Prototypes: Object-Orientation, Functionally

Michal Atlas

October 30, 2023

What are we looking at today?

The Paper's first page

```
(define (fix p b)
  (define f (p (lambda i (apply f i)) b)) f)
(define (mix c p)
  (lambda (f b) (c f (p f b))))
```

We will make the case that the above two definitions summarize the essence of Object-Oriented Programming (OOP), and that all the usual OOP concepts can be easily recovered from them—all while staying within the framework of pure Functional Programming (FP).

Back to the Paper

Is this a struct?

```
(define (x1-y2 msg)
  (case msg
    ((x) 1)
    ((y) 2)
    (else (error "unbound slot" msg))))
```

A Prototype

```
;; Super calls the "parent class"
(define ($x3 self super)
  (λ (msg) (if (eq? msg 'x) 3 (super msg))))

(define ($z<-xy self super)
  (λ (msg) (case msg
             ((z) (+ (self 'x) (* 0+1i (self 'y))))
             (else (super msg)))))
```

Applying the prototype

```
(define x3-y2 (fix $x3 x1-y2))

(x3-y2 'x) ;=> 3
(x3-y2 'y) ;=> 2
```

Mixing Prototypes

```
(define z6+2i
  (fix (mix $z<-xy (mix $double-x $x3)) x1-y2))

(map z6+2i '(x y z))
;=> '(6 2 6+2i)
```

Curb your codegolf!!!

```scheme
;; ...
(define (fix p b)
  (define f (p (lambda i (apply f i)) b)) f)
```

```scheme
;; ...
(define (mix c p)
  (lambda (f b) (c f (p f b))))
```

Curb your codegolf!!! - Whiteboard time

```scheme
;; FIX
(define (instantiate-prototype prototype base-super)
  (define self
    (prototype (λ i (apply self i)) base-super))
  self)

;; MIX
(define (compose-prototypes child parent)
  (λ (self super2) (child self (parent self super2))))
```

Any language

JS

```
const fix = (p,b) => f = p((i) => f(i), b)
const mix = (c,p) => (f,b) => c(f, p(f,b))
```

PY

```
def fix(p, b):
    f = p(lambda i: f(i), b)
    return f

def mix(c, p):
    return lambda f, b: c(f, p(f, b))
```

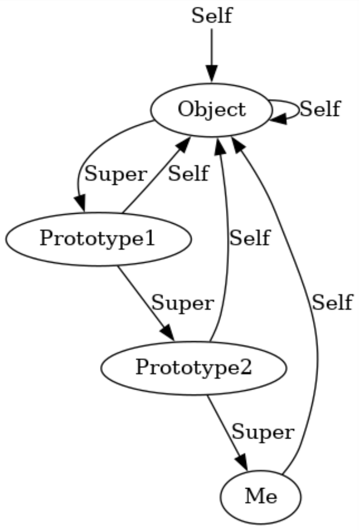
ANY language

```
typedef struct object_t {
  char *slot_name;
  SCM (*fn)(struct object_t *,
            struct object_t *);
  struct object_t *prototype;
} object_t;
```

Mixing Prototypes

```
(define z6+2i
  (fix (mix $double-x
            (mix $z<-xy (mix $double-x $x3))) x1-y2))

(map z6+2i '(x y z))
;=> '(12 2 12+2i)
```

Why I'm not self



Let’s make it nice to use

$slot-gen

```
(define ($slot-gen k fun)
  (λ (self super)
    (λ (msg)
      (define (inherit) (super msg))
      (if (equal? msg k) (fun self inherit) (inherit)))))

(define ($slot k v)
  ($slot-gen k (λ (_self _inherit) v)))

(define ($slot-modify k modify)
  ($slot-gen k (λ (_ inherit) (modify (inherit)))))

(define ($slot-compute k fun)
  ($slot-gen k (λ (self _) (fun self))))
```

Usage

```
(define $x3 ($slot 'x 3))

(define $double-x ($slot-modify 'x (λ (x) (* 2 x))))

(define $z<-xy
  ($slot-compute
   'z
   (λ (self) (+ (self 'x) (* 0+1i (self 'y))))))
```

Building up the utilities

```
(define (identity-prototype self super) super)

(define (compose-prototype-list prototype-list)
  (foldr compose-prototypes
         identity-prototype prototype-list))

(define (instantiate-prototype-list
         prototype-list base-super)
  (instantiate-prototype
   (compose-prototype-list prototype-list) base-super))
```

Instantiation

```
(define (bottom . args) (error "bottom" args))

(define (instance . prototype-list)
  (instantiate-prototype-list prototype-list bottom))
```

Introspections

```
(define ($slot-gen/keys k fun)
  (λ (self super)
    (λ (msg) (cond ((equal? msg k)
                     (fun self (λ () (super msg))))
                    ((equal? msg 'keys)
                     (cons k (super 'keys)))
                    (else (super msg))))))
```

POWER

```
((instance $z<-xy $x3 $y2) 'keys)
;=> '(z x y)
```

Ordering shenanigans

```
(define ($number-order self super)
  (λ (msg) (case msg
             ((<) (λ (x y) (< x y)))
             ((=) (λ (x y) (= x y)))
             ((>) (λ (x y) (> x y)))
             (else (super msg)))))

(define ($string-order self super)
  (λ (msg) (case msg
             ((<) (λ (x y) (string<? x y)))
             ((=) (λ (x y) (string=? x y)))
             ((>) (λ (x y) (string>? x y)))
             (else (super msg)))))
```

Get Ready

```
(define ($compare<-order self super)
  (λ (msg) (case msg
             ((compare)
              (λ (x y) (cond (((self '<) x y) '<)
                             (((self '>) x y) '>)
                             (((self '=) x y) '=)
                             (else (error "incomparable"
                                          x y)))))
                (else (super msg)))))

(define number-order
  (instance $number-order $compare<-order))
(define string-order
  (instance $string-order $compare<-order))
```

Delegation

```
(define ($symbol-order self super)
  (λ (msg)
    (case msg
      ((< = > compare)
       (λ (x y) ((string-order msg)
                 (symbol->string x)
                 (symbol->string y))))
      (else (super msg)))))
```

Our Instance was an algorithm not just a struct

Usage

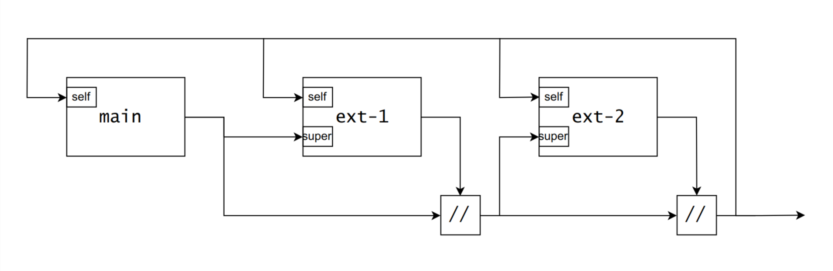
```
((string-order 'compare) "Foo" "FOO")
;=> '>

((string-order 'compare) "42" "42")
;=> '=
```

Too big to show

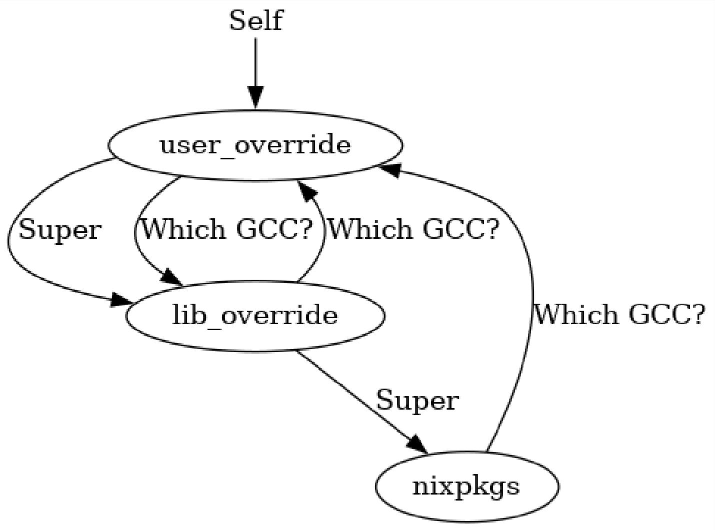
```
(define symbol-tree-map
  (instance $binary-tree-map
            ($slot 'Key symbol-order)))


(define Dict
  (instance $avl-tree-rebalance
            $binary-tree-map
            ($slot 'Key symbol-order)))
```

Where we find this in the wild

Nix



Why?


Self
user_override
Super
Which GCC?
Which GCC?
lib_override
Which GCC?
Super
nixpkgs

Prototype vs. Instance though...

```
rec { ssl = 4; gcc.ssl = ssl; }

  { gcc = { ssl = 4; }; ssl = 4; }

lib.fix' (self: { ssl = 4; gcc.ssl = self.ssl; })

  {
    __unfix__ = «lambda @ «string»:1:16»;
    gcc = { ssl = 4; };
    ssl = 4;
  }
```

__unfix__

```
obj.__unfix__ { ssl = 8; }

  { gcc = { ssl = 8; }; ssl = 4; }

obj.__unfix__ { ssl = 8; } // { ssl = 8; }

  { gcc = { ssl = 8; }; ssl = 8; }
```

So the Instance (Self) is the fixed-point of a prototype

JavaScript!!!

Simple objects

```
x = { foo: 2, bar: 5 }
console.log(x)
console.log(x.foo)
```

```
{ foo: 2, bar: 5 }
2
```

Objects with prototypes

```
$p = { foo: 8, bar () { return this.foo } }
i = { foo: 16, __proto__: $p }
console.log(i.bar())
```

```
16
```

We even have Super

```
$p1 = { foo: 2 }
$p2 = { up ()   { return this.foo },
        down () { return super.foo },
        __proto__: $p1
      }

i = { foo: 8, __proto__: $p2 }

console.log(i.up())
console.log(i.down())

8
2
```

Classes are syntax sugar over prototypes

And we can manipulate them

```
class A { }
before = new A

console.log(before.foo)

A.prototype.foo = 8

after = new A

console.log(before.foo)
console.log(after.foo)
```

```
undefined
8
8
```

Because being in a class means having the prototype

```
class A { }
i1 = new A
i2 = new A

console.log(A.prototype === i1.__proto__)
console.log(A.prototype === i2.__proto__)

true
true
```

Because being in a class means having the prototype

```
class A { }
i1 = new A

console.log(i1 instanceof A)
i1.__proto__ = {}
console.log(i1 instanceof A)
```

```
true
false
```

Because being in a class means having the prototype

```
class A { }
class B { }
i1 = new A

console.log(i1 instanceof B)
i1.__proto__ = B.prototype
console.log(i1 instanceof B)
```

```
false
true
```

JS Constructors

```
function Constructor(i) {
    this.foo = i
}

Constructor.prototype = { bar: 20 }

c = new Constructor(2)
console.log(c.foo)
console.log(c.bar)
console.log(c.__proto__)
```

```
2
20
{ bar: 20 }
```

If any of this seemed cool, do go read the original, it’s very fun and pleasant

Thanks for listening