The Essence of Object-Orientation: Modularity and Incrementality, or: Lambda, the Ultimate Object

François-René Rideau

Mutual Knowledge Systems, Inc.
`fare@mukn.com`

Abstract. We argue that the essence of Object-Orientation (OO) is a mechanism for reified (in-language) Incremental Modularity: we do it by first making a semi-formal problem statement, then identifying the simplest solution from first principles, thereby reconstructing the basic concepts of OO on top of the pure λ-calculus. We discuss how various features of OO languages can facilitate or hinder this Incremental Modularity, from forms of inheritance, to classes themselves, to mutation. Our exploration yields answers that sometimes coincide with prevalent academic discourse or industrial practice, but sometimes goes against one or both.

1 The Essence of OO

1.1 OO in 2 lines of FP

Our previous paper (Rideau et al. 2021) shows how to reduce Object-Orientation (OO) to a few lines of Functional Programming (FP). Its kernel consists of just two one-line functions—reprised and detailed in section 3.2. These functions slightly generalize formulas known in theory for many decades (Bracha and Cook 1990), and actually used as the basis of practical implementations for many years (Simons 2015). That paper then uses and extends this technique to implement several complete Object Systems, in a few tens of lines of code each, in any language that can express higher-order functions.

1.2 OO as Incremental Modularity

In the above paper, we briefly mention how ***OO is a mechanism to specify computations in modular increments***, and how modularity justifies using multiple inheritance over mixin inheritance, or conflating of prototypes and instances (or classes and types) over keeping them separate.

In this present paper, we will elaborate on this relationship between OO and Incremental Modularity. Without presenting a complete theory of Modularity (sketched in Rideau (2016)) we introduce some semi-formal criteria for what Modularity and Incrementality mean. We can then make our previous claims about OO and Modularity more explicit and less informal.

1.3 Claims

The present paper claim the following original contributions:

– Dispel common misconceptions as to what OO is about (section 1.4).
– Propose criteria for Modularity (section 2.1) and Incrementality (section 2.2) in terms of information needed to make software modifications.
– Elucidate how Incrementality and Modularity go together (section 2.3).
– Map the basic concepts of OO to modularity and incrementality (section 3.1), as embodied in the simplest kind of OO Prototypes using mixin inheritance (section 3.2).
– Explain how single inheritance is less expressive and modular than mixin inheritance (section 4.2), that is less so than multiple inheritance (section 4.3).
– Show how "structs" with the performance benefits of single-inheritance can be expressed in a system with multiple-inheritance (section 4.3.6).
– Discuss how purity and laziness make OO more modular, and solve difficult initialization order issues (section 5.1).
– Discuss how purity and laziness make OO more modular, and solve difficult initialization order issues (section 5.1).
– Expose the conflation between prototypes and instances (or classes and types) at the heart of most OO, and why it contributes to modularity (section 5.3).
– Clarify the relationship between Prototype OO and Class OO, and why Prototypes, being first-class, enable more modularity (section 6).
– Generalize OO methods from fixed slots to functional lenses, very simply enable modular features like method combinations (section 7.1.5).
– Show how the "typeclass" approach can be more composable and thus more modular than the "class" approach (section 7.1.3).
– Provide a pure functional modular solution to issues with multiple dispatch vs single dispatch, friend classes or mutually recursive classes, by making library namespace management an explicit part of the language (section 7.1.7).

Many of the concepts and relationships we tackle have long been part of OO practice and lore, yet have been largely neglected in scientific literature and formalization attempts.

1.4 What OO is *not* about

We make the bold claim that the essence of OO is Incremental Modularity. Yet, many other slogans or concepts have been claimed to be essential to OO in the past. We can summarily dismiss those claims as follows:

Classes Many think that classes, as introduced by Simula 67 (Dahl et al. 1968) (though implementing a concept previously named by Hoare (Hoare 1965)), are essential to OO, and only ever care to implement, use, formalize, study, teach or propagandize class-based OO (a.k.a. Class OO).

Yet the existence since 1976 (Adams and Rees 1988; Borning 1977, 1979, 1981; Kahn 1976, 1979; Rees and Adams 1982) of languages using class-less prototype-based OO (a.k.a. Prototype OO) (Borning 1986; Chambers et al. 1989; Lawall and Friedman 1989; Lieberman 1986), and the fact that the single most used OO language in the world, JavaScript (GitHub 2022), uses prototypes (International 2015), provide clear counter-evidence to this belief. The original inventors of OO also later unified classes, prototypes and procedures into a general notion of "patterns" (Kristensen et al. 1987), which also voids any appeal to their authority in declaring classes as such as essential to OO.

Of course, classes are an *important* concept in OO. The situation is similar to that of types in FP, in which they are an important though not essential concept, as evidenced by the historical preexistence and continued use of the untyped λ-calculus and the wide adoption of dynamically typed functional languages like Scheme or Nix. Actually, we'll demonstrate below in section 6 how classes are a indeed special case of prototypes, and how they precisely relate to types.

Imperative Programming Many people assume that OO requires that all slots of all objects should be mutable, or be so by default, and that OO requires mutation deep in its object initialization protocols. Furthermore, they assume the same eager evaluation model for function calls and variable definitions as in every common imperative language.

Meanwhile, many have of late claimed that purity (the lack of side-effects including mutable state) is essential to FP, making it incompatible with OO. Some purists even argue that normal-order evaluation (call-by-name or call-by-need) is also essential for true FP, making it even more incompatible with OO.

Now there are many good historical reasons, having to do with speed and memory limitations at runtime as well as compile-time for which the first OO languages, as well as most languages until recently, were using state and side-effects everywhere, and an eager evaluation model, at least by default. Yet with early 1980s slogans like "objects are a poor man's closures" and "closures are a poor man's objects" (Adams and Rees 1988), the problem back then was clearly not whether OO could be done purely with functions, but whether it made practical sense to program purely with functions in general. That question that would only be slowly answered positively, in theory in the early 1990s and in practice in the mid 2000s to mid 2010s, as Haskell grew up to become a practical language.

Yet, there are (a) pure models of OO such as those of Kamin, Reddy, Cook and Bracha (Bracha and Cook 1990; Cook 1989; Kamin 1988; Reddy 1988), (b) pure lazy dynamic OO languages such as Jsonnet or Nix (Cunningham 2014; Dolstra and Löh 2008), and pure OO systems such as presented in this paper and its predecessors (Rideau et al. 2021) and (c) languages happily combining

OO and FP such as Common Lisp or Scala with plenty of libraries restricting themselves to pure functional objects only. These provide ample evidence that OO does not at all require mutation, but can be done in a pure setting, and is very compatible with FP, purity, and even with laziness and normal-order evaluation. We could even argue that Haskell typeclasses embody OO (Rideau 2012; Wadler and Blott 1989), though its designers might not wholly embrace the OO tradition.

Inheritance as opposed to Composition Some argue that the essence of OO is to choose a side in a conflict between Inheritance and Composition, wherein one has to model every possible domain in terms of inheritance, especially so where it can be preferred compared to alternatives not involving it, and even more so when such alternative involves FP and composition.

But OO and FP are just distinct concepts neither of which subsumes the other, that thus fit distinct sets of situations. It makes no sense to oppose them, especially not when we see that OO can be expressed in a few lines of FP, whereas most modern OO languages contain FP as a subset.

The argument is actually a distortion of a legitimate question of OO design, wherein one has to decide whether some aspect of a class (respectively prototype or pattern) embodied as slots or method functions, should be included directly in the class (a) by inheriting from another class defining the aspect (the class *is-a* subclass of it — inheritance of classes), or (b) indirectly by the class having a slot containing an object of that other class (the class *has-a* slot that is it — composition of classes seen as object constructor functions).

The answer of course depends on expectations about how the class will be further specialized within a static or dynamically evolving schema of data structures and algorithms. If the schema is small, static, well-understood and won't need to evolve, it doesn't really matter which technique is used to model it. But as it grows, evolves and boggles the mind, a more modular and incremental approach is more likely to enable adapting the software to a changing situation, at which point thoughtful uses of inheritance can help a lot.[1]

[1] *Is* a car a chassis (inheritance), or does it *have* a chassis while not being it (composition)? If you're writing a program that is interested in the length of objects, you may model a `car` as a `lengthy` object with a `length` slot, and a `chassis` too. Now if your program will only ever be interested but in the length of objects, you may altogether skip any object modelling: and only use numeric length values directly everywhere for all program variables. Is a car a chassis? Yes, they are both their length, which is the same number, and you may unify the two, or let your compiler's optimizer unify the two variables as they get initialized them from the same computation. Now if you know your program will evolve to get interested in the width of objects as well as their length, you might have records with length and width rather than mere numbers, and still unify a car and its chassis. But if your program eventually becomes interested in the height, weight or price of objects, you'll soon enough see that the two entities may somehow share some attributes yet be actually distinct: ultimately, both `car` and `chassis` *are* `lengthy`, but a `car` *has* a `chassis` and *is not* a `chassis`.

Encapsulation Many OO pundits claim that an essential concept in OO is "encapsulation" or "information hiding" (DeRemer and Kron 1975), though there is no consensus as to what this or these concepts mean, and no clear definition.

Inasmuch as some people identify encapsulation as the presence of specific visibility mechanisms (with some slots or methods being public, private or something in–between), we'll easily dismiss the claim that it is an essential aspect of OO by showing that many quintessential OO languages like Smalltalk or Common Lisp lack any such specific mechanism, whereas many non-OO languages possess mechanisms to achieve the same effect, in the form of modules defining but not exporting identifiers (e.g. not declaring them `extern` in C), or simply lexical scoping as present in FP (Rees 1995).

On the other hand, inasmuch as this "encapsulation" informally denotes an aspect of modularity, we'll argue that the claim of encapsulation being essential to OO partakes in our better formalized argument according to which OO is about modularity (and incrementality). See section 2.

Message Passing Alan Kay, who invented Smalltalk and coined the term "Object-Oriented Programming" notably explained (Kay 2020) that by that he originally meant a metaphor of computation through independent (concurrent, isolated) processes communicating by passing asynchronous messages. This metaphor also guided the modifications originally brought by Simula to Algol (Dahl and Nygaard 1966).

However, neither Simula, nor Smalltalk nor any claimed "OO" language actually fits that metaphor. Instead, the only commonly used language ever to fit it is Erlang (Johnson and Armstrong 2010); yet Erlang is not part of the OO tradition, and its authors have instead described its paradigm as "Concurrency-Oriented Programming". Meanwhile the theory of computation through message-passing processes was studied with various "process calculi", that are also foreign to the OO tradition, and largely unembraced by the OO community.

Moreover, many OO languages generalize and extend their method dispatch mechanism from "single dispatch" to "multiple dispatch" (Bobrow et al. 1988; Bobrow et al. 1986; Chambers 1992); their "multimethods" are attached to several objects or classes, and there is no single object, class, or single independent entity of any kind capable of either "receiving" or "sending" a message. Mechanisms like typeclasses, while not usually considered part of the OO tradition, can be seen as isomorphic to classes (Rideau 2012), yet also lack any specific object to "receive" a message.

Thus, whatever historical role the paradigm of message-passing processes may have had in inspiring the discovery of OO, it remains a completely different paradigm, with its own mostly disjoint tradition and very different concerns.

What is usually meant by OO, is a paradigm for organizing code development for modularity and reuse, with a notable focus on "inheritance" through classes or prototypes (or at times "patterns"), usually in a synchronous evaluation framework within a single thread. Whatever clear or murky correspondance between names and concepts others may use, this paradigm is what we will call OO and discuss in this article, systematically reducing it to elementary concepts.

2 Modularity and Incrementality

2.1 Modularity

Division of Labor Modularity (Dennis 1975; Parnas 1972) is the organization of software source code in order to support division of labor, dividing it into "modules" that can each be understood and worked on mostly independently from other modules.

A Meta-linguistic Feature Most modern programming languages offer *some* builtin notion of modules as "second-class" entities, entities that exist at compile-time but are not available as regular runtime values A few languages even offer a notion of modules as "first-class" entities, that can be manipulated as values at runtime.[2] But many (most?) languages offer no such notion; indeed modules are a complex and costly feature to design and implement, and few language designers and implementers will expend the necessary efforts toward it at the start of language's development.[3]

Yet modularity is foremost a *meta-linguistic* concept: even in a language that provides no support whatsoever for modules *within* the language itself (such as C), programmers will find manual and automated means to achieve and support modularity *outside* the language. They will:

– copy and paste sections of code as poor man's modules;
– automate organized concatenation of code snippets with preprocessors;
– divide code in files they can "transclude", "link" or "load" together;
– transclude "include" files in lieu of interfaces;
– orchestrate building of software with utilities such as "make";
– bundle software into "packages" they exchange and distribute online;
– create "package managers" to handle those bundles.

When for the sake of "simplicity", "elegance", or ease of development or maintenance, support for modularity is lacking within a language, this language then becomes but the kernel of a haphazard collection of tools cobbled together to palliate the weakness of this kernel. The result inevitably ends up being extremely complex, ugly, and hard to develop and maintain.

[2] In between the two, some languages offer a "reflection" API that gives some often limited runtime access to representations of the module entities. This API is often limited to introspection only or mostly; for instance, it won't normally let you call the compiler to define new modules or the linker to load them. YEt some languages support APIs to dynamically evaluate code, that can be used to define new modules; and some clever hackers find ways to call a compiler and dynamic linker, even in languages that don't otherwise provide support APIs for it.

[3] Unless they develop their language within an existing modular framework for language-oriented programming, such as Racket, from which they inherit the module system.

Criterion for Modularity ***A design is modular if it enables developers to cooperate without having to coordinate***, compared to alternative designs that enable less cooperation or require more coordination, given some goals for developers, a space of changes they may be expected to enact in the future, etc.

For instance, the object-oriented design of ASDF (Rideau and Goldman 2010) made it simple to configure, to extend, and to refactor to use algorithms in *O(n)* rather than *O(n^3)* or worse, all of it without any of the clients having to change their code. This makes it arguably more modular than its predecessor MK-DEFSYSTEM (Kantrowitz 1991) that shunned use of objects (possibly for portability reasons at the time), was notably hard to configure, and resisted several attempts to extend or refactor it.

2.2 Incrementality

Small Changes Developers quickly lose direction, motivation, support from management and buy-in from investors and customers when they do not have tangible results to show for their work. Incrementality is the ability for a system to deliver more rewards for fewer efforts, compared to alternatives. In other words, incrementality supports a short feedback loop in software development.

A Developer-Interface Feature Incrementality should be understood within a framework of what changes are or aren't "small" for a human (or AI?) developer, rather than for a fast and mindless algorithm. Otherwise, the most "incremental" design would be to have code produced by `gunzip` or some similar decompressor, that can expand a few bits of incremental change into a large amount of code.

Thus, for instance, changing some arithmetic calculations to use bignums (large variable-size integers) instead of fixnums (builtin fixed-size integers) in C demands a whole-program rewrite with a different program structure; in Java involves some changes all over though straightforward and preserving the program structure; in Lisp or Haskell requires no changes, or minimal and local. Thus with respect to this and similar kinds of change, if expected, Java has a more incremental design than C, but less than Lisp or Haskell.

There again, incrementality is usually a meta-linguistic notion, wherein changes happen as pre-syntactic operations on the source code, rather than semantic operations within the language itself. And yet, using reflection and/or considering the entire "live" interactive development environment as "the system" rather than a "dead" program in a programming language, these pre-syntactic operations can be internalized.

A Criterion for Incrementality ***A design is incremental if it enables developers to enact change through small local modifications*** compared to alternative designs that require larger (costlier) rewrites or more global modifications (or prohibit change, same as making its cost infinite).

2.3 Incremental Modularity

A Dynamic Duo Modularity and Incrementality work hand in hand: ***Modularity means you only need to know a small amount of old information to make software progress. Incrementality means you only need to contribute a small amount of new information to make software progress.*** Together they mean that a finite-brained developer can make more software progress with a modular and incremental design than with a less-modular and less-incremental design.

Reducing Costs vs Moving them Around Beware that many designs have been wrongfully argued as more modular and/or incremental based on moving code around: these myopic designs achieve modest development savings in a few modules under focus by vastly increasing the development costs left out of focus, in extra boilerplate and friction, in other modules having to adapt, inter-module glue being made harder, or module namespace curation getting more contentious.

For instance microkernels or microservices may make each "service" look smaller, but only inasmuch as the overall code has been butchered into parts between which artificial runtime barriers were added; yet each barrier added involves extra code, actually increasing the incidental complexity of the code in direct proportion to the alleged benefits, without doing anything whatsoever to address its intrinsic complexity. These misguided designs stem from the inability to think about meta-levels and distinguish between compile-time and runtime organization of code.

Incremental Modularity, Interactively Incrementality does not necessarily mean that a complex addition or refactoring can be done in a single small change; rather, code evolution can be achieved in many small changes, wherein the system can assist the developer into only having to care about a small change at a time, while the system tracks down what are all the small remaining changes necessary.

For instance, a rich static type system can often be used as a tool to guide large refactorings by dividing them in manageably small changes, making the typechecker happy one redefinition at a time after a type modification. This example also illustrates how ***Incrementality and Modularity usually happen through meta-linguistic mechanisms rather than linguistic mechanisms***, i.e. through tooling outside the language rather than expressions inside the language.

3 Prototypes

3.1 Internal Incremental Modularity

Internalized Feature Now what if modular increments of computational specifications could be embodied as linguistic expressions *within* a programming language, that could be manipulated at runtime and studied formally, rather than just as semi-formal meta-linguistic interactions?

We dub* prototype *such an embodiment of incremental modularity within a language. And to narrow the discussion down to a formal context, let's consider programming languages with a functional programming core, i.e. that contain some variant of the lambda-calculus as a fragment, either untyped or with suitably expressive types (to be determined later).

Embodying Specification To embody some concept in the functional programming core of a language that has one, you will necessarily use a function. Since the concept is the specification of a computation, the function must eventually return that specific computation as an output value, given some inputs to be determined. The type of its output is the type of the target value.

Embodying Modularity Each prototype should be able to contribute information that other modules can use while using information from other modules it depends on. In functional terms, it will be or contain a function with the former among its outputs and the latter among its input.

Now to maximize the expressiveness of this Modularity in a functional setting, a prototype specifying one aspect of a computation should be able to make (forward) references to the complete computation being specified itself, so as to pass it as argument to higher-order functions extracting information about arbitrary aspects of it. This means the prototype should be or contain a function with the computation `self` as input for self-reference, and returns as output a computation with the specified structure that uses `self` in an *open recursion* for all "self-reference" to aspects the final computation (possibly further refined, extended or overridden). That function then specifies (part of) a larger specification function of which the complete computation will be a *fixed-point*.

Embodying Incrementality Each prototype should be able to refer not only to the complete computation with all available information, but also to the partial computation with only the information specified *so far*. Thus, it may examine so-far specified aspects and use them to contribute small modifications to these existing aspects as well as new aspects based on this information.

In functional terms, the prototype function will take an additional input `super` based on which to specify a computation. Thus, to embody incremental modularity, a prototype will be or contain a prototype function of `self` and `super` returning an enriched `self`.

Prototype Primitives Prototypes of course depend on whichever primitive operations support the type of computation being specified; but those are not specific to prototypes as such. The minimal set of prototype-specific primitives follows:

– A function that given a prototype specifying a computation (and possibly some context) returns a complete computation exactly as specified, closing the open recursion; this function we call `instantiate`, or `fix` (for reasons that will soon be obvious).
– A function to compose, chain, juxtapose and/or cross-reference multiple smaller prototypes (at least two, maybe more) each specifying some aspects of a computation, return a larger prototype that contains all these combined aspects, yet ready to be further composed, keeping the recursion open; this function we call `mix`, or `inherit` (for reasons that will also soon be obvious)

3.2 Simplest prototypes

Mixin Functions The very simplest possible design for *prototypes* is thus as "mixin" functions with the following minimal type:
`Mixin self super = self ⊂ super ⇒ self → super → self`
where `self` is the type of the computation as completely specified, `super` the type of the computation as partially specified so far, `self ⊂ super ⇒` is the constraint that `self` should be a subtype of `super`, and `Mixin` is the name introduced by Cannon (Cannon 1982) and reprised and popularized by Cook and Bracha (Bracha and Cook 1990).

The mixin instantiation and inheritance primitives are as follows:

```
instantiate : Mixin instance base → base → instance
instantiate = λ mixin base ↦ Y (λ instance ↦ mixin instance base)

inherit : Mixin instance intermediate → Mixin intermediate inherited
          → Mixin instance inherited
inherit = λ child parent ↦ λ instance inherited ↦
              child instance (parent instance inherited)
```

or equivalently:

```
fix : Mixin self top → top → self
fix = λ mixin top ↦ Y (λ self ↦ mixin self top)

mix : Mixin self super → Mixin super duper → Mixin self duper
mix = λ child parent self duper ↦ child self (parent self duper)
```

Elucidating Mixin Instantiation The `instantiate` function above computes a fixed-point `instance` for a `mixin` given as extra argument a type-appropriate `base` value that serves as seed of the computation being instantiated: an empty record `{}`, a function that always fails `⊤ = λ _ ↦ ⊥`, etc. The type of `base`, a.k.a. `top`, is thus a base type for the specified computation: a supertype of the type `instance` being computed, a.k.a. `self`. In a monomorphic setting, `base` is just `instance` itself; with a rich-enough type system, it can be a “top” type for many distinct types of computations, carrying no information.

The `Y` combinator is the usual fixed-point combinator, chosen to match the variant of λ-calculus being used (e.g. using eager or lazy evaluation).

Elucidating Mixin Inheritance Mixin inheritance combines two *mixins* `child` and `parent` into one that given two *instances* `instance` and `inherited` passes `(parent instance inherited)` as the `super` argument to `child`.

By the time the complete `instance` and `inherited` value so far are provided (if ever), the combined mixin itself may be but part of a wider combination, with further mixins both to the right and to the left. The provided `instance` will then be the fixed-point of the entire wider combination (involving further children to the left, then `child` and `parent`, then further parents to the right). Meanwhile, the `inherited` value will only contain the information from applying the further parent mixins to the right to the provided `base` object. The `parent` will be able to extend (enrich or override) any method definition from the `inherited` computation; the `child` may further extend it, and further mixins to the left yet more.

The function matches the `mix` function from the introduction modulo α-renaming, well-named since its essence is to compose or “mix” mixins. The function is associative, with identity mixin `idm = λ s t ↦ t`. As usual, a change of representation from `p` to `cp = inherit p` would enable use regular function composition for `mix`, whereas `fix` would retrieve `p` as `cp idm`; but that would make the types unnecessarily more complex.

Stricter, More Modular Types The types given in section 3.2.1 work well, but then must be carefully chosen so the `self` and `super` used during mixin definition should precisely match those used during mixin composition and instantiation. This is not a problem if a mixin is used only once (as in single inheritance, see section 4.2), but it is a problem in the more general case of mixin inheritance (and in multiple inheritance, see section 4.3).

A more refined type that can be used for mixins is then:

```
Mixin self super = self ⊂ super ⇒
  ∀ eself ⊂ self, ∀ esuper ⊂ super, eself ⊂ esuper ⇒
     eself → esuper → eself
```

where `self` and `super` are the minimal types intrinsic to the mixin, and `eself` and `esuper` are the *effective* types, respective subtypes of the above, as will actually be used, depending on the context of instantiation.

This type is an intersection of all variants of the previous type for subtypes `eself` and `esuper` of `self` and `super` respectively. It allows a mixin to be defined in its most general form, then used multiple times, each in a distinct more specialized context, making the mixin definition and its typechecking *more modular*. In exchange for this modularity, the mixin is restricted to only act in a uniform manner, that monotonically preserves arbitrary additional information passed as arguments to it.

Minimal Design, Maximal Outreach We have just derived from first principles a minimal design of prototypes-as-mixin-functions to embody modular increments of software specification inside a functional programming language. And this design closely reproduces that of existing models and languages:

(a) It reproduces the earliest general semantic model of OO (Bracha and Cook 1990).
(b) It also reproduces the formal semantics (though not the implementation) of objects in the pure lazy dynamic functional prototype object language Jsonnet (Cunningham 2014), a popular choice to generate distributed software deployment configurations for Kubernetes or AWS, and was started as a conceptual cleanup of
(c) the Google Control Language GCL (Bokharouss 2008) (née BCL, Borg Control Language), which has been used to specify all of Google's distributed software deployments since about 2004 (but uses dynamic rather than static scoping, causing dread among Google developers).
(d) It furthermore reproduces not just the semantics but the actual implementation of "extensions" (Simons 2015) as a user-level library in the pure lazy dynamic functional language Nix; these extensions are heavily used by NixOS (Dolstra and Löh 2008), a Nix-based software distribution for Linux and macOS, one with thousands of contributors.[4]

The main difference between our minimal model and the above works is that our model generalizes them by not being tied to any specific encoding of records, or indeed to records at all (see section 5.2)

This simplest of object-oriented designs, purely functional prototypes as mixin functions, has thus been proven capable to literally support specification and deployment of software on a world-wide scale. As we'll see, this design embodies the primitive core of OO, to which other forms of OO can be reduced. In the end, we can rightfully claim that the essence of OO in historical intent as well as practical extent is the incremental modularity embodied as language entities, and that prototypes are the most direct form of this embodiment.

3.3 Working with Records

Records, Methods, Instances Most OO tradition, including the precedents cited above, follows the historical restriction of only enabling modular and incremental specification of *"records"* mapping names to values (Cook 1989; Hoare 1965). The names, the values they are bound to, and/or the bindings, are at times called *"methods"*, "slots", "fields", "attributes", "properties", "members", "variables", or otherwise, depending on the specific sub-tradition.

[4] These extensions were reinvented semi-independently by Peter Simons, who did not know anything about their relationship to Prototypes, Mixins or OO, but was inspired by examples by and discussions with Andres Löh and Conor McBride, who were more versed in this literature.

The records themselves will be suitably wrapped into a proper computation result *instance*: a class (in Class OO), an object (in Prototype OO), a typeclass (in FP with typeclasses, though its users may deny the OO tradition), wherein the record will embody the "method dispatch table", "attribute set", "dictionary" or whatchamacallit of the aforementioned entity.

Note that this meaning of the word *instance* itself comes from the Prototype OO tradition, and does not match what the meaning of the word in the class OO tradition; in the latter tradition, "instance" instead refers to an element of the class seen as a type, whereas that type would be the instance in the prototype OO tradition. For now we will focus on the simplest and most primitive kind of OO, Prototype OO, in its simplest form where the instances are the records themselves. We will extend our point of view in section 4 and later.

Encoding Records We will assume that, either with some language primitives, some "builtin modules" to import from, or some variant of Church encoding, our Functional Language is suitably extended with the usual essential data structures: numbers, booleans, strings, tuples, lists. Record keys can be of a language-appropriate type with a decidable equality predicate: integers (sometimes as named constants at the meta-level), strings, or optionally symbols (interned strings) or identifiers (source code tracking entities).

Records can be defined from the empty record `rtop` and a constructor `rcons k v r` that given a key `k`, a value `v` and a previous record `r` returns a new record that extends `r` with a new or overriding binding of `k` to `v`. The three simplest encodings of a record would then be as a function, an *alist*, or a mapping table, as follow.

Records as functions is the simplest encoding, and accessing the value for a key is done by just calling the function with the key. However, overriding and deletion will leak memory and access time; also they don't support iteration over bindings — an introspection operation that is very much desired in contexts like I/O automation, though best kept hidden in contexts like analysis or restriction of software effects. The two constructors are as follows:

```
ftop = ⊤ = λ _ ↦ ⊥
fcons = λ k v r m ↦ if m == k then v else r m
```

The traditional Lisp "alist" (association list) data structure, singly-linked list of (key,value) pairs, solves the previous encoding's issues with memory leak and lack of introspection, but is still inefficient with linear-time operations. Its two constructors are as follows:

```
atop = []
acons = λ k v r ↦ [(k,v), ...r]
```

Records as pure mapping tables can provide logarithmic-time operations; but their implementation can be complex if not provided as a language primitive. Binding accessor, binding presence test, binding deletion, etc., are left as an exercise to the reader. We will write their constructors as follows:

```
mtop = {}
mcons = λ k v r ↦ {k: v, ...r}
```

In our previous article (Rideau et al. 2021) we showed how you could start with a simple of records as function, use OO style to incrementally and modularly specify a more elaborate mapping table data structure, and thereafter use that data structure in the definition of more efficient further records. That's our first case of a "meta-object protocol" (Kiczales et al. 1991), one that illustrates how to *bootstrap* more elaborate variants of OO from simpler variants.

Mixins and Helpers for Records Abstracting over the specific encoding for records, the primitive way to define a mixin that adds a method to a record being specified is with:

```
methodG = λ rkons k f s t ↦ rkons k (f s t) t
```

wherein the argument `k` is a key naming the method, `f` is a function that takes the instance `s` of type `self` and a inherited record `t` of type `super` and returns a value `v` to which to bind the method in a record that extends the inherited record, according to the record encoding defined by `rkons`.

In practice, OO language implementations provide a fixed builtin encoding for records, with specialized instantiation function `fixR` and method-addition mixin `methodR`:

```
fixR = λ mixin ↦ fix mixin rtop
methodR = methodG rcons
```

For a mixin that binds a method to a constant value `v`, you can then use

```
methodK k v = methodR k (λ _ _ ↦ v)
```

Common helpers could similarly be defined for mixins that bind a method to a value that only depends on the instance `s` of type `self` and not the inherited value `t` of type `super`, or vice versa.

Further helpers could help define more than one method at once e.g. by somehow appending record contents rather than consing bindings one at a time. Furthermore, given macros in the base language, specialized syntax could help make such definitions concise.

With or without macros, we will assume a syntax `a.b` for calling an appropriate record accessor with record `a` and method name `b` suitably encoded as a key. For simplification purposes, we will hereafter assume method names are strings.

Meanwhile, we will assume the following helpers to handle lists of mixins without having to awkwardly nest lots of applications of the `mix` function, assuming bracketed and comma-delimited lists, with `[head, ...tails]` patterns:

```
mix* [] = idm
mix* [h, ...t] = mix h (mix* t)
fix* base l = fix base (mix* l)
fixR* = fix* rtop
```

Giving polymorphic types to these list helpers may require not only subtyping but also some form of type indexing for those lists. Doing it without requiring full dependent types is left as an exercise to the reader.

Example Records built from Mixins We can now define the usual point and colored-point example as follows, where `$point` is the *prototype* for the point (in our simplest prototypes-as-mixin model), and `point` its *instance*:

```
$point = mix (methodK "x" 3.0) (methodK "y" 4.0)
point = fixR $point
$blue = (methodK "color" "blue")
coloredPoint = fixR* [$blue, $point]
```

Assuming a primitive `assert` that checks that a boolean value is true, and an equality predicate that behaves properly for records, we can then assert:

```
assert (point == {x: 3.0, y: 4.0})
assert (coloredPoint == {x: 3.0, y: 4.0, color: "blue"})
```

We can further define and use a radius-defining mixin, assuming functions `sqr` and `sqrt` for square and square roots of numbers respectively:

```
$radius == methodR "radius" λ s _ ↦ sqrt ((sqr s.x) + (sqr s.y))
pointWithRadius = fixR* [$radius, $point]
assert (pointWithRadius == {x: 3.0, y: 4.0, radius: 5.0})
```

Mixin Caveats Note that in the above examples, all the mixins commute, and we could have changed the order in which we define those methods — because they never use inheritance nor overrode any method, and instead pairwise define disjoint sets of methods. Thus ***merging disjoint commuting mixins embodies modularity, but not incrementality***: incrementality can still be achieved in an extralinguistic way by rebuilding modules in different ways from smaller modules; but to achieve it intralinguistic, you need a way to operate on existing modules, which by definition is not commutative.

As a counterpoint, the mixins below do override or inherit previous method bindings, and therefore do not commute, and instead yield different results when mixed in different orders:

```
$v1 = methodK "v" 1
$v2 = methodK "v" 2
$v10 = methodR "v" λ _ t ↦ t.v * 10
assert (fixR* [$v1,$v2] == {v: 1})
assert (fixR* [$v2,$v1] == {v: 2})
assert (fixR* [$v1,$v10] == {v: 1})
assert (fixR* [$v10,$v1] == {v: 10})
```

Finally note that trying to instantiate `$v10` alone would fail: it would try to multiply by 10 the inherited value of `v`, but the base record `rtop` has no such value and this would result in an error. Even without inheritance, the prototype `$radius` above would also fail to instantiate alone, because it will try to access undefined methods `x` and `y`. This illustrates how not every prototype can be successfully instantiated, which is actually an essential feature of prototypes (whether implemented as simple mixins or not), since the entire point of a prototype is to provide a *partial* specification of a small aspect of an overall computation, that in general depends on other aspects being defined by other prototypes.

4 Mixin, Single, and Multiple Inheritance

4.1 Mixin Inheritance

The Last Shall Be First The inheritance (Taivalsaari 1996) mechanism described above is called *mixin inheritance*. It is arguably the simplest kind of inheritance to formalize *given the basis of FP*. It also maps directly to the concepts of Modularity and Incrementality we are discussing. And for these reasons we introduced it first.

However, historically it was discovered last, because FP wasn't mature until much after the time the need for Modularity and Incrementality was felt. It is also relatively more obscure, probably because, in addition to the above, it is less modular than the more complex but previously discovered multiple inheritance (discussed below in section 4.3).

And yet, we already saw above in section 3.2.5 that object prototypes with mixin inheritance are used to specify software configurations at scale. An elaborate form of mixin inheritance is also notably used in the class-based OO system used by Racket's GUI (Flatt et al. 2006).

Mixin Semantics We saw above (section 3.2) that mixin inheritance involves just one type constructor `Mixin` and two functions `fix` and `mix`:

```
Mixin self super = self ⊂ super ⇒ self → super self
fix : Mixin self top → top → self
fix = λ mixin top ↦ Y (λ self ↦ mixin self top)
mix : Mixin self super → Mixin super duper → Mixin self duper
mix = λ child parent self duper ↦ child self (parent self duper)
```

4.2 Single inheritance

Simple and Efficient Historically, the first inheritance mechanism discovered was *single inheritance*, though it was not known by that name until later. In (Dahl et al. 1968), a "class" of records (Hoare 1965) uses a previous class as a "prefix", reusing all its field definitions and method functions; the text of the resulting class is then the "concatenation" of the direct text of all its transitive prefix classes. In modern terms, we call the prefix a superclass, the extended class a subclass. Single inheritance was made popular circa 1971 by Smalltalk and later circa 1995 by Java (International 2015).

Single inheritance is easy to implement without higher-order functions; method lookup can be compiled into a simple and efficient array lookup at a fixed index — as opposed to some variant of hash-table lookup in the general case for mixin inheritance or multiple inheritance. In olden days, when resources were scarce, and before FP was mature, these features made single inheritance more popular than the more expressive but costlier alternatives. Even some more recent languages that support multiple inheritance (section 4.3) also support single inheritance for some classes (or "structures"), and sometimes the consistent combination of the two (section 4.3.6).

Semantics of Single Inheritance In single inheritance, the prototypes at stake, i.e. the entities that embodied increments of modularity, are not the mixin functions of mixin inheritance, but simpler *generators* that only take a `self` as open recursion parameter and return a record using `self` for self-reference. The semantics can reduced to the following types and functions: :

```
Gen self = self → self
Y : Gen self → self
base : Gen top → top
base = λ _ ↦ rtop
extend : Mixin self super → Gen super → Gen self
extend = λ mixin parent self ↦ mixin self (parent self)
```

Note how `Gen self` is the type of generators for instances of type `self`; the instantiation function for a generator is the usual fixed-point combinator `Y`; the `base` object to extend is the generator that always returns the empty record (for whichever encoding is used for records); and the `extend` function creates a child generator from a parent generator and a mixin (as in mixin inheritance above), where `self` is constrained to be a subtype of `super`.

Mind again that in the single-inheritance paradigm, *the prototype is the generator, not the mixin.* A prototype-as-generator may thus be the `base` generator that returns the empty record `rtop` or otherwise base instance, or a generator created by extending a *single* `parent` generator with a `mixin`. Since the same constraint applies recursively to the parent generator, a prototype-as-generator can be seen as repeatedly extending that `base` generator with an ordered list of mixins to compose. Just like in mixin inheritance, an *instance* can thus still be seen as the fixed point of the composition of a list of elementary mixins as applied to a base instance. However, since generators, not mixins, are the prototypes, the "native" view of single inheritance is more to see the parent specified in `extend` as a direct super prototype, and the transitive supers-of-supers as indirect super prototypes; each prototype is considered as not just the mixin it directly contributes, but as the list of all mixins directly and indirectly contributed.

Single Inheritance with Second-Class Mixins While single-inheritance requires some form of mixin, most single-inheritance object systems don't allow mixins as first-class entities that can be independently composed. Rather mixins are only linear second-class syntactic entities and can only be used once, immediately, as part of an extension. You cannot consider a mixin or list of mixins independently, and *append* such lists together; you cannot abstract a base or super instance away from a generator to extract its mixin; you can only *cons* a single new elementary mixin to the list of mixins implicit in a previous generator and already applied to its base.

This will particularly matter when we see that in most Class OO languages, prototype inheritance happens in a restricted language at the type level, one with limited abstraction and no way to express appending from consing.

Then again, if language starts with single-inheritance OO, but *does* allow mixins as first-class entities that can be composed, then it actually supports mixin inheritance, not just single inheritance, just like the Racket class system does (Flatt et al. 2006), or like typical uses of extensions in Nix go. It thus only makes sense to speak of single inheritance in a context where the language syntax, static type system, dynamic semantics, or socially-enforced coding conventions somehow disallow or strongly discourage mixins as first-class entities.

Lack of expressiveness and modularity The limitations to single inheritance translate into lack of expressiveness relative to mixin inheritance. Thus, in an OO language with single inheritance, you can define a prototype `Point` with two coordinates `x` and `y` with two children prototypes `ColoredPoint` and `WeightedPoint` that respectively extend it with an attribute `color` and an attribute `weight`. But if you want a `WeightedColoredPoint` that has both `color` and `weight` attributes, you have to choose at most one of the two prototypes `ColoredPoint` and `WeightedPoint` to inherit from, and repeat all the definitions of the other's mixin.

In case you want a prototype to possess all the methods defined in each of two or more long mixins or long lists of mixins are involved, you will have to repeat all the definitions from all but one existing list of mixins. You can always resort to copy/pasting the definitions from one class to the other; but that is unreliable and fragile as maintenance operations now need to happen simultaneously in multiple copies that the developer must track down, and that can easily grow subtly out-of-synch as the developer is fallible. Worse, this is an extra-linguistic means, so that inasmuch as you then still achieve incremental modularity, it is no longer *within* the language, only *outside* it. By contrast with mixin inheritance or multiple inheritance, you could easily combine together all the elementary mixins from each of the many prototypes-as-mixins that you want to simultaneously extend.

This concludes our proof that single inheritance is strictly less expressive (Felleisen 1991) and less modular than mixin and multiple inheritance.

4.3 Multiple inheritance

More Sophisticated A third kind of inheritance is *multiple inheritance*, that historically appeared before mixin inheritance was formalized (Cannon 1982), and that is more sophisticated than the two above. It was popularized by the Lisp object systems Flavors, Common Loops, New Flavors and CLOS (Bobrow et al. 1988), then by Self and C++. These days it is notably used in Python, Scala or Rust.

Like mixin inheritance, multiple inheritance allows developer to create a new prototype using more than one existing prototype as super prototype, lifting the main limitation of single inheritance. Like single inheritance, multiple inheritance allows developer to declare dependencies between prototypes, such that a prototype can have indirect, transitive dependencies implicitly included as super prototypes, as well as direct super prototypes.

Prototypes as a DAG of mixins Since each prototype inherits from multiple parents rather than a single one, the inheritance hierarchy is not a list, but a Directed Acyclic Graph (DAG). Each prototype is a node in the overall DAG. The super prototypes explicitly listed as dependencies it inherits from when defining it are called its *direct supers*. But the set of all a prototype's super prototypes includes not only those direct supers, but also indirectly the supers of those supers, etc., in a transitive closure.

The prototype's supers thus constitute a DAG, that is an "initial" sub-DAG of the DAG of all prototypes, that includes all the prototypes directly or indirectly "above" the considered prototype, that is at the very bottom of its DAG. The super prototype relation can also be viewed as a partial order on prototypes (and so can its opposite sub prototype relation).

This is in contrast with single inheritance, where this relation is a total order, each prototype's super hierarchy constitute a list, and the overall hierarchy of all prototypes is a tree. This is also in contrast with mixin inheritance, where each mixin's inheritance hierarchy can be viewed as a composition tree, that since it is associative can also be viewed flattened as a list, and the overall hierarchy is a multitree… except that a super prototype (and its own supers) can appear multiple times in a prototype's tree.

Each prototype is thus a node in the inheritance DAG. To represent it, a prototype will then be not just a mixin function, but a tuple of:

(a) a mixin function, as in mixin inheritance, that contributes an increment to the modular specification,
(b) an ordered list of direct super prototypes it inherits from, that specify increments of information on which it depends, and
(c) a unique name (fully qualified path, string, symbol, identifier, or other tag) to identify each prototype as a node in the inheritance DAG.

Precedence Lists Then comes the question of how to instantiate a prototype's inheritance DAG into a complete specification, of how to reducing it to a *generator* as for single inheritance.

A general solution could be to compute the instance, or some seed value based on which to compute the instance, as an *inherited attribute* of that inheritance DAG. For instance, a generator (as in single-inheritance above) of which to take a fixed-point, could be computed by having each mixin function be of type `self` → `super` → `self` where each direct super prototype is of type `super_i` and `super` is the *product* of the `super_i`.

However, the increment of specification from each prototype must be taken into account *once and only once* in the overall specification; and the order in which these increments are taken into account must be *consistent* from one method computation to another. If individual mixin functions had to take a tuple or list of inherited attributes, they would have a hard time untangling the already mixed in effects of other mixins, to reapply them only once, what more in a consistent order.

Therefore, multiple inheritance uses a more refined mechanism, wherein the inheritance DAG for a prototype is reduced to a list of prototypes, the *precedence list*. An instance is then as per mixin inheritance (or, equivalently, single inheritance), by combining the mixin functions of the supers, in the order given by the precedence list, with a universal top value. Each mixin function remains of type `self → super → self`, where the `super` argument is the *intersection* of the types `super_i`, not just of its direct super prototypes, but, effectively, of all the super prototypes after it in the precedence list (see section 3.2.4 for handling that gap).

The precedence list is itself computed, either by walking the DAG or as an inherited attribute, using the prototype names to ensure the unique appearance of each super in the resulting list. The precedence list can be viewed as a total order that extends and completes the partial order of the inheritance DAG. Modern algorithms like C3 (Barrett et al. 1996; Wikipedia 2021) further ensure "monotonic" consistency between the precedence list of a prototype and those of its supers, such that the former extends the latter as well as the list of supers itself.

Complete implementations of prototypes using multiple inheritance in a few tens of lines of code are given in our previous paper using Scheme (Rideau et al. 2021), or in a proof of concept in Nix (Rideau 2021). Our production-quality implementation in Gerbil Scheme (Rideau 2020) including many features and optimizations fits in about a thousand lines of code.

More Expressive than Mixin Inheritance Multiple inheritance requires measurably more sophistication than mixin inheritance, and hence an additional cognitive burden. Why would anyone use that instead of just using mixins? Because it is more expressive, and more modular, and its cognitive burden pays for itself by alleviating the other cognitive burdens from developers.

The multiple inheritance is no less expressive than mixin inheritance is simple enough to prove: you can macro-express (Felleisen 1991) mixin inheritance within multiple inheritance. Replace each mixin function by a prototype using that mixin function, a empty direct super list. Keep around lists of such prototypes rather than mix them, then before you instantiate, create a single prototype with an identity mixin that depends on the list of mixins as direct super prototypes, where each mixin was given a fresh name, to ensure that multiple copies are all used indeed.

This trick with fresh names at the last minute is necessary to defeat multiple inheritance otherwise ensuring that a given prototype (as identified by its name) will be used once and only once in the precedence list. But this unicity is actually a feature that the users usually want (and if they somehow do want multiple uses of a mixin, they can explicitly use multiple copies of it with distinct names).

More Modular than Mixin Inheritance In practice there is always a dependency order between prototypes, whether it is reified as an automatically managed in-language entity as with multiple inheritance, or left as an extra-language entity that developers must manually keep track of as with mixin inheritance. Thus, a prototype may depend on a method having been declared or implemented by a (transitive) parent, so it may use or override it. That parent that must appear before it in the precedence list of prototypes (in the right-to-left order of application to the base instance with the convention we use above). Moreover, each prototype should appear only once in the precedence list, because its effects may not be idempotent, or may cancel the effects of other prototypes found in between two copies.

For instance, consider a dependency DAG such as follows, where among other things, `Z` depends on `K2` that depends on `D` that depends on `O`:

$$\begin{tikzcd}
 & & O & & \\
C \arrow[urr] & A \arrow[ur] & B \arrow[u] & D \arrow[ul] & E \arrow[ull] \\
 & K1 \arrow[ul] \arrow[u] \arrow[ur] & K3 \arrow[ul] \arrow[ur] & K2 \arrow[ul] \arrow[u] \arrow[ur] & \\
 & & Z \arrow[ul] \arrow[u] \arrow[ur] & &
\end{tikzcd}$$

The only way to compute precedence lists for `O`, `A`, `B`, `C`, `D`, `E` yields the respective precedence lists `[O]`, `[A O]`, `[B O]`, `[C O]`, `[D O]`, `[E O]`. No problem.

However, consider the precedence list for `K1`. If computed naively by concatenating the precedence lists of the prototypes it directly depends on without eliminating duplicates, you get `[K1 C O A O B O]`. This can be a big problem if re-applying `O` will undo some of the effects of `A` or of `B`. The problem is the same for `K2` and `K3` and only worse for `Z`. Even when all prototypes at stake are idempotent and commute, this naive strategy will cause an exponential explosion of prototypes to mix as the graph becomes deeper. Meanwhile, a proper linearization as given by the C3 algorithm would be `[K1 C A B O]` for `K1` and `[Z K1 C K3 A K2 B D E O]` for `Z`. It avoids issues with duplicated prototypes, and grows linearly with the total number of prototypes however deep the graph.

With mixin inheritance, developers would have to manually curate the order in which they mix prototypes, extra-linguistically. When using prototypes defined in other modules, they would have to know not just the prototypes they want to use, but all the detail about the transitive prototypes they depend on. Their dependency DAG will not be a hidden implementation detail, but part of the interface. And when some upstream module modifies the dependency DAG of a prototype, all the prototypes in all the modules that transitively depend on it will have to be updated by their respective maintainers to account for the change.

This requires much more information to understood and provided by developers than if these developers were instead using multiple inheritance, that automates the production of that precedence list, and its update when upstream modules are modified. The transitive parts of DAG can largely remain a hidden implementation detail from those developers who only care about some direct dependencies. Thus, mixin inheritance is indeed less modular than multiple inheritance.

Single and Multiple Inheritance Together Some languages such as CLOS (Bobrow et al. 1988) allow for both single-inheritance `struct`s and multiple-inheritance `class`es with uniform ways of defining object and methods. Thus, programmers can benefit from the performance advantage in slot access or method dispatch possible where there is no multiple-inheritance, while still enjoying the expressiveness and modularity of multiple-inheritance in the general case. They can explore without constraint, and simply change a flag when later optimizing for performance.

However, in CLOS, structs and classes constitute disjoint hierarchies. Some languages further allow structs and classes to inherit from each other, within appropriate constraints. Thus Scala (Odersky and Zenger 2005) allows a single struct to inherit from classes (except, to fit the Java and Smalltalk traditions rather than Lisp tradition, it calls the single-inheritance structs "classes", and the multiple-inheritance classes "traits"). Gerbil Scheme supports the least set of constraints that preserve the coherence of both structs and classes, by suitably extending the C3 algorithm.

C3 crucially frames the problem of superclass linearization in terms of constraints between the precedence lists of a class and of its superclasses: notably, the precedence list of a superclass must be an ordered subset of that of the class, though its elements need not be consecutive. To support structs and their optimizations, we only need add a constraint that the precedence list of a struct must be a suffix of that of its substructs (when considered in the order from most specific to least specific, as is customary in languages with multiple inheritance, after the Lisp original).

At that point, we realize that what characterizes structs is not exactly "single inheritance" since a struct can now have multiple superclasses, and a class can now inherit from a struct indirectly via multiple superclasses. There is still single inheritance of sorts between structures, in the sense that the superstructures of a structure constitute a finite total order, when you ignore the other classes in the inheritance. But by this observation, by ignoring these other classes, fails to characterize structs. Instead, what characterizes structs is this "suffix" constraint on precedence lists, which include all classes, not just structs. This characterization in turn harkens back to the original Simula name of "prefix" for a superclass: Simula was then considering its single-inheritance precedence list in the opposite order, from least specific to most specific superclass (though the vocabulary to say so didn't exist at the time). And this semantic constraint can be expressed in a system that has multiple inheritance.

Under-Formalized Many notable papers offer proper treatment of multiple inheritance as such (Allen et al. 2011).

However, multiple inheritance often remains unjustly overlooked, summarily dismissed, or left as an exercise to the reader in academic literature that discusses the overall formalization of programming languages and OO (Abadi and Cardelli 1997; Friedman et al. 2008; Khrisnamurthi 2008; Pierce 2002).

Many computer scientists interested in the semantics of programming languages seem to either fail to understand or fail to value the modularity enhancement from multiple inheritance over single inheritance or mixin inheritance; or they are not ready to deal with the extra complexity needed to formalize multiple inheritance, for instance due to requiring richer type systems. (Cardelli 1984)

And yet languages that care more about expressiveness, modularity and incrementality than about ease of writing performant implementations with simpler type systems, will choose multiple inheritance over the less expressive and less modular alternatives: see for instance Common Lisp, C++, Python, Scala, Rust.

5 Missing Insights into OO

Here are some topics that are largely neglected by both academic literature and public discourse about OO, even more so than multiple inheritance, yet that can yield essential insights about it. Some of these insights may already be known, but often only implicitly so, and only by a few experts or implementers.

5.1 Pure Laziness

Lazy makes OO Easy In a lazy functional language such as Nix, you can use the above definitions for `fix`, `mix`, `methodG` and `methodR` as is and obtain a reasonably efficient object system; indeed this is about how "extensions" are defined in the Nix standard library (Simons 2015).

Now, in an eager functional language such as Scheme, using these definitions as-is will also yield correct answers, modulo a slightly different `Y` combinator. However applicative order evaluation may cause an explosion in redundant recomputations of methods, and sometimes infinite loops. Moreover, the applicative `Y` combinator itself requires one extra layer of eta-expansion, such that only functions (including thunks) can be directly used as the type for fixed-points. Unneeded computations and infinite loops can be averted by putting computations in thunks, protected by a λ; but computations needed multiple times will lead to an exponential duplication of efforts as computations are nested deeper, because eager evaluation provides no way to share the results between multiple calls to a same thunk, especially those from the `Y` combinator. The entire experience is syntactically heavy and semantically awkward.

Happily, Scheme has `delay` and `force` special forms that allow for both lazy computation of thunks and sharing of thusly computed values. Other applicative functional languages usually have similar primitives. When they don't, they usually support stateful side-effects based on which the lazy computation primitives can be implemented. Indeed, an applicative functional language isn't very useful without such extensions, precisely because it is condemned to endlessly recompute expressions without possibility of sharing results across branches of evaluation — except by writing everything in continuation-passing style with some kind of state monad to store such data, which would involve quite a non-modular cumbersome global code transformation.

Computations vs Values To reprise the Call-By-Push-Value paradigm (Blain Levy 1999), prototypes incrementally specify *computations* rather than *values*: instructions for recursive computing processes that may or may not terminate (which may involve a suitable monad) rather than well-founded data that always terminates in time proportional to its size (that only involve evaluating pure total functions). Others may say that the fixed-point operation that instantiates prototypes is coinductive rather than inductive.

And indeed, laziness (call-by-need) is the best good way to reify a computation as a value, bridging between the universes of computations and values. Compared to mere thunking (call-by-name) that can also bridge between these universes, laziness enables sharing, with advantages both in terms of performance and semantic expressiveness, without requiring any stateful side-effect to be observable in the language, thus preserving equational reasoning. Thunking can still be expressed on top of a lazy language, but laziness cannot be expressed on top of a language with thunks only, without using side-effects.

Method Initialization Order Traditional imperative OO languages often have a problem with the order of slot initialization. They require slots must be initialized in a fixed order, usually from most specific mixin to least specific, or the other way around. But subprototypes may disagree on the order of initialization of their common variables. This leads to awkward initialization protocols that are (a) inexpressive, forcing developers to make early choices before they have the right information, and/or (b) verbose, requiring developers to explicitly call super constructors in repetitive boilerplate, sometimes passing around a lot of arguments, sometimes unable to do so. Often, slots end up undefined or initialized with nulls, with later side-effects to fix them up after the fact; or a separate cumbersome protocol involves “factories” and “builders” to accumulate all the initialization data and process it before to initialize a prototype.

By contrast, lazy evaluation enables modular initialization of prototype slots: Slots are bound to lazy formulas to compute their values, and these formulas may access other slots as well as inherited values. Each prototype may override some formulas, and the order of evaluation of slots will be appropriately updated. Regular inheritance with further prototypes, is thus the regular way to further specify how to initialize what slots are not yet fully specified yet.

Pure lazy prototypes offer many advantages over effectful eager object initialization protocols:

– The slot initialization order needs not be the same across an entire prototype hierarchy: new prototypes can modify or override the order set by previous prototypes.

– When the order doesn't require modification, no repetitive boilerplate is required to follow the previous protocol.
– There are no null values that become ticking bombs at runtime, no unbound slots that at least explode immediately but are still inflexible.
– There are no side-effects that complicate reasoning, no computation yielding the wrong value because it uses a slot before it is fully initialized, no hard-to-reproduce race condition in slot initialization.
– At worst, there is a circular definition, which can always be detected at runtime if not compile-time, and cause an error to be raised immediately and deterministically, with useful context information for debugging purposes.
– There is seldom the need for the "builder pattern", and when builders are desired they require less code.

If it's so good... Some may wonder why OO languages don't use pure lazy functional programming for OO, if the two are meant for each other.

Well, they do: as we'll see in section 6, class-based OO is prototype-based OO at the type-level for type descriptors; and the type-level meta-programming language with which to define and use those prototypes at compile-time, thus where OO actually takes place, is invariably pure functional: languages with static classes have have no provision for modifying a class after it is defined at compile-time, and disclaim all guarantees if reflection facilities are used to modify them at runtime. The compile-time languages in which classes are defined is often quite limited; but a few languages have a powerful such compile-time language, famously including C++ and its "templates". Templates support lazy evaluation with `typedef`, or, since C++11, with `using ... = ...`. Even when eagerly evaluated, multiple occurrences of a same type-level template expression share their computed values, similar to lazy evaluation.

As for prototype OO, while early languages with prototypes, like T or Self, or later popular ones like JavaScript, were applicative and stateful, we already discussed in section 3.2.5 how in the last ten years, Jsonnet and Nix have brought out the happy combination of pure lazy functional programming and prototypes. We have also been using in production a lazy functional prototype object system as implemented in a few hundred lines of Gerbil Scheme (Rideau 2020).

Thus, we see that contrary to what many may assume from common historical usage, not only OO does not require the usual imperative programming paradigm of eager procedures and mutable state — OO is more easily expressed in a pure lazy functional setting. Indeed, we could argue that OO *as such* is almost never practiced in a mutable setting, but rather as a pure functional static metaprogramming technique to define algorithms that often use mutation (but don't need to).

Of course, it is also possible to embrace imperative style and stateful side-effects when either using or implementing OO as such. For instance, many Lisp and Scheme object systems have allowed dynamic redefinition of classes or prototypes and their inheritance hierarchy, while the language Self had mutable *parent slots* to specify prototype inheritance, and JavaScript objects have a mutable `__proto__` slot. Often, mutation makes for much more complex semantics, with uglier edge cases, or expensive invalidation when the inheritance hierarchy changes, but faster execution in the common case thanks to various optimizations. See section 7.1.1.

5.2 Instances Beyond Records

Prototypes for Numeric Functions Looking back at the definitions for `Mixin`, `fix` and `mix`, we see that they specify nothing about records. Not only can they be used with arbitrary representations of records, they can also be used with arbitrary instance types beyond records, thereby allowing the incremental and modular specification of computations of any type, shape or form whatsoever (Rideau et al. 2021).

For instance, a triangle wave function from real to real could be specified by combining three prototypes, wherein the first handles 2-periodicity, the second handles parity, and the third the shape of the function on interval `[0,1]`:

```
twf = (λ p q r ↦ fix (mix p (mix q r)) λ x ↦ ⊥)
      (λ self super x ↦ if x > 1 then self (x - 2) else super x)
      (λ self super x ↦ if x < 0 then self (- x) else super x)
      (λ self super x ↦ x)
```

The prototypes are reusable and can be combined in other ways: for instance, by keeping the first and third prototypes, but changing the second prototype to specify an odd rather than even function (having the `then` case be `- self (- x)` instead of `self (- x)`), we can change the function from a triangle wave function to a sawtooth wave function.

Now these real functions are very constraining by their monomorphic type: every element of incremental specification has to be part of the function. There cannot be a prototype defining some score as MIDI sequence, another prototype defining sound fonts, and a third producing sound waves from the previous. Actually, one could conceivably encode extra information as fragments of the real function to escape this stricture, but that would be very awkward: For instance, one could use the image of floating-point `NaN`s or the indefinite digits of the image of a special magic number as stores of data. But it's much simpler to incrementally define a record, then extract from the record a slot bound to a numeric function—in, in what can be seen as a use of the "builder pattern".

Records are thus a better suited target for general-purpose incremental modular specification, since they allow the indefinite further specification of new aspects, each involving slots and methods of arbitrary types, that can be independently specialized, modified or overridden. Still, the kernel of OO is agnostic with respect to instance types and can be used with arbitrarily refined types that may or may not be records, may or may not be functions, and may or may not generalize or specialize them in interesting ways.

Conflating Records and Functions Many languages solve the above issue by allowing an instance to be simultaneously both a record and a function. Thus, prototype definitions can use extra record slots to store ancillary data (such as

MIDI sequence and sound font in the example above), yet simultaneously specify a the behavior of a function.

Thus, back in 1981, Yale T Scheme (Rees and Adams 1982) was a general-purpose programming environment with a graphical interface written using a prototype object system (Adams and Rees 1988). It lived by the dual slogans that “closures are a poor man’s objects” and “objects are a poor man’s closures”; its functions could have extra entry points, which provided the basic mechanism on top of which methods and records were built.

Many later OO languages offer similar functionality, though they build it on top of OO rather than build OO on top of it: CLOS has `funcallable-instance`, C++ lets you override `operator ()`, Java has *Functional Interfaces*, Scala has `apply` methods, JavaScript has the `Function` prototype, etc. Interestingly, in Smalltalk, a “function” is just an object that can reply to the message `value:`, and an object can similarly be not just a function, but an array, a dictionary, or a stand-in for any of the “builtin” primitive data types, “just” by defining the methods that comprise the interface for each data type.

Such functionality does not change the expressiveness of a language, since it is equivalent to having records everywhere, with a specially named method instead of direct function calls. Yet, it does improve the ergonomics of the language, by reducing the number of extra-linguistic concepts, distinctions and syntactic changes required for all kinds of refactorings. It also opens new ways for programmers to shoot themselves in the foot, but programmers already have plenty of them, and these record-function instances don’t make that particularly easier.

Now, to a mathematician, this may mean that those instances aren’t functions strictly speaking, but an implicit product of a record and a function, and maybe more things. The mathematical notion of “function” isn’t directly represented in the programming language, only somehow implemented or expressed in it. Programmers may retort that such is the reality in any programming language anyway, and some languages are more honest about it than others, and won’t let a lie stop them from building more ergonomic features. Mathematicians might insist that sometimes they really want to represent just a function, with no other hidden capabilities, and more generally, to maximally restrict what a program can do, so as more feasibly to reason about it. Programmers may retort that they still can in such a language, if they insist.

Our purpose is not to repeat the debate whether or not making objects callable is a good or bad idea, even less to take sides in it—but instead to notice and make explicit this important and useful notion of implicit product of several things, whether record, function, or more, whether resolved syntactically at compile-time (when possible) or dynamically at runtime (otherwise). We will call this implicit product a *conflation.*

Freedom of and from Representation We already saw in section 3.3.2 that were many ways to represent records, that affect performance, memory usage, the ability to introspect values, etc. There are even more ways to represent them in conflation with functions, arrays, and more. And a language implementer may find themselves with an embarrassment of choices for what exact specific underlying data type to use to represent the instances of their specifications.

However, having neatly separated the core concepts of prototype, inheritance and instantiation, as tools of incremental specification of software, from any specific type of instance being specified, we now find we are not only free to choose the instance type, but also free *not* to choose: we can keep the concepts of prototypes, inheritance, etc., as abstract entities that can work on any instance type a programmer may want to apply them to, instead of only supporting a single privileged instance type. This makes prototypes a more general and more modular notion that can be used in multiple ways in a same language ecosystem.

OO without Objects At this point, we may realize we have been explaining and implementing all key concepts of "Object Orientation" without ever introducing any notion of object, much less of class.

There are prototypes, and there are instances; but neither is an object. Prototypes are uninstantiated specifications, often incomplete therefore uninstantiable; you can't call methods on them, or do anything that you can expect to do on an object. Instances are plain values of any type whatsoever, sometimes just simple real functions; you can't combine them with inheritance, or do any OO-related operation on them. If either is an "object", then the word "object" is utterly empty of meaning.

Indeed, we wrote code exactly in this object-less "OO" style to generate presentation slides for this work (Rideau et al. 2021). We could express without objects everything that is usually done with objects, but for one caveat discussed below in section 5.3.2.

Thus maybe "Object Orientation" was always a misnomer, born from the original confusion of a time before science identified and clarified the relevant concepts. Maybe the field should be named after Inheritance, or Prototypes, or Incremental Modularity, and banish the word "Object" forevermore from its name.

Yet misnamed as OO may be, objects are possible and a useful concept in it.

5.3 Objects: The Power of Conflation

Conflating Prototype and Instance While neither a prototype nor an instance is an object, the *conflation* of the two, is. This is exactly what objects are in pure prototype OO languages like Jsonnet and Nix, and a slight simplification of what they are in stateful prototype OO languages: every object can be seen as either an instance, when querying the values of its slots, or as a prototype, when combining it with other objects using inheritance.

Indeed in a pure functional language, without side-effects, there is a unique instance associated to any prototype, up to observable equality: its fixed-point. Thus, it always makes sense to consider "the" instance for a prototype, and to see it as but another aspect of it. Evaluating the fixed-point may or may not converge, but thanks to lazy evaluation, you don't have to care about whether that is the case to refer to the two together, and once computed once the result can be cached for performance.

If the language has side-effects, there may be multiple distinct instances to a prototype, and a `clone` construct will generate a new object from an existing object, and still keep instance and prototype together. Even in such a language, a laziness construct can help build a simpler and nicer object system.

Note that these prototype objects correspond to *classes at compile-time* in class OO languages, that use the word "object" differently. See section 6.

Keeping Extensibility Modular Specifying software with prototypes yet without objects works great, as long as it's clear at all times which entities are prototypes and which are instances. This is simple enough when the specification all happens in a single phase, and everything is a big prototype with a big fixed point operation around it, and plenty of explicit fixed point operations within, one for every sub-prototype. But what if the specification involves multiple phases, where the "same" entity is sometimes used as an instance, sometimes as a prototype, what more without it always being used as a prototype before it is used as an instance? What if some entity, complete and useful in itself, is later extended by another programmer, overriding parts that the original programmer didn't anticipate would be overridden? What if unextended and extended variants of it are used in a same program?

The conflation of prototype and instance into an object enables future phasing and extensions without the original programmers having to anticipate how their code will be used and to factor it accordingly. Programs can be written that can refer to previous or other programs without having to track and distinguish which parts are instantiated at which point. No need to decide at every potential extension point whether and when to either compute a fixed-point or defer its computation, in what leads to a combinatorial explosion of potential interfaces. No need to defer everything until the last minute, and make it expensive to use any intermediary value to make a decision before that last minute, while contaminating the entire computation to turn everything into explicit prototypes. No need to construct and remember access paths or lenses that you'll have to use in two different contexts to access both aspects of the "same" object.

In the end, conflation of prototype and instance allows programmers to write and refer to objects with less mutual coordination with respect to when an object is being used or extended. By the criteria in section 2, this conflation indeed makes OOP more modular.

Conflating More Features For mixin inheritance, we wanted a prototypes to be just a mixin function. For multiple inheritance, we wanted a prototype to *also* have a list of direct supers; and for good measure, we wanted to cache rather than expensively recompute every time the prototype's precedence list of transitive supers. Further features can be added by conflating further aspects into the notion of prototypes.

For instance, we can add a "default values" feature, by conflating an additional map from slot to value that is only consulted when no override is provided. Compile-time type restrictions or runtime assertions on slots, slot visibility information, debugging information, online documentation, examples and test cases, generators and minimizers for property-based testing, introspectable method definitions, etc., can be added as in the same way: as additional conflated aspects of a prototype, factors in the prototype as (implicit) product, or equivalently slots in the prototype seen itself as a record instance.

Distinction and Conflation Conflating many aspects of prototypes, instances and together objects in an implicit product brings better ergonomics and extensibility. But doing it without having explicit notions of these aspects as distinct and separate entities leads to a hell of ununderstandably complex semantics as all the aspects are inextricably weaved together: ***Conflation without Distinction is Confusion***.

Previous presentations of OO, whether in programming language documentation, teaching materials or academic literature, have largely or wholly omitted both the implicit conflation of prototypes and instances in objects (for Prototype OO) or classes (for Class OO), and the explicit distinction between the two notions, with indeed much confusion as both cause and consequence. And yet by necessity those who implement compilers and interpreters by necessity abide by this conflation.

By insisting on both conflation and distinguish of the two concepts of instance and prototype, we aim at dispeling the confusion often reigns in even the most experienced OO practitioners when trying to reason about the fine behavior of OO programs.

6 Classes

6.1 Class OO as Type-Level Prototype OO

Type Prototypes Having fully elucidated Prototype OO in the previous sections, including its notion of Object as conflationg of Prototype and Instance, we can now fully elucidate Class OO including its notion of Class: ***A Class is a Prototype for a Type***.

Class OO is therefore a special case of Prototype OO, though one where prototype computations only happen at the type-level. The instances incrementally specified by these prototypes are *types*—or more precisely *type descriptors*, usually available at compile-time only in Class OO languages, in a form of staged metaprogramming.

Thus when we claimed in section 1.4.1 that the situation of classes in OO was similar to that of types in FP, we meant it quite literally.

Class OO makes classes Second-Class Now, the language in which these type prototypes are defined and composed is not the usual "base language" that the programmer is usually programming in (e.g. C++, Java, C#), but instead a distinct *type-level language* in which the types and the base-level functions operating on them are being incrementally specified.

The type-level language used in a language with Class OO is usually is restricted in expressiveness, in an often deliberate attempt to keep it from being "Turing-equivalent". This attempt sometimes succeeds (as in OCaml), but more often than not utterly fails, as computational power emerges from unforeseen interactions between language features added over time (as in C++, Java, Haskell).

The attempts do usually succeed, however, at making these type-level languages require a completely different mindset and very roundabout design patterns to do anything useful, a task then reserved for experts.

Computationally powerful or not, the type-level language of a Class OO language is almost always very different from the base language: the type-level languages tend to be pure functional or logic programming languages with pattern-matching and laziness but without any I/O support, even though the base languages themselves tend to be eager stateful procedural languages with lots of I/O support and often without pattern-matching or laziness (or limited ones as afterthoughts).

In the end, classes are thus not *first-class* entities in Class OO (subject to arbitrary programming at runtime), but *second-class* entities (restricted to limited compile-time programming), though many languages offer limited reflection capabilities at runtime. By contrast, classes are first-class entities in Prototype OO; and indeed, one of the first applications of Prototype OO in any language is often to build rich runtime type descriptors, that include features not usually expressible with compile-time type descriptors or their runtime representation as sometimes accessible through "reflection", such as extra constraints, context-dependent I/O, property-based testing support, etc.

More Popular yet Less Fundamental Class OO was historically discovered (1967) nine years before Prototype OO (1976), and remains overall more popular in the literature. The most popular OO language, JavaScript, started with Prototype OO only (1995), but people were constantly reimplementing classes on top, and twenty years later classes were added to the language itself (International 2015).

And yet we will argue that Class OO is less fundamental than Prototype OO: it can indeed be very easily expressed in terms of Prototype OO and implemented on top of it (as exemplified many times over in JavaScript), such that inheritance among classes is indeed a special case of inheritance among the underlying prototypes, whereas the opposite is not possible: Class OO offers little to no advantage in implementing Prototype OO over directly implementing it on top of FP, and it is not universally possible to build Prototype OO such that a prototype's inheritance structure is verily the inheritance of an underlying class (since the former is always first-class but the latter usually second-class).

6.2 Typing Records

Now, a type system with suitable indexed types and subtyping is required to use rich records. With a less-expressive type system, each use of mixins will be monomorphic; at the very least, methods will have to be options to support prototypes that say nothing about them; dynamic typing may have to be reimplemented on top of static typing to support more advanced cases; and users will have to do a lot of wrapping and unwrapping to use mixins, adding a lot of overhead to the cost of incremental specification. This may explain why using the above implementation kernel for OO in FP has so far only been found non-trivial use but in dynamically typed languages.

Record classes were initially identified with record types and subclassing with subtyping (Hoare 1965). However, the assumption soon proved to be false; many attempts were made to find designs that made it true or ignored its falsity, but it was soon enough clear to be an impossible mirage. Without expressive-enough subtyping, prototypes are still possible, but their types will be very monomorphic. Users can still use them to store arbitrary data, by awkwardly emulating dynamic types on top of static types to achieve desired results.

This also makes them hard to type without subtypes.

Type descriptors are themselves often a monomorphic type that does not require subtyping, at least not unless the type system accommodates dependent types, or at least staging.

7 BLAH START (RE)WRITING FROM HERE

7.1 FOOOOOOOOOOOO

Mutation The performance optimizations and semantic issues related to mutability in OO.

Also, what the relationship between object systems that allow mutation of the inheritance DAG (Smalltalk, Self, CLOS) and their pure sematic models?

Inasmuch as mutation is seen as meaning “anything can become anything else at the drop of a hat”, then the static semantics of everything is essential trivial; there is total chaos and uncertainty in the mind of the software analyst. But inasmuch as mutation is seen as meaning “inheritance hierarchies are being set up before they are used, but don’t change while being used though they might change before, after and between uses”, with mutation happening at some notional meta-level or staging area with respect to the inheritance hierarchy, then the pure semantic model does help describe how the system behaves while the inheritance hierarchy is being used in a given locally unchanging state.

Now, if a system uses mutation to crucially modify “itself” in general and its inheritance hierarchy in particular while executing, then indeed the pure semantic model will prove insufficient to describe the behavior of the system. A more refined, lower-level model of how mutation of the inheritance hierarchy interferes with flow control in ongoing operations will become necessary. Yet the pure system remains a benchmark for how the system does or should behave in the extents during which the inheritance hierarchy was left undisturbed.

Monotonicity Why Subclassing is rarely Subtyping, and other questions of monotonicity, (co-, contra- and in-) variance in Functor Mixins and Fixed-Point Operators.

Typeclasses The relationship between Classes and Typeclasses. How typeclasses make object creation less ad hoc and more modular.

Autowrapping The relationship between Mutable or Immutable objects, linear typing and subtyping.

Optics The generalization of OO from overriding methods in records to overriding arbitrary aspects of arbitrary computations using functional lenses or zippers, and how this generalization can accommodate advanced OO practices like method combinations.

Method Combination, Instance Combination Specializing inheritance with respect to how increments are combined. generalizing precedence lists with DAG attribute grammars. Metaobject-compatibility.

Global Open Recursion A pure functional solution, already widely used in practice, yet neglected in the literature, to the problem of "multimethods", "friend classes" or "orphan typeclasses", and the according implications on designing and growing a language.

Multimethods (multiple dispatch) can enable more modular extension, but require constraints on the definition and use of methods after the fact. In particular, the ability to add methods retroactively change the shape of the method DAG, and thus make previous naive manual DAG joins ineffectual; more careful DAG joins (that explicitly take all directly inherited methods as parameters) can become tedious and costly to write and run. Automated DAG joins through reduction to a method monoid via a precedence list make a lot of sense in this context; no manual joins needed.

Meta-Object Protocols tying together all the bells and whistles in defining bindings, representations, objects, classes, methods, combinations, etc. We can adapt and generalize the techniques from AMOP in a pure functional setting.

Runtime Reflection Controlling Meta-Objects, from Synchronous Message-Passing Proxies to Fully Abstract Asynchronous Containers. We can only briefly survey this topic, maybe reusing the Collapsing Towers of Interpreters.

8 Conclusion

8.1 Related Work

As far as our bibliographical search goes, all these concepts have been largely or completely neglected by previous literature trying to provide formal semantics to OO (the kind that can be used for logical reasoning about objects): the underlying knowledge undoubtedly has existed for a long time, at least among implementers, yet it remained implicit or ad hoc rather than explicit and systematic. We are grateful to any reviewer, pre- or post- publication, who can pinpoint previous works that did make these concepts explicit, named them, and explained the relationship between the human factors and the formal model, and we will issue according addenda to our bibliography. We also welcome pointers to more informal literature that may have discussed these concepts without an attempt at formal semantics, though such works were unlikely to build a bridge between the two paradigms. In the end, the two paradigms OO and FP are as complementary as sums and products.

Note how Objects themselves appear only half way through the exposition, while Classes and Mutability appear even further. These are obviously all essential concepts to fully understand OO, yet they are not as *primitive* as the concepts introduced before them in a suitable theory of OO. Indeed, they are much more elaborate constructions, the semantics of which can be simple, clear and general when decomposing it into the preceding concepts, but hopelessly complex, confusing and ad hoc when failing to.

8.2 Parting Words

OO and FP are best friends. My concepts come with constructive implementations in terms of the λ-calculus either normal or applicative or both, pure or mutable, with or without (sub)types, with or without staging. Any λ-capable language can now be equipped with an Object System à la carte in a few tens of lines of code, the formal semantics of which can be nicely decomposed in a few orthogonal concepts. Say “No” to languages with missing or badly designed Object Systems — use our principled approach to build your own OO above or underneath them.

We hope that our explanations will convince some FP practitioners that OO both has sound meaning and practical value, despite many of them having only seen heaps of inarticulate nonsense in much of the OO literature. OO can be simply expressed on top of FP, and should be a natural part of the FP ecosystem.

Conversely, we hope that our explanations will convince some OO practitioners that OO can be given a simple formal meaning using solid general principles based on which they can safely reason about their programs, and not just vague informal principles and arbitrary language-specific rules. FP provides a robust foundation for OO, and should be a natural part of the OO ecosystem.

Bibliography

Martín Abadi and Luca Cardelli. *A theory of objects.* 1997. doi:10.1007/978-1-4419-8598-9

Norman Adams and Jonathan Rees. Object-Oriented Programming in Scheme. In *Proc. LFP*, 1988. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.954`

Eric Allen, Justin Hilburn, Scott Kilpatrick, Victor Luchangco, Sukyoung Ryu, David Chase, and Guy Steele. Type checking modular multiple dispatch with parametric polymorphism and multiple inheritance. In *Proc. OOPSLA*, 2011. `https://people.mpi-sws.org/~skilpat/papers/multipoly.pdf`

Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A Monotonic Superclass Linearization for Dylan. In *Proc. OOPSLA*, 1996. doi:10.1145/236337.236343

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. doi:10.5555/645894.671755

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. doi:10.1007/bf01806962

Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefyk, and Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *Proc. OOPSLA*, 1986. doi:10.1145/28697.28700

Ibrahim Bokharouss. GCL Viewer: a study in improving the understanding of GCL programs. Eindhoven University of Technology, 2008. `https://research.tue.nl/en/studentTheses/gcl-viewer`

Alan Hamilton Borning. ThingLab— an Object-Oriented System for Building Simulations using Constraints. In *Proc. 5th International Conference on Artificial Intelligence*, 1977. `https://www.ijcai.org/Proceedings/77-1/Papers/085.pdf`

Alan Hamilton Borning. ThingLab— A Constraint-Oriented Simulation Laboratory. PhD dissertation, Stanford University, 1979. `https://constraints.cs.washington.edu/ui/thinglab-tr.pdf`

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. doi:10.1145/357146.357147

Alan Hamilton Borning. Classes Versus Prototypes in Object-Oriented Languages. In *Proc. 1986 Fall Joint Computer Conference*, 1986. doi:10.5555/324493.324538

Gilad Bracha and William Cook. Mixin-Based Inheritance. In *Proc. OOPLSA/ECOOP*, 1990. doi:10.1145/97945.97982

Howard Cannon. Flavors: A non-hierarchical approach to object-oriented programming. 1982. `https://www.softwarepreservation.org/projects/LISP/MIT/nnnfla1-20040122.pdf`

Luca Cardelli. A Semantics of Multiple Inheritance. In *Proc. Information and Computation*, 1984. `https://api.semanticscholar.org/CorpusID:13032155`

C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proc. OOPSLA*, 1989. doi:10.1145/74877.74884

Craig Chambers. Object-oriented multi-methods in Cecil. In *Proc. ECOOP*, 1992. http://www.laputan.org/pub/papers/cecil-ecoop-92.pdf

William R. Cook. A Denotational Semantics of Inheritance. PhD dissertation, Brown University, 1989. https://www.cs.utexas.edu/~wcook/papers/thesis/cook89.pdf

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

Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. SIMULA 67 Common Base Language. 1968. https://web.archive.org/web/20131225084408/http://www.edelweb.fr/Simula/scb-1.pdf

Ole-Johan Dahl and Kristen Nygaard. SIMULA: An ALGOL-Based Simulation Language. *Commun. ACM* 9(9), pp. 671–678, 1966. doi:10.1145/365813.365819

Jack B. Dennis. Modularity. 1975. doi:10.1007/3-540-07168-7_77

Frank DeRemer and Hans Kron. Programming-in-the Large versus Programming-in-the-Small. In *Proc. Proceedings of the International Conference on Reliable Software*, 1975. doi:10.1145/800027.808431

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. https://edolstra.github.io/pubs/nixos-jfp-final.pdf

Matthias Felleisen. On the Expressive Power of Programming Languages. Rice University, 1991. https://www.cs.rice.edu/CS/PLT/Publications/Scheme/

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. https://www2.ccs.neu.edu/racket/pubs/asplas06-fff.pdf

Daniel P. Friedman, Christopher T. Haynes, and Mitchell Wand. *Essentials of Programming Languages (3rd Ed.)*. 2008. https://www.eopl3.com/

GitHub. The top programming languages. 2022. https://octoverse.github.com/2022/top-programming-languages

C.A.R. Hoare. Record handling. *Algol Bulletin* 21, pp. 39–69, 1965. http://archive.computerhistory.org/resources/text/Knuth_Don_X4100/PDF_index/k-9-pdf/k-9-u2293-Record-Handling-Hoare.pdf

Ecma International. *ECMAScript 2015 Language Specification*. 2015. http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf

Ralph Johnson and Joe Armstrong. Ralph Johnson, Joe Armstrong on the State of OOP (Interview at QCon). 2010. https://www.infoq.com/interviews/johnson-armstrong-oop/

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. doi:10.1145/1024273.1024278

Kenneth Michael Kahn. Creation of computer animation from story descriptions. PhD dissertation, MIT, 1979. https://dspace.mit.edu/handle/1721.1/16012

Samuel N. Kamin. Inheritance in Smalltalk-80: A Denotational Definition. In *Proc. POPL*, 1988. doi:10.1145/73560.73567

Mark Kantrowitz. Portable Utilities for Common Lisp. School of Computer Science, Carnegie-Mellon University, CMU-CS-91-143, 1991. https://cl-pdx.com/static/CMU-CS-91-143.pdf

Alan Kay. What thought process would lead one to invent Object-Oriented Programming? 2020. https://www.quora.com/What-thought-process-would-lead-one-to-invent-object-oriented-programming/answer/Alan-Kay-11

Shriram Khrisnamurthi. *Programming Languages: Application and Interpretation (1st Ed.)*. 2008. https://plai.org/

Gregor Kiczales, Jim Des Rivières, and Daniel Gureasko Bobrow. *The Art of the Meta-Object Protocol*. 1991. https://direct.mit.edu/books/book/2607/The-Art-of-the-Metaobject-Protocol

Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. *DAIMI Report Series*(229), 1987. https://www.semanticscholar.org/paper/The-BETA-Programming-Language-Kristensen-Madsen/3971897f708518e13809d3c0772105bd86526e50

Julia L. Lawall and Daniel P. Friedman. Embedding the Self Language in Scheme. 1989. https://legacy.cs.indiana.edu/ftp/techreports/TR276.pdf

Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proc. OOPLSA*, 1986. https://homepages.cwi.nl/~storm/teaching/reader/Lieberman86.pdf

Martin Odersky and Matthias Zenger. Scalable component abstractions. *SIGPLAN Not.* 40(10), pp. 41–57, 2005. doi:10.1145/1103845.1094815

D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15(12), pp. 1053–1058, 1972. doi:10.1145/361598.361623

Benjamin C. Pierce. *Types and Programming Languages*. 2002. https://www.cis.upenn.edu/~bcpierce/tapl/

Uday Reddy. Objects as Closures - Abstract Semantics of Object Oriented Languages. In *Proc. LFP*, 1988. doi:10.1145/62678.62721

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. https://www.researchgate.net/publication/221252249_T_a_dialect_of_Lisp_or_LAMBDA_The_ultimate_software_tool

Jonathan Allen Rees. A Security Kernel Based on the Lambda Calculus. PhD dissertation, Massachusetts Institute of Technology, 1995. http://mumble.net/~jar/pubs/secureos/

François-René Rideau. LIL: CLOS Reaches Higher-Order, Sheds Identity and has a Transformative Experience. In *Proc. International Lisp Conference*, 2012. http://github.com/fare/lil-ilc2012/

François-René Rideau. Build Systems and Modularity. 2016. https://ngnghm.github.io/blog/2016/04/26/chapter-9-build-systems-and-modularity/

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

François-René Rideau. Pure Object Prototypes. 2021. https://github.com/divnix/POP

François-René Rideau and Robert Goldman. Evolving ASDF: More Cooperation, Less Coordination. In *Proc. International Lisp Conference*, 2010. http://common-lisp.net/project/asdf/doc/ilc2010draft.pdf

François-René Rideau, Alex Knauth, and Nada Amin. Prototypes: Object-Orientation, Functionally. In *Proc. Scheme and Functional Programming Workshop*, 2021. `https://github.com/metareflection/poof`

Peter Simons. Nixpkgs fixed-points library. 2015. `https://github.com/NixOS/nixpkgs/blob/master/lib/fixed-points.nix`

Antero Taivalsaari. On the Notion of Inheritance. *ACM Comput. Surv.* 28(3), pp. 438–479, 1996. doi:10.1145/243439.243441

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL*, 1989. doi:10.1145/75277.75283

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