Prototypes: Object-Orientation, Functionally

FRANÇOIS-RENÉ RIDEAU, *Mutual Knowledge Systems, Inc.*, USA
ALEX KNAUTH, *Mutual Knowledge Systems, Inc.*, USA
NADA AMIN, *Harvard University*, USA

This paper elucidates the essence of Object-Oriented Programming (OOP), using a constructive approach: we identify a minimal basis of concepts with which to synthesize existing and potential object systems. We reduce them to constructions atop the pure untyped lambda calculus, thereby obtaining both denotational semantics and effective implementation. We start from the simplest recognizable model of prototype-based OOP, so simple it arguably does not even have "objects" as such. We build further models of increasing sophistication, reproducing a growing subset of features found in past object systems, including original combinations. We also examine how our approach can deal with issues like typing, modularity, classes, mutation. We use Scheme to illustrate our approach.

1 PROTOTYPES, BOTTOM UP

1.1 The Essence of OOP

1.1.1 Object Orientation in 38 `cons` *cells of standard Scheme.*

```
(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).

1.1.2 Claims. Our approach emphasizes the following original contributions:

(a) the explicit conceptual distinction between instances, prototypes, wrappers and generators (section 1), objects (section 4.2), classes and elements (section 5),
(b) promoting *composition* of wrappers rather than their *application* to a generator as the algebraic structure of interest (sections 1, 3)
(c) both *how* to implement multiple inheritance with pure prototypes and *why* use them based on a model of modularity (section 4.3),
(d) how to derive class OOP from the more primitive prototype OOP (section 5),
(e) a constructive model that does not rely on mutation (sections 1, 2, 3), yet that can be extended to play well with it (section 6),
(f) overall, a pure functional approach that provides not only denotational semantics atop the untyped lambda-calculus, but also a practical constructive implementation.

1.1.3 Plan. In section 1, we explain how the above functions implement a minimal but recognizable model of OOP, with open recursion and inheritance. In section 2, we illustrate this OOP model actually enables incremental definition of complex data structures. In section 3, we show how prototypes are useful beyond traditional notions of objects, and identify how they have a special place in computer science. In section 4, we examine how our model can be extended to support more advanced features of OOP. In section 5, we demonstrate how classes are "just" prototypes for type descriptors. In section 6, we discuss our pure functional model relates to models with mutable objects. In section 7, we compare our work to salient papers in fifty years of research on OOP. Finally in section 8, we propose a path for further research. Along the way, we relate our approach to both OOP and FP traditions, both decades-old and recent.

1.1.4 Modular Increments of Computation. OOP consists in specifying computations in modular increments each contributing their part based on the combined whole and the computation so far. The ability to abstract over which increments are part of a computation is called *ad hoc* polymorphism (Strachey 1967).

In general, we will call *prototype* such a computation increment. In the rest of section 1, we will present an initial model where a *prototype* will be encoded as prototype *wrapper*: a function of two arguments, `self` and `super` (or, above, `f` and `b`, for fixed-point and base value). In section 4, we will enrich prototypes to be more than prototype wrappers, so as to model more advanced OOP features; but until then, we will use the two interchangeably.

Prototypes come with two primary operations. Function `fix` *instantiates* a prototype `p` given a super/base value `b`. Function `mix` has *child* prototype `c` *inherit* from *parent* prototype `p` so they operate on a same fixed-point `f` while chaining their effects on `b`, yielding a new prototype.

Given some instance type `Self` and a super-type `Super` of `Self`, a prototype for `Self` from `Super` will thus be a function from `Self` and `Super` to `Self`. In the type notation of Appendix C:

```
; (deftype (Proto Self Super) (Fun Self Super -> Self st: (<: Self Super)))
; fix : (Fun (Proto Self Super) Super -> Self st: (<: Self Super))
; mix : (Fun (Proto Self Super) (Proto Super Sup2) -> (Proto Self Sup2))
```

The first argument `self` of type `Self` will hold the instance resulting as a fixed point from the entire computation. When composing multiple prototypes, every prototype will receive the *same* value as their `self` argument: the complete instance that results from applying each prototype in order. This allows prototypes to "cooperate" with each other on *different* aspects of the computation, wherein one prototype defines some aspects (e.g. "methods" in some dictionary) while relying on aspects to be defined by other prototypes (e.g. other methods), accessed through the `self` argument in what is called "late binding".

The second argument `super` by contrast holds the partial result of the fixed-point computation after applying only the "next" prototypes. When composing multiple prototypes, each prototype will (presumably) receive a different value. The last prototype in the list (rightmost, most ancestral parent) will receive the "base" or "bottom" value from the fix function (often literally the bottom value or function in the language), then the "previous" prototype (its child, to the left) will receive ("inherit") the result of that "next" computation (its parent, to the right), and so on until the first prototype (leftmost, most recent child) inherits its `super` value from the rest and computes the final instance. This allows prototypes to cooperate with other prototypes on a *same* aspect of the instance computation, wherein children prototypes can accumulate, modify or override the method values inherited from the parent prototypes.

1.1.5 Applicability to other Programming Languages. The two definitions above can be easily translated to any language with closures and either dynamic types or dependent types. However, their potential is not fully realized in languages with mere parametric polymorphism (see section 3.4). Furthermore, for an *efficient* implementation of objects with our formulas, we will also require lazy evaluation (see section 3), whether an ubiquitous language feature or an optional one (possibly user-implemented with side-effects). We do not otherwise require side-effects—though they can be used for the usual optimizations in the common "linear" case (see section 6).

1.2 A minimal object system

1.2.1 Records as functions. Let us relate the above two functions to objects, by first encoding "records" of multiple named values as functions from symbol (the name of a slot) to value (bound to the slot). In accordance with Lisp tradition, we will say *slot* where others may say "field" or

"member" or "method", and say that the slot is *bound* to the given value rather than it "containing" the value or any such thing.

Thus function x1-y2 below encodes a record with two slots x, y bound respectively to 1 and 2.

```
; x1-y2 : (Fun 'x -> Nat | 'y -> Nat)
(define (x1-y2 msg) (case msg ((x) 1)
                              ((y) 2)
                              (else (error "unbound slot" msg))))
```

We can check that we can indeed access the record slots and get the expected values:

```
> (list (x1-y2 'x) (x1-y2 'y))
'(1 2)
```

Note that we use Scheme symbols for legibility. Poorer languages could instead use any large enough type with decidable equality, such as integers or strings. Richer languages (e.g. Racket or some Scheme with syntax-case) could use resolved hygienic identifiers, thereby avoiding accidental clashes.

1.2.2 Prototypes for Records. A *prototype* for a record is a function of two record arguments self, super that returns a record extending super. To easily distinguish prototypes from instances and other values, we will follow the convention of prefixing with $ the names of prototypes. Thus, the following prototype extends or overrides its super-record with slot x bound to 3:

```
; $x3 : (Proto (Fun 'x -> Nat | A) A)
(define ($x3 self super) (λ (msg) (if (eq? msg 'x) 3 (super msg))))
```

This prototype extends a record with a new slot z bound to a complex number computed from real and imaginary values bound to the respective slots x and y of its self:

```
; $z<-xy : (Proto (Fun (Or 'x 'y) -> Real | 'z -> Complex | A)
;  (Fun (Or 'x 'y) -> Real | A))
(define ($z<-xy self super)
  (λ (msg) (case msg
             ((z) (+ (self 'x) (* 0+1i (self 'y))))
             (else (super msg)))))
```

That prototype doubles the number in slot x that it *inherits* from its super record:

```
; $double-x : (Proto (Fun 'x -> Number | A) (Fun 'x -> Number | A))
(define ($double-x self super)
  (λ (msg) (if (eq? msg 'x) (* 2 (super 'x)) (super msg))))
```

More generally a record prototype extends its super record with new slots and/or overrides the values bound to its existing slots, and may in the process refer to both the records self and super and their slots, with some obvious restrictions to avoid infinite loops from circular definitions.

1.2.3 Basic testing. How to test these prototypes? With the fix operator using the above record x1-y2 as base value:

```
; x3-y2 : (Fun 'x -> Nat | 'y -> Nat)
> (define x3-y2 (fix $x3 x1-y2))
> (list (x3-y2 'x) (x3-y2 'y))
'(3 2)
; z1+2i : (Fun 'x -> Nat | 'y -> Nat | 'z -> Complex)
> (define z1+2i (fix $z<-xy x1-y2))
```

```
> (list (z1+2i 'x) (z1+2i 'y) (z1+2i 'z))
'(1 2 1+2i)
; x2-y2 : (Fun 'x -> Nat | 'y -> Nat)
> (define x2-y2 (fix $double-x x1-y2))
> (list (x2-y2 'x) (x2-y2 'y))
'(2 2)
```

We can also `mix` these prototypes together before to compute the `fix`:

```
; z6+2i : (Fun 'x -> Nat | 'y -> Nat | 'z -> Complex)
> (define z6+2i (fix (mix $z<-xy (mix $double-x $x3)) x1-y2))
> (map z6+2i '(x y z))
'(6 2 6+2i)
```

And since the `$z<-xy` prototype got the `x` and `y` values from `self` and not `super`, we can freely commute it with the other two prototypes that do not affect either override slot `z` or inherit from it:

```
> (list ((fix (mix $z<-xy (mix $double-x $x3)) x1-y2) 'z)
        ((fix (mix $double-x (mix $z<-xy $x3)) x1-y2) 'z)
        ((fix (mix $double-x (mix $x3 $z<-xy)) x1-y2) 'z))
'(6+2i 6+2i 6+2i)
```

`mix` is associative, and therefore the following forms are equivalent to the previous ones, though the forms above (folding right) are slightly more efficient than the forms below (folding left):

```
> (list ((fix (mix (mix $z<-xy $double-x) $x3) x1-y2) 'z)
        ((fix (mix (mix $double-x $z<-xy) $x3) x1-y2) 'z)
        ((fix (mix (mix $double-x $x3) $z<-xy) x1-y2) 'z))
'(6+2i 6+2i 6+2i)
```

Now, since `$double-x` inherits slot `x` that `$x3` overrides, there is clearly a dependency between the two that prevents them from commuting:

```
; x6-y2 : (Fun 'x -> Nat | 'y -> Nat)
> (define x6-y2 (fix (mix $double-x $x3) x1-y2))
; x3-y2 : (Fun 'x -> Nat | 'y -> Nat)
> (define x3-y2 (fix (mix $x3 $double-x) x1-y2))
> (list (x6-y2 'x) (x3-y2 'x))
'(6 3)
```

1.2.4 Building record prototypes. Here are general utility functions to build record prototypes. To define a record with a slot `k` mapped to a value `v`, use:

```
; $slot : (Fun k:Symbol V -> (Proto (Fun 'k -> V | A) A))
(define ($slot k v) ; k v: constant key and value for this defined slot
  (λ (self super) ; self super: usual prototype variables
    (λ (msg) ; msg: message received by the instance, a.k.a. method name.
      (if (equal? msg k) v ; if the message matches the key, return the value
        (super msg))))) ; otherwise, recurse to the super instance
```

What of inheritance? Well, we can modify an inherited slot using:

```
; $slot-modify : (Fun k:Symbol (Fun V -> W)
;  -> (Proto (Fun 'k -> W | A) (Fun 'k -> V | A) A) st: (<: V W))
(define ($slot-modify k modify) ; modify: function from super-value to value
```

```
    (λ (self super) (λ (msg)
      (if (equal? msg k) (modify (super msg))
        (super msg)))))
```

What if a slot depends on other slots? We can use this function

```
; $slot-compute : (Fun k:Symbol (Fun A -> V) -> (Proto (Fun 'k -> V | A) A))
(define ($slot-compute k fun) ; fun: function from self to value.
  (λ (self super)
    (λ (msg)
      (if (equal? msg k) (fun self)
        (super msg)))))
```

A general function to compute and override a slot would take as arguments both `self` and a function to `inherit` the super slot value, akin to `call-next-method` in CLOS (Gabriel et al. 1991):

```
; $slot-gen : (Fun k:Symbol (Fun A (Fun -> V) -> W)
;  -> (Proto (Fun 'k -> W | A) (Fun 'k -> V | A) A) st: (<: V W))
(define ($slot-gen k fun) ; fun: from self and super-value thunk to value.
  (λ (self super)
    (λ (msg)
      (define (inherit) (super msg))
      (if (equal? msg k) (fun self inherit) (inherit)))))
```

We could redefine the former ones in terms of that latter:

```
(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))))
```

Thus you can re-define the above prototypes as:

```
(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))))))
```

Here is a universal bottom function to use as the base for fix:

```
; bottom-record : (Fun Symbol -> _)
(define (bottom-record msg) (error "unbound slot" msg))
```

To define a record with a single slot `x` bound to `3`, we can use:

```
; x3 : (Fun 'x -> Nat)
> (define x3 (fix $x3 bottom-record))
> (x3 'x)
3
```

To define a record with two slots `x` and `y` bound to `1` and `2` respectively, we can use:

```
; x1-y2 : (Fun 'x -> Nat | 'y -> Nat)
> (define x1-y2 (fix (mix ($slot 'x 1) ($slot 'y 2)) bottom-record))
> (map x1-y2 '(x y))
'(1 2)
```

1.3 Prototype Basics

1.3.1 Long-form Basics. Let's rewrite `fix` and `mix` in long-form with variables `self` and `super` instead of `f` and `b`, etc. Thus, the instantiation function `(fix p b)` becomes:

```
; instantiate-prototype : (Fun (Proto Self Super) Super -> Self)
(define (instantiate-prototype prototype base-super)
  (define self (prototype (λ i (apply self i)) base-super))
  self)
```

A more thorough explanation of the above fixed-point function is in Appendix D. Meanwhile, the composition function `(mix p q)` becomes:

```
; compose-prototypes : (Fun (Proto Self Super) (Proto Super Super2)
;   -> (Proto Self Super2) st: (<: Self Super Super2))
(define (compose-prototypes child parent)
  (λ (self super2) (child self (parent self super2))))
```

Note the types of the variables and intermediate expressions:

```
child : (Proto Self Super)                   parent : (Proto Super Super2)
self : Self                                  super2 : Super2
(child self (parent self super2)) : Self     (parent self super2) : Super
```

When writing long-form functions, instead of vying for conciseness, we will use the same naming conventions as in the function above:

- `child` for a *prototype* at hand, in leftmost position;
- `parent` for a *prototype* that is being mixed with, in latter position;
- `self` for the *instance* that is a fixed point of the computation;
- `super` for the *instance* so-far accumulated or at the base of the computation.

Note the important distinction between *prototypes* and *instances*. Instances are the non-composable complete outputs of instantiation, whereas prototypes are the composable partial inputs of instantiation. Prototypes, increments of computation, are functions from an instance (of a subtype `Self`) and another instance (of a supertype `Super`) to yet another instance of the subtype `Self`.

1.3.2 Composing prototypes. The identity prototype as follows is a neutral element for mix. It does not override any information from the super/base instance, but only passes it through. It also does not consult information in the final fixed-point nor refers to it.

```
; identity-prototype : (Proto Instance Instance)
(define (identity-prototype self super) super)
```

Now, prototypes are interesting because `compose-prototypes` (a.k.a. `mix`) is an associative operator with neutral element `identity-prototype`. Thus prototypes form a monoid, and you can compose or instantiate a list of prototypes:

```
; compose-prototype-list : (Fun (IndexedList I (λ (i)
;   (Proto (A_ i) (A_ (1+ i))))) -> (Proto (A_ 0) (A_ (Card I))))
(define (compose-prototype-list prototype-list)
  (foldr compose-prototypes identity-prototype prototype-list))
; instantiate-prototype-list : (Fun (IndexedList I (λ (i)
;   (Proto (A_ i) (A_ (1+ i))))) (A_ (Card I)) -> (A_ 0))
(define (instantiate-prototype-list prototype-list base-super)
  (instantiate-prototype (compose-prototype-list prototype-list) base-super))
```

1.3.3 The Bottom of it. In a language with partial functions, such as Scheme, there is a practical choice for a universal base super instance to pass to `instantiate-prototype`: the `bottom` function, that never returns, but instead, for enhanced usability, may throw a helpful error.

```
; bottom : (Fun I ... -> O ...)
(define (bottom . args) (error "bottom" args))
```

Thus, in dialects with optional arguments, we could make `bottom` the default value for `base-super`. Furthermore, in any variant of Scheme, we can define the following function `instance` that takes the rest of its arguments as a list of prototypes, and instantiates the composition of them:

```
; instance : (Fun (IndexedList I (λ (i) (Proto (A_ i) (A_ (1+ i)))))...
;   -> (A_ 0)))
(define (instance . prototype-list)
  (instantiate-prototype-list prototype-list bottom))
```

What if you *really* wanted to instantiate your list of prototypes with some value `b` as the base super instance? "Just" tuck `($constant-prototype b)` at the tail end of your prototype list:

```
; $constant-prototype : (Fun A -> (Proto A _))
(define ($constant-prototype base-super) (λ (_self _super) base-super))
```

Small puzzle for the points-free Haskellers reading this essay: what change of representation will enable prototypes to be composed like regular functions without having to apply a binary function like `mix`? Solution in footnote.[1]

1.3.4 Single Inheritance. Most object systems do not offer programmers prototype composition, but only a much less expressive operation: prototype application.

Consider a *generator* to be a function from a function type to itself. To instantiate a generator is to compute its fixed point. There is a surjection from prototype and base value to generator, such that instantiating the generator is same as instantiating the prototype with the base value. Applying prototype `p` to the generator obtained from prototype `q` and base value `b` is the same as obtaining a generator from `(mix p q)` and `b`.

```
; (deftype (Generator A) (Fun A -> A))
; fix-generator : (Fun (Generator A) -> A)
(define (fix-generator g) (define f (g (λ i (apply f i)))) f)
; proto->generator : (Fun (Proto A B) B -> (Generator A))
(define (proto->generator p b) (λ (f) (p f b)))
; (== (fix-generator (proto->generator p b)) (fix p b))
; apply-proto : (Fun (Proto A B) (Generator B) -> (Generator A))
(define (apply-proto p g) (λ (f) (p f (g f))))
; (== (apply-proto p (proto->generator q b)) (proto->generator (mix p q) b))
```

"Single inheritance" consists in prototypes being second-class entities that you can only apply to generators immediately after having defined them, with a fixed base instance for all generators. Instead of being able to abstract over prototypes and compose them, you can only use constant prototypes and apply them to generators (that themselves are usually also second-class entities but can be abstracted over somewhat). To incrementally define an instance, you must `cons` prototypes one by one onto the list to instantiate.

---

[1] A general purpose trick to embed an arbitrary monoid into usual composable functions is to curry the composition function (here, `mix`) with the value to be composed. Thus, the functional embedding of prototype `p` will be `(λ (q) (mix p q)) : (Fun (Proto Super S2) -> (Proto Self S2))`. To recover `p` from that embedding, just apply it to `identity-prototype`.

Composition of first-class prototypes is obviously more expressive than single-inheritance. You can mix prototypes or append prototype lists, and programmatically compose code from many increments of code. Prototypes are thus more akin to the “mixins” or “traits” of more advanced objects systems (Bracha and Cook 1990; Cannon 1982; Flatt et al. 2006). Prototype composition however, does not by itself subsume multiple inheritance, that we will study in section 4.

2 PURE OBJECTIVE FUN

2.1 Using prototypes to incrementally define simple data structures

2.1.1 Prototypes for Order. Let’s use prototypes to build some simple data structures. First, we’ll write prototypes that offer an abstraction for the ability to compare elements of a same type at hand, in this case, either numbers or strings.

```
(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)))))
```

We can add a “mixin” for a compare operator that summarizes in one call the result of comparing two elements of the type being described. A mixin is a prototype meant to extend other prototypes. See how this mixin can be used to extend either of the prototypes above. Also notice how, to refer to other slots in the eventual instance, we call (self '<) and such.

```
(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))

> (list ((number-order '<) 23 42) ((number-order 'compare) 8 4)
  ((string-order '<) "Hello" "World") ((string-order 'compare) "Foo" "FOO")
  ((string-order 'compare) "42" "42"))
'(#t > #t > =)
```

We can define an order on symbols by delegating to strings!

```
(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)))))
(define symbol-order (instance $symbol-order))
```

```
> (list ((symbol-order '<) 'aardvark 'aaron) ((symbol-order '=) 'zzz 'zzz)
  ((symbol-order '>) 'aa 'a) ((symbol-order 'compare) 'alice 'bob)
  ((symbol-order 'compare) 'b 'c) ((symbol-order 'compare) 'c 'a))
'(#t #t #t < < >)
```

2.1.2 Prototypes for Binary Trees. We can use the above `order` prototypes to build binary trees over a suitable ordered key type `Key`. We'll represent a tree as a list of left-branch, list of key-value pair and ancillary data, and right-branch, which preserves the order of keys when printed:

```
(define ($binary-tree-map self super)
  (λ (msg)
    (define (node l kv r) ((self 'node) l kv r))
    (case msg
      ((empty) '())
      ((empty?) null?)
      ((node) (λ (l kv r) (list l (list kv) r)))
      ((singleton) (λ (k v) (node '() (cons k v) '())))
      ((acons)
       (λ (k v t)
         (if ((self 'empty?) t) ((self 'singleton) k v)
             (let* ((tl (car t)) (tkv (caadr t)) (tk (car tkv)) (tr (caddr t)))
               (case (((self 'Key) 'compare) k tk)
                 ((=) (node tl (cons k v) tr))
                 ((<) (node ((self 'acons) k v tl) tkv tr))
                 ((>) (node tl tkv ((self 'acons) k v tr))))))))
      ((ref)
       (λ (t k e)
         (if ((self 'empty?) t) (e)
             (let ((tl (car t)) (tk (caaadr t)) (tv (cdaadr t)) (tr (caddr t)))
               (case (((self 'Key) 'compare) k tk)
                 ((=) tv)
                 ((<) ((self 'ref) tl k e))
                 ((>) ((self 'ref) tr k e)))))))
      ((afoldr)
       (λ (acons empty t)
         (if ((self 'empty?) t) empty
           (let ((tl (car t)) (tk (caaadr t)) (tv (cdaadr t)) (tr (caddr t)))
             ((self 'afoldr)
              acons (acons tk tv ((self 'afoldr) acons empty tl)) tr)))))
      (else (super msg)))))
```

With this scaffolding, we can define a dictionary data structure that we can use later to differently represent objects:

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

However, when we use it, we immediately find an issue: trees will too often be skewed, leading to long access times, especially so when building them from an already ordered list:

```
> (define my-binary-dict ; heavily skewed right, height 5
    (foldl (λ (kv t) ((symbol-tree-map 'acons) (car kv) (cdr kv) t))
```

```
          (symbol-tree-map 'empty)
          '((a . "I") (b . "II") (c . "III") (d . "IV") (e . "V"))))
> my-binary-dict
'(()
  ((a . "I"))
  (() ((b . "II")) (() ((c . "III")) (() ((d . "IV")) (() ((e . "V")) ())))))
```

But binary trees otherwise work:

```
> (map (λ (k) ((symbol-tree-map 'ref) my-binary-dict k (λ () #f))) '(a b c d e z))
'("I" "II" "III" "IV" "V" #f)
```

2.1.3 Prototypes for Balanced *Binary Trees.* We can incrementally define a balanced tree data structure (in this case, using the AVL balancing algorithm) by overriding a single method of the original binary tree prototype:

```
(define ($avl-tree-rebalance self super)
  (λ (msg)
    (define (left t) (car t))
    (define (kv t) (caadr t))
    (define (height t) (if (null? t) 0 (cdadr t)))
    (define (right t) (caddr t))
    (define (balance t) (if (null? t) 0 (- (height (right t)) (height (left t)))))
    (define (mk l kv r)
      (let ((lh (height l)) (rh (height r)))
        (or (member (- rh lh) '(-1 0 1)) (error "tree unbalanced!"))
        (list l (cons kv (+ 1 (max lh rh))) r)))
    (define (node l ckv r)
      (case (- (height r) (height l))
        ((-1 0 1) (mk l ckv r))
        ((-2) (case (balance l)
                ((-1 0) (mk (left l) (kv l) (mk (right l) ckv r)))
                ((1) (mk (mk (left l) (kv l) (left (right l)))
                         (kv (right l)) (mk (right (right l)) ckv r)))))
        ((2) (case (balance r)
               ((-1) (mk (mk l ckv (left (left r)))
                         (kv (left r)) (mk (right (left r)) (kv r) (right r))))
               ((0 1) (mk (mk l ckv (left r)) (kv r) (right r)))))))
    (case msg ((node) node) (else (super msg)))))
(define Dict
  (instance $avl-tree-rebalance $binary-tree-map ($slot 'Key symbol-order)))
```

Our dictionary is now well-balanced, height 3, and the tests still pass:

```
> (define my-avl-dict
    (foldl (λ (kv t) ((Dict 'acons) (car kv) (cdr kv) t))
           (Dict 'empty)
           '((a . "I") (b . "II") (c . "III") (d . "IV") (e . "V"))))
> my-avl-dict
'((() ((a . "I") . 1) ())
  ((b . "II") . 3)
```

```
   ((() ((c . "III") . 1) ()) ((d . "IV") . 2) (() ((e . "V") . 1) ())))
> (map (λ (k) ((Dict 'ref) my-avl-dict k (λ () #f))) '(a b c d e z))
'("I" "II" "III" "IV" "V" #f)
```

3 BEYOND OBJECTS AND BACK

Prototypes are not just for records as functions from symbol to value: they can be used to incrementally specify any kind of functions!

3.1 Prototypes for Numeric Functions

Let's see how prototypes can be used to build functions from real numbers to real numbers. The following prototype is a mixin for an even function, that delegates to other prototypes the images of positive values and returns the image of the opposite for negative values:

```
(define ($even self super) (λ (x) (if (< x 0) (self (- x)) (super x))))
```

The following prototype is a mixin for taking the cube of the parent value:

```
(define ($cube self super) (λ (x) (let ((y (super x))) (* y y y))))
```

We can instantiate a function out of those of prototypes, and test it:

```
> (define absx3 (instance $even $cube ($constant-prototype (λ (x) x))))
> (map absx3 '(3 -2 0 -1))
'(27 8 0 1)
```

3.1.1 Number Thunks. Now, the simplest numeric functions are thunks: nullary functions that yield a number.

```
> (define (p1+ _ b) (λ () (+ 1 (b))))
> (define (p2* _ b) (λ () (* 2 (b))))
> (list ((fix (mix p1+ p2*) (λ () 30)))
        ((fix (mix p2* p1+) (λ () 30))))
'(61 62)
```

3.2 Prototypes are for Computations not Values

Prototypes for number thunks can be generalized to prototypes for any kind of thunks: you may incrementally specify instances of arbitrary types using prototypes, by first wrapping values of that type into a thunk. An alternative to thunks would be to use Scheme's `delay` special form, or whatever form of *lazy evaluation* is available in the language at hand.

To reprise the Call-By-Push-Value paradigm (Blain Levy 1999), prototypes incrementally specify *computations* rather than *values*. In applicative languages we can reify these computations as values either as functions (as in section 1), or as delayed values as follows:

```
; (deftype (δProto Self Super) (Fun (Delayed Self) (Delayed Super) -> Self))
; δfix : (Fun (δProto Self Super) (Delayed Super) -> (Delayed Self))
(define (δfix p b) (define f (delay (p f b))) f)
; δmix : (Fun (δProto Self Mid) (δProto Mid Super) -> (δProto Self Super))
(define (δmix c p) (λ (f b) (c f (delay (p f b)))))
; δ$id : (δProto X X)
(define (δ$id f b) (λ (_self super) (force super)))
```

Now, most interesting prototypes will only lead to error or divergence if you try to instantiate them by themselves—they are "mixins", just like `$compare<-order` or `$avl-tree-rebalance` above,

designed to be combined with other prototypes. Indeed, the entire point of incremental programming is that you want to define and manipulate fragments that are not complete specifications. To use these fragments in a total language where all computations terminate will require attaching to each prototype some side-condition as to which aspects of a computation it provides that other prototypes may rely on, and which aspects of a computation it requires that other prototypes must provide.

3.3 Prototypes are Useful Even Without Subtyping

The above prototypes for numeric functions also illustrate that even if a language's only subtyping relationship is the identity whereby each type is the one and only subtype or itself (or something slightly richer with syntactic variable substitution in parametric polymorphism), then you can use prototypes to incrementally specify computations, with the following monomorphic types:

```
; (deftype (MProto A) (Fun A A -> A))
; fix : (Fun (MProto A) A -> A)
; mix : (Fun (MProto A) (MProto A) -> (MProto A))
```

As per the previous discussion, if the language does not use lazy evaluation, `A`'s may be constrained to be functions, or to they may have to be wrapped inside a `Lazy` or `delay` special form.

Lack of subtyping greatly reduces the expressive power of prototypes; yet, as we'll see in section 5, the most popular use of prototypes in Object-Oriented Programming is completely monomorphic: prototypes for type descriptors, a.k.a. classes; only this common use of prototypes happens at the meta-level, classes being second-class objects (pun not intended by unaware practitioners).

4 BETTER OBJECTS, STILL PURE

In section 1, we implemented a rudimentary object system on top of prototypes. Compared to mainstream object systems, it not only was much simpler to define, but also enabled mixins, making it more powerful than common single-inheritance systems. Still, many bells and whistles found in other object systems are missing. Now that we have a nice and simple semantic framework for "objects", can we also reimplement the more advanced features of object systems of yore?

4.1 Slot Introspection

4.1.1 Bug and feature. When representing records as functions from symbol to value, it is not generally possible to access the list of symbols that constitute valid keys. Most object systems do not allow for this kind of introspection at runtime, and further introduce scoping or access control rules to enable modular reasoning about slot accesses, advertising such restrictions as "encapsulation" or "information hiding" features. Yet, introspection can be useful to e.g. automatically input and output human-readable or network-exchangeable representations of an instance, and has also been advertised as a "reflection" feature. Some languages even offer both contradictory sets of features.

4.1.2 Same concrete representation, abstract constructor. Slot introspection can be achieved while keeping records as functions from keys to values, by adding a "special" key, e.g. `keys`, that will be bound to the list of valid keys. To save themselves the error-prone burden of maintaining the list of keys by hand, programmers would then use a variant of `$slot-gen` that maintains this list, as in:

```
(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))))))
```

4.1.3 Different instance representation. Slot introspection can also be achieved by using a different representation for records. Instead of functions, mappings from symbols to value could be represented using hash-tables, or, preferrably in a pure setting, balanced trees. Purposefully, we implemented such balanced trees in section 2. Thus, we could represent records as thunks that return a `symbol-avl-map`, as follows:

```
(define ($slot-gen/dict k fun)
  (λ (self super)
    (define (inherit) ((Dict 'ref) (super) k bottom))
    (λ () ((Dict 'acons) k (fun self inherit) (super)))))
```

However, while the above definition yields the correct result, it potentially recomputes the entire dictionary `(super)` twice at every symbol lookup, with exponential explosion as the number of super prototypes increases. We could carefully implement sharing by precomputing `(define super-dict (super))`. But the instance is itself a thunk that would be recomputing the entire dictionary at every call, and is better evaluated only once. Now, if we adopt lazy evaluation, we can automatically share results across multiple copies of a computation as well as multiple consumers of a same copy, without having to manually reimplement this caching every time. Thus, using `δfix` and `δmix` instead of `fix` and `mix`, we can have:

```
(define (δ$slot-gen/dict k fun)
  (λ (self super)
    (delay (let ((inherit ((Dict 'ref) (force super) k (λ () (delay (bottom))))))
             ((Dict 'acons) k (delay (fun self inherit)) (force super))))))
```

4.1.4 Same instance representation, different prototype representation. Finally, slot introspection can be achieved while preserving the same instance representation by instead changing the representation for *prototypes*. Though the concise functional representation we offered gives an enlightening expression of the essence of object systems, we can maintain equivalent semantics with a more efficient implementation, for instance a pair `pk` of the same prototype as before and a list of keys. The `mix` and `fix` functions become:

```
(define (mix/pk child parent)
  (cons (mix (car child) (car parent))
        (append (cdr child) (cdr parent))))
(define (fix/pk proto base)
  (fix (mix ($slot 'keys (cdr proto)) (car proto)) base))
```

One could represent sets of keys by a data structure with a union more efficient than list append. One could also put `$slot` invocation after the `(car proto)` rather than before it, to allow prototypes to intercept slot introspection.

4.1.5 Abstracting over representations. We can generalize the above solutions by realizing that both instances and prototypes may be may be wrapped, unwrapped or otherwise represented in a variety of ways, and augmented with various other information, when designing and implementing an object system for usability and performance. Yet, the representation given in section 1 is a good guide to the semantics of OOP, and the conceptual distinction between instance and prototype is instrumental to keeping this semantics simple and understandable.

4.2 Unifying Instances and Prototypes

4.2.1 OOP without Objects. We have so far insisted on the distinction between instance and prototype. Each embody one aspect of what is usually meant by "object", yet neither embody all of

it. We are thus in a strange situation wherein we have been doing "Object-Oriented Programming" without any actual object!

This is a practical problem, not a mere curiosity, as this distinction makes it painful to write extensible specifications for nested or recursive data structures: you need to decide for each slot of each data structure at each stage of computation whether it contains an extensible prototype or an instance to call the proper API. This incurs both a psychological cost to programmers and a runtime cost to evaluation, that without careful design may in the worst case cause an exponential explosion of the code the programmers must write and/or of the recomputations of sub-expressions that the evaluator will do at runtime.

4.2.2 Conflation without confusion. Jsonnet and Nix (see section 7) both confront and elegantly solve the above issue, and in the very same way, with one small trick: they bundle and conflate together instance and prototype in a same single entity, the "object". Indeed, in a pure context, and given the base super value for the given prototype representation, there is one and only one instance associated to a prototype up to any testable equality, and so we may usefully cache the computation of this instance together with the prototype. The same "object" entity can thus be seen as an instance and queried for method values, or seen as a prototype and composed with other objects (also seen as prototypes) into a new object.

Thanks to the conflation of instance and prototype as two aspects of a same object, configurations can be written that can refer to other parts of the configuration without having to track and distinguish which parts are instantiated at which point, and it all just works. Still, distinguishing the two concepts of instance and prototype is important to dispel the confusion that can often reign in even the most experienced OO practitioner regarding the fine behavior of objects when trying to assemble or debug programs.

In Jsonnet, this conflation is done implicitly as part of the builtin object system implementation. In Nix, interestingly, there are several unassuming variants of the same object system, that each store the prototype information in one or several special fields of the `attrset` that is otherwise used for instance information. Thus, in the simplest Nix object system, one could write:

```
fix' (self: { x = 1; y = 2 + self.x; })
```

and it would evaluate to an attrset equivalent to:

```
{ x = 1; y = 3; __unfix__ = self: { x = 1; y = 2 + self.x; }; }
```

Nix contains many variants of the same object system, that use one or several of `extend`, `override`, `overrideDerivation`, `meta`, etc., instead of `__unfix__` to store composable prototype information.

4.2.3 Practical Conflation for Fun and Profit. In the rest of this article, we'll represent an object as a pair of an instance and a prototype:

```
(define (make-object instance prototype) (cons instance prototype))
(define (object-instance object) (car object))
(define (object-prototype object) (cdr object))
```

In a more robust implementation, we would use an extension to the language Scheme to define a special structure type for objects as pairs of instance and prototype, disjoint from the regular pair type. Thus, we can distinguish objects from regular lists, and hook into the printer to offer a nice way to print instance information that users are usually interested in while skipping prototype information that they usually are not.

To reproduce the semantics of Jsonnet, instances will be a delayed `Dict` (as per section 2), mapping symbols as slot names to delayed values as slot computations; meanwhile the prototype will be a prototype wrapper of type `(δProto Object Object)` (as in section 4.1.3), that preserves the prototype while acting on the instance. Thus the basic function `slot-ref` to access a slot,

and the basic prototypes to define one, analogous to the above `$slot-gen`, `$slot`, `$slot-modify`, `$slot-compute` are as follow:

```
(define (slot-ref object slot)
  (force ((Dict 'ref) (force (object-instance object)) slot bottom)))
(define ($slot/gen k fun)
  (λ (self super)
    (make-object ((Dict 'acons) k (fun self (delay (slot-ref super k)))
                  (object-instance (force super)))
                 (object-prototype (force super)))))
(define ($slot/value k v) ($slot/gen k (λ (_self _inherit) (delay v))))
(define ($slot/modify k modify)
  ($slot/gen k (λ (_ inherit) (delay (modify (force inherit))))))
(define ($slot/compute k fun) ($slot/gen k (λ (self _) (delay (fun self)))))
```

And for the common case of just overriding some slots with constant values, we could use:

```
(define ($slot/values . kvs)
  (if (null? kvs) identity-prototype
    (compose-prototypes ($slot/value (car kvs) (cadr kvs))
                        (apply $slot/values (cddr kvs)))))
```

4.3 Multiple Inheritance

4.3.1 The inheritance modularity issue. To write incremental OO programs, developers need to be able to express dependencies between objects, such that an object `Z` depends on super objects `K1`, `K2`, `K3` being present after it in the list of prototypes to be instantiated. We'll also say that `Z` *inherits from* from these super objects, or *extends* them, or that they are its direct super objects.

But what if `K1`, `K2`, `K3` themselves inherit from super objects `A`, `B`, `C`, `D`, `E`, e.g. with `K1` inheriting from direct supers `A B C`, `K2` inheriting from direct supers `D B E`, and `K3` inheriting from direct supers `D A`, and what more each of `A`, `B`, `C`, `D`, `E` inheriting from a base super object `O`? (See Appendix B for details on this example.)

With the basic object model offered by Nix, Jsonnet, or our section 1, these dependencies could not be represented in the prototype itself. If the programmer tried to "always pre-mix" its dependencies into a prototype, then `K1` would be a pre-mix `K1 A B C O`, `K2` would be a pre-mix `K2 D B E O`, `K3` would be a pre-mix `K3 D A O`, and when trying to specify `Z`, the pre-mix `Z K1 A B C O K2 D B E O K3 D A O` would be incorrect, with unwanted repetitions of `A`, `B`, `D`, `O` redoing their effects too many times and possibly undoing overrides made by `K2` and `K3`. Instead, the programmer would have to somehow remember and track those dependencies, such that when he instantiates `Z`, he will not write just `Z`, but `Z` followed a topologically sorted *precedence list* where each of the transitive dependencies appears once and only once, e.g. `Z K1 K2 K3 D A B C E O`.

Not only does this activity entail a lot of tedious and error-prone bookkeeping, it is not modular. If these various objects are maintained by different people as part of separate libraries, each object's author must keep track not just of their direct dependencies, but all their transitive indirect dependencies, with a proper ordering. Then they must not only propagate those changes to their own objects, but notify the authors of objects that depend on theirs. To get a change fully propagated might required hundreds of modifications being sent and accepted by tens of different maintainers, some of whom might not be responsive. Even when the sets of dependencies are properly propagated, inconsistencies between the orders chosen by different maintainers at

different times may cause subtle miscalculations that are hard to detect or debug. In other words, while possible, manual maintenance of precedence lists is a modularity nightmare.

4.3.2 Multiple inheritance to the rescue. With multiple inheritance, programmers only need declare the dependencies between objects and their direct super objects: the object system will automatically compute a suitable precedence list in which order to compose the object prototypes. Thus, defining objects with dependencies becomes modular.

The algorithm that computes this precedence list is called a linearization: It considers the dependencies as defining a directed acyclic graph (DAG), or equivalently, a partial order, and it completes this partial order into a total (or linear) order, that is a superset of the ordering relations in the partial order. The algorithm can also detect any ordering inconsistency or circular dependency whereby the dependencies as declared fail to constitute a DAG; in such a situation, no precedence list can satisfy all the ordering constraints, and instead an error is raised. Recent modern object systems seem to have settled on the C3 linearization algorithm, as described in Appendix C.

4.3.3 A prototype is more than a function. But where is the inheritance information to be stored? A prototype must contain more than the function being composed to implement open-recursion. *A minima* it must also contain the list of direct super objects that the current object depends on. We saw in 4.1.4 how we can and sometimes must indeed include additional information in a prototype. Such additional information will include a precomputed cache for the precedence list below, but could also conceivably include type declarations, method combinations, and support for any imaginable future feature.

We could start by making the prototype a pair of prototype wrapper and list of supers; if more data elements are later needed, we could use a vector with every element at a fixed location. But since we may be adding further features to the object system, we will instead make the prototype itself an object, with a special base case to break the infinite recursion. Thus, the same functions as used to query the slot values of an object's instance can be used to query the data elements of the prototype (modulo this special case). But also, the functions used to construct an object can be used to construct a prototype, which lays the foundation for a meta-object protocol (Kiczales et al. 1991), wherein object implementation can be extended from the inside.

When it is an object, a prototype will have slot `function` bound to a prototype wrapper as previously, and slot `supers` bound to a list of super objects to inherit from, as well as a slot `precedence-list` bound to a precomputed cache of the precedence list. When it is the special base case, for which we below chose the empty list, slots `supers` and `precedence-list` are bound to empty lists, and slot `function` is bound to a function that overrides its super with constant slots from the instance.

```
(define (Dict->Object dict) (make-object dict '()))
(define (object-prototype-function object)
  (define prototype (object-prototype object))
  (if (null? prototype)
    (λ (self super)
      (make-object (Dict-merge (object-instance object) (object-instance super))
                   (object-prototype super)))
    (slot-ref prototype 'function)))
(define (object-supers object)
  (define prototype (object-prototype object))
  (if (null? prototype) '() (slot-ref prototype 'supers)))
(define (object-precedence-list object)
```

```scheme
    (define prototype (object-prototype object))
    (if (null? prototype) '() (slot-ref prototype 'precedence-list)))
  (define (compute-precedence-list object)
    (c3-compute-precedence-list object object-supers object-precedence-list))
  (define base-dict (Dict 'empty))
  (define (instantiate supers function)
    (define proto
      (Dict->Object ((Dict 'acons) 'function (delay function)
                     ((Dict 'acons) 'supers (delay supers)
                      ((Dict 'acons) 'precedence-list (delay precedence-list)
                       (Dict 'empty))))))
    (define base (make-object base-dict proto))
    (define precedence-list (compute-precedence-list base))
    (define prototype-functions (map object-prototype-function precedence-list))
    (instantiate-prototype-list prototype-functions (delay base)))
  (define (object function . supers) (instantiate supers function))
```

4.4 Further Features

Other advanced OOP features found in CLOS (Bobrow et al. 1988), such as multi-methods (a.k.a multiple dispatch) or method combinations, can also be expressed in a pure setting. However, they require further extensions to the model we propose, that we discuss in our Appendix A.

5 CLASSES

5.1 Classes on top of Prototypes

5.1.1 Layering Language Features. So far we have reconstructed prototype-based OOP; yet class-based OOP comes first both in history and in popularity. There have been implementations of classes on top of prototypes in the past, notably for JavaScript (International 2015). But these implementations relied heavily on side-effects over some object encoding, to achieve efficiency within some existing language. Instead we will propose our own reconstruction on how to *macro-express* (Felleisen 1991) classes on top of our pure functional prototypes. This reconstruction will shed light on the essence of the relationship between classes and prototypes.

5.1.2 Prototypes for Type Descriptors. In our reconstruction, a *class* is “just” a prototype for type descriptors. Type descriptors, as detailed below, are a runtime data structure describing what operations are available to recognize and deal with elements of the given type. Type descriptors therefore do not have to themselves be objects, and no mention of objects is required to describe type descriptors themselves. They can be just a type on which to apply the monomorphic prototypes of section 3.4. Still, it is typical in OOP to conflate into a “class” both the instance of a type descriptor and the prototype for the type descriptor. Our distinction of the two concepts can then help avoid a lot of the confusion present in classical presentations of OOP.

Every compiler or language processor for a statically typed language (even without OOP) necessarily has type descriptors, since the language’s compile-time is the compiler’s runtime. But in most statically typed languages, there are no dependent types, and the types themselves (and the classes that are their prototypes) are not first-class entities (Strachey 1967) that can be arguments and results of runtime computations, only second-class entities that are fully resolved at compile-time. Therefore, class-based languages only have second-class classes whereas only prototype-based languages have first-class classes.

5.2 Type Descriptors

5.2.1 I/O Validation and Beyond. One common programming problem is validation of data at the inputs and outputs of programs. Static types solve the easy cases of validation in languages that have them. Dynamically typed languages cannot use this tool so often grow libraries of "type descriptors" or "data schemas", etc.

These runtime type descriptors often contain more information than typically available in a static type, such as: methods to encode and decode values, to print and parse them, to display them or interact with them in a graphical interface; default values for use in user interfaces or simple tests; pseudo-random value generators and value compressors for use in automated testing and other search algorithms; algebraic operations whereby this type implements an interface, satisfies a constraint, or instantiates a typeclass; etc. Types used at compile-time would have additional data to deal with how to infer them, how they lead to further inferences, how to compute unions or intersections with other types, how to generate code for operations that deals with them, etc.

Therefore, even statically typed languages often involve runtime type descriptors. Better languages will provide "reflection" facilities or "macros" to automatically generate those descriptors without humans having to try keeping two different representations in synch as programs evolve.

5.2.2 Simple Types. In a dynamic language, all types would have at least a recognizer slot `is?`, bound to a function to recognize whether a runtime value is element of the type. In the most basic types, `Top` (that tells nothing about everything) and `Bottom` (that tells everything about nothing), this might be the only slot (though an additional `name` slot would help), with trivial values:

```
(define Top (object ($slot/value 'is? (λ (_) #t))))
(define Bottom (object ($slot/value 'is? (λ (_) #f))))
```

Other simple types might include the type of representable values; here they will sport a single additional method `->sexp` to turn a value into a type-specific source expression, but a real library would have many more I/O operations. Numbers would also have various arithmetic operations.

```
(define Number (object ($slot/values 'is? number? '->sexp identity
                                     '+ + '- - 'zero 0 'one 1)))
```

5.2.3 Parameterized Types. A parameterized type at runtime can be function that return a type descriptor given its parameter, itself a type descriptor, or any runtime value (yielding a runtime dependent type). Thus, monomorphic lists might be:

```
(define (list-of? t x) (or (null? x)
  (and (pair? x) ((slot-ref t 'is?) (car x)) (list-of? t (cdr x)))))
(define (ListOf t) (object ($slot/value 'is? (λ (x) (list-of? t x)))))
```

5.2.4 More Elaborate Types. By using object prototypes, we have, in a few hundreds of lines of code (Rideau 2020), defined type descriptors for functions with given input and output types; for dicts, objects, and objects having specific slots with values of specific types; for slot and type descriptors, and functions that compute type descriptors from type descriptors or other values; etc. The type descriptor system can describe itself as well as all pieces of the system, generating routines for all kinds of automation or user interaction.

Along the way, prototype-based OOP enables a compact and extensible style of programming, leveraging all the advantages of classes or typeclasses for defining pure functional types.

5.2.5 Classes and Typeclasses. Runtime type descriptors correspond to the "dictionaries" passed around at runtime in implementations of typeclasses (Peterson and Jones 1993) in various FP languages. They also correspond to the "vtable" describing an "object"'s type at runtime in implementations of class-based OOP languages. Indeed, the correspondance can be made formal,

with automated transformations between the two styles (Rideau 2012). Note however how "object" denotes very different notions in the two styles: what is called "object" in class-based OOP is the notional pair of the type descriptor and an element of the described type, whereas the type descriptor is (an instance of) an "object" in our prototype-based formalism, handled independently from the elements of the described type, that themselves may or may not be objects.

6 MUTABILITY

6.1 Mutability as Pure Linearity

6.1.1 From Pure to Mutable and Back. Note how all the objects and functions defined in previous sections were pure. They did not use any side-effect. No `set!`. No `call/cc`. Only laziness at times, which still counts as pure. This contrasts with all OOP literature from the 1960s to the 1980s, and most since. But what if we are OK with side-effects? How much does that simplify computations? What insights does the pure case offer on the mutable case, and vice versa?

A simple implementation of mutable objects is to store pure object values into mutable cells. Conversely, a mutable object can be viewed as a monadic thread of pure object state references, with an enforced linearity constraint wherein effectful operations return a new state reference after invalidating the previous one. Indeed, this isomorphism is not just a curiosity from category theory, it is a pair of transformations that can be and have been automated for practical purposes (Rideau 2012).

6.1.2 Combining Mutable and Functional Wins. Mutability allows for various optimizations, wherein objects follow common linearity constraints of consuming some arguments that are never referenced again after use, at which point their parts can be transformed or recycled "in place" with local modifications in $O(1)$ machine operations, rather than global copies in $O(\log n)$ or $O(n)$ machine operations. As a notable simplification, the "super" computation as inherited by a prototype is linear (or more precisely, affine: the prototype may either consume the super value, or ignore it). Thus, the entire fixed-point computation can be done in-place by updating slots as they are defined. This is a win for mutability over purity.

Yet, even mutable (or linear) rather than pure (and persistent), the functional approach solves an important problem with the traditional imperative approach: traditionally, programmers must be very careful to initialize object slots in a suitable order, even though there is no universal solution that possibly works for all future definitions of inheriting objects. The traditional approach is thus programmer-intensive in a subtle way, and leads to many errors from handling of unbound slots. Depending on the quality of the implementation, the consequences may range from a error being raised with a helpful message at compile-time or at runtime, to useless results, wrong results, memory corruption, or catastrophic failures. By defining objects functionally or lazily rather than imperatively and eagerly, programmers can let the implementation gracefully handle mutual references between slots; an initialization order can be automatically inferred wherein slots are defined before they are used, with a useful error detected and raised (at runtime) if no such order exists.

6.1.3 Simplified Prototype Protocol. A prototype for a mutable object can be implemented with a single mutable data structure argument `self` instead of two immutable value arguments `self` and `super`: the *identity* of that data structure provides the handle to future complete computation, as previously embodied in the `self` argument; and the current *storage* of the data structure provides state of the computation so far, as previously embodied in the `super` argument.

A mutable prototype wrapper would then be of type `(deftype μProto (Fun Object ->))`, where the `Object`'s instance component would typically contain a mutable hash-table mapping slot

names (symbols) to effective methods to compute each slot value. These effective methods would be either thunks or lazy computations, and would already close over the identity of the object as well as reuse its previous state. Overriding a slot would update the effective method in place, based on the new method, the self (identity of the object) and the super-inherited entry previously in the hash-table. Since this protocol is based on side-effects, no need to return `self` in the end; the fix operator variant will also rely on side-effects.

```
(define (fix! p b) (def f (hash-copy b)) (p f) f)
(define (mix! p q) (λ (f) (q f) (p f)))
(define ($slot-gen! k fun)
  (λ (self) (define inherit (hash-ref self k (delay (bottom))))
            (hash-set! self k (fun self inherit))))
```

6.2 Cache invalidation

There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors (Phil Karton). One issue with making objects mutable is that modifications may make some previously computed results invalid. There are several options then, none universally satisfactory.

The object system may embrace mutation and just never implicitly cache the result of any computation, and let the users explicitly insert any cache desired. That's the design commonly followed by prototype systems in effectful languages from T to JavaScript. But constant recomputation can be extremely expensive; moreover, many heavy computations may be below the level at which users may intervene—including computing precedence lists. This probably explains why these languages tend not to support multiple inheritance, and if so to handle it specially.

At the opposite end of the spectrum, the object system may assume purity-by-default and cache all the computations it can. It is then up to users to explicitly create cells to make some things mutable, or flush some caches when appropriate. We used this option in Rideau (2020).

In between these two opposites, the system can automatically track which mutations happen, and invalidate those caches that need be, with a tradeoff between how cheap it will be to use caches versus to mutate objects. However, to be completely correct, this mutation tracking and cache invalidation must be pervasive in the entire language, not just the object system implementation, at which point the language is more in the style of higher-order reactive or incremental programming.

7 RELATED WORK

7.1 Prototype Object Systems

7.1.1 Director and ThingLab. The first prototype object system might have been Director a.k.a. Ani (Kahn 1976, 1979), an actor system to create animations from story constraints, written in MacLisp at MIT. The system is informally described as a modifiable hierarchy of message-passing objects that inherit from other objects to which they "pass the buck" when they receive a message that doesn't match any of the patterns they directly have rules for. At about the same time, ThingLab (Borning 1977, 1979, 1981) implemented a classless object system atop Smalltalk, and introduced the term "prototype", as part of a similar project to specify the constraints of an environment to simulate. Both systems were followed by many others in their respective traditions.

7.1.2 T. In 1981, Yale T Scheme (Rees and Adams 1982) included a mature variant of such an object system as part of a general-purpose programming environment. A paper was later published describing the object system (Adams and Rees 1988). The T design was notably reprised by YASOS in 1992 (Dickey 1992) and distributed with SLIB since.

T's object system is already essentially equivalent to the initial design in our section 1, though not exactly isomorphic. The major differences stem from T's heavy reliance on mutation whereas we aim at purity: T was running on resource-limited platforms by today's standards, and mutation was naturally ubiquitous, with explicit state management, and no attempt at implicitly caching effective method values.

There are minor differences between our model and T's. Importantly, T lives by the slogan that "closures are a poor man's objects" and "objects are a poor man's closures": T objects are callable functions that have extra named entry points; in our model we can add a special method for the default entry point, but we need language support to hook the regular function call syntax into that method. As for nomenclature, what we call "instance", T calls "object" or "instance", and instead of "prototypes" it has "components" with a slightly different API. Also, T assumes methods are all function-valued, to be immediately called when looked up, as if all accesses to an object in our model went through the `operate` function below. Methods in T can be directly represented as slots with a function value in our approach. Non-function-valued slots in our approach above can be represented in T by nullary methods returning a constant value.

```
(define (operate instance selector . args) (apply (instance selector) args))
```

7.1.3 Prototypes become mainstream. In 1986, several publications (Borning 1986; Lieberman 1986) popularized the term and concept of prototypes, and to a point that of "delegation" for the variant of inheritance associated with them.

That year also appeared SELF, a language in the Smalltalk tradition but with prototypes instead of classes. SELF was influential, notably thanks to its complete graphical environment that could run on SparcStations, then widespread in universities and research centers. However, SELF could only usably run on high-end workstations, despite many innovations to make it efficient (Chambers et al. 1989). The optimization effort and ubiquitous mutation including of the inheritance hierarchy contributed to making its semantic model complex.

JavaScript came out in 1994, with a prototype object system (Guha et al. 2015; International 2015), and brought prototype object orientation to the masses.

All the above efforts happened in a stateful rather than pure context, optimizing for efficient implementation where memory and computation are expensive. None of them sought to establish a general model for all OOP, what is more with simple semantics.

7.2 The Pure Functional Tradition

7.2.1 Denotational and Operational Semantics. Reddy (Reddy 1988) used objects as closures to offer denotational semantics for Smalltalk. Cook (Cook 1989) independently took the same approach, but made it more general. Cook uses the same model as in our section 1, just restricted to records (unlike our section 3), and with single inheritance only, no composition. Cook then shows how to adapt this model to describe many class-based languages of his day. He notably factors the record extension mechanism as a parameter of his protocol. Cook doesn't address prototype systems, and mentions but doesn't adequately tackle multiple inheritance and other advanced features from Flavors.

Bracha first documented and implemented prototype composition, which he called mixin composition (Bracha and Cook 1990), then in his thesis (Bracha 1992) generalized the concept and vastly expanded on it. He notably explained how to express single and multiple inheritance in terms of this composition, and relating it to both previous class and prototype OOP. A lot of the concepts in the present paper are explicitly present or a least latent in Bracha's work, though we arrange them in slightly different ways: Bracha has more of a deconstructive approach, breaking down the elementary mechanisms with which to reconstruct any past and future system of modules,

objects or first-class environments; our paper has a more constructive presentation of the same ideas, emphasizing the simplicity of minimal definitions of object systems on top of FP.

7.2.2 GCL, Jsonnet, Nix. Jsonnet (Cunningham 2014) in 2014 was the first publicly available language with prototype objects in the context of a pure lazy functional language with dynamic types. It was also the first widely-used language (excluding Bracha's experiments above) that made prototype composition the primary algebraic operation on objects: the model of our section 1, restricted to records extensions. Jsonnet objects also combine in a same entity the two aspects, instance and prototype, as in our section 4.2. Note that Jsonnet uses the empty object as the implicit base super object for inheritance; this is equivalent to using the bottom function in the representation from our section 1, but not in theirs! Jsonnet also has field introspection, and flags fields as either visible or hidden for the sake of exporting JSON. Jsonnet itself started as a simplified reconstruction and cleanup of GCL, the decade-older and reputedly clunky Google Configuration Language, which remains unpublished.

Nix (Dolstra and Löh 2008), the pure lazy dynamic functional configuration language for NixOS, has several variations of an "extension system", all of them essentially equivalent to Jsonnet's object system (minus field visibility flagging), except done as a handful of user-defined functions, rather than as builtin primitives. Peter Simons wrote the initial one in 2015 to support for multiple versions of the GHC ecosystem. However, neither the code nor its documentation describe to these systems as object systems, though each has a small comment about object-orientation; furthermore, while prototype composition is implemented, the codebase tends to stick to single inheritance.

Jsonnet and Nix illustrate how prototype objects are a great fit to simply express flexible configurations. Pure functional programmers may have unjustly neglected prototypes until recently because of the limitations of their languages' type systems. These two object systems in turn inspired the present authors to use, study and improve prototype object systems.

No academic publication was made on any of GCL, Jsonnet or Nix, and it is unclear which of their design elements were influenced by which previous systems, and which were original inventions or independent reconstructions. Still, we suspect that the influence of previous prototype systems on these three languages was indirect and weak. If this suspicion is correct, then the fact that, across several decades, closely matching designs were independently reinvented many times without explicit intent to do so, is a good sign that this design is a "fixed point in design space", akin to the notion of "fixed point in time" in Dr Who (Unknown 2021): however you may try to reach a different conclusion, you still end-up with that fixed-point eventually.

7.2.3 Types for Objects. Object Systems present many challenges to authors of static type systems (Abadi and Cardelli 1997; Pierce 2002). Proper treatment of common usage requires existential types to deal with fixed-points, subtypes for ad hoc polymorphism, covariance or contravariance to commute subtypes with fixed-points, row polymorphism to support records, etc. One logical feature too many and the type system will become undecidable, or worse, inconsistent.

Bruce et al. (Bruce et al. 2006) summarize one basic challenge in encoding objects. Their first model, OR, fits our section 1. Their other three, OE, OBE, ORE, do not apply to prototype-based OOP, but are tailored to class-based OOP. OE and ORBE naturally emerge when using our reduction of class-based OOP to prototype-based OOP, OE as our "typeclass" representation, ORBE as our "class" representation.

Most object typing papers, like the one above, focus on class-based OOP and generators only, because it makes for much simpler types. Prototypes and wrappers are reduced away at the type-level at compile-time, distinction between instance and prototype is static, etc. Handling full prototype OOP would require dependent types or some notion of staged computations.

7.3 Other Relevant Work

7.3.1 The Lisp tradition. The Lisp tradition includes many object systems that were class-based but innovated in other ways: Flavors (Cannon 1982) brought multiple inheritance, mixins, extensible method combinations, default initialization values, etc. CommonLoops (Bobrow et al. 1986) brought generic functions and multiple dispatch, meta-object protocols, etc. CLOS (Bobrow et al. 1988) standardized it all. Dylan brought sealing and the C3 linearization algorithm (Barrett et al. 1996). Some of these innovations have been individually copied here and there, but by and large, CLOS remains decades ahead of object systems outside the Lisp tradition in most ways but static typing.

7.3.2 The Haskell tradition. The limited type system of Haskell does not support OOP for first-class entities. Yet Haskell popularized a form of second-class *ad hoc* polymorphism in the form of typeclasses (Peterson and Jones 1993). Despite these limitations, Oliveira (Oliveira 2009) uses essentially our section 1 construction to amazing effects. Remarkably, he doesn't try at all to model records, which wouldn't work, but instead uses prototype wrappers (that he calls "mixins") for arbitrary function types, as in our section 3. He then leverages typeclass constraints on the context to incrementally specify the *means* of a computation, even though the monomorphic type limitation won't allow incremental specification of a computation's *results*. Oliveira explicitly relates his work to Cook's (Cook 1989), yet importantly identifies composition rather than application as the interesting algebraic structure.

8 FUTURE WORK

In the future, we would like to explore the Generalized Prototypes mentioned in Appendix A to also implement multiple dispatch and method combination. Multiple dispatch raises interesting questions regarding the semantics of extending existing objects. Method combination meanwhile may require attaching meta-data to method prototypes regarding how they are to be combined in the end, which means we will have to explore what insights our approach may bring into Meta-Object Protocols (Kiczales et al. 1991).

Another important kind of meta-data we may want to attach to prototypes is type information and/or logical constraints, that may be enforced either at runtime or at compile-time. For a fully formal treatment, we would have to formalize such type information in a dependent type system. While Church-style systems like Coq, Lean, Agda, or Idris are more popular, we suspect that a Curry-style system like Cedille might be more appropriate to seamlessly extend our untyped approach into a dependently typed one.

Finally, we may want to study how prototype OOP interacts with staged computations or cache invalidation so as to require less-than-dependent types, or enable various optimizations at compile-time especially with respect to object representation.

BIBLIOGRAPHY

Martín Abadi and Luca Cardelli. *A theory of objects*. 1997.

Norman Adams and Jonathan Rees. Object-Oriented Programming in Scheme. In *Proc. ACM Conference on Lisp and Functional Programming*, 1988.

Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A Monotonic Superclass Linearization for Dylan. 1996.

Paul Blain Levy. Call-by-Push-Value: A Subsuming Paradigm. In *Proc. Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, 1999.

D. G. Bobrow, L. D. DeMichiel, R. P. Gabriel, S. E. Kleene, G. Kiczales, and D. A. Moon. Common Lisp Object Specification X3J13. *SIGPLAN Notices 23 (Special Issue)*, 1988.

Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefyk, and Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. *OOPSLA*, 1986.

Alan Hamilton Borning. ThingLab— an Object-Oriented System for Building Simulations using Constraints. In *Proc. 5th International Conference on Artificial Intelligence*, 1977.

Alan Hamilton Borning. ThingLab— A Constraint-Oriented Simulation Laboratory. PhD dissertation, Stanford University, 1979.

Alan Hamilton Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Trans. Program. Lang. Syst.* 3(4), pp. 353–387, 1981. https://doi.org/10.1145/357146.357147

Alan Hamilton Borning. Classes Versus Prototypes in Object-Oriented Languages. In *Proc. 1986 Fall Joint Computer Conference*, 1986.

Gilad Bracha. The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. PhD dissertation, University of Utah, 1992.

Gilad Bracha and William Cook. Mixin-Based Inheritance. In *Proc. Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, 1990.

Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing Object Encodings. In *Proc. Journal of Functional Programming*, 2006.

Howard Cannon. Flavors: A non-hierarchical approach to object-oriented programming. 1982.

C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proc. Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, 1989.

Craig Chambers. Object-oriented multi-methods in Cecil. In *Proc. Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, 1992.

William R. Cook. A Denotational Semantics of Inheritance. PhD dissertation, Brown University, 1989.

Dave Cunningham. Jsonnet. 2014. https://jsonnet.org

Ken Dickey. Scheming with objects. *AI Expert* 7(10), pp. 24–33, 1992.

Eelco Dolstra and Andres Löh. NixOS: A purely functional Linux distribution. In *Proc. Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, 2008.

Matthias Felleisen. On the Expressive Power of Programming Languages. Rice University, 1991.

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Proc. In Asian Symposium on Programming Languages and Systems (APLAS) 2006*, 2006.

Nate Foster, M. Greenwald, J. T. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 2007.

Richard P Gabriel, Jon L White, and Daniel G Bobrow. CLOS: Integrating object-oriented and functional programming. *Communications of the ACM* 34(9), pp. 29–38, 1991.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. *CoRR* abs/1510.00925, 2015.

Ecma International. *ECMAScript 2015 Language Specification*. 2015.

Kenneth Michael Kahn. An Actor-Based Computer Animation Language. In *Proc. Proceedings of the ACM/SIGGRAPH Workshop on User-Oriented Design of Interactive Graphics Systems*, 1976. https://doi.org/10.1145/1024273.1024278

Kenneth Michael Kahn. Creation of computer animation from story descriptions. PhD dissertation, MIT, 1979.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming (ECOOP)*, 1997.

Gregor Kiczales, Jim Des Rivières, and Daniel Gureasko Bobrow. *The Art of the Meta-Object Protocol*. 1991.

Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proc. OOPLSA*, 1986.

Bruno C. d. S. Oliveira. The Different Aspects of Monads and Mixins. 2009.

John Peterson and Mark Jones. Implementing Type Classes. In *Proc. Proceedings of the 2000 ACM SIGPLAN Haskell Workshop, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science*, 1993.

Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor Optics: Modular Data Accessors. *The Art, Science, and Engineering of Programming* 1(2), 2017.

Benjamin C. Pierce. *Types and Programming Languages*. 2002.

Uday Reddy. Objects as Closures - Abstract Semantics of Object Oriented Languages. In *Proc. ACM Symposium on LISP and Functional Programming*, 1988.

Jonathan A. Rees and Norman I. Adams. T: a dialect of LISP or, Lambda: the ultimate software tool. In *Proc. Symposium on Lisp and Functional Programming, ACM*, 1982.

François-René Rideau. LIL: CLOS Reaches Higher-Order, Sheds Identity and has a Transformative Experience. In *Proc. International Lisp Conference*, 2012.

François-René Rideau. Gerbil-POO. 2020. https://github.com/fare/gerbil-poo

Lee Salzman and Jonathan Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In *Proc. ECOOP 2005, Proceedings, LNCS*, 2005.

Christopher Strachey. Fundamental Concepts in Programming Languages. 1967.

Unknown. Fixed Point in Time. 2021. https://tardis.fandom.com/wiki/Fixed_point_in_time

Wikipedia. C3 linearization. 2021. https://en.wikipedia.org/wiki/C3_linearization

A ADVANCED OOP

A.1 Multiple Dispatch

A.1.1 Generic Functions. Some object systems, starting with CommonLoops (Bobrow et al. 1986) then CLOS (Bobrow et al. 1988), feature the ability to define "generic functions" whose behavior can be specialized on each of multiple arguments. For instance, a generic multiplication operation will invoke different methods when called with two integers, two complex numbers, an integer and a floating-point number, a complex number and a vector of complex numbers, etc.

Selection of behavior based on multiple arguments is called "multiple dispatch", as contrasted with "single dispatch" which is selection of behavior based on a single argument. Methods that may specialize on multiple arguments are sometimes called "multi-methods" to distinguish them from single dispatch methods. It is possible to macro-expand multiple dispatch into single dispatch, by chaining dispatch of the first argument into a collection of functions that each dispatch on the second argument, and so on. But the process is tedious and non-local, and better left to be handled by an automated implementation of multiple dispatch.

Since this feature is rather involved yet was already implemented in previous prototype systems (Chambers 1992; Salzman and Aldrich 2005), we'll restrict our discussion to additional challenges presented in a pure or mostly pure functional setting.

A.1.2 Extending previous objects. Generic functions and their multi-methods are associated to multiple objects, thus they cannot be defined as part of the definition of a single object; these methods, and the generic function that they are part of, are necessarily defined outside of (some) objects. Therefore, language designers and implementers must resolve a first issue before they may implement generic functions: adding "multi-methods" to existing functions or objects after they have been initially defined. Indeed, generic functions when introduced, involve adding new methods that specialize the behavior of the new function on existing objects; and when new objects

are introduced, they often involve adding new methods to specialized the behavior of the existing functions on the new objects.

Moreover, when generic functions and objects are introduced in independent code libraries (e.g. some graphical user-interface library and some data structure library), the specialized behavior might be introduced in yet another library (e.g. that defines a graphical interface for this data structure). Worse, there might be conflicts between several such libraries. Even without conflicting definitions, a definition might conflict in time with the absence of the same definition, if there is any chance that some code depending on its presence is compiled or run both before and after the definition was evaluated.

That is why, for instance, the Glasgow Haskell Compiler issues warnings if you declare an "orphan instance": a typeclass instance (rough analogue to methods of generic functions) in a file other than one where either the typeclass is defined (analogous to generic functions) or the type is defined (analogous to objects). The language offers weak guarantees in such situations. One way to avoid "orphan" situations might be to declare either new typeclasses (new generic functions) or newtype aliases (new objects) that will shadow or replace the previous ones in the rest of the program; but this is a non-local and non-composable transformation that potentially involves wrapping over all the transitive dependencies of an application, and defeat the purpose of incremental program specification. Another approach suitable in a more dynamic language would be to maintain at runtime a "negative" cache of methods previously assumed to be absent, and issue a warning or error when a new method is introduced that conflicts with such an assumption.

Then again, a solution might be to eschew purity and embrace side-effects: the original T just did not cache method invocation results, and re-ran fixed-point computations, with their possible side-effects, at every method invocation (objects can specify their own explicit cache when desired). The behavior of new generic functions on previously existing objects would be optionally specified in a default method, to be called in lieu of raising a "method not found" error. The T object paper (Adams and Rees 1988) mentions an alternate approach that was rejected in its implementation, though it is essentially equivalent in behavior, wherein default methods are added to a "default" object used as the base super value instead of an empty object when (re)computing instance fixed-points.

A.2 Method Combinations

A.2.1 Combining Method Fragments. With method combination, as pioneered in Flavors (Cannon 1982), then standardized by CLOS (Bobrow et al. 1988) via (Bobrow et al. 1986) and made slightly popular outside the Lisp tradition by Aspect-Oriented Programming (Kiczales et al. 1997), object methods to be specified in multiple fragments that can be subsequently combined into the "effective method" that will be called.

Thus, in CLOS, "primary" methods are composed the usual way, but "before" methods are executed beforehand for side-effects from most-specific to least-specific, and "after" methods are executed afterwards for side-effects from least-specific to most-specific, and "around" methods wrap all the above, with from most-specific wrapper outermost to least-specific inner-most.

Furthermore, programmers may override the above standard "method combinations" with alternative method combinations, either builtin or user-specified. The builtin method combinations, `progn + list nconc and max or append min` in CLOS, behave as if the calls to the (suitably sorted) methods had been wrapped in one of the corresponding Lisp special form or function. User-specified method combinations allow for arbitrary behavior based on an arbitrary set of sorted, labelled (multi)methods (though, in CLOS, with a fixed finite set of labels).

A.2.2 Generalized Prototype Combination. As determined above, generic functions are made of many labelled fragments (multi-methods); fragments of a same label are sorted according to some

partial or total order (and, in the case of multiple dispatch, filtered for applicability); they are then composed via some representation-dependent mixing function; a fixed-point is extracted from the composed fragments, with some base value; finally a representation-dependent wrapper is applied to the fixed-point to instantiate the effective method… this is very much like an object! Actually, since we have accepted in section 3 that prototypes and "object orientation" are not just to compute records that map slot names to values, but for arbitrary computations, then we realize there is a more general protocol for computing with prototypes.

A full treatment of generalized prototypes is a topic for future work. Still, here are a few hints as to what they would be. A generalized prototype would involve: (a) a *lens* (Foster et al. 2007; Pickering et al. 2017) that can extract or update the method from the current partial computation of the raw prototype fixed-point; (b) an object-setter that "cooks" the raw prototype fixed-point into a referenceable `self` object; (c) a method-wrapper that turns the user-provided "method" into a composable prototype.

The lens, passed below as two function arguments `getter` and `setter`, generalizes the fetching or storing of a method as the entry in a `Dict`, or as the response to a message; it expresses how a prototype overrides some specific fragment of some specific method in some specific sub-sub-object, encoded in some specific way. The object-setter may apply a method combination to extract a function from fragments; it may transcode an object from a representation suitable for object production to a representation suitable for object consumption; it may be the setter from a lens making the prototype-based computation that of a narrow component in a wider computation, at which point the corresponding getter allows a method to retrieve the computation at hand from the overall computation; it may be a composition of some of the above and more. The method-wrapper may to automate some of the builtin method combinations; for instance, in a `+` combination, it would, given an number, return the prototype that increments the super result by that number; it may also handle the merging of a `Dict` override returned by the method into the super `Dict` provided by its super method; etc.

Here then is a generalization that subsumes the above `$slot-gen` or `$slot/gen`:

```
(define ($lens-gen setter getter wrapper method)
  (λ (cooked-self raw-super)
    (setter ((wrapper method) cooked-self (delay (getter raw-super)))
            raw-super)))
```

B CODE LIBRARY

We have used Racket to develop this document in such a way that the very same file is used as source file for a reusable Racket library, a test module, and the printable document. The code is available under the Apache License, version 2.0, and adapting it to run on any Scheme implementation should take minimal effort. Alternatively, you could use the full-fledged library we built on the same general model in another Scheme dialect (Rideau 2020): it features many practical optimizations and syntactic abstractions for enhanced usability, as well as a extensive support for type descriptors.

In this appendix, we include library code of relatively obvious or well-known functions, that provide no original insight, yet that we rely upon in the implementation of the object system. This code is included for the sake of completeness and reproducibility. It is also required to get this file running, though we could have kept the code hidden.

B.1 C3 Linearization Algorithm

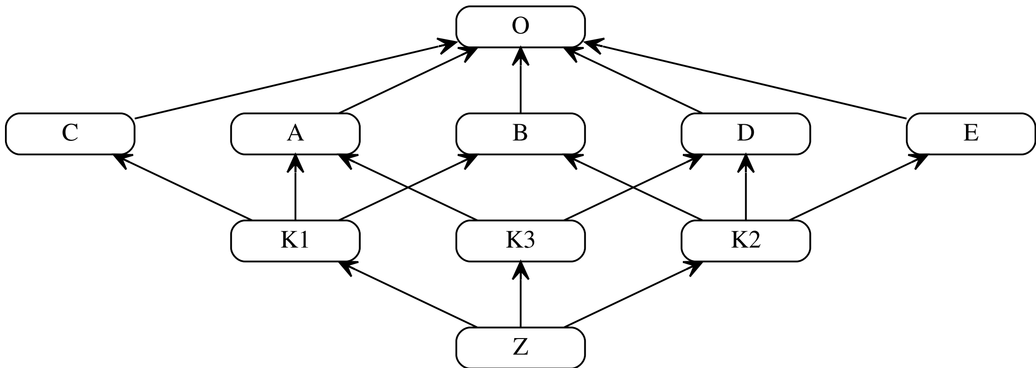
Below is the C3 Linearization algorithm to topologically sort an inheritance DAG into a precedence list such that direct supers are all included before indirect supers. Initially introduced in Dylan (Barrett et al. 1996), it has since been adopted by many modern languages, including Python, Raku, Parrot, Solidity, PGF/TikZ.

The algorithm ensures that the precedence list of an object always contains as ordered sub-lists (though not necessarily with consecutive elements) the precedence list of each of the object's super-objects, as well as the list of direct supers. It also favors direct supers appearing as early as possible in the precedence list.

We import the standard library `srfi/1` for the sake of the standard functions `every` and `filter`.

```
; The (require srfi/1) below imports SRFI 1 list functions into Racket
; YMMV if you use another Scheme implementation
(require srfi/1)

; not-null? : Any -> Bool
(define (not-null? l) (not (null? l)))

; remove-nulls : (List (List X)) -> (List (NonEmptyList X))
(define (remove-nulls l) (filter not-null? l))

; remove-next : X (List (NonEmptyList X)) -> (List (NonEmptyList X))
(define (remove-next next tails)
  (remove-nulls (map (λ (l) (if (equal? (car l) next) (cdr l) l)) tails)))

; c3-compute-precedence-list : A (A -> (List A)) (A -> (NonEmptyList A))
;  -> (NonEmptyList A)
(define (c3-compute-precedence-list x get-supers get-precedence-list)
  (define supers (get-supers x))
  (define super-precedence-lists (map get-precedence-list supers))
  (define (c3-select-next tails)
    (define (candidate? c) (every (λ (tail) (not (member c (cdr tail)))) tails))
    (let loop ((ts tails))
      (when (null? ts) (error "Inconsistent precedence graph"))
      (define c (caar ts))
      (if (candidate? c) c (loop (cdr ts)))))
  (let loop ((rhead (list x))
             (tails (remove-nulls (append super-precedence-lists (list supers)))))
    (cond ((null? tails) (reverse rhead))
          ((null? (cdr tails)) (append-reverse rhead (car tails)))
          (else (let ((next (c3-select-next tails)))
                  (loop (cons next rhead) (remove-next next tails)))))))
```

We will now test this linearization algorithm on the inheritance graph from section 4.3. This test case was originally taken from the Wikipedia article on C3 linearization (Wikipedia 2021), that usefully includes the following diagram (that nevertheless fails to represent the ordering constraints imposed on A, B, C, D, E by their order of appearance in the supers list of K1, K2, K3):



The definitions for this test case are as follows:

```
(define test-inheritance-dag-alist
  '((O) (A O) (B O) (C O) (D O) (E O)
    (K1 A B C) (K2 D B E) (K3 D A) (Z K1 K2 K3)))
(define (test-get-supers x) (cdr (assoc x test-inheritance-dag-alist)))
(define (test-compute-precedence-list x)
  (c3-compute-precedence-list x test-get-supers test-compute-precedence-list))
```

And the test are thus:

```
> (map not-null? '(() (1) (a b c) nil))
'(#f #t #t #t)
> (remove-nulls '((a b c) () (d e) () (f) () ()))
'((a b c) (d e) (f))
> (map test-compute-precedence-list '(O A K1 Z))
'((O) (A O) (K1 A B C O) (Z K1 K2 K3 D A B C E O))
```

B.2 More Helper Functions

To help with defining multiple inheritance, we'll also define the following helper functions:

```
(define (alist->Dict alist)
  (foldl (λ (kv a) ((Dict 'acons) (car kv) (cdr kv) a)) (Dict 'empty) alist))

(define (Dict->alist dict)
  ((Dict 'afoldr) (λ (k v a) (cons (cons k v) a)) '() dict))

(define (Dict-merge override-dict base-dict)
  ((Dict 'afoldr) (Dict 'acons) base-dict override-dict))
```

B.3 Extra tests

Here are some tests to check that our functions are working.

```
(define test-instance
  (alist->Dict `((a . ,(delay 1)) (b . ,(delay 2)) (c . ,(delay 3)))))
(define test-prototype
  (alist->Dict `((function . ,(delay (λ (self super)
                                 (Dict-merge (force test-instance) (force super)))))
                 (supers . ,(delay '()))
```

```
                  (precedence-list . ,(delay '()))))))
(define test-object (make-object test-instance test-prototype))
(define test-p1 ($slot/value 'foo 1))
```

```
> (map (λ (x) (slot-ref test-object x)) '(a b c))
'(1 2 3)
> (slot-ref (instantiate '() test-p1) 'foo)
1
> (map (slot-ref (ListOf Number) 'is?) '(() (1 2 3) (1 a 2) (1 . 2)))
'(#t #t #f #f)
```

C DIGRESSION ABOUT TYPE NOTATION

There is no standard type system for the language Scheme, so we will keep our type declarations as semi-formal comments that are unenforced by the compiler, yet that document what the types for our functions would be in a hypothetical dependent type system (with possible effect extensions?) that would be powerful enough to describe our computations.

C.1 Variables and Type Variables

We will use lowercase letters, such as `a`, `f`, `x`, to denote variables, that may be bound any value in the language. We will use uppercase letters, such as `A`, `F`, `X`, to denote *type variables*. That is, variables representing a type, that is not currently specified, but such that the formulas we write must hold for any type, in a hypothetical type system that one could layer on top of the language.

As common in programming languages such as ML and Haskell, we will assume that each top-level function type declaration introduces its own scope, wherein free variables are implicitly universally quantified.

C.2 Variable Rows and Type Variable Rows

We will write a `...` for a "row" of multiple values, such as may be used as input arguments to a function, or return values of a function. We will write A `...` for a "row" of multiple types, such as may be used to type the inputs or outputs of a function. Indeed, in Scheme, a function may take multiple inputs arguments and and return multiple output values.

For instance, a row of types could be:

```
Integer
```

for the single type of integers, or:

```
String
```

for the single type of strings, or:

```
Integer String
```

for the two types (in order) `Integer` and `String`, or:

```
Integer Symbol ...
```

for the types of one integer followed by zero or many symbols.

C.3 Function Types

The type of functions that take inputs `I ...` and return outputs `O ...` we will write as any one of the following:

```
(Fun I ... -> O ...)
```

As usual, the arrows are associative such that these denote the same type:

```
(Fun A -> B -> C)
(Fun A -> (Fun B -> C))
```

Note that in this paper we follow the common convention of left-to-right arrows, as above. We also follow this convention when contributing to other people's code bases that follow it. However, in our own code base, we favor an opposite convention of right-to-left arrows, that makes them covariant with the right-to-left flow of information from arguments to function in the traditional prefix notation for function application. Still, we revert to left-to-right arrows when we use concatenative languages or stack virtual machines that use the "Reverse Polish Notation" as in FORTH, PostScript or the much missed HP RPL.

C.4 Type Constraints

We will use the keyword `st:` (being a short form for "such that") to denote type constraints, as in:

```
st: Constraint1 Constraint2 ...
```

The constraints we will consider will be subtyping constraints of the form:

```
(<: A B C ...)
```

meaning `A` is a subtype of `B`, which is a subtype of `C`, etc.

Thus, the type `(Fun Self Super -> Self st: (<: Self Super))` means "for all types `Self` and `Super` such that `Self` is a subtype of `Super`, a function of two arguments of respective types `Self` and `Super`, returning a value of type `Self`", the universal quantification being implicit, at the top-level declaration.

D FIXED-POINT FUNCTIONS

In section 1, we quickly went over the fixed-point function `fix` that we used to instantiate prototypes.

Reminder: A function prototype `(Proto A B)` is function `p : (Fun A B -> A)` where `A` and `B` are function types, and `A` is a subtype of `B`. Thus, `p` takes a function `f` of type `A`, and a function `B` of type `B`, and returns a new function `g` of same type `A`.

To instantiate a prototype is get a function of type `A` from a function of type `B`. Given these premises, there is one and only one construction that allows us to and only one way to get an `A`: it's by calling `p`. But to call `p`, we need to already have an `A`! Where do we get this function of type `A` to begin with? That's where the magic of fixed-point functions comes in: they will somehow *tie a knot*, and get a reference to the function being defined, even before the function is defined.

Here is how we want a fixed point to look like:

```
(define (overly-simple-fix p b)
  (define f (p f b))
  f)
```

Unhappily, this does not work in Scheme, because Scheme is eager: the call to `p` needs to fully evaluate its arguments, but `f` has not been fully defined yet, so the call is invalid. Some Scheme implementations may detect that this definition tries to use `f` before it is defined and raise an error at compile-time. Some implementations will initially bind `f` to a magic "unbound" marker, and trying to use `f` before it is defined will result in an error at runtime. In yet other implementations, `f` will initially be bound to some default "null" value such as `#f` or `(void)` that will be used without immediately raising an error—until you try to call `f` while expecting it to be a function, and then the implementation will raise a runtime error while you are left wondering why the program is

trying to call this null value. Finally, some reckless implementations may try to use f before the data frame was even properly initialized at all, and some random low-level value is used that might not make sense with respect to the garbage collector, and you'll eventually dance fandango on core.

Yet, in a lazy language, the above definition works! Indeed in Nix, you can write the equivalent definition:

```
let fix = p: b: let f = p f b; in f
```

In Scheme, we can similarly write:

```
(define (delayed-fix p b)
  (define f (delay (p f b)))
  f)
```

But in this only works if p accepts delayed computations as arguments rather than direct function values (and still eagerly computes the result from them). Then we have will have:

```
; (deftype (DelayedProto A B) (Fun (Delayed A) (Delayed B) -> A))
; delayed-fix : (Fun (DelayedProto A B) (Delayed B) -> (Delayed A))
```

On the other hand, this works for arbitrary types A and B, and not just for function types!

So, how do we get around this issue without delay? One solution would be as follows—can you tell why it works, and why it is not fully satisfactory?

```
(define (fix--0 p b)
  (define (f . i) (apply (p f b) i))
  f)
```

First, why it works: by making f a function, we can recursively refer to f from within the body of function f itself, since by the time this reference is used, f was called, and by the time f was called, f was defined. Thus we can give f to p to compute the fixed-point value (p f b). But by that time, we're trying to call the fixed point, so we take all the input arguments passed to f in a list i, and we pass them all forward to the fixed-point expression using the builtin function apply. All is well. Try it, it works.

However, there is an issue: fix--0 calls p again at every invocation of the fixed-point f. Therefore, if p makes expensive computations, it will pay to recompute them every time from scratch. Worse, if p wants to build data structures meant to be shared between invocations, such as a cache, this sharing will be lost between calls to f. There can be no sharing of information between calls to f. No pre-computation, no cacheing, no memoization, no shared mutable state.

Therefore a better solution, that does allow for sharing computations and state between invocations of the fixed-point result, is:

```
(define (fix--1 p b)
  (define f (p (λ i (apply f i)) b))
  f)
```

That's the same as the fix function from section 1. Note how the anonymous lambda closure does part of the "protection" or "delay" whereby the recursive data structure will only be called after f is defined, but relies on p not causing its first argument to be called during its evaluation, only stored in a data structure or in a closure to be called later after p has returned.

If you do not like internal defines, you can write the same function equivalently using letrec, as:

```
(define (fix--2 p b)
  (letrec ((f (p (λ i (apply f i)) b)))
```

```
    f))
```

And if you do not even like `letrec`, you can use a Y-combinator variant:

```
(define (fix--3 p b)
  ((λ (yf) (yf yf)) (λ (yf) (p (λ i (apply (yf yf) i)) b))))
```

In practice, a lazy variant such as the one using `delay` above will be the most usable as well as the most general, though laziness is not colloquial in the applicative language Scheme.

E NOTE FOR CODE MINIMALISTS

In our introduction, we described the `fix` and `mix` functions in only 38 `cons` cells or 109 characters of Scheme (counting one extra `cons` cell per top-level form, but no `cons` cell for the implicit `begin`). We can do even shorter with various extensions. MIT Scheme and after it Racket, Gerbil Scheme, and more, allow you to write curried function definitions, to cut 2 `cons` cells and 9 characters.

```
(define ((mix p q) f b) (p f (q f b)))
```

And then we'd have Object Orientation in only 36 `cons` cells, 100 characters.

Then again, in Gerbil Scheme, we could get it down to only 34 `cons` cells, and 88 characters, counting newline:

```
(def (fix p b) (def f (p (cut apply f <>) b)) f)
```

Or, compressing spaces, to 24 `cons` cells and 73 characters, not counting newline, since we elide unnecessary spaces:

```
(def(fix p b)(def f(p(cut apply f <>)b))f)(def((mix p q)f b)(p f(q f b)))
```

Science in general, and Computer Science in particular, is about conceptual minimalism. While algorithmic complexity is defined only up to an additive constant determined up to a choice of a base computing system, simple variants of the λ-calculus provide sensible conventional such bases, and the pure subset of Scheme we use is one of them. The above "code golf" then suggests that our proposed approach to formalizing Object-Oriented Programming indeed exhibits simple foundational principles.