Typology of programming languages

History of Genericity

Table of Contents

Barbara Liskov

Barbara Liskov

- Nov. 7, 1939
- Stanford
- PhD supervised by J. McCarthy
- Teaches at MIT
- CLU (pronounce “clue”)
- John von Neumann Medal (2004)
- A. M. Turing Award (2008)

CLU syntax and semantic

CLU looks like an Algol-like language,
but its semantics is like that of Lisp

History of CLU:
`ftp://ftp.lcs.mit.edu/pub/pclu/CLU/3.Documents/clu-history.PS`

Problem Statement

How to write a data structure or algorithm that can work with elements of many different types?

Quote on CLU by B. Liskov

> *An abstract data type is a concept whose meaning is captured in a set of specifications [...] An implementation is correct if it "satisfies" the abstraction's specification.*
>
> – *B. Liskov*

Genericity in CLU

- First ideas of generic programming date back from CLU (in 1974, before it was named like this [HOPL'93]).
- Some programming concepts present in CLU:
  - data abstraction (encapsulation)
  - iterators (well, *generators* actually)
  - type safe variants (*oneof*)
  - multiple assignment (`x, y, z = f(t)`)
  - parameterized modules

Genericity in CLU

- In CLU, modules are implemented as *clusters*
  programming units 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)
- Syntax: *module*[*parameter*]
- *Dynamic instantiation* of parameterized modules.

Implementation of parameterized modules in CLU

- 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.

Table of Contents

Genericity in Ada 83

Introduced with the **generic** keyword

```
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"):

```ada
-- 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;
```

Table of Contents

A History of C++ Templates

- 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 (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.

A History of C++ Templates (cont.)

- 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

```cpp
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 with non-member functions (such as C++).

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

- Class templates can make up for the lack of generic functions in most uses cases (through **fonctor**).
- Eiffel does not feature generic function at all.
- Java and C-sharp provide only generic *member* functions.

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: boolean vector has its own definition, different from type T vector
- Mechanism close to *function overloading* in spirit, but distinct.

Alexander Alexandrovich Stepanov (Nov. 16, 1950)

Алекса́ндр Алекса́ндрович Степа́нов

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 it is part of the standard
- Introduces the notion of *concept*: a set of *syntactic* and *semantic* requirements over one (or several) types.
- But the language does not enforce them.

Example

```
template<typename T>
concept Hashable =
requires(T a) {
  { std::hash<T>{}(a) } ->
   std::convertible_to
            <std::size_t>;
};


// constrained C++20
// function template
template<Hashable T>
void f(T);
```

Summary

templates

generics

partial specialization

Constraints