Prototypes: Object-Orientation, Functionally

François-René Rideau, Alex Knauth, and Nada Amin

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

(define (mix c p)
  (lambda (f s)
    (c f (p f s))))
```

https://github.com/metareflection/poof

Prototypes: Object-Orientation, Functionally

François-René Rideau, Alex Knauth, and Nada Amin

```
(define (instantiate proto base)
  (define self (proto (lambda i (apply self i)) base))
  self)

(define (inherit child parent)
  (lambda (self super)
    (child self (parent self super))))
```

https://github.com/metareflection/poof

What IS in the Paper

What IS in the Paper

Object Systems defined in the λ-calculus

Fundamental concepts established

Inheritance elucidated

Prototypes before Classes

Purity before Mutation

Constructive Semantics

What is Object-Orientation about?

Incrementality

Open Recursion

Modularity

Ad hoc Polymorphism

What is Object-Orientation about?

Incrementality

Open Recursion

Modularity

Ad hoc Polymorphism

What is Object-Orientation NOT about?

Classes

“Encapsulation”

Inheritance being opposed to Composition

Mutation everywhere

What is Object-Orientation NOT about?

Classes

“Encapsulation”

Inheritance being opposed to Composition

Mutation everywhere

Fundamental Concepts

Incrementality: Instances and Prototypes

Inheritance: Wrappers and Generators

Generality: Prototypes beyond records

Multiple inheritance: modular dependencies

Conflation: Object = Prototype × Instance

Type Prototypes: Classes and Elements

Simplest Incrementality

Simplest Incrementality

Instance: value to specify incrementally

Prototype: increment of specification

Simplest Incrementality

Instance: value to specify incrementally

Prototype: increment of specification

```
instantiate: prototype → instance

inherit: prototype prototype → prototype
```

Simplest Instances: Records as Functions

Record: Symbol → Value

```
(define (my-point msg)
  (case msg ((x) 1)
            ((y) 2)
            (else (error "invalid field"))))
> (my-point 'y)
2
```

Simplest Prototypes: Wrappers

```
; (deftype (Proto Self Super)
;   (Fun Self Super → Self st: (⊂ Self Super)))

; : (Proto Self Super) Super -> Self
(define (instantiate proto base)
  (define self (proto (λ i (apply self i)) base))
  self)

; : (Proto Self Super) (Proto Super S2) -> (Proto Self S2)
(define (inherit child parent)
  (lambda (self super)
    (child self (parent self super))))
```

Simple Prototypes at work

```
(define (my-point msg) (case msg ((x) 1) ((y) 2) (else (⊥))
```

Simple Prototypes at work

```
(define (my-point msg) (case msg ((x) 1) ((y) 2) (else (⊥))

(define ($x3 self super)
  (λ (msg) (if (eq? msg 'x) 3 (super msg))))
(define ($double-x self super)
  (λ (msg) (if (eq? msg 'x) (* 2 (super 'x)) (super msg))))
(define ($z<-xy self super)
  (λ (msg) (case msg
            ((z) (+ (self 'x) (* 0+1i (self 'y))))
            (else (super msg)))))
```

Simple Prototypes at work

```
(define (my-point msg) (case msg ((x) 1) ((y) 2) (else (⊥))

 (define ($x3 self super)
   (λ (msg) (if (eq? msg 'x) 3 (super msg))))
 (define ($double-x self super)
   (λ (msg) (if (eq? msg 'x) (* 2 (super 'x)) (super msg))))
 (define ($z<-xy self super)
   (λ (msg) (case msg
             ((z) (+ (self 'x) (* 0+1i (self 'y))))
             (else (super msg)))))

(define $your-point (inherit $z<-xy (inherit $double-x $x3)
(define your-point (instantiate $your-point my-point))
> (your-point 'z)
6+2i
```

Compare: Single Inheritance

```
; (deftype (Gen A) (Fun A -> A))
; instantiate-generator : (Fun (Gen A) -> A)
(define (instantiate-generator g)
  (define f (g (λ i (apply f i)))) f)

; proto->generator : (Fun (Proto A B) B -> (Gen A))
(define (proto->generator p b) (λ (f) (p f b)))
; (== (instantiate-generator (proto->generator p b))
;     (instantiate p b))

; apply-proto : (Fun (Proto A B) (Gen B) -> (Gen A))
(define (apply-proto p g) (λ (f) (p f (g f))))
; (== (apply-proto p (proto->generator q b))
;     (proto->generator (inherit p q) b))
```

Beyond Simple Records

Prototypes for any type of instance...

Prototypes build computations, not values (CBPV)

Functions, thunks, delayed or lazy values

Useful even without record subtyping

Multiple Inheritance

Make `Wrapper` dependencies modular

Users specify dependency DAG in local increments

System computes and linearizes global DAG

```
Prototype = Wrapper × List(Prototype) × …
```

Conflation of Instance and Prototype

We can do all OOP without “objects”,

maintaining instance/prototype distinction, but…

`Object = Prototype × Instance`

Conflation works better with purity

Conflation without Distinction ⇒ Confusion

Classes

Class OO = Prototype OO at meta-level

Instance = Type descriptor (fields, operations…)

Class = Prototype for Type descriptor

Abstract vs Concrete Class = Prototype vs Instance

Subclass ≠ Subtype

Classes: pure at meta-level (but multimethods…)

‘object’, ‘instance’ meanings differ in Class vs Proto

Mutation

Easy to extend pure model with mutation

More efficient in linear case, less with sharing

Simplified `self/super` protocol

Challenge: cache invalidation

- mutable slots vs derived slots
- mutable supers vs precedence list

Constructive Semantics

Denotational Semantics × Practical Implementation

30 loc prototype OOP in any λ language

50 loc more for multiple inheritance

No side-effect needed, but better with laziness

Records need subtyping or dynamic types

Classes also need staging or dependent types

Related Work

(Stateful) Prototypes: 1970s: Director, ThingLab;
1980s: T, SELF; 1990s: JavaScript

Semantics: 1980s Semantics (Reddy; Cook...);
1990s Types (Cardelli; Pierce...)

Composable Mixins: 1990s StrongTalk… (Bracha);
2000s Racket, Scala, Haskell…

Pure Functional Prototypes: 2004 GCL (Google);
2014 Jsonnet, 2015 Nix

Future Work

Multiple dispatch (multimethods)

Method Combinations

Generalized Prototypes (with lenses)

Usable static types

Better caching control

Paper Claims (redux)

Define Object Systems in the λ-calculus

Establish fundamental concepts

Elucidate Inheritance

Prototypes before Classes

Purity before Mutation

Constructive Semantics

Meta Claims

Humility, not fanaticism

Incommensurable paradigms? Go wider!

Simplicity matters

λ's for Semantics, macros for Syntax

Questions?

Paper (23 pages, 33 w/ appendices)

`https://github.com/metareflection/poof`

Gerbil Scheme implementation (3 kloc w/ library)

`https://github.com/fare/gerbil-poo`

Nix implementation (~80 loc w/ multiple inheritance)

`https://github.com/NixOS/nixpkgs/pull/116275`

We're hiring at MuKn!

`jobs@mukn.io`