Typology of programming languages

Concepts behind generics

Problem Statement

How to implement **Generics**?

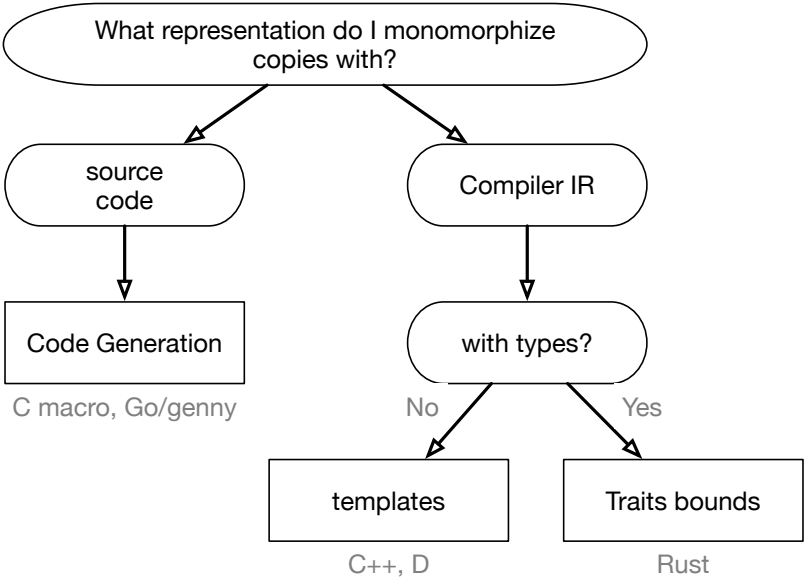
Table of Contents

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, Ada

Various Monomorphization



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

```
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

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

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

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

Rust polyomorphization

An optimisation which determines when functions, closures and generators could remain polymorphic during code generation.

Polymorphization will identify A as being unused

```
fn foo<A, B>() {
    let x: Option<B> = None;
}

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

Table of Contents

Boxing: main idea

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

Boxing: main idea

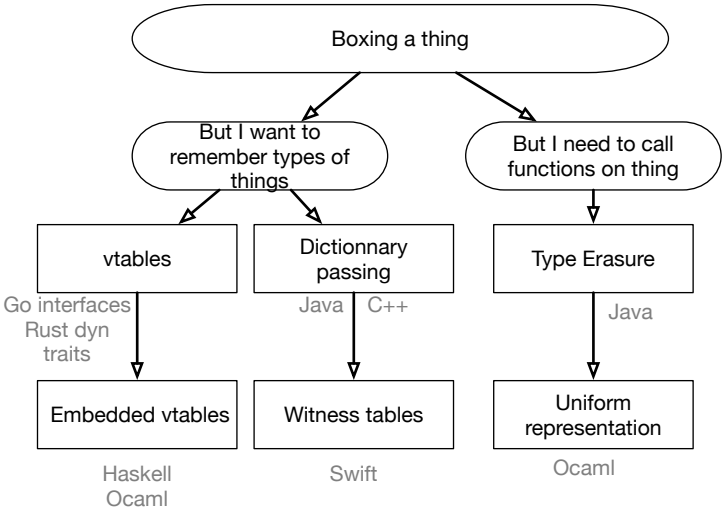
Put everything in uniform "boxes" so that they all act the same way

- The data structure only handles pointers
- 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

Various boxing



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.

  $\Rightarrow$ 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 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 => 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 that 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");
// (type error) compilation-time 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!

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:

- 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
  $\Rightarrow$ there is a reserved bit in machine word

Introducing Interfaces

Limitation with boxing

The boxed types are completely opaque!

(generic sorting function need 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 to **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

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

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

Metaprogramming

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

Rust metaprogramming

- Function-like procedural macros define public function
- Derive macros append functions to structs
- Attributes macro add fields to structs

```
#[proc_macro]
pub fn make_answer(_item: TokenStream)
-> TokenStream {
    "fn answer() -> u32 { 42 }"
        .parse().unwrap()
}

make_answer!();
fn main() {
    println!("{}", answer());
}
```

C++ metaprogramming

```
template <unsigned int n>
struct factorial {
  enum { value = n *
    factorial<n - 1>::value };
};

template <>
struct factorial<0> {
        enum { value = 1 };
};
```

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

Reflection

With vtables, it's 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.

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.

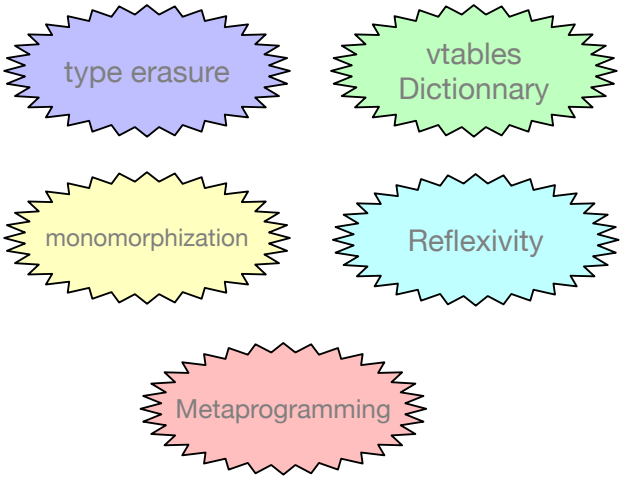
```
public boolean classequal(Object o1,
                          Object o2){
  Class c1 = o1.getClass();
  Class c2 = o2.getClass();
  return (c1 == c2);
}
```

- **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();
```

Summary


type erasure
vtables
Dictionnary
monomorphization
Reflexivity
Metaprogramming