Typology of Programming Languages

Genericity

May 2025

Exercise

What is generic programming?

Exercise

What is generic programming?

Definition

Generic programming consists in writing algorithms and data structures that can be used with multiple types interchangeably. These generic functions and types are then *specialized* or *instantiated* for a given concrete type.

Exercise

What is generic programming?

Definition

Generic programming consists in writing algorithms and data structures that can be used with multiple types interchangeably. These generic functions and types are then *specialized* or *instantiated* for a given concrete type.

Genericity is the possibility of abstracting over multiple types, whereas **polymorphism** refers to representing multiple types at once.

Generic types

```
template <typename T>
struct List<T> { ... };
```

`List<T>` is a *generic* (or *polymorphic*) type: represents multiple data types.

- `T` is called a *type parameter*.
- `List` is called a *type constructor*.
  - can be seen as a function taking types as parameters and returning types (cf Zig generics).
  - Applying types parameters to a type constructor gives a concrete, *monomorphic* (*i.e.* no longer generic) type.

Generic functions

```
template <typename T>
void print( const List<T>& l ) { ... }
```

`print` is a *generic function*: can be used with multiple data types for its arguments interchangeably.

Section 1

A History of Generics

Subsection 1

CLU

Barbara Liskov

Barbara Liskov

- Born Nov. 7, 1939
- Stanford
- PhD supervised by J. McCarthy
- Teaches at MIT
- CLU (pronounced “clue”) (1975)
- John von Neumann Medal (2004)
- A. M. Turing Award (2008)

CLU syntax and semantic

CLU looks like an Algol-like language but with semantics closer to Lisp.

- A History of CLU:
  https://dspace.mit.edu/bitstream/handle/1721.1/149734/MIT-LCS-TR-561.pdf
- An interesting interview of Liskov from 2016:
  https://amturing.acm.org/interviews/liskov_1108679.cfm

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

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*.

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)
```

Inside `set` functions, the only valid operation on `t` values is `equal`.

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 could be used.
- Complete compile-time check of parameterized modules.
- Generation of a single unit of code.

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

- Common code is shared accross different instantiations of the same parameterized module.
- 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.

Subsection 2

Ada

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;
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.ads (specification, or header)
generic
    type T is private;
    zero: T;
    unity: T;
    with function "+" (a, b: T) return T;
    with function "*" (a, b: T) return T;
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

```
-- matrices.adb (implementation, or body)
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;
```

Subsection 3

C++

A History of C++ Templates

- Initial motivation: provide parameterized containers.
- Previously, *macros* were used to provide such containers
- 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 once per compile unit, *manually*.
  - No recursivity.

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 until 2020 (though `enable_if` tricks were used before that).
- 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:
    using value_type = T;
    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 that allows non-member functions (called “free functions” in 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 have generic functions 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.

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 now 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.
  - Initially planned as a language extension in the C++11/14/17 standard…
  - …and finally adopted in C++20.

Alexander Alexandrovich Stepanov (Nov. 16, 1950)

Example

```cpp
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);
```

Section 2

Behind Generics

Generics in C?

Exercise

How do you implement a generic linked list in C?

Generics in C?

Exercise

How do you implement a generic linked list in C?

```c
struct list {
  void* data;
  struct list* next;
};
```

```c
#define LIST(T) \
struct list_##T { \
  T data; \
  struct list_##T* next; \
}
```

Generics in C?

Exercise

What are the pros and cons of both approaches?

Generics in C?

Exercise

What are the pros and cons of both approaches?

	Pros	Cons
Macro	Strong typing, homogenous	Many implementations, slower to compile, need to instantiate for each type
`void*`	Only one implementation, heterogenous	Weak typing, need to have a pointer to a `free` function for the type hidden by `void*`, need runtime checks

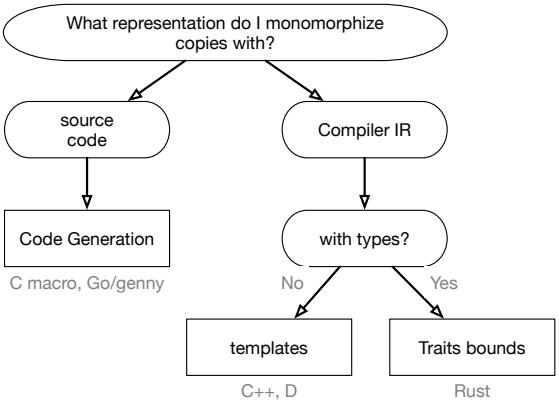
Subsection 1

Monomorphization

Monomorphization

The **monomorphization** approach outputs multiple versions of the code for each type we want to use it with.

- C++ template
- (C macros, in a way)
- Rust generics (and procedural macros, in away)
- D, Ada



Monomorphization strategies

C monomorphization

```
#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;
}
```

C++ 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;
}
```

Rust monomorphization

```rust
fn printer<T: Display>(t: T) {
    println!("{}", t);
}

// Bounding restricts the parameter T to types that conform
// to the bounds.
struct S<T: Display>(T);

// Error! `Vec<i32>` does not implement `Display`.
// This specialization will fail.
let s = S(vec![1]);
```

Rust polymorphization

```rust
fn foo<A, B>(_: B) { }

fn main() {
    foo::<u64, u32>(2);
    foo::<u32, u32>(2);
    foo::<u16, u32>(1);
}
```

- An optimisation which determines when functions, closures and generators could remain polymorphic during code generation.
- Polymorphization will identify A as being unused: only one instance for `foo<_, u32>`.

Subsection 2

Boxing

Boxing: main idea

Put everything in uniform "boxes" so that they all act the same way

- The data structure only handles pointers
- Pointers to different types act the same way
- ... so the same code can deal with all data types!

Boxing: main idea

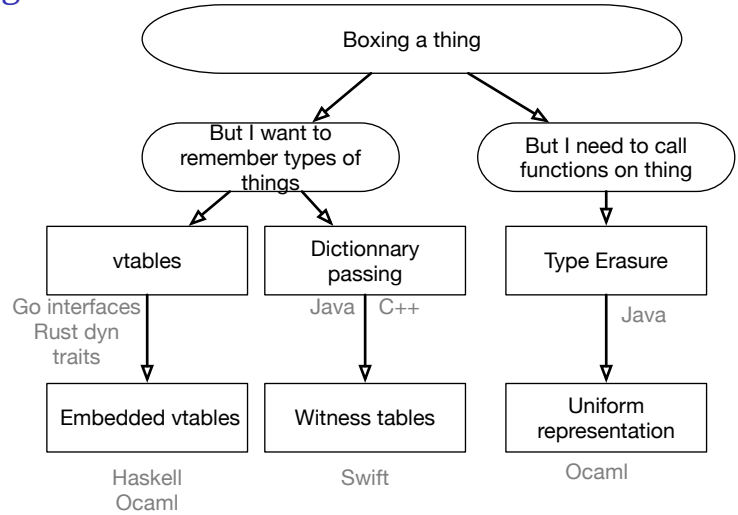
Put everything in uniform "boxes" so that they all act the same way

- The data structure only handles pointers
- Pointers to different types act the same way
- ... so the same code can deal with all data types!

Widely used strategy:

- C: use void pointers + dynamic cast
- Go: interface
- Java (pre-generics): Objects
- Objective-C (pre-generics): id

Various boxing



Boxing strategies

Pro/cons with the boxing approach

Pros:

- Easy to implement in (any) language

Cons:

- Casts for every read/write in the structure
  $\Longrightarrow$ runtime overhead!
- Error-prone: type-checking
  $\Longrightarrow$ Sometimes harder for the compiler to prevent us from manipulating elements of incompatible types

Type-erased boxed generics

- Idea: add generics functionality to the type system
- BUT use the basic boxing method exactly as before at runtime.

  $\Rightarrow$ This approach is often called **type erasure**, because the types in the generics system are "erased" and all become the same type

Java and Objective-C both started out with basic boxing but added features for generics with type erasure

Java Example

Without Generics (pre Java 4.0):
Throws *java.lang.ClassCastException*

```
List v = new ArrayList();

v.add("test");

// A String cannot be cast to
// an Integer => Run time error.
Integer i = (Integer) v.get(0);
```

Java Example

Without Generics (pre Java 4.0):
Throws *java.lang.ClassCastException*

```
List v = new ArrayList();

v.add("test");

// A String cannot be cast to
// an Integer => Run time error.
Integer i = (Integer) v.get(0);
```

With Generics:
Fails at compile time!

```
List<String> v =
  new ArrayList<String>();
v.add("test");


// Compile-time (type) error.
Integer i = v.get(0);
```

Inferred boxed generics with a uniform representation

- Problem with simple boxing:
  - In the previous approach, generic data structures cannot hold primitive types!
  - …hence `Integer` *vs* `int` in Java: explicit boxing of primitive types.

- OCaml's solution: *Uniform representation* where no primitive types require an additional boxing allocation!

Inferred boxed generics with a uniform representation (cont'd)

In OCaml:

- no additional boxing allocation for primitive types (ints do not need to be turned into an `Integer` object)
- everything is either already boxed or represented by a pointer-sized integer
  $\Longrightarrow$ everything is accessed via one machine word
- Problem: the language (especially the garbage collector) needs to know what is a pointer (and thus should be dereferenced) and what is a self-contained value (to be used directly) $\Rightarrow$ the most significant bit of the word is reserved and used as a "marker"
  - (hence 31 or 63 bits-wide integers in OCaml!)

Introducing Interfaces

Limitation with boxing

Once compiled, the boxed types are completely opaque!
A generic sorting function needs some extra functionality, like a type-specific comparison function.

$\Rightarrow$ **Dictionnary passing**
$\Rightarrow$ **Interface vtables**

Dictionnary passing

Dictionary passing Haskell (type class), OCaml (modules)

- Pass a table of the required function pointers along to generic functions that need them
- similar to constructing Go-style interface objects at the call site

A note on Dictionnary passing

Swift Witness Tables

- Use dictionary passing and put the size of types and how to move, copy and free them into the tables,
- Provide all the information required to work with any type in a uniform way
- ...without boxing them (monomorphization).

Going further

Have a look at **Intensional Type Analysis**.

Interface vtables

Interface vtables Rust (*dyn traits*) & Golang (*interface*)

- When casting to interface type it creates a wrapper
- The wrapper contains:
  1. a pointer to the original object, and
  2. a pointer to a vtable of the type-specific functions for that interface

Go example

A stack without any constraint on the contained type:

```
type Stack struct {
  values []interface{}
}

func (this *Stack) Push(value interface{}) {
  this.values = append(this.values, value)
}
```

Section 3

Genericity in OCaml

Basics

```
let id x = x
(* Type according to the REPL: *)
(* val id : 'a -> 'a *)
```

```
let pair x y = (x, y)
(* val pair : 'a -> 'b -> 'a * 'b *)
```

```
let a = pair 42 (id "42")
(* val a : int * string = (42, "42") *)
let b = pair (id 1) 2
(* val b : int * int = (1, 2) *)
```

`id` is a generic function: it takes an argument of a type `'a` (*i.e.* any given type) and returns a value of the same type.

`pair` is generic as well. It takes two arguments of type `'a` and `'b` and returns a pair of those types.

`a` and `b` are variables.

Operating on polymorphic types

Exercise

What if we want to perform operations on a given type `'a`?

Operating on polymorphic types

Exercise

What if we want to perform operations on a given type `'a`?

Pass a function along with the value!

```
let rec map l f =
  match l with
  | [] -> []
  | h :: t -> f h :: (map t f)
```

Operating on polymorphic types

Exercise

What if we want to perform operations on a given type `'a`?

Pass a function along with the value!

```
let rec map l f =
  match l with
  | [] -> []
  | h :: t -> f h :: (map t f)

(* val map : 'a list -> ('a -> 'b) -> 'b list *)
```

Packaging types and values

For a function or type generic on a type `'t`, we may need many functions working on T:

```
val compare : 't -> 't -> int
val equal : 't -> 't -> bool
val dump : 't -> unit
val hash : 't -> int
(* ... *)
```

Passing each of these functions as a separate argument is not reasonnable. We need to package them along with the type `'t`.

Modules

Modules can be used to bundle types and values (variables/functions) together.

```
module M = struct
  type t = ...

  let x = ...

  let f x y = ...

  module SubModule = struct
    let nested_function () = ...
  end
end

let _ = M.Submodule.nested_function ()
```

Module types/signatures

Modules have *types* (or *signatures*).

```
module X = struct
  type x = int * string
  let check (n, s) = String.equal (Int.to_string n) s
end

(*
module X : sig
  type x = int * string
  val check : int * string -> bool
end
*)
```

Each file is a module

```
(* In set.ml *)
type 'a t = 'a list
let foo = ()
let empty = []
let is_empty = function [] -> true | _ -> false
let singleton x = [ x ]

(* In set.mli, the interface for the module of set.ml *)
type 'a t (* no definition provided for t: it is an abstract type *)
val empty : 'a t
val is_empty : 'a t -> bool
val singleton : 'a -> 'a t
(* foo is not in the module interface: the symbol is not exported *)

(* In main.ml *)
let my_set = Set.empty in
let my_other = Set.singleton 42 in
assert (is_empty my_set && not (is_empty singleton))
```

Functors

Modules parameterized by other modules. Genericity through dictionnary passing.

```
module type HasEqual = sig
  type t
  val equal : t -> t -> bool
end

module MySet (Elt : HasEqual) = struct
  type t = Elt.t list
  let empty = []

  let rec add set value =
    match set with
    | [] -> [value]
    | hd :: tl ->
        if Elt.equal hd value then set
        else hd :: (add tl value)
end
```

(Obviously a poor implementation of a set, but you get the idea.)

Functors

```
module IntSet = MySet(struct
  type t = int
  let equal a b = a = b
end)

let singleton = IntSet.add (IntSet.empty) 42
(* val singleton : IntSet.t = [42] *)

(* ...or using the stdlib's Int module *)
module IntSet = MySet(Int)

let still_singleton = IntSet.add (singleton) 42
(* val still_singleton : IntSet/2.t = [42] *)
```

Bad module type

```
module IntSet = MySet(struct
  type t = int
  let equal a b = a + b
end)

(*
Error: Modules do not match: sig type t = int val equal : t -> t -> t end
     is not included in HasEqual
     Values do not match:
       val equal : t -> t -> t
     is not included in
       val equal : t -> t -> bool
     The type t -> t -> t is not compatible with the type t -> t -> bool
     Type t is not compatible with type bool
*)
```

Advanced: Inheriting and overriding modules

```
module type Foo = sig
  type t
  val equal : t -> t -> bool
  val compare : t -> t -> int
  val foo : t -> t
end
module M (E : Foo) = struct end

module IntM_Ko = M(Int) (* Error! No foo function in Int*)

module IntM_Ok = M(struct
  include Int (* Inherit all definitions in Int *)
  let foo n = n + 1 (* Add foo *)
  let compare a b = compare b a (* override compare *)
end)
```

Advanced: First-class modules

Since modules have types, they can also be used directly as values.

```
type picture = (* ... *)
module type DEVICE = sig
  val draw : picture -> unit
  (* ... *)
end
let devices : (string, (module DEVICE)) Hashtbl.t = Hashtbl.create 17

module SVG = struct (* ... *) end
let _ = Hashtbl.add devices "SVG" (module SVG : DEVICE)

module PDF = struct (* ... *) end
let _ = Hashtbl.add devices "PDF" (module PDF : DEVICE)
```

Advanced: First-class modules

First-class modules allow for parametrizing code with modules without using functors.

```
let sort (type s) (module Set : Set.S with type elt = s) l =
  Set.elements (List.fold_right Set.add l Set.empty)

(* val sort : (module Set.S with type elt = 's) -> 's list -> 's list = <fun> *)

let make_set (type s) cmp =
  let module S = Set.Make(struct
    type t = s
    let compare = cmp
  end) in
  (module S : Set.S with type elt = s)

(* val make_set : ('s -> 's -> int) -> (module Set.S with type elt = 's) = <fun> *)
```