Typology of Programming Languages

Advanced paradigms and concepts

May 2025

Section 1

Metaprogramming

Metaprogramming

“Writing programs that write programs”.

Some language have a clean and dedicated way of doing code generation:

- Syntax tree macros: produce AST types in macros written in the language (like in Rust)
- Template: work on types and type substitution
- Functions evaluated at compile time (like C++ constexpr)

Rust metaprogramming

Procedural macros can produce or *modify* part of the AST. They come in three flavors:

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

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

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

C++ metaprogramming

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

template <>
struct facto<0> {
  enum { val = 1 };
};

int main(void) {
  cout << facto<5>::val;
}
```

Prints 120, computed at compile time.

```
// Or, since C++11:
constexpr int facto(int n) {
  if (n == 0)
    return 1;
  else
    return n * facto(n-1);
}

int main(void) {
  constexpr int n = facto(5);
  std::cout << n;
}
```

From interface vtables to Reflection (1/3)

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!
Most functionnal languages can create new types!
Some languages like Python and Ruby have super-powered reflection systems that are used for everything.

From interface vtables to Reflection (2/3)

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 generate tables with extra type information like field names, types and locations

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();
```

Section 2

Prototype-based object orientation

Engineering Properties, L.Caardelli 1996

- **Economy of execution.**
  How fast does a program run?
- **Economy of compilation.**
  How long does it take to go from sources to executables?
- **Economy of small-scale development.**
  How hard must an individual programmer work?
- **Economy of large-scale development.**
  How hard must a team of programmers work?
- **Economy of language features.**
  How hard is it to learn or use a programming language?

Problem Statement

Traditional class-based OO languages are based on a deep-rooted duality:

- **Classes**: defines behaviours of objects.
- **Object instances**: specific manifestations of a class

Unless one can predict with certainty what qualities a set of objects and classes will have in the distant future, one cannot design a class hierarchy properly

Metaclass class class class claaahhhh

Self

Invented by David Ungar and Randall B. Smith in 1986 at Xerox Park

Overview:

- Neither classes nor meta-classes
- Self objects are a collection of *slots*. Slots are accessor methods that return values.
- Self object is a stand-alone entity
- An object can delegate any message it does not understand itself to the parent object
- Inspired from Smalltalks blocks for flow control
- Generational garbage collector

Example in self

- Copy the `lecture` object and set the copy's title to TYLA

```
tyla := lecture copy title: 'TYLA'.
```

- add a slot to an object

```
tyla _AddSlots: (| remote <- 'false'|).
```

- change who is the parent at runtime

```
myObject parent: someOtherObject.
```

Impacts

- Javascript
- NewtonScript
- Io
- Rust
- Go

Rust, Go, …

> *Gang of 4 quote Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations.*
>
> –
>
> **Elements of Reusable Object-Oriented Software Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides**

hard Helm, Ralph Johnson, and John Vlissides* \end{shadequote}

> *Even though structs and enums with methods aren’t called objects, they provide the same functionality, according to the Gang of Four’s definition of objects.*
>
> –
>
> *Rust documentation*

Example in Rust

```rust
trait Describe {
    fn my_tostring(&self) -> String;
}

impl Describe for u8 {
    fn my_tostring(&self) -> String {
        format!("u8: {}", *self)
    }
}

impl Describe for String {
    fn my_tostring(&self) -> String {
        format!("string: {}", *self)
    }
}

fn do_something<T: Describe>(x: T) {
    x.my_tostring();
}
```

Duck Typing

If it walks like a duck and it quacks like a duck, then it must be a duck

Duck Typing

If it walks like a duck and it quacks like a duck, then it must be a duck

```python
# In python, dynamically
class Duck:
    def swim(self):
        print("Duck swimming")

    def fly(self):
        print("Duck flying")

class Whale:
    def swim(self):
        print("Whale swimming")

for animal in [Duck(), Whale()]:
    animal.swim()
```

Duck Typing

If it walks like a duck and it quacks like a duck, then it must be a duck

```python
# In python, dynamically
class Duck:
    def swim(self):
        print("Duck swimming")

    def fly(self):
        print("Duck flying")

class Whale:
    def swim(self):
        print("Whale swimming")

for animal in [Duck(), Whale()]:
    animal.swim()
```

```go
// In Go, statically
type Duck struct {}
func (Duck) Swim() {
    fmt.Println("Duck swimming")
}
type Whale struct {}
func (Whale) Swim() {
    fmt.Println("Whale swimming")
}
func main() {
    animals := []interface{Swim()}{
        Duck{}, Whale{}
    }
    for _, animal := range animals {
        animal.Swim()
    }
}
```

CLOS

Developed in mid 80’s.

Overview:

- Metaobject Protocol
- Meta-class
- Multiple Inheritance
- Multiple dispatch
- Generic Functions
- Method Qualifier
- Introspection

Small Example

```
(defclass human () (name size birth-year))
(make-instance 'human)



(defclass Shape () ())
(defclass Rectangle (Shape) ())
(defclass Ellipse (Shape) ())
(defclass Triangle (Shape) ())

(defmethod intersect ((r Rectangle) (e Ellipse))
    ...)

(defmethod intersect ((r1 Rectangle) (r2 Rectangle))
    ...)

(defmethod intersect ((r Rectangle) (s Shape))
    ...)
```