h:70:
   candidates are:
   bool _List_iterator<int,int &,int *>::operator !=
     (const _List_iterator<int,int &,int *> &) const
```

(A Bit Less) Poor Error Messages

Some progress: G++ 3.3.

```
list-invalid.cc: In function ‘int main()’:
list-invalid.cc:13: error: conversion from
   ‘std::_List_iterator<int, const int&, const int*>’
   to non-scalar type
   ‘std::_List_iterator<int, int&, int*>’ requested
```

G++ 3.4, 4.0 and 4.1, 4.2, 4.3 and 4.4.

```
list-invalid.cc: In function ‘int main()’:
list-invalid.cc:13: error: conversion from
   ‘std::_List_const_iterator<int>’ to non-scalar type
   ‘std::_List_iterator<int>’ requested
```

G++ 4.5.

```
list-invalid.cc: In function ‘int main()’:
list-invalid.cc:13:50: error: conversion from
   ‘std::list<int>::const_iterator’ to non-scalar type
   ‘std::list<int>::iterator’ requested
```

(A Bit Less) Poor Error Messages

G++ 4.6 and 4.7.

```
list-invalid.cc: In function ‘int main()’:
list-invalid.cc:13:50: erreur: conversion from
   ‘std::list<int>::const_iterator {aka std::_List_const_iterator<int>}’
   to non-scalar type
   ‘std::list<int>::iterator {aka std::_List_iterator<int>}’ requested
```

G++ 4.8 (forthcoming).

```
list-invalid.cc: In function 'int main()':
list-invalid.cc:13:50: error: conversion from
  'std::list<int>::const_iterator {aka std::_List_const_iterator<int>}'
  to non-scalar type
  'std::list<int>::iterator {aka std::_List_iterator<int>}' requested
   for (std::list<int>::iterator i = list2.begin ();
                                                 ^
```

(A Bit Less) Poor Error Messages

ICC 8.1 and 9.1.

```
list-invalid.cc(8):
   remark #383: value copied to temporary, reference
                to temporary used
   list.push_back (1);
                   ^
[...]
list-invalid.cc(13): error: no suitable user-defined conversion
   from
   "std::list<int, std::allocator<int>>::const_iterator" to
   "std::list<int, std::allocator<int>>::iterator" exists
    for (std::list<int>::iterator i = list2.begin ();
                                      ^
```

ICC 10.0 and 11.0.

```
list-invalid.cc(13): error: no suitable user-defined conversion
  from "std::_List_const_iterator<int>"
  to "std::_List_iterator<int>" exists
    for (std::list<int>::iterator i = list2.begin ();
                                      ^
```

(A Bit Less) Poor Error Messages

Clang 1.1 (LLVM 2.7)

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
  for (std::list<int>::iterator i = list2.begin ();
                                ^   ~~~~~~~~~~~~~~
In file included from list-invalid.cc:2:
In file included from /usr/include/c++/4.2.1/list:69:
/usr/include/c++/4.2.1/bits/stl_list.h:113:12: note: candidate
      constructor (the implicit copy constructor) not viable:
      no known conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'struct std::_List_iterator<int> const' for 1st argument
    struct _List_iterator
           ^
1 error generated.
```

(A Bit Less) Poor Error Messages

Clang 2.8 (LLVM 2.8).

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
  for (std::list<int>::iterator i = list2.begin ();
                                ^   ~~~~~~~~~~~~~~
In file included from list-invalid.cc:2:
In file included from /usr/include/c++/4.2.1/list:69:
/usr/include/c++/4.2.1/bits/stl_list.h:112:12: note: candidate
      constructor (the implicit copy constructor) not viable:
      no known conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::_List_iterator<int> const &' for 1st argument
    struct _List_iterator
           ^
1 error generated.
```

(A Bit Less) Poor Error Messages

Clang 2.9 (LLVM 2.9).

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
  for (std::list<int>::iterator i = list2.begin ();
                                ^   ~~~~~~~~~~~~~~
In file included from list-invalid.cc:2:
In file included from /usr/include/c++/4.2.1/list:69:
/usr/include/c++/4.2.1/bits/stl_list.h:112:12: note: candidate
      constructor (the implicit copy constructor) not viable:
      no known conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'const std::_List_iterator<int> &' for 1st argument
    struct _List_iterator
           ^
1 error generated.
```

(A Bit Less) Poor Error Messages

Clang 3.0 (LLVM 3.0) and Clang 3.1 (LLVM 3.1).

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
  for (std::list<int>::iterator i = list2.begin ();
                                ^   ~~~~~~~~~~~~~~



/usr/include/c++/4.2.1/bits/stl_list.h:112:12: note: candidate
      constructor (the implicit copy constructor) not viable:
      no known conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'const std::_List_iterator<int> &' for 1st argument;
    struct _List_iterator
           ^
1 error generated.
```

Templates in the C++ Standard Library

The Standard Template Library (STL)

- A library of containers, iterators, fundamental algorithms and tools, using C++ templates.
- Designed by Alexander Stepanov at HP.
- The STL is not the Standard C++ Library (nor is one a subset of the other)—although most of the STL is part of the ISO C++ Standard [3]
- Introduces the notion of *concept*: a set of *syntactic* and *semantic* requirements over one (or several) types.
- But the language does not enforce them.
- Initially planned as a language extension in the C++1x standard...
- ...but eventually will not be a part of it. :-(

An example of Concept: *Container*

`http://www.sgi.com/tech/stl/Container.html`

Refinement of

Assignable

Associated types

Type	typedef	Meaning (abridged)
Value type	`X::value_type`	The type of the object stored.
Iterator type	`X::iterator`	The type of iterator used to iterate.
Const iterator type	`X::const_iterator`	Likewise, does not modify elements.
Reference type	`X::reference`	A type that behaves as a reference.
Const reference type	`X::const_reference`	A type that behaves as a const ref.
Pointer type	`X::pointer`	A type that behaves as a pointer.
Distance type	`X::difference_type`	Type used to represent a distance between two iterators.
Size type	`X::size_type`	Type for nonnegative distance.

An example of Concept: *Container* (cont.)

Valid expressions (abridged)

Name	Expression	Return type
Beginning of range	`a.begin()`	`iterator` if `a` is mutable, `const_iterator` otherwise
End of range	`a.end()`	`iterator` if `a` is mutable, `const_iterator` otherwise
Size	`a.size()`	`size_type`
Maximum size	`a.max_size()`	`size_type`
Empty container	`a.empty()`	Convertible to `bool`
Swap	`a.swap(b)`	`void`

An example of Concept: *Container* (cont.)

Complexity guarantees

- The copy constructor, the assignment operator, and the destructor are linear in the container's size.
- `begin()` and `end()` are amortized constant time.
- `size()` is linear in the container's size. `max_size()` and `empty()` are amortized constant time. If you are testing whether a container is empty, you should always write `c.empty()` instead of `c.size() == 0`. The two expressions are equivalent, but the former may be much faster.
- `swap()` is amortized constant time.

An example of Concept: *Container* (cont.)

Invariants

Valid range	For any container a,[`a.begin()`, `a.end()`) is a valid range.
Range size	`a.size()` is equal to the distance from `a.begin()` to `a.end()`.
Completeness	An algorithm that iterates through the range [`a.begin()`, `a.end()`) will pass through every element of a.

Models

- std::vector

Static Metaprogramming

- Metaprograms: programs manipulating programs.
- Static metaprograms: programs “running” at compile-time.
- Notions of two-stage programming (compile and run times), code generation.
- Limited form of static introspection and reflection.
- C++ templates can be used to implement template metaprograms.
- Template metaprogramming is Turing-complete.
- Applications : compile-time functions, functions on types, static assertions, code factoring, etc.

An Example of Compile-Time Function

A compile-time definition of factorial:

```
template <int n>
struct fact
{
  static const int val =
    n * fact<n - 1>::val;
};

template<>
struct fact<0>
{
  static const int val = 1;
};

int main()
{
  int x = fact<4>::val; // == 24
}
```

- “Function” implemented as a class template.
- “Argument(s)” passed as template parameter(s).
- “Return value” returned as a class (static) attribute.
- Pure function: no side effects (except compilation errors).
- Uses recursive template instantiations.

Nature and Origins of Template Metaprogramming

- Template metaprograms are very dependent of C++ idiosyncrasies with respect to templates.
  - Explicit specialization mechanism.
  - Implicit (automatic) template instantiation.
- Verbose and unfriendly syntax.
- Template metaprogramming discovered almost by accident by Erwin Unruh, who wrote a program printing out a list of prime numbers at compile-time as error messages.
- Term "template metaprogramming" coined by Todd Veldhuizen.
- A major programming paradigm of modern C++ (used in many Boost libraries, etc.).

A Metaprogramming Example of the Tiger Compiler

Problem

- Problem: We need two hierarchies of visitors to traverse Abstract Syntax Trees (ASTs):
  - a read-only version: `Visitor`
  - a read-write version: `ConstVisitor`.
- Likewise for default traversals (`DefaultVisitor` and `DefaultConstVisitor`).
- Similar to STL's `iterator` and `const_iterator`.

A Metaprogramming Example of the Tiger Compiler

Visitor vs ConstVisitor

```
class Visitor
{
  virtual void operator() (NilExp& e);
  virtual void operator() (IntExp& e);
  virtual void operator() (StringExp& e);
  virtual void operator() (CallExp& e);
  // ...
};

class ConstVisitor
{
  virtual void operator() (const NilExp& e);
  virtual void operator() (const IntExp& e);
  virtual void operator() (const StringExp& e);
  virtual void operator() (const CallExp& e);
  // ...
};
```

A Metaprogramming Example of the Tiger Compiler

Solutions

- Duplicate the code.
  - → Very bad: error prone, not robust to code evolution, etc.
- Generate the code using C++ macros
  - → Painful and low-level approach (see previous examples about macro-based genericity).
- Generate the code using a third-party language, e.g. M4.
  - → Adds an extra dependency.
- Generate the code at compile-time using template metaprogramming.
  - → Best compromise between maintenance efforts, dependency minimization and debugging difficulty.

Factoring visitors with respect to const

A First Idea

```
template <type_qualifier Constness>
class GenVisitor
{
  virtual void operator() (Constness NilExp& e);
  virtual void operator() (Constness IntExp& e);
  // ...
};
```

Not applicable as-is in C++ ...

Factoring visitors with respect to const

Making Constness a Function

```
template <type_function Constness>
class GenVisitor
{
  virtual void operator() (Constness(NilExp)& e);
  virtual void operator() (Constness(IntExp)& e);
  // ...
};
```

where `Constness` can be a function on types such as :

- $T \mapsto T$ (identity); or
- $T \mapsto$ `const` T (const-ification of T).

Remarks:

- Still invalid C++ syntax, but...
- ... can be implemented in valid C++ using template metaprogramming!

Factoring visitors with respect to const

Functions on types

Traits (functions on types) from tc's 'lib/misc/select_const.hh':

```
/// Return \a T as is.
template <typename T>
struct id_traits
{
  typedef T type;
};

/// Return \a T constified.
template <typename T>
struct constify_traits
{
  typedef const T type;
};
```

- "Return value" expressed as a `typedef`.
- "Call" syntax:
  - `id_traits<int>::type`
  - `constify_traits<int>::type`
- Traits invocations preceded by the `typename` keyword in template contexts.

Factoring visitors with respect to const

Using traits to implement GenVisitor

```
template <template <typename> class Const>
class GenVisitor
{
  virtual void operator() (typename Const<NilExp>::type& e);
  virtual void operator() (typename Const<IntExp>::type& e);
  // ...
};

typedef GenVisitor<id_traits> Visitor;
typedef GenVisitor<constify_traits> ConstVisitor;
```

Bibliography I

Russell R. Atkinson, Barbara H. Liskov, and Robert W. Scheifler.
Aspects of implementing CLU.
In *Proceedings of the 1978 annual conference*, ACM '78, pages 123–129, New York, NY, USA, 1978. ACM.

Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock.
A comparative study of language support for generic programming.
In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, And Applications (OOPSLA)*, pages 115–134, New York, NY, USA, 2003. ACM Press.

Bibliography II

ISO/IEC.
ISO/IEC 14882:2003 (e). Programming languages — C++, 2003.

M. Jazayeri, R. Loos, and D. Musser, editors.
Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers, volume 1766 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

Barbara Liskov.
A history of CLU.
In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 133–147, New York, NY, USA, 1993. ACM.

Bibliography III

Bertrand Meyer.
Genericity versus inheritance.
ACM SIGPLAN Notices, 21(11):391–405, 1986.

Bjarne Stroustrup.
The Design and Evolution of C++.
ACM Press/Addison-Wesley Publishing Co., 1994.

The Boost Project.
Generic programming techniques.
`http://www.boost.org/community/generic_programming.html`, 2008.