Generics

Akim Demaille, Etienne Renault, Roland Levillain

March 29, 2020

Table of Contents

Problem Statement

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

A Definition of Generic Programming

> *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.*
>
> —
>
> *Jazayeri et al., 2000, Garcia et al., 2003*

A Definition of Generic Programming (cont'd)

> *Key ideas include:*
>
> —
>
> *Jazayeri et al., 2000, Garcia et al., 2003*

A Definition of Generic Programming (cont'd)

> *Key ideas include:*
>
> - *Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.*

—

Jazayeri et al., 2000, Garcia et al., 2003

A Definition of Generic Programming (cont'd)

> *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.*
>
> —
>
> *Jazayeri et al., 2000, Garcia et al., 2003*

A Definition of Generic Programming (cont'd)

"

—

Jazayeri et al., 2000, Garcia et al., 2003

A Definition of Generic Programming (cont'd)

"

- *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.*

—

Jazayeri et al., 2000, Garcia et al., 2003

A Definition of Generic Programming (cont'd)

"

- *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.

—

Jazayeri et al., 2000, Garcia et al., 2003

Table of Contents

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)

Genericity in CLU

- First ideas of generic programming date back from CLU [HOPL'93] (in 1974, before it was named like this).

Genericity in CLU

- First ideas of generic programming date back from CLU [HOPL'93] (in 1974, before it was named like this).
- 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

- First ideas of generic programming date back from CLU [HOPL'93] (in 1974, before it was named like this).
- 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
- In CLU, modules are implemented as *clusters* programming units grouping a data type and its operations.

Genericity in CLU

- First ideas of generic programming date back from CLU [HOPL'93] (in 1974, before it was named like this).
- 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
- 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.

Parameterized modules in CLU

- Initially: parameters checked at run time.
- Then: introduction of **where**-clauses (requirements on parameter(s)).

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.

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.

$\rightarrow$ Complete compile-time check of parameterized modules.

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)

Implementation of parameterized modules in CLU

- Notion of *instantiation*:
  binding a module and its parameter(s)
- Syntax: *module*[*parameter*]

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

- Notion of *instantiation*:
  binding a module and its parameter(s)
- Syntax: *module*[*parameter*]
- *Dynamic instantiation* of parameterized modules.
- For a given module, each distinct set of parameters is represented by a (run-time) object.

Implementation of parameterized modules in CLU

- Notion of *instantiation*:
  binding a module and its parameter(s)
- 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.

Implementation of parameterized modules in CLU

- Notion of *instantiation*:
  binding a module and its parameter(s)
- 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:

Implementation of parameterized modules in CLU

- Notion of *instantiation*:
  binding a module and its parameter(s)
- 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).

Implementation of parameterized modules in CLU

- Notion of *instantiation*:
  binding a module and its parameter(s)
- 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.

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;
```

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);
```

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);
```

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.

A History of C++ Templates

- Initial motivation: provide parameterized containers.
- Previously, *macros* were used to provide such containers (in C and C with classes).

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:

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.

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*.

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.

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.

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).

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.

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.)

- 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.

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.
- No explicit constraints on parameters.
- Implicit (automatic) template instantiation (though explicit instantiation is still possible).

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

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

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**).

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.

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.

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.

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

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)

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.

The Standard Template Library (STL)

- A library of containers, iterators, fundamental algorithms and tools, using C++ templates.
- Designed by Alexander Stepanov at HP.

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

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.

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.

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.
- Initially planned as a language extension in the C++11/14/17 standard...

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.
- Initially planned as a language extension in the C++11/14/17 standard...
- ...but abandonned shortly before the standardization. :-(

Example

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

struct meow {};

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

Table of Contents

Problem Statement

How to implement **Generics**?

Table of Contents

Boxing: main idea

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

Boxing: main idea

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

- The data structure only handles pointers

Wideley used strategy:

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

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

Wideley used strategy:

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

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!

Wideley used strategy:

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

Go example

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

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

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
  $\Longrightarrow$ No mechanism to prevent us putting elements of different types into the structure

Type-erased boxed generics

Idea

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

Type-erased boxed generics

Idea

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

  → This approach is often called **type erasure**, because the types in the generics system are ”erased” and all become the same type

Type-erased boxed generics

Idea

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

  → 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 add 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 that cannot be cast to an Integer
Integer i = (Integer) v.get(0); // Run time error
```

Java Example

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

```
List v = new ArrayList();
v.add("test"); // A String that cannot be cast to an Integer
Integer i = (Integer) v.get(0); // Run time error
```

- With Generics
  Fails at compile time

```
List<String> v = new ArrayList<String>();
v.add("test");
Integer i = v.get(0); // (type error) compilation-time error
```

Todo Wildcard?

Inferred boxed generics with a uniform representation

Problem with simple boxing

In the previous approach, generic data structures cannot hold primitive types!

Inferred boxed generics with a uniform representation

Problem with simple boxing

In the previous approach, generic data structures cannot hold primitive types!

Ocaml's Solution

Uniform representation where there are no primitive types that requires an additional boxing allocation !

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

Ocaml's apporach:

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

Ocaml's apporach:

- no additional boxing allocation (like int needing to be turned into an Integer

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

Ocaml's apporach:

- no additional boxing allocation (like int needing to be turned into an Integer
- everything is either already boxed or represented by a pointer-sized integer
  $\Longrightarrow$ everything is one machine word

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

Ocaml's apporach:

- no additional boxing allocation (like int needing to be turned into an Integer
- everything is either already boxed or represented by a pointer-sized integer
  $\Longrightarrow$ everything is one machine word
- Problem :garbage collector needs to distinguish pointers from integers

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

Ocaml's apporach:

- no additional boxing allocation (like int needing to be turned into an Integer
- everything is either already boxed or represented by a pointer-sized integer
  $\Longrightarrow$ everything is one machine word
- Problem :garbage collector needs to distinguish pointers from integers
- ... there is a reserved bit in machine word

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

Ocaml's apporach:

- no additional boxing allocation (like int needing to be turned into an Integer
- everything is either already boxed or represented by a pointer-sized integer
  $\Longrightarrow$ everything is one machine word
- Problem :garbage collector needs to distinguish pointers from integers
- ... there is a reserved bit in machine word
  - integer size is only 31/63 bits
  - pointer size is only 31/63 bits

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

Ocaml's apporach:

- no additional boxing allocation (like int needing to be turned into an Integer
- everything is either already boxed or represented by a pointer-sized integer
  $\Longrightarrow$ everything is one machine word
- Problem :garbage collector needs to distinguish pointers from integers
- ... there is a reserved bit in machine word
  - integer size is only 31/63 bits
  - pointer size is only 31/63 bits
  - the 32/64 bit for integer is 1
  - the 32/64 bit for valid aligned pointers is 0

Introducing Interfaces

Limitation with boxing

The boxed types are completely opaque!
(generic sorting function need some extra functionality,
like a type-specific comparison function.)

Introducing Interfaces

Limitation with boxing

The boxed types are completely opaque!
(generic sorting function need some extra functionality,
like a type-specific comparison function.)

Two families of solutions

- **Dictionary passing**: Haskell (type class) and 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
- **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

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.
- $\rightarrow$ swift uses monomorphization (later in lecture)

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.
- $\rightarrow$ swift uses monomorphization (later in lecture)

Going further

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

- boxed types is augmented to add a type ID
- generate functions for each interface method
- Dispatch using big switch statement over all the types

From interface vtables to Reflection (1/3)

In Object-oriented programming (like Java)

- No need to have separate interface objects
- the vtable pointer is embedded at the start of every object
- inheritance and interfaces that can be implemented entirely with these object vtables
- $\rightarrow$ construct new interface types with indirection is no longer required.

Reflection

With vtables, itffs not difficult to have reflection since the compiler can generates tables of other type information like field names, types and locations

From interface vtables to Reflection (2/3)

Reflection is the ability of a program to examine, introspect, and modify its own structure and behavior at runtime.

From interface vtables to Reflection (2/3)

Reflection is the ability of a program to examine, introspect, and modify its own structure and behavior at runtime.

Reflection is not limited to OOP!
and most functionnal languages can create new types!
Python and Ruby have super-powered reflection systems that are used for everything.

From interface vtables to Reflection (3/3)

- Introspection: ability to observe and therefore reason about its own state.

```
public boolean classequal(Object o1, Object o2){
    Class c1, c2;           c1 = o1.getClass();
    c2 = o2.getClass();     return (c1 == c2);
}
```

- Intercession: ability to modify its execution state or alter its own interpretation

```
Class c = obj.getClass();
Object o = c.newInstance();
```

```
String s = "FooBar".
Class c = Class.forName(s);
Object o = c.newInstance();
```

Table of Contents

Monomorphization

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

- C++ template
- Rust procedural macros
- D

Metaprogramming

Writing programs that write programs.
Some language a clean way of doing code generation

- Syntax tree macros: the ability to produce AST types in macros written in the language
- Template: reason about types and type substitution
- Compile time functions