Lambda: the Ultimate Object

Object-Orientation Elucidated

François-René Rideau

January 8, 2026

This book is a work in progress. Please send feedback to fahree at gmail.[1]

As a software practitioner, you have not only heard of Object-Orientation (OO), but seen it or used it, loved it or hated it. Yet you may have been frustrated that there never seems to be clear answers as to what exactly OO is or isn't, what it is *for*, when and how to use it or not use it. There are many examples of OO; but everyone does it differently; every OO language offers an incompatible variant. There is no theory as to what common ground there is if any, even less so one on the best way to do OO. Certainly, none that two computer scientists can agree about. By comparison, you well understand Functional Programming (FP).

Can you explain OO in simple terms to an apprentice, or to yourself? Can you reason about OO programs, what they do, what they mean? Can you make sense of the tribal warfare between OO and FP advocates? Maybe you've enjoyed OO in the past, or been teased by colleagues who have, and are wondering what you are or aren't missing? Maybe you'd fancy implementing OO on top of the OO-less language you are currently using or building, but from what you know it looks too complicated? Indeed do you really understand why to implement which of no inheritance, single inheritance, mixin inheritance, or multiple inheritance? can you weigh the arguments for multiple inheritance done C++ or Ada style, versus Lisp, Ruby, Python or Scala style? Is there a best variant of inheritance anyway? And do concepts like prototypes, method combinations and multiple dispatch seem natural to you, or are they mysteries that challenge your mental model of OO? Last but not least... have you had enough of us Lispers bragging about how our 1988 OO system is still decades ahead of yours?

If any of these questions bother you, then this book is for you. This book offers a Theory of OO that it elucidates in simple terms on top of FP—as Intra-linguistic Modular Extensibility. A mouthful, but actually all simple concepts you already use, though you may not have clear names for them yet. This Theory of OO can answer all the questions above, and more. The answers almost always coincide with *some* existing academic discourse or industry practice; but obviously, they cannot possibly coincide with *all* the mutually conflicting discourses and practices out there; and, often enough, this theory will reject currently prevalent majority views and promote underrated answers.

But this Theory of OO is not just connecting previously known yet disparate lore; nor is it yet another *a posteriori* rationalization for the author's arbitrary preferences. This theory is *productive*, offering new, never before articulated ways to think about OO, based on which you can implement OO in radically simpler ways, in a handful of short functions you can write in any language that has higher-order functions; and it can *objectively* (hey!) justify every choice made. This theory reconciles Class OO, Prototype OO, and even a more primitive classless OO that few computer scientists are even aware exists. What is easily underappreciated, this theory can demarcate this common domain of OO from a lot of related but quite distinct domains that may look like OO and even share some of its vocabulary, yet can be shown to be conceptually foreign. The crown of this Theory of OO though is a new algorithm, C4, that allows combining single and multiple inheritance in a way that is better—and provably so—than the alternatives used in any programming language so far.

[1] For your convenience, a current draft is available in PDF at `http://fare.tunes.org/files/cs/poof/ltuo.pdf` and in HTML at `http://fare.tunes.org/files/cs/poof/ltuo.html`. The source code is at `https://github.com/metareflection/poof`.

Contents

Chapter 1

Introduction

1.1 Wherefore this Book

1.1.1 Curiosity about OO, Familiarity with FP

> It was probably in 1967 when someone asked me what I was doing, and I said: "It's object-oriented programming".
>
> Alan Kay

Object-Oriented Programming (OOP), or Object-Orientation (OO), is a paradigm for programming in terms of "objects".

What even are objects? What aren't? What is OO? What isn't? What is it for? What is a paradigm to begin with? How do I use OO to write programs? How do I make sense of existing OO programs? How may I best think about OO programs, to design them? How may I best reason about them, to debug them? What are variants of OO? How do I compare them? How can I build OO if I don't have it yet (or not a good variant of it)? And why are so many people so into OO, and so many others so against it? Which of their arguments is right or wrong when? Am I missing something by not using OO, or by using it?

These are the kinds of questions this book will help you answer. To get there, I will have to introduce many concepts. Don't worry: when others may relish in complexity, I instead aspire to simplicity. You will learn some new words and new ideas. But if you practice programming, and think about your practice, then you are in my target audience; and you will find it will be easier to program with the right ideas than with the wrong ones. And the Internet is certainly full of wrong and sometimes toxic ideas about OO and programming in general.

To answer those questions about OO, I will assume from my readers a passing familiarity with Functional Programming (FP). You don't have to be an expert at FP; you just need basic knowledge about how to read and write "anonymous higher-order"

functions in your favorite programming language.[1]

1.1.2 Decades too late, but still decades ahead

> Each new generation born is in effect an invasion of civilization by little barbarians, who must be civilized before it is too late.
>
> Thomas Sowell

Why write a book about OO in 2026? It is present year; don't people know everything they need to know (about OO or otherwise) by now, unlike the barbarians of times past? No, people of past years were not barbarians though they were ignorant of what we now know; and neither are we barbarians for failing to know what our successors will. Every mind is just too busy with knowledge from their time, that would have been useless earlier, and will soon be useless again.

Conceived around 1967 with Dahl and Nygaard's Simula and Alan Kay's musings, OO was actually born in 1976 when these and other influences collided, resulting in Smalltalk-76. OO took off from there, at first reserved to the happy few who could use the most high-end systems from Xerox or MIT. OO became popular among researchers in the 1980s, and at some point was the Next Big Thing™. In the 1990s, OO finally became available to every programmer, accompanied by endless industry hype to promote it. By the mid 2000s it had become the normal paradigm to program in. Then, at some point in the mid 2010s it started to become as boring as ubiquitous. Now in the mid 2020s it is on its way to become forgotten, at least among the Cool Kids. Yet, one thing OO never was, was understood. Until now.

As a USENET wiseman wrote: "Every technique is first developed, then used, important, obsolete, normalized, and finally understood." This book then is here to bring the final nail on the coffin of OO: understanding it. Too bad no one reads books anymore, except AIs. If that book, and most importantly its understanding, had come a few decades earlier, it could have saved a lot of people a lot of trouble. OO sure baffled me for a long time, and many around me. Now that I am not baffled anymore, I can bring you all my explanations—but long after the battle.

It would be nice to hear a few of my old colleagues tell me: "so *that* is what OO was about all along!" But I have little hope of convincing many in the old generations of the benefits of OO done right (yes, I am one of those Lispers bragging about how their 1988 OO system is still decades ahead of yours). And even if I did, they will be retiring soon. However, a new generation of programmers is born every year, and it is always time to inspire and educate the generation, that they do not fall as low as their

[1] Anonymous functions are just functions that do not need to have a name. Higher-Order means that they can take other functions as arguments, and return functions as results, both old and new (by e.g. applying or composing previous functions). These days, in 2026, most mainstream languages have such functions, quite unlike 20 years ago. It is also possible to emulate such functions in languages that do not have them yet; that is a topic I will not cover, but I can refer you to classic books about programming languages that do it well (Abelson and Sussman 1996; Friedman et al. 2008; Khrisnamurthi 2008; Pierce 2002; Queinnec 1996). I do love these books, however I find their treatment of OO lacking—otherwise I wouldn't be writing the present book.

predecessors, or lower. And even if the AIs take over programming, they too will need education.

1.1.3 Towards a Rebirth of OO

> If you want to build a ship, don't drum up the men to gather wood, divide the work, and give orders. Instead, teach them to yearn for the vast and endless sea.
>
> Antoine de Saint-Exupéry, creatively misquoted.

I actually think OO is a fantastic programming paradigm to build software, one that I tremendously enjoy when I use it in Lisp, and miss when I can't. But what passes for OO in mainstream programming languages disappoints me.

When I mention OO becoming boring, then forgotten, or talk of nailing its coffin, I'm not celebrating OO's decline. I'm lamenting that a bad version of OO became popular, then faded. Meanwhile, there's little interest or funding in either industry or academia to further improvement to OO—a topic wrongly considered already understood.

These days, bright programmers gravitate toward Functional Programming (FP), a paradigm unjustly neglected in the industry during OO's heyday. As distributed systems became widespread, FP proved more practical than imperative style for managing state and avoiding the problematic interactions that make concurrent programs slow and buggy. Because OO had been sold in a package deal with imperative programming, it fell out of fashion alongside it.

Grumpy old Lispers like me yell at the clouds that there was never an opposition between OO and FP—that we Lispers have been enjoying both together since the 1970s, and that OO can be so much more than you "blub" programmers can even imagine. My hope—my faith, even, despite available evidence—is that a better OO can and will rise from the dead: Simpler than what was once popular. Unbundled from imperative programming. More powerful. With the advanced features of Lisp OO at long last adopted by the mainstream.

But I have to admit my defeat so far: I have yet to build a system to my liking that would attract a critical mass of adopters to become self-sustaining. And so, like all researchers who title their papers "Towards a . . . " when they fail to achieve their goals, I am switching to plan B: trying to convince others that there's gold in them thar hills, so they will go dig it—because I can't dig it all by myself. In writing this book, I am sharing the treasure map with you.

1.2 Why this Book

> If there's a book you really want to read but it hasn't been written yet, then you must write it.
>
> Toni Morrison

This section is about me, not you. So feel free to skip to the next section. Come back if you're ever curious about the backstory of my theory of OO.

1.2.1 Proximate Cause

> The world will never starve for want of wonders, but for want of wonder.
>
> Gilbert K. Chesterton

After I used the Prototype OO programming language Jsonnet (Cunningham 2014) in production, then discovered that Nix (Simons 2015) implemented the very same object model in two lines of code, OO finally clicked for me. After all those years of experiencing how great or terrible OO in various forms could be, yet never quite being able to explain to myself or others what OO even was, certainly not in clear and simple terms—at last, I understood. And then I realized I might be the only one who understood OO from both the theoretical side of programming language semantics, and the practical side of actually building large systems—with the advanced features of untyped Lisp OO as well as with the advanced types of feature-poor OO systems like Scala's. At least the only one who cared enough to talk about it.

So I tried to get the Good News out, by getting a paper published. And I did get a paper published (Rideau et al. 2021), but only at the Scheme Workshop, a small venue of sympathetic Lispers, who already understood half of it and did not need much effort to understand the rest, but who already had plenty of good object systems to play with. Meanwhile, my repeated attempts at publishing in more mainstream Computer Science conferences or journals were met with incomprehension. Also with great technical feedback, of course, that helped me learn and improve a lot, and for which I am most grateful; but fundamentally, with deep misunderstanding about what I was even talking about. As Gabriel (2012) would say, my reviewers were trying to evaluate my work while making sense of it from an Incommensurable Paradigm.

Certainly, I could try to explain myself, to translate between their language and mine; spend time explaining the denotations and connotations of my words as I was using them, and defusing those that may mistakenly be heard by various readers from different communities; tell my readers to put aside the concepts they think they know, and somehow teach them the concepts I am putting behind the words, so they understand. But that takes a lot of time and space. For me. And for my readers. Resources that we both lack, especially scientific publications limited to 12-25 pages, depending on the venue. That was barely enough to address actual misunderstandings experienced by previous reviewers, and make my claims clear—see how much that takes in chapter 2 and chapter 3 respectively—with no space left to properly explain and subtantiate them. Attempts to compress that information into this kind of format would again lead to loss of clarity, and the inevitable misunderstanding by reviewers, and frankly, the readers they rightfully stand for.

Or, I could take the time and space to explain things right. But then, I'd have to abandon the hope of fitting in existing venues. I would have to write a book. This book.

1.2.2 Ultimate Cause

> Some mathematicians are birds, others are frogs. Birds fly high in the air and survey broad vistas of mathematics out to the far horizon. They delight in concepts that unify our thinking and bring together diverse problems from different parts of the landscape. Frogs live in the mud below and see only the flowers that grow nearby. They delight in the details of particular objects, and they solve problems one at a time. Manin is a bird. I happen to be a frog, but I am happy to introduce this book which shows us his bird's-eye view of mathematics.
>
> Freeman Dyson, in his foreword to Manin (2007)

I am not new to ideas that are hard to publish. My thesis on Reconciling Reflection and Semantics not only remains unpublished (Rideau 2018b), none of the many ideas within it could be published in an academic venue, except for a very short summary in a small workshop (Rideau 2018a). One explanation is that these ideas are hard to compress to fit within the size limits of publishable papers: out of the four parts in my thesis, the earlier parts can seem trivial, and mostly pointless (except for a few useful definitions and some cool insight), unless you understand the applications in the latter parts; but the applications in the latter parts seem impossible or don't even make sense unless I introduce the concepts from the first parts.

I repeatedly develop theories too large to publish in parts because I tend to think in terms of big pictures—or what others call big pictures, for I am also an aphantasiac: one with no mind's eye except when dreaming. A "bird", I like to explain large-scale human behaviors, or tie together seemingly disparate phenomena, by identifying common causal patterns that you can observe from "above". Ideas that are hard to communicate to "frogs", who don't think at a high enough level of abstraction. And I suspect that even other "birds" who do think at a high enough level, sometimes higher, not being aphantasiac, are overwhelmed or distracted by their visual imagery, and miss structural patterns I perceive non-visually.

However in the case of this book on Object-Orientation, there is the difference that I actually have three decades of practical as well as theoretical experience with OO. I studied the semantics of programming languages in college. I professionally wrote or maintained OO programs in Lisp, Python, Java, JavaScript, Jsonnet, Scala, C++. I kept writing papers about OO while working in the industry (Rideau 2012; Rideau et al. 2021). And for many years, I have been implementing OO, and maintaining two object systems for Gerbil Scheme (Rideau 2020; Vyzovitis 2016). OO is a topic both easier and more concrete, in general and for me in particular. A topic where I have direct experience as a frog, and where I can stand as a bird on the shoulders of giants who already solved many of the foundational problems. On this mature topic, I am ready and capable, and can explain a complete Theory of OO that is also fully implemented and immediately usable.

Ultimately, then, OO is the topic where my bird's view and frog's experience meet, where I made worthy findings that won't fit in a short publication, but that I can share in the form of a book.

1.3 What this Book

1.3.1 A Theory of OO

> There is nothing so practical as a good theory.
>
> Kurt Lewin

OO is the Paradigm of Programming with Inheritance. Despite what its name says, the actual central concept in Object-Orientation is *Inheritance*, a mechanism for programming by modularly extending partial specifications of code. OO usually depends on explicit support from the Programming Language (PL) at hand, then called an Object-Oriented (Programming) Language (OOPL).

This characterization of OO should be retrospectively obvious to all familiar with OO. Yet remarkably, some programmers explicitly reject it, eminent professors even![2] There is thus a need to elucidate the words and concepts of OO, behind the hype and confusion, and justify the choices made by OO (or then again, their amendment). Such is the main purpose of this book: to offer a *Theory of OO*.

This Theory of OO is Meaningful. A theory is *meaningful* if it is a body of explanations necessary and sufficient to explain what we do when we do OO, and we don't when we don't. It can clearly state the problems to be solved by or for OO, and the criteria by which we can judge some solutions as better than others, good or bad, acceptable or unacceptable. In these explanatory abilities, the negative is as important

[2] A notable dissident to this characterization is William Cook, a respected academic who made many key contributions to understanding the semantics of inheritance (Bracha and Cook 1990; Cook and Palsberg 1994; Cook and Palsberg 1989; Cook et al. 1989; Cook 1989) yet also argued that Inheritance was orthogonal to OO and that OO is about "classes" of "objects" that can only be accessed through "interfaces" (Cook 2009, 2012; Cook 1991).

However, coding against an SML module would count as OO by Cook's criteria, and indeed Cook explicitly calls the untyped λ-calculus "the first object-oriented language", while dismissing Smalltalk as not OO enough because its integers are not pure objects (Cook 2009). Cook's definition, that embraces the modular aspect of OO while rejecting its extensible or dynamic aspect, runs contrary to all practice. It brings no light on any of the languages commonly considered OO yet derided by Cook as not being OO enough, no light on any of the Functional Programming (FP) languages blessed by Cook as actually being OO to the surprise of their users, and no light on the difference between the two.

Cook's many works on OO over the years also systematically neglect important concepts in OO, such as prototypes, multiple inheritance, method combination or multiple dispatch. In the end, Cook's PhD and subsequent academic career grew out of brilliantly modeling the key mechanism of OO (Inheritance) from the foreign point of view of FP; but his lack of appreciation and understanding for the OO tradition, indeed missing the point of it all, were such, that they have become proverbial: immortalized in Gabriel's essay "The Structure of a Programming Language Revolution" (Gabriel 2012) as a prototypical failure to understand a phenomenon when viewed through a scientific paradigm incommensurable with the one that produced it. The problem is not just that Cook solved Inheritance as frog and failed to take the big picture as a bird: he did take a bird's view, and still couldn't see what his paradigm couldn't express.

Cook is well worth mentioning precisely to illustrate the lack of common vocabulary, common concepts, and common paradigms among those who practice and study OO, even or especially among notable academics with deep expertise in the field. And yet, there are undeniably common practices, common phenomena, common concepts, common language features, common design patterns, common goals, common aspirations, worth understanding, conceptualizing, defining and naming in the rich (though sometimes mutually conflicting) traditions that grew around OO.

as the positive: it allows to demarcate the concept of OO from other concepts; to distinguish that which partakes in it, that explains it, from that which doesn't, and would only corrupt it if accepted as part of OO.

This Theory of OO is Consistent: to remain *consistent*, a theory shall not contain internal contradictions. Consistency as such is easy: just avoid saying much, stick to tautologies and things already known for sure. Consistency, however, is much harder when you want to actually say a lot of useful things that interact with each other—which is why we want the theory to also be relevant.

This Theory of OO is Relevant A theory of OO is *relevant* to OO, and not of something else, if it matches the lore of OO: it will restate most if not all the things we know and care about OO, especially what has been well-known for decades.

To be both consistent and relevant at the same time in a lore full of controversial and mutually contradictory opinions, a theory must be critical when recalling the existing lore: it must organize the information, distinguish the wheat from the chaff, promoting some views deemed correct and denouncing incorrect ones, whether or not they are backed by majority opinions. Even if it contradicts the explanations of previous theories though, a theory must still account for and be consistent with the same observed and verifiable phenomena; or else it is a theory of something different altogether.

This Theory of OO is Productive. To be actually interesting, a theory must be *productive*: it must positively contribute new information. Just contributing new criteria to make sense of the existing lore can be enough to be productive. But I will go further and contribute more lore, too: useful ideas never published about OO, that make it simpler.

This Theory of OO is Constructive. While I will discuss informal principles, I will include running code in Scheme that you can easily adapt to your favorite programming language (see section 4.4.7 regarding choosing Scheme). I will explain which features are needed beyond the mere applicative λ-calculus, why, and how to typically implement them in existing programming languages. Remarkably, the main feature needed is lazy evaluation, or ways to emulate it, as OO is most naturally defined in a pure lazy functional setting, and eager evaluation of OO without side-effects leads to exponential recomputations.

One aspect for which I do not provide a construction, however, is static typing. I am a proficient user of types, but am no expert at the design, implementation or theory of types. Therefore I will only provide semi-formal designs for what better static types for OO should look like. And I will refer to the better papers among the many I have read, for what I believe are good foundations for typing OO (Allen et al. 2011; Eifrig et al. 1995b,a). Among other things, good OO types should work not just for Second-Class Classes, but also for First-Class Prototypes; they require recursive types, subtyping, and some form of existential types. Dependent types are not necessary.

1.3.2 Multiple Variants of Inheritance

> When you come to a fork in the road, take it.
>
> Yogi Berra

Now since nearly the very beginning of OO, there have been multiple variants of inheritance to choose from (Taivalsaari 1996). Many prefer Single Inheritance for its simplicity and performance (Dahl and Nygaard 1967; Kay 1993). Others prefer Multiple Inheritance, for its greater expressiveness and modularity, and this multiple inheritance itself comes in multiple flavors, notably divided on whether to use a technique called "linearization" (Bobrow and Winograd 1976; Cannon 1979). A few prefer Mixin Inheritance, a variant in some sense intermediary between the two above, but in another sense more fundamental, more composable (Bracha and Cook 1990).

With this variety of options, programmers (respectively programming language designers) face a choice of which of several variants of inheritance to use (respectively implement), if any at all.[3] Is there an objectively superior form of inheritance—whether an existing variant or some combination—with respect to expressiveness, modularity, extensibility, runtime performance, and whatever else might matter? Is one of the usual variants superior to the others in every way? If not, is there a combination of them, or a superset of them, that is? Some languages notably support forms of both single inheritance and multiple inheritance, though with some constraints: Lisp (Steele 1990), Ruby, Scala (Odersky and Zenger 2005). Would the best way to do inheritance subsume these combinations? If so, how does it relate to the multiple flavors of multiple inheritance?

And of course, critics of OO argue against using inheritance at all. What are the reasons to use or not use inheritance to begin with? I will use the absence of inheritance as a baseline against which to evaluate our variants.

1.3.3 Optimal Inheritance

> An extreme optimist is a man who believes that humanity will probably survive even if it doesn't take his advice.
>
> John McCarthy

I will claim that indeed (a) there is a best way to combine single and multiple inheritance, that (b) it involves linearization of the inheritance graph, that (c) there are enough constraints on linearization for the optimal algorithm to be well-defined up to some heuristic, and that (d) there are good reasons to prefer a specific heuristic. ***The C4 algorithm implements this Optimal Inheritance.*** I implemented C4 as part of the builtin object system of Gerbil Scheme (Vyzovitis 2016).[4]

[3] And then there are dubious variants published in obscure papers, that I will not discuss. Hopefully, after reading chapter 7, you will be able to understand why they are either trivially expressible in terms of the above variants, or fundamentally flawed, if you should come across them. That said, when, in old papers, I see pioneers struggling to find solutions to problems few if anyone else suspected existed, then make mistakes, and take wrong turns—I laugh, I cry, but I root for them. On the other hand, when, in more recent papers, I see researchers propose wrong solutions to problems that were solved long ago, I just shake my head, and express sadness that scientific communication and education are not working well.

[4] Scheme is a language with a rather minimalist definition. Dozens of mutually incompatible implementations of Scheme exist that each provide their own extensions on top of this common minimal core, with various degrees of compliance to various standards, to offer a usable programming environment. However,

It is unusual for a book to claim some significant innovation like that: usually, a researcher would publish it at some conference. However, C4 in isolation might only look mildly interesting: it "just" combines a couple well-known ideas and a speed optimization. The concepts I develop are a prerequisite for the claim of optimality of C4 to even make sense, yet require this book to properly articulate. C4 itself is a notable improvement, that crowns the theory as productive. But the theory behind it is the real achievement.

What that means for you in practice, though, is that a good theory brought you a better inheritance algorithm with which to improve your existing (or future) languages.

1.4 How this Book

1.4.1 Plan of the Book

> — Would you tell me, please, which way I ought to go from here?
> — That depends a good deal on where you want to get to.
>
> Lewis Carroll

Chapter 1, which you are presently reading, is an introduction to the book itself. The remaining chapters focus on OO itself.

In chapter 2, I dispel common misconceptions about OO, to ensure that my theory isn't met with misunderstanding due to misconceptions or disagreements about what is being theorized.

In chapter 3, I provide a quick overview of Object Orientation, and the three variants of inheritance in common use. This chapter serves as a map of the concepts and of the words I use to describe them, necessary because there is no common theory of OO, and no unambiguous shared vocabulary to name what common concepts there are. Importantly, I introduce the essential yet oft-ignored notion of Conflation between Specification and Target value. I then describe the relationship between Specifications, Prototypes, Classes and Objects.

In chapter 4, I explain what I mean by Internal Extensible Modularity, the rationale for OO. This chapter remains informal, but lays the conceptual groundwork for the formal approach I take in the rest of this book.

In chapter 5, I introduce minimal formal models of Modularity and Extensibility. Using pure Functional Programming (FP) as a foundation, with Scheme syntax, I derive from first principles a minimal OO system, in two lines of code. This minimal OO system uses mixin inheritance, and, remarkably, has neither objects nor prototypes, much less classes, only specifications and targets.

In chapter 6, I rebuild all the mainstream features and appurtenances of OO as additions or modifications to the minimal system from chapter 5: prototypes, classes,

there is no common object system, instead plenty of different object systems that span the entire design space for OO—except for their generally lacking static types. Gerbil Scheme provides its own builtin object system, not compatible with any standard, but with arguably the best inheritance of any object system to date (as of 2026).

types, mutation, etc. I notably clarify the all-too-common confusion between subtyping and subclassing, and discuss the actual relationship between OO and imperative programming, when the natural framework for OO is actually pure lazy functional programming.

In chapter 7, I discuss in detail the main forms of inheritance: single inheritance, multiple inheritance and mixin inheritance. I examine issues surrounding method conflict and resolution, or harmonious combination. I explain the known consistency constraints that matter for linearization algorithms in the context of multiple inheritance, and the state-of-the-art in satisfying them, the C3 algorithm. Finally, I discuss how to combine multiple and single inheritance, and examine the existing solutions adopted by Common Lisp (Steele 1990), Ruby, and Scala (Odersky and Zenger 2005). I then propose my solution, a linearization algorithm I call C4, that satisfies all the constraints of C3 plus those for combining single and multiple inheritance. I explain why the residual heuristic I also adopt from C3 is arguably the best one.

In chapter 8, I discuss more advanced topics including Focused Modular Extensions, Method Combination, Multiple Dispatch (Multimethods), Monotonicity, Orphan Typeclasses, and Global Fixpoints.

Finally, in chapter 9, I conclude by recapitulating my findings.

1.4.2 Stop and Go

> Begin at the beginning and go on till you come to the end: then stop.
>
> Lewis Carroll

This book has a mostly linear narrative: Every chapter builds on the previous ones. The end of every chapter is a nice place to stop reading, and restart later if you are still engaged.

The narrative goes from the most informal, most generic and most basic information about OO, to the most formal, most specific and most advanced. Some readers may prefer to stop when the material becomes more formal. Others may want to skip the informal discussion and jump directly to the formal parts at chapter 5, about a third into the book.

The most enthusiastic among you will read the book cover to cover, including footnotes and bibliographical notes, and go all the way into using and implementing the most advanced OO techniques of the later chapters (see chapter 8, chapter 9), end up building your OO system, and writing a sequel to this book. If you do, why not contact me and join me to build and write them together?

But you don't have to be that enthusiastic to read and hopefully enjoy this book. You may read casually, skip parts that don't interest you. You can look only at the section titles and the informal principles in bold italic. You can focus on the formal code definitions, or their type declarations. You can search for an argument on a specific controversy. You can scan the bibliography for more things to read.

What will enhance your experience, however, will be the ability to interact with a computer and play with the code. If you have an electronic copy, you may copy/paste

the code, use text search to compensate for the lack of a word index, click directly into the sections that interest you, and even ask AI assistants for navigation help.

1.4.3 Self-Description

> An adjective is autological if it describes itself (e.g., "short" is short). An adjective is heterological if it does not describe itself (e.g., "long" is not long). Now consider the adjective "heterological": Is it heterological?
>
> Grelling–Nelson paradox

This book includes enough self-descriptions ahead of each section that you hopefully may make reasonable decisions of which parts to read, to skip, to skim, to read attentively, to keep, to throw away—or, if you're ambitious, rewrite.

At times, I will highlight some short sentence in bold italic, to make it clear I believe it is an important principle, as follows: ***"If you have exceptions to your stated principle, ask yourself by what standard you make the exceptions. That standard is the principle you really hold."*** The above principle describes principles; in this case, it was written not by me, but by my friend Jesse Forgione.

I intend to include more examples in a future edition. But good examples take time to write, and space in the book; they can be too much for some readers and not enough for others. I am seeking the perfect concise teachable example in each case, that neatly illustrates what I mean without taking too much space or explanations. Until I find it, I must direct you to online resources, where OO code in general is abundant. If you are looking specifically for code that uses Prototype OO and multiple inheritance, you may look at my library Gerbil-POO (Rideau 2020): it provides a practical but short implementation of a prototype object system; and it builds interesting type descriptors on top of that object system, including a nice trie data structure, that is further specialized in the gerbil-persist library. AI assistants may also be able to find examples tailored to your needs. If you struggle with a particular concept that lacks an example, please tell me. And if you find good illustrative examples for ideas you or others struggled with, please send them my way.

1.4.4 Being Objectively Subjective

> Be… suspicious… of all those who employ the term 'we' or 'us' without your permission. This is [a] form of surreptitious conscription… Always ask who this 'we' is; as often as not it's an attempt to smuggle tribalism through the customs.
>
> Christopher Hitchens

You will see me using the first person singular a lot in this book. That doesn't mean I don't want to include you in my narrative. Believe me, I would most delight me if you could feel the same joy at exploring this topic as I do. Every sentence of this very

book is an invitation for you to see and practice OO my way. That doesn't mean I am bragging, either. That means I am taking responsibility for my actions.

Too many authors hide the responsibility for a decision among multiple authors (even when there is only one), or include the readers in a collective decision they did not make. Using an unwarranted "we" is a trick commonly used by conmen, narcissists and politicians (but I repeat myself), and I find it misleading even when done innocently. I once set myself and my friends a "cuss box" in which to drop a dollar when any of us did it, even on social media. But maybe Mark Twain put it best: "Only presidents, editors, and people with tapeworms have the right to use the editorial 'we'." And I don't think presidents have that right, either.

I will still say "we" on occasions, speaking for me and you readers, or for all humans. That "we" will then be passive, as things that we are, experience, or happen to us, or are constrained by the laws of logic and history: "we saw that example in a previous section" (you who read and I who wrote), "that experiment tells us" (us who saw it), "we cannot solve the termination problem" (we subject to logic). It will not be a trick to hide an action or decision that some among us made while shifting praise or blame or responsibility onto others: "we got one Nobel prize each—on average" (Marie Curie and I, but she did all the work), "we killed that poor man" (I did, but I'm trying to implicate you), "we must help that poor widow" (you all must, I'll take a large cut of the funds).

1.4.5 Technical Nomenclature

> When words are unfit, speech is unadapted and actions are unsuccessful.
>
> Confucius

As I restate well-known and less-known lore of Object Orientation, I will endeavor to precisely define the terms I use. Defining terms is especially important since various authors from diverse communities around many OO languages each use conflicting terminologies, with different words for the same concepts, or—which is worse—the same words for different concepts. This tower of Babel can cause much confusion when trying to communicate ideas across communities, as people ascribe conflicting assumptions and connotations to the words used by other people, and talk past each other while incorrectly believing they understand what the other said.

Thus, when multiple nomenclatures conflict, I will try to identify the *least ambiguous* word for each concept, even if it is neither the most popular word for the concept, nor the oldest, even if I sometimes make one up just for this book. Ideally, I can find a word that will be unambiguously understood by all my readers; but if that is not the case, I will prefer an awkward word that causes readers to pause and reflect, to a deceptively familiar word that causes them to unwittingly misunderstand the sometimes subtle points I make.

In particular, I will conspicuously avoid using the unqualified words "object" and "class" unless strictly necessary, because they carry different connotations for each reader, depending on their adopted traditions, that are at odds with the theory I am

laying out. I will also avoid the word "class" when talking about the most general kind of entity subject to inheritance, since a class is but a quite limited special case of a *prototype*, that is itself derivative of what I'll call a *specification*. I will define those terms precisely in chapter 3, chapter 4, chapter 5, chapter 6.

Chapter 2

What Object-Orientation is *not*

> It's not ignorance that does so much damage; it's knowing so darned much that ain't so.
>
> Josh Billings

Before I explain in detail what OO *is*, I shall cast aside a lot of things it *isn't* that too many people (both proponents and opponents) falsely identify with OO. This is important, because attempts at building or explaining a theory of OO often fail due to authors and readers having incompatible expectations about what OO is supposed to be.

If you find yourself shocked and in disagreement, that's fine. You don't have to agree at this point. Just consider that what I call OO and discuss at length in this book may be something slightly different from what you currently call OO. Then please allow me to narrow down what I mean, and make my argument. Or don't and close this book. But I hope you'll give my ideas a fair hearing.

2.1 OO isn't Whatever C++ is

> I made up the term 'object-oriented', and I can tell you I didn't have C++ in mind.
>
> Alan Kay, at OOPSLA '97 (near peak C++ popularity)

The most popular OO language in the decades that OO was a popular trend (roughly 1980 to 2010), C++ indeed supports some form of OOP. But C++ is a rich language with many aspects completely independent of OO (e.g. efficient bit-banging, RAII, template metaprogramming, pointer aliasing, a memory model), whereas the OO aspect that it undoubtedly offers is very different from how OO works in most other OO languages, and colloquial C++ often goes against the principles of OO. Therefore, C++

is in no way representative of OO in general, and if what you know of "Object Orientation" comes from C++, please put it aside, at least while reading this book, and come with a fresh mind.

This is especially true with regard to multiple inheritance, that will be an important topic later in this book. C++ boasts support for multiple inheritance, and many people, when thinking of multiple inheritance, think of what C++ offers. Yet, while C++ supports single inheritance well, what it calls "multiple inheritance" (Stroustrup 1989) is not at all the same as what almost everyone else calls "multiple inheritance":[1] It is actually a modified kind of mixin inheritance with some kind of "duplication" of superclasses (for non-`virtual` classes, with members renamed along the inheritance tree), and a subset of multiple inheritance (for `virtual` classes and members, with restriction from a "conflict" view of inheritance, see section 7.3.4). Notably, C++ lacks the proper method resolution that enables a lot of the modularity of multiple inheritance in other languages.

Now, you can use C++'s powerful template language to reconstitute actual mixin inheritance and its method resolution on top of C++'s weird variant of inheritance (Smaragdakis and Batory 2000); and you could no doubt further implement proper multiple inheritance on top of that.[2] But this technique is quite uncolloquial, syntactically heavy, slower than the colloquial ersatz, and programmers have to rigorously follow, enforce and maintain some complex design patterns.

Finally, and at the very least, consider that unless you explicitly tag your classes and their members `virtual`, C++ will deliberately eschew the "dynamic dispatch" of OO and use "static dispatch" instead for the sake of performance (at doing the wrong thing). In the end, C++ is many great and not-so-great things, but only few of those things are OO, and even most of those that look like OO are often different enough that ***C++ does not reliably inform about OO in general***.[3]

[1] Interestingly, the design of C++ non-virtual classes is very similar to the solution from Snyder's CommonObjects (Snyder 1986), even though Stroustrup does not cite Snyder: redefine the problem to be whatever the desired "solution" does—a Tree instead of a DAG—and hope the users won't notice the difference. On the other hand, Stroustrup does cite the Lisp Machine Manual (Weinreb and Moon 1981), and rejects Flavors because it is not "sufficiently simple, general and, efficient enough to warrant the complexity it would add to C++", which is exceedingly ironic considering Flavors was 1.4kloc (in October 1980, when cited), and C++ ~100kloc (in 1989, when citing), with Flavors having much richer and more general OO functionality than C++.

[2] One could achieve multiple inheritance as a design pattern on top of mixin inheritance, as I will describe later in this book, wherein developers would manually compute and specify each class's superclass precedence list; but this cancels some of the modularity benefits of multiple inheritance versus single and mixin inheritance. Alternatively, someone could extend the above technique to also reimplement the entire superclass linearization apparatus within the C++ template metaprogramming language. Template metaprogramming is most definitely powerful enough for the task, though it will take a very motivated developer to do the hard work, and the result will still be a burden for any developer who wants to use it. Moreover, for all that cost, classes defined that way would only interoperate with other classes following the exact same pattern. Maybe the library implementing the pattern could eventually be included in some semi-standard library, until, if it gets any traction, the language itself is eventually amended to do the Right Thing™.

[3] The situation is similar for Ada, that adopted multiple inheritance in 2003 by seemingly copying the general design of C++. Now even when C++ got multiple inheritance wrong, ignorance was no valid excuse, since Lisp got it right ten years earlier (Cannon 1979) and Stroustrup even cited it via (Weinreb and Moon 1981). Ignorance is even less excusable in the case of Ada copying C++'s "multiple inheritance" yet 14 years later. By contrast, many languages got it right in the same time frame, including Python (1991), Ruby (1995), Scala (2004).

2.2 OO isn't Classes Only

> The class/instance distinction is not needed if the alternative of using prototypes is adopted.
>
> Lieberman (1986)

Many claim that classes, as first implemented by Simula 67 (Dahl and Nygaard 1967) (though implementing a concept previously named by Hoare (Hoare 1965)), are essential to OO, and only ever care to implement, use, formalize, study, teach, promote, or criticize class-based OO (a.k.a. Class OO). Books from luminaries in Programming Languages (Friedman et al. 2008; Khrisnamurthi 2008; Pierce 2002), in their chapter about OO, barely even mention any other kind of OO if at all, much less study it.

Yet KRL (Bobrow and Winograd 1976; Winograd 1975), the second recognizable precursor to OO, whose authors introduced the words "inheritance" and "prototypes" with the same meaning as in OO in the context of their language (though the words were initially used as descriptions rather than definitions), has what I would now call prototype-based OO (a.k.a. Prototype OO). The modern concept of OO can be traced back to Smalltalk adopting inheritance in 1976, naming inheritance after KRL's usage, and popularizing the word and concept of it among programming language designers (KRL, a layer on top of Lisp, is arguably not a *programming* language, though it integrates with one). Certainly, Smalltalk was class-based. Yet contemporary with Smalltalk or immediately after it were prototype-based languages Director (Kahn 1976, 1979a,b) and ThingLab (Borning 1977, 1979, 1981).[4] Plenty more Prototype OO or "class-less" OO languages followed (Adams and Rees 1988; Chambers et al. 1989; Cunningham 2014; Hewitt et al. 1979; Lawall and Friedman 1989; Rees and Adams 1982; Rideau et al. 2021; Salzman and Aldrich 2005; Simons 2015). There are a lot more Prototype OO languages than I could have time to review (Wikipedia 2025b), but prominent among them is JavaScript (Eich 1996), one of the most used programming languages in the world (GitHub 2022), maybe the top one by users (though it relatively recently also adopted classes on top of prototypes (International 2015)).

Moreover, I will argue that Prototype OO (Borning 1986) is more general than Class OO, that is but a special case of it (Lieberman 1986) (see section 3.2.2, section 6.2). And I will even argue that you can recognizably have OO with neither prototypes nor classes, as in T (Adams and Rees 1988) (see section 3.3, chapter 5, chapter 6). Despite common misinformed opinions to the contrary, ***Class-less OO is part and parcel of the OO tradition***, historically, conceptually, and popularly.

Now of course, classes, while not *essential* to OO, are still *important* in its tradition. The situation is similar to that of types in Functional Programming (a.k.a. FP): the historical preexistence and continued relevance of the untyped λ-calculus and the wide adoption of dynamically typed functional languages like Scheme or Nix are ample evidence that types are not essential to FP; yet types are undoubtedly an important

[4] ThingLab was built on top of Smalltalk by members of the same team at PARC, and oscillated between having or not having classes in addition to prototypes.

topic that occupies much of the theory and practice of FP. Actually, the analogy goes further since, as we'll see, classes are precisely an application of OO to types (see section 3.2, section 6.2).

2.3 OO isn't Imperative Programming

> Objects are a poor man's closures.
>
> Norman Adams

> Closures are a poor man's objects.
>
> Christian Queinnec

Many people assume that OO requires mutation, wherein all attributes of all objects should be mutable, or at least be so by default, and object initialization must happen by mutation. Furthermore, they assume that OO requires the same applicative (eager) evaluation model for procedure calls and variable references as in every common imperative language. On the other side, many now claim 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 (they say) even more incompatible with OO.

However, there are many good historical reasons, related to speed and memory limitations at both runtime and compile-time, why early OO and FP languages alike, from the 1960s to the 1980s, as well as most languages until relatively recently, were using mutable state everywhere, and an eager evaluation model, at least by default. And with 1990s slogans among Lispers like "objects are a poor man's closures" (Dickey 1992), and "closures are a poor man's objects" (Queinnec 1996), the problem back then was clearly not whether OO could be done purely with functions (obviously it could), but whether it made practical sense to program purely without side-effects in general. That question would only be slowly answered positively, in theory in the early 1990s (Moggi 1991) and in practice in the mid 2000s to mid 2010s, as Haskell grew up to become a practical language.[5]

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

[5]Some may identify darcs (2003) as the first widely used real-world application written in Haskell. After it came innovations such as bytestring (2005), cabal (2005) (and the "cabal hell" it started causing around 2006 until later solved by stack), ghc6 (2006), that made Haskell much more practical to use, and new notable applications appeared like pandoc (2006), or xmonad (2007). A turning point was perhaps the publication of "Real World Haskell" (O'Sullivan et al. 2008). Eventually, Stack (2015) made non-trivial haskell programs and scripts repeatable. Now there's obviously a lot of subjectivity in deciding when exactly Haskell became "practical"—but one should expect the transition to practicality to be an S curve, such that whichever reasonable yet somewhat arbitrary threshold criteria you choose, the answer would be at about the same time. In any case, making a practical language pure functional was just not an option before 2010 or so, and it is absurd to declare any programming language concept intrinsically stateful merely because all its practical implementations before 2010 were stateful. You could similarly make the absurd claim that logic programming, functional programming, or linear algebra are intrinsically stateful.

dynamic OO languages such as Jsonnet or Nix (Cunningham 2014; Dolstra and Löh 2008; Simons 2015), and pure lazy OO systems for Scheme (Rideau et al. 2021), (c) languages happily combining OO and FP such as Common Lisp or Scala, with plenty of libraries restricting themselves to pure functional objects only (Chiusano and Bjarnason 2014; Rideau 2012), and (d) last but not least, Oleg Kiselyov's implementation of OO, even stateful OO if you want, in the pure FP language Haskell(!) (Kiselyov and Lämmel 2005).

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. Actually, I will argue based on studying of the semantics of OO that ***Pure Lazy Functional Programming is the natural setting for OO***.

2.4 OO isn't Encapsulation

> A half-truth is a whole lie.
>
> Yiddish proverb

Many OO pundits claim that an essential concept in OO is "encapsulation" or "information hiding" (DeRemer and Kron 1975). Some instead speak of "data abstraction" or some other kind of "abstraction". There is no consensus as to what this or these concepts mean, and no clear definition, but overall, these words refer either (a) to part or all of what I call *modularity* (see section 3.1.2, section 4.1), or (b) to some specific set of visibility primitives in some OO languages.

Indeed, "encapsulation" usually denotes the ability to code against an interface, with code on either side not caring which way the other side implements its part of the interface, not even being able to distinguish between multiple such implementations, even less to look inside at the state of the other module. Viewed broadly, this is indeed what I call modularity, which in my theory is indeed half of the essence of OO. But the word modularity much better identifies the broader purpose, beyond a mere technical property. And even then, modularity only characterizes half of OO, so that people who try to equate OO with that half only crucially miss the other half—*extensibility* (see section 3.1.3, section 4.2)—and thus fail to properly identify OO.

Now, insofar as some people identify encapsulation narrowly as the presence of specific visibility mechanisms such as found in C++ or Java (with some attributes or methods being `public`, `private` or something in-between, whose precise semantics the designers of different languages cannot agree on), I'll easily dismiss such mechanisms as not essential to OO, since 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 (Rees 1995).

Certainly, these mechanisms can be very useful, worthy features to add to an OO language. They are just not essential to OO and not specific to it, though of course their adaptation to OO languages will follow the specific shape of OO constructs not found

in non-OO languages. Misidentifying OO as being about these mechanisms rather than about the modularity they are meant to support can only lead to sacrificing the ends to the means.

2.5 OO isn't opposite to FP

> ¿Por qué no los dos? (Why not both?)
>
> Old El Paso

Some argue that there is an essential conflict between OO and FP, between Inheritance and Composition, wherein OO is about modeling every possible domain in terms of inheritance, and FP is about modeling every possible domain in terms of composition, and the two must somehow duel to death.

But OO and FP, inheritance and composition, are just pairs of distinct concepts. Neither of which subsumes the other; each fits a distinct set of situations. It makes no sense to oppose them, especially not when I see that OO can be implemented in a few lines of FP, whereas most modern OO languages contain FP as a subset—and Lisp has harmoniously combined OO and FP together ever since they both emerged in the 1970s, decades before anyone had the idea to fantasize a conflict between the two.

The argument is actually a distortion of a legitimate question of OO design, wherein one has to decide whether some aspect of a class[6] embodied as attributes or methods, should be included directly in the class (a) by inheriting from another class defining the aspect (the class *is-a* subclass of the aspect class—inheritance of classes), or (b) indirectly by the class having as an attribute an object of that other class (the class *has-a*n attribute of the aspect class—composition of classes seen as 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 extensible approach is more likely to enable adapting the software to changing situations, at which point thoughtful uses of

[6] My counter-argument also works for prototypes or arbitrary OO specifications, but since the argument is usually given for classes, I will use classes in this section.

inheritance can help a lot[7] [8].

In the end, ***OO and FP are complementary, not opposite***. If there is a real opposition, it is not between two perfectly compatible techniques, but between two mindsets, between two tribes of programmers each locked into their narrow paradigm (Gabriel 2012) and unable to comprehend what the other is saying.

2.6 OO isn't Message Passing

> Name the greatest of all inventors. Accident.
>
> Mark Twain

Alan Kay, who invented Smalltalk and coined the term "Object-Oriented Programming" circa 1967 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 to Algol by Simula (Dahl and Nygaard 1966). It is also present in notable early object systems such as Director (Kahn 1976, 1979a,b) and ThingLab (Borning 1977, 1979, 1981).

However, neither Simula nor Smalltalk nor any popular OO language actually fits that metaphor. Some less popular Actor languages might (Hewitt et al. 1979), but they remain marginal in the tradition. Instead, the only widely-used language to truly embody this metaphor is Erlang (Johnson and Armstrong 2010); yet Erlang is

[7] *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 only 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 three, or let your compiler's optimizer unify the two variables as you initialize them from the same computation. Now if you know your program will evolve to become 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`.

[8] There is also an old slogan of OO design, notably found in the famous "Gang of Four" ("GoF") book (Gamma et al. 1994), that you should "favor object composition over class inheritance". GoF argues not to create an exponential number of subclasses that specialize based on static information about what is or could be a runtime value, because classes are compile-time and human-developer-time objects that are less flexible and costlier in human effort than runtime entities. These arguments of course do not apply for regular Prototype OO, wherein umpteen combinations of prototypes (and classes as a particular case) can be generated at runtime at no additional cost in human effort. Still, the point can be made that if a programmer is confused about which of is-a or has-a to use in a particular case, it's a good heuristic to start with has-a, which will quickly lead to an obvious showstopper issue if it doesn't work, whereas picking is-a where has-a was the better choice can lead to a lot of complications before it is realized that it won't work right. Yet it is always preferable to understand the difference between "is" and "has", and to use the correct one based on understanding of the domain being modeled, rather than on vague heuristics that substitute for lack of understanding. At any rate, this slogan, though oft quoted out of context in online debates, actually has nothing to do with the OO vs FP debate—it is about using OO effectively.

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 unacknowledged by the OO community. Indeed Erlang crucially lacks inheritance, or support for the "extreme late binding of all things" that Alan Kay also once mentioned was essential for OO.[9]

Moreover, many OO languages generalize and extend their method dispatch mechanism from "single dispatch" to "multiple dispatch" (Allen et al. 2011; Bobrow et al. 1988; Bobrow et al. 1986; Chambers 1992). Their "multimethods" are attached to tuples of prototypes or classes, and there is no single prototype, class, or single independent entity of any kind capable of either "receiving" or "sending" a message. Instead, they are attached to a "generic function" that handles the dispatch based on the types of its arguments.[10] While multimethods are obviously not essential to OO since there are plenty of OO languages without them, they are a well-liked, age-old extension in many OO languages (CLOS, CECIL, Dylan, Fortress, Clojure, Julia) and extensions

[9]In Erlang, each process is a dynamic pure applicative functional language enriched with the ability to exchange messages with other processes. Now, as we'll see, you need fixpoints to express the semantics of OO; but in a pure applicative context, you cannot directly express sharing the results of a computation, so the pure fixpoint combinators lead to exponential recomputations as deeper self-references are involved (see section 5.2.6). OO is therefore possible using the applicative pure functional fragment of the language within an Erlang process, but the result will not scale very well; see for instance the example "object-via-closure" that Duncan McGreggor wrote as part of LFE. Or OO could be achieved indirectly, by using a preprocessor that expands it away, or a compile-time only extension to the compiler, as in most static Class OO languages. Or OO could be achieved as a design pattern of maintaining some global table to store the state of the many shared lazy computations in each process. Or, more in line with the Actor model that Erlang embodies, OO could be achieved by spawning one or multiple processes for each shared lazy or stateful computation (including each super-object of each object), which might require some strict object lifetime discipline (not colloquial in Erlang), or garbage collection of processes (not part of the Erlang language, beyond the process tree); see for instance the example "object-via-process" that Duncan McGreggor wrote as part of LFE. None of these solutions would qualify as supporting OO much more than assembly language "supports" OO or any Turing-universal language "supports" any paradigm, though. In the end, the essence of OO, which is Prototype OO, directly fits in the pure lazy functional paradigm, but only fits indirectly in other paradigms, including the pure applicative functional paradigm.

[10]The "generic function" functionality from the Common Lisp Object System (CLOS) can be viewed as isomorphic to the "protocols" functionality of Clojure; and Common Lispers also use the word "protocol" informally to designate a set of generic functions. They would in turn be isomorphic to the "typeclasses" of Haskell or the "traits" of Rust... if only these latter two supported inheritance, which they don't. These idioms all denote a set of related function names and type signatures, that are implemented differently for different configurations, where each configuration is associated to *one or multiple* types of arguments (and, in Haskell, also different types of expected results). Other crucial property of these idioms: these traits, typeclasses or protocols can be defined *after the fact*, so that new traits, typeclasses or protocols can be defined for configurations of existing types, and new types can be added to existing typeclasses, etc. This second property is in sharp contrast with "interfaces" in Java or C#, wherein the author of the class must specify in advance all the interfaces that the class will implement, yet cannot anticipate any of the future extensions that users will need. Users with needs for new protocols will then have to keep reinventing variants of existing classes, or wrappers around existing classes, etc. — and again when yet another protocol is needed. Protocols are therefore much more modular than Java-style "interfaces", and more extensible than Rust "traits" or Haskell "typeclasses", making them modular at a finer grain (protocol extensions rather than protocol definitions), which in turn makes them more modular. Note also how what Rust recently popularized as "trait" is something completely different from what Smalltalk, and after it Mesa or Scala, call "trait". In these languages, with an anterior claim to the word, a "trait" is just a class that partakes in multiple inheritance, defining a single type and associated methods, and not after the fact. Once again, be careful that there is no common vocabulary across programming language communities.

exist for C++, Java, JavaScript, TypeScript, C#, Python, Ruby, etc. The "message passing" paradigm, having no place for multimethods, thus falls short compared to other explanations of OO that accommodate them.[11]

In conclusion, whatever historical role it may have had in inspiring the discovery of OO, ***the paradigm of message-passing processes is wholly distinct from OO***, with its own mostly disjoint tradition and very different concerns, that describes a different set of programming languages and patterns.[12]

2.7 OO isn't a Model of the World

> If you call a tail a leg, how many legs has a dog? Five? No!
> Calling a tail a leg doesn't make it a leg.
>
> Abraham Lincoln, explaining the difference between lexical scoping and dynamic scoping

Some have claimed that OO is meant to be *the* way to model the world, or at least *a* way, often in association with the concurrent message passing model I already established above was not quite OO, or with some class-based OO framework they sell.

However, while OO can indeed be of great use in modeling a lot of problems, especially where the modeling language needs modularity and extensibility, it by no means is supposed to be a Theory of Everything that subsumes Relativity and Quantum Mechanics, Constitutional Law, Darwinism, Aristotelian Poetics, etc. Even if I stick to software, there are plenty of paradigms other than OO that OO does not subsume: functional programming, logic programming, machine learning, operational research, relational databases, reactive programming, temporal logic, concurrent programming, dataflow programming, homomorphic encryption, etc. Inasmuch as OO languages can be used to implement any of these paradigms, so can any Turing Tar-Pit. And inasmuch

[11] Now, the message passing paradigm can be extended with a notion of "group messaging" where one object sends a "message" to a "group" of objects as a collective entity (rather than each member of the target group) or to a "chemical" paradigm where a "chemical reaction" may involve multiple entities in and multiple entities out, with "message" entities conveying the changes in intermediary steps. But even with these extensions to the paradigm, you would still have to also specifically shoe-horn extensibility and method resolution into the paradigm to fit OO and its method inheritance, whether with single dispatch or multiple dispatch.

[12] Now, there is no doubt, from their later testimonies as well as then published papers, that Erlang's Concurrency Oriented Programming is clearly what the authors of Simula, Smalltalk, Actors, etc., were all *aiming at*. But, due to hardware as well as software limitations of the 1960s and 1970s, they all failed to actually reach that goal until the mid 1980s. However, on their way to an intended destination, they instead serendipitously stumbled on something altogether different, inheritance, that would soon become (pun intended) a vastly successful programming language feature, as often misunderstood, abused and hated as understood, well-used and loved, that came to define a new style of programming, called "Object-Oriented Programming".

That's how invention always works: if you knew beforehand what you would discover, you would already have discovered it. An invention is always surprising, original, and never, ever, exactly what you knew in advance it would be—or else the invention happened earlier and *then* was surprising and original. Also, an invention is shaped by the technical constraints of the time—some of which the inventor may lift, but not always those anticipated.

as any of these paradigms can be harmoniously combined with OO, that does not make either a subset of the other. People seriously studying OO should not take at face value the claims of Snake Oil and Silver Bullet salesmen, either about what their products can do, or about whether these products indeed embody OO. Mostly, they do not.

Consider methodologies such as UML that claim to do OO modeling, drawing diagrams of relations between classes including inheritance. Besides the fact that classes are not essential to OO as seen previously, UML and similar languages do not even meaningfully have classes: there is no proper semantics to inheritance, especially in presence of fields that recursively refer back to a class: should the child class have a link to the parent class or to the child class? Assume a classic case of modeling humans as animals, wherein animals can have offspring that are animals of the same kind: Should human offspring be modeled as arbitrary animals, or should they be modeled as human only? Conversely, if some animals eat other animals, does that mean that humans automatically eat humans, or only some other animals? In presence of recursion, UML falls apart, by failing to distinguish between subclassing and subtyping, between self-reference and reference to a constant (see section 6.3).

Interestingly, Amílcar Sernadas or Bart Jacobs's categorical theories of "objects" and "inheritance" (Costa et al. 1994; Jacobs 1995, 1996) actually model UML and refinement, and not at all actual objects and inheritance as used in Programming Languages; a hijacking of the same words for completely different meanings, with the only similarity being that both sets of meanings involve arrows between specifications. At least Jacobs explicitly embraces early on the limitation whereby self-reference or recursion is prohibited from field definitions. Just like UML, his co-algebra utterly fails to model OO; but at least his theory is internally consistent if not externally.

UML, co-algebras and other similar methodologies are actually relational data modeling disguised as OO. As we'll see later, their "classes" are extensible indeed, but in a trivial way that fails to support modularity.[13] UML and co-algebras describe the "easy case" of OO, where objects are just a convenient way of merging records of elementary data types (or "constant" data types, for co-algebras) — an easy case without recursion, where subclassing indeed coincides with subtyping. But these methodologies avoid crucial features of OO programming, where records can recursively refer to other records, where the operations of interest are higher-level than getting or setting fields, where you incrementally extend not just data types but also algorithms, where there are "binary methods" that involve two objects at once, or even more elaborate higher-order functions, etc. More broadly, these methodologies lack any effective semantics of inheritance, of method resolution in computing properties of objects along their class hierarchies, or of anything that has the precision required to specify code that can actually run and be reasoned about. But specifying code is exactly where the

[13]Note that there is nothing wrong at all with relational data modeling as such: it is a fine technique for many purposes, despite being deliberately limited in abstraction, and, therefore, in modularity—and sometimes *thanks to this limitation*. Restrictions to expressiveness can be very useful, in the necessarily restricted or imprecise cases that they apply. Indeed, in some cases, relational data modeling, not OO, is what you need to organize your data and your code. Moreover, Category Theory are a great way to improve on previous approaches to relational data, as witness by the field of Categorical Databases. However, what is very wrong, and intellectually dishonest, was to sell relational data modeling as OO back when OO was trendy, based on a superficial and at times deliberate misunderstanding of OO by either or both sellers and buyers, resulting in more confusion.

conceptual difficulties and gains of OO are both to be found with respect to software construction. In fact, these handwaving methodologies[14] are specifically designed to fool those incapable or unwilling to wrestle with computation into believing they understand all there is to know about software modeling. Yet the nature and correctness of software lies precisely in this gap they are unable or unwilling to explore.

An actual theory of types for OO must confront not just products of elementary data types, but sum types, function types, subtyping, constrained type parameters, existential and universal types, and more—including, especially, fixpoints (recursion). And you can always go beyond with session types, substructural types, temporal types, separation types, dependent types, etc. In the end, if you care about modeling the types in your software (and you usually should), you should write your software in a language with a rich and strong type system, one that is logically consistent or at least whose inconsistencies are well mapped and can be avoided, one that is statically enforced by the compiler or at least that you will systematically enforce socially. Then you should use that type system to describe not just records of elementary data types over the wire or on disk, but all the rich entities within your software, their interactions and interrelations. This will provide much more help with design and safety than any code-less methodology can. And if you picked an OO-capable language like C++, Java, C# or Scala, (or, with manually enforced dynamic types, Lisp, Ruby or Python), you can actually use OO as you do it.

[14]Not all uses of Category Theory in "OO" are handwaving. Goguen, who invokes Category Theory in his papers, and is cited by these later categorical imitators, develops precise formal specification of code, refinement of such specifications, and actual implementation of executable code. On the other hand, Goguen's claim to do "OO" is dubious, despite some attempts in later works to retrofit some actual OO concepts into his systems; instead, what he developed turns out to be a completely different and orthogonal paradigm—term rewriting. Term rewriting is a wonderfully interesting paradigm to study, but has never seen any adoption for practical programming, though it has found use in reasoning about programs. What Goguen calls "inheritance" most of the time is actually code refinement, a technique that can be used to build proven-correct compilers, though it is not a general theory of code implementation applicable to arbitrary such compilers.

Chapter 3

What Object-Orientation *is* — Informal Overview

> Ce qui se conçoit bien s'énonce clairement,
> Et les mots pour le dire arrivent aisément.
> (What is clearly conceived is clearly expressed,
> And the words to say it flow with ease.)
>
> Nicolas Boileau

In this chapter, I map out the important concepts of OO, as I develop in the rest of this book. I provide only what explanation is strictly necessary to relate concepts to one another and prevent their misinterpretation. More details, and justifications, will follow in subsequent chapters.

3.1 Extensible Modular Specifications

> Say what you mean and mean what you say.
>
> Lewis Carroll

Object-Orientation ("OO") is a technique that enables the specification of programs through extensible and modular *partial* specifications, embodied as entities *within* a programming language (see chapter 4, chapter 5).

3.1.1 Partial specifications

A program is made of many parts that can be written independently, enabling division of labor, as opposed to all logic being expressed in a single monolithic loop.[1]

[1]The entire point of partial specifications is that they are not complete, and you want to be able to manipulate those incomplete specifications even though of course trying to "instantiate" them before all the

3.1.2 Modularity (Overview)

A programmer can write or modify one part (or "module") while knowing very little information about the contents of other parts, enabling specialization of tasks. Modularity is achieved by having modules interact with each other through well-defined "interfaces" only, as opposed to having to understand in detail the much larger contents of the other modules so as to interact with them.

3.1.3 Extensibility (Overview)

A programmer can start from the existing specification and only need contribute as little incremental information as possible when specifying a part that modifies, extends, specializes or refines other parts, as opposed to having to know, understand and repeat existing code nearly in full to make modifications to it.

3.1.4 Internality

Those partial programs and their incremental extensions are entities *inside* the language, as opposed to merely files edited, preprocessed or generated *outside* the language itself, which can be done for any language.

3.2 Prototypes and Classes

> SIR, I admit your gen'ral Rule
> That every Poet is a Fool:
> But you yourself may serve to show it,
> That every Fool is not a Poet.
>
> Alexander Pope (or Jonathan Swift), adapting a French quip

3.2.1 Prototype OO vs Class OO

These in-language entities are called *prototypes* if first-class (manipulated at runtime, and specifying values, most usually records), and *classes* if second-class (manipulated at compile-time only, and specifying types, most usually record types). A language offers prototype-based object orientation ("Prototype OO") if it has prototypes, and class-based object orientation ("Class OO") if it only has classes.

The first arguably OO language used classes (Dahl and Nygaard 1967), but the second one used prototypes (Winograd 1975), and some provide both (International 2015). Class OO is the more popular form of OO, but the most popular OO language, JavaScript, started with Prototype OO, with Class OO only added on top twenty years later.

information has been assembled should fail. Type systems and semantic frameworks incapable of dealing with such incomplete information are thereby incapable of apprehending OO.

3.2.2 Classes as Prototypes for Types

A class is a compile-time prototype for a type descriptor: a record of a type and accompanying type-specific methods, or some meta-level representation thereof across stages of evaluation.

Class OO is therefore a special case of Prototype OO, which is therefore the more general form of OO (Lieberman 1986; Rideau et al. 2021). And indeed within Prototype OO, you can readily express prototypes for runtime type descriptors for your language, or prototypes for type descriptors for some other language you are processing as a meta-language.

In this book, I will thus discuss the general case of OO, and thus will seldom mention classes, that are a special case of prototypes. Exceptions include when I elucidate the relationship between Class OO and Prototype OO, or mention existing systems that are somehow restricted in expressiveness so they only support classes.[2] Still my discussion of OO, and my exploration of inheritance in particular, directly applies just as well to the special case that is Class OO.

3.2.3 Classes in Dynamic and Static Languages

Dynamic languages with reflection, such as Lisp, Smalltalk or JavaScript, can blur the distinction between runtime and compile-time, and thus between Prototype OO and Class OO. Indeed many languages implement Class OO on top of Prototype OO exactly that way, with classes being prototypes for type descriptors.

Static languages, on the other hand, tend to have very restricted sublanguages at compile-time with termination guarantees, such that their Class OO is vastly less expressive than Prototype OO, in exchange for which it is amenable to somewhat easier static analysis. A notable exception is C++, that has a full-fledged Pure Functional Lazy Dynamically-Typed Prototype OO language at compile-time: templates. On the other hand, Java and after it Scala are also Turing-universal at compile-time, but not in an intentional way that enables practical metaprogramming, only an unintentional way that defeats guarantees of termination (Grigore 2016).

3.3 More Fundamental than Prototypes and Classes

> Simplicity does not precede complexity, but follows it.
>
> Alan Perlis

[2] There would be plenty to discuss about classes and types from the point of view of library design and ecosystem growth, patterns that OO enables and antipatterns to avoid, tradeoffs between expressiveness and ease of static analysis, etc. There are plenty of existing books and papers on Class OO, OO software libraries, techniques to type and compile OO, to get inspiration from, both positive and negative. While some of the extant OO lore indeed only applies to defining classes, a great deal of it easily generalizes to any use of inheritance, even without classes, even without prototypes at all, even when the authors are unaware that any OO exists beyond Class OO.

3.3.1 Specifications and Targets

As I reconstruct the semantics of OO from first principles, I will see that more so than prototype, class, object, or method, ***the fundamental notions of OO are*** **specification** ***and*** **target**: *target* computations are being specified by extensible and modular, partial *specifications* (see chapter 5).

The target can be any kind of computation, returning any type of value. In Prototype OO, the target most usually computes a record, which offers simpler opportunities for modularity than other types. In Class OO, the target is instead a record *type*, or rather a type descriptor, record of a record type and its associated methods, or of their compile-time or runtime representation.

A (partial or complete) specification is a piece of information that (partially or completely) describes a computation. Multiple specifications can be combined together into a larger specification. From a complete specification, a target computation is specified that can be extracted; but most specifications are incomplete and you can't extract any meaningful target from them. A specification is modular inasmuch as it can define certain aspects of the target while referring to other aspects to be defined in other specifications. A specification is extensible inasmuch as it can refer to some aspects of the target as partially defined so far by previous specifications, then amend or extend them. In Prototype OO, access to this modular context and to the previous value being extended are usually referred to through respective variables often named `self` and `super`. In Class OO, such variables may also exist, but there is an extra layer of indirection as they refer to an element of the target type rather than to the target itself.

3.3.2 Prototypes as Conflation

A specification is not a prototype, not a class, not a type, not an object: it is not even a value of the target domain. It is not a record and does not have methods, fields, attributes or any such thing.

A target value in general is not an object, prototype or class either; it's just an arbitrary value of that arbitrary domain that is targeted. It needs not be any kind of record nor record type; it needs not have been computed as the fixpoint of a specification; indeed it is not tied to any specific way to compute it. It cannot be inherited from or otherwise extended according to any of the design patterns that characterize OO.

Rather, ***a prototype (a.k.a. "object" in Prototype OO, and "class" but not "object" in Class OO) is the*** **conflation** ***of a specification and its target***, i.e. an entity that depending on context at various times refers to either the specification (when you extend it) or its target (when you use its methods).

Formally speaking, a conflation can be seen as a cartesian product with implicit casts to either component factor depending on context. In the case of a prototype, you'll implicitly refer to its specification when speaking of extending it or inheriting from it; and you'll implicitly refer to its target when speaking of calling a method, looking at its attributes, reading or writing a field, etc.

3.3.3 Modularity of Conflation

In first-class OO without conflation, developers of reusable libraries are forced to decide in advance and in utter ignorance which parts of a system they or someone else may want to either extend or query in the future. Having to choose between specification and target at every potential extension point leads to an exponential explosion of bad decisions to make: Choosing target over specification everywhere would of course defeat extensibility; but choosing specification over target for the sake of extensibility, especially so without the shared computation cache afforded by conflation, would lead to an exponential explosion of runtime reevaluations. By letting programmers defer a decision they lack information to make, the conflation of specification and target is an essential pragmatic feature of Prototype OO: it increases intertemporal cooperation between programmers and their future collaborators (including their future selves) without the need of communication between the two (that would require time-travel). Thus, in first-class OO, ***Conflation Increases Modularity***.

This modularity advantage, however, is largely lost in static languages with second-class Class OO.[3] In such languages, all class extensions happen at compile-time, usually in a restricted language. Only the type descriptors, targets of the specifications, may still exist at runtime (if not inlined away). The specifications as such are gone and cannot be composed or extended anymore. Thus, you can always squint and pretend that conflation is only a figure of speech used when describing the system from the outside (which includes all the language documentation), but not a feature of the system itself. However, this also means that in these static OO languages, conflation of specification and target is but a minor syntactic shortcut with little to no semantic benefit. Yet its major cost remains: the confusion it induces, not just among novice developers, but also experts, and even language authors.

3.3.4 Conflation and Confusion

Conflation without Distinction is Confusion. Those who fail to distinguish between two very different concepts being conflated will make categorical errors of using one concept or its properties when the other should be used. Inconsistent theories will lead to bad choices. Developers will pursue doomed designs they think are sound, and eschew actual solutions their theories reject. At times, clever papers will offer overly simple solutions that deal with only one entity, not noticing an actual solution needed to address two. In rare cases, heroic efforts will lead to overly complicated solutions that correctly deal with both conflated entities at all times. Either way, incapable of reasoning correctly about OO programs, developers will amend bad theories with myriad exceptions and "practical" recipes, rather than adopt a good theory that no one is offering. Murky concepts will lead to bad tooling. Confused developers will write subtle and persistent application bugs. Researchers will waste years in absurd quests and publish nonsense along the way, while fertile fields lay unexplored. See the NNOOTT

[3] Note that this does not include dynamic Class OO languages like Lisp, Smalltalk, Ruby or Python. In these languages, compilation can keep happening at runtime, and so programmers still benefit from Conflation.

(section 6.3.3) regarding decades of confusion between subtyping and subclassing due to confusing target (subtyping) and specification (subclassing).

Still when you clearly tease the two notions apart, and are aware of when they are being conflated for practical purposes, so you can distinguish which of the two aspects should be invoked in which context, then the semantics of OO becomes quite simple. Shockingly, conflation was first explicitly discussed only in Rideau et al. (2021) even though (a) the concept is implicitly older than OO, going at least as far back as Hoare (1965), and (b) the implementation of various Prototype OO systems has to explicitly accommodate for it (see e.g. the `__unfix__` attribute in Simons (2015)) even when the documentation is silent about it.

3.4 Objects

> Computer Science is no more about computers than astronomy is about telescopes.
>
> E. W. Dijkstra

3.4.1 An Ambiguous Word

In Prototype OO, a prototype, conflation of a specification and its target, is also called an "object" or at times an "instance", especially if the target is a record. Note that laziness is essential in computing the target record or its entries, since most specifications, being partial, do not specify a complete computation that terminates without error in finite time, yet this expected non-termination should not prevent the use of the conflated entity to extract and extend its specification (see section 6.1).

In Class OO, a prototype, conflation of a specification and its target, is instead called a "class", and the target is specifically a type descriptor rather than an arbitrary record, or than a non-record value. What, in Class OO, is called an "object" or an "instance" is an element of a target type as described. A class being a prototype, its regular prototype fields and methods are called "class fields" or "class methods" (or "static" fields and methods, after the keyword used in C++ and Java)—but be mindful that they only involve the target type, not the specification. "Object methods" are semantically regular methods that take one implicit argument in front, the object (i.e. element of the target type). "Object fields" are regular fields of the object as a record (see section 6.2).

Finally, many languages, systems, databases, articles or books call "object" some or all of the regular runtime values they manipulate:[4] these "objects" may or may not be records, and are in no way part of an actual OO system extensible with inheritance.

[4] Alan Kay, in his Turing Award lecture, remarks: "By the way, I should mention that, you know, the name, the term object predates object-oriented programming. Object, in the early 60s, was a general term that was used to describe compound data structures, especially if they had pointers in them."

The authors will not usually claim that these objects are part of an OO framework actual or imagined, but then again sometimes they may.[5].

3.4.2 OO without Objects

Furthermore, the fundamental patterns of OO can exist and be usefully leveraged in a language that lacks any notion of object, merely with the notions of specification and target: Indeed, Yale T Scheme has a class-less "object system" (Adams and Rees 1988), wherein the authors call "object" any language value, and "instance" those records of multiple function entry points used as the non-extensible targets of their extensible specifications, themselves called "components", that use mixin inheritance (see chapter 5, section 6.1.1).

Therefore ***the word "object" is worse than useless when discussing OO in general***. It is actively misleading. It should never be used without a qualifier or outside the context of a specific document, program, system, language, ecosystem or at least variant of OO, that narrows down the many ambiguities around its many possible mutually incompatible meanings.[6] Meanwhile, the word "class" is practically useless, denoting a rather uninteresting special case of a prototype. Even the word "prototype", while meaningful, is uncommon to use when discussing OO in general. If discussing inheritance, one will only speak of "specifications". And if discussing instantiation, one will speak of "specification" and "target". Prototypes only arise when specifically discussing conflation. To avoid confusion, I will be careful in this book to only speak of "specification", "target", "prototype", and (target type) "element" and to avoid the words "object" or "class" unless necessary, and then only in narrowly defined contexts.[7]

This is all particularly ironic when the field I am studying is called "Object Orientation", in which the most popular variant involves classes. But fields of knowledge are usually named as soon as the need is felt to distinguish them from other fields, long before they are well-understood, and thus based on misunderstandings; this misnomer is thus par for the course.[8]

On the other hand, this book is rare in trying to study OO in its most general form. Most people instead try to *use* OO, at which point they must soon enough go from the general to the particular: before a programmer may even write any code, they have to pick a specific OO language or system in which to write their software. At that point,

[5]This situation can be muddled by layers of language: Consider a language without OO itself implemented in an OO language. The word "object" might then be validly denote OO from the point of view of the implementer using the OO meta-language, yet not from the point of view of the user using the non-OO language. Conversely, when an OO language is implemented using a non-OO language, calling some values "objects" may validly denote OO for the user yet not for the implementer.

[6]It's a bit as if you had to discuss Linear Algebra without being able to talk about lines, or had to discuss Imperative Programming without being able to talk about the Emperor. Ridiculous. Or then again just an artefact of etymology.

[7]I am however under no illusion that my chosen words would remain unambiguous very long if my works were to find any success. They would soon be rallying targets not just for honest people to use, but also for spammers, cranks, and frauds to subvert—and hopefully for pioneers to creatively misuse as they make some unforeseen discovery.

[8]The wider field of study is similarly misnamed. E. W. Dijkstra famously said that Computer Science is not about computers. Hal Abelson completed that it is not a science, either.

the context of the language and its ecosystem as wide as it is, is plenty narrow enough to disambiguate the meanings of all those words. And likely, "object", and either or both of "prototype" or "class" will become both well-defined and very relevant. Suddenly, the programmer becomes able to utter their thought and communicate with everyone else within the same ecosystem... and at the same time becomes more likely to misunderstand programmers from other ecosystems who use the same words with different meanings. Hence the tribal turn of many online "debates".

3.5 Inheritance Overview

> Discovery consists of seeing what everybody has seen and thinking what nobody has thought.
>
> Albert Szent-Györgyi

3.5.1 Inheritance as Modular Extension of Specifications

Inheritance is the mechanism by which partial modular specifications are incrementally extended into larger modular specifications, until the point where a complete specification is obtained from which the specified target computation can be extracted.

There have historically been three main variants of inheritance, with each object system using a variation on one or two of them: single inheritance, multiple inheritance and mixin inheritance. For historical reasons, I may speak of classes, subclasses and superclasses in this subsection, especially when discussing previous systems using those terms; but when discussing the general case of prototypes and specifications, I will prefer speaking of specifications and their parents, ancestors, children and descendants—also specifications.

3.5.2 Single Inheritance Overview

Historically, the first inheritance mechanism discovered was *single inheritance* (Dahl and Nygaard 1967), though it was not known by that name until a decade later. In an influential paper (Hoare 1965), Hoare introduced the notions of "class" and "subclass" of records (as well as, infamously, the `null` pointer). The first implementation of the concept appeared in Simula 67 (Dahl and Nygaard 1967). Alan Kay later adopted this mechanism for Smalltalk 76 (Ingalls 1978), as a compromise instead of the more general but then less well understood multiple inheritance (Kay 1993). Kay took the word "inheritance" from KRL (Bobrow and Winograd 1976; Winograd 1975), a "Knowledge Representation Language" written Lisp around Minsky's notion of Frames. KRL had "inheritance of properties", which was what we would now call "multiple inheritance". The expressions "single inheritance" and "multiple inheritance" can be first be found in print in Stansfield (1977), another Lisp-based frame system. Many other languages adopted "inheritance" after Smalltalk, including Java that made it especially popular circa 1995.

In Simula, a class is defined starting from a previous class as a "prefix". The effective text of a class (hence its semantics) is then the "concatenation" of the direct text of all its transitive *prefix classes*, including all the field definitions, method functions and initialization actions, in order from least specific superclass to most specific.[9] In modern terms, most authors call the prefix a superclass in general, or direct superclass when it is syntactically specified as a superclass by the user. I will be more precise, and after Snyder (1986), I will speak of a parent for what users explicitly specify, or, when considering transitively reachable parents of parents, an ancestor; and in the other direction, I will speak of child and descendant. The words parent and ancestor, in addition to being less ambiguous, also do not presume Class OO. I will call *inheritance hierarchy* of a specification, the set of its ancestors, ordered by the transitive parent relation. When using single inheritance, this hierarchy then constitutes a list, and the ancestor relation is a total order.

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. Moreover, calls to the "super method" also have a simple obvious meaning. 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 and less understood alternatives.

[9] Simula later on also allows each class or procedure to define a "suffix" as well as a "prefix", wherein the body of each subclass or subprocedure is the "inner" part between this prefix and suffix, marked by the `inner` keyword as a placeholder. This approach by Simula and its successor Beta (Kristensen et al. 1987) (that generalized classes to "patterns" that also covered method definitions the same way) is in sharp contrast with how inheritance is done in about all other languages, that copy Smalltalk. The "prefix" makes sense to initialize variables, and to allow procedure definitions to be overridden by latter more specialized definitions; the "suffix" is sufficient to do cleanups and post-processing, especially when procedures store their working return value in a variable with the same name as the procedure being defined, in Algol style. The entire "inner" setup also makes sense in the context of spaghetti code with GOTOs, before Dijkstra made everyone consider them harmful in 1968, as well as return values controlled by assigning a value to a variable named after the function. But frankly, it is both limited and horribly complex to use in the post-1968 context of structured code blocks, not to mention post-1970s higher-order functions, etc., even though in Beta you can still express what you want in a round-about way by explicitly building a list or nesting of higher-order functions that re-invert control back the way everyone else does it, that you call at the end. And so, while Simula was definitely a breakthrough, its particular form of inheritance was also a dead-end; no one but the Simula inventors want anything resembling `inner` for their language; no other known language uses it: after Smalltalk, languages instead let subclass methods control the context for possible call of superclass methods, rather than the other way around). Beta behavior is easily expressible using user-defined method combinations in e.g. CLOS (Bobrow et al. 1988), or can also be retrieved by having methods explicitly build an effective method chained the other way around. Thus I can rightfully say that inheritance, and OO, were only invented and named through the interaction of Bobrow's Interlisp team and Kay's Smalltalk team at PARC circa 76, both informed by ideas from Simula, and Minsky's frames, and able to integrate these ideas in their wider respective AI and teachable computing experiments thanks to their dynamic environments, considerably more flexible than the static Algol context of Simula. In the end, Simula should count as a precursor to OO, or at best an early draft of it, but either way, not the real, fully-formed concept. Dahl and Nygaard never invented, implemented, used or studied OO as most of us know it: not then with Simula, not later with Beta, and never later in their life either. Just like Columbus never set foot on the continent of America. Yet they made the single key contribution thanks to which the later greater discovery of OO became not just possible, but necessary. They rightfully deserve to be gently mocked for getting so close to a vast continent they sought yet failing to ever set foot on it. But this only makes me admire them more for having crossed an uncharted ocean the vast extent of which no one suspected, beyond horizons past which no one dared venture, to find a domain no one else dreamed existed.

Even today, most languages that support OO only support single inheritance, for its simplicity.

3.5.3 Multiple Inheritance Overview

Discovered a few years later, and initially just called *inheritance*, in what in retrospect was prototype OO, in KRL (Bobrow and Winograd 1976; Winograd 1975), Multiple inheritance allows a specification (frame, class, prototype, etc.) to have multiple direct parents. The notion of (multiple) inheritance thus predates Smalltalk-76 (Ingalls 1978) adopting the term, retroactively applying it to Simula, and subsequently inventing the terms "single" and "multiple" inheritance to distinguish the two approaches as well as recognize their commonality.

Although some more early systems (Bobrow and Stefik 1983; Borning and Ingalls 1982; Borning 1977; Curry et al. 1982; Goldstein and Bobrow 1980) used multiple inheritance, ***multiple inheritance only became usable with the epochal system Flavors*** (Cannon 1979; Weinreb and Moon 1981), refined and improved by successor Lisp object systems New Flavors (Moon 1986), CommonLoops (Bobrow et al. 1986) and CLOS (Bobrow et al. 1988; Steele 1990). Since then, many languages including Ruby, Perl, Python and Scala correctly adopted the basic design of Flavors (though none of its more advanced features)—I will call them *flavorful*.[10] On the other hand, influential or popular languages including Smalltalk, Self, C++ and Ada failed to learn from Flavors and got multiple inheritance largely wrong—I will call them *flavorless*.

With multiple inheritance, the structure of a specification and its ancestors is a Directed Acyclic Graph ("DAG"). The set of all classes is also a DAG, the subclass relation is a partial order. Most OO systems include a common system-wide base class at the end of their DAG; but it is possible to do without one.[11]

The proper semantics for a specification inheriting multiple different methods from a non-linear ancestry proved tricky to get just right. The older (1976) "flavorless" viewpoint sees it as a conflict between methods you must override. The newer (1979) "flavorful" viewpoint sees it as a cooperation between methods you can combine. Unhappily, too many in both academia and industry are stuck in 1976, and haven't even heard of the flavorful viewpoint, or, when they have, still don't understand it. For this reason, despite its being more expressive and more modular than single inheritance, flavorful multiple inheritance still isn't as widely adopted..[12]

[10] To be fair, these languages all include the capability for a method to call a super-method, that was not directly possible in Flavors (1979) without writing your own method-combination, but only introduced by CommonLoops (1986) with its run-super function, known as call-next-method in CLOS (1988).

[11] Indeed, Flavors allowed you not to include the system-provided `vanilla-flavor` at the end of its precedence list, so you could write your own replacement, or do without.

[12] Out of the top 50 most popular languages in the TIOBE index, 2025, 6 support flavorful multiple inheritance (Python, Perl, Ruby, Lisp, Scala, Solidity), 3 only support flavorless multiple inheritance (C++, Ada, PHP), 2 require a non-standard library to support multiple inheritance (JavaScript, Lua), 17 only support single inheritance (Java, C#, VB, Delphi, R, MATLAB, Rust, COBOL, Kotlin, Swift, SAS, Dart, Julia, TypeScript, ObjC, ABAP, D), and the rest don't support inheritance at all (C, Go, Fortran, SQL, Assembly, Scratch, Prolog, Haskell, FoxPro, GAMC, PL/SQL, V, Bash, PowerShell, ML, Elixir, Awk, X++, LabView, Erlang).

3.5.4 Mixin Inheritance Overview

Mixin inheritance was discovered last (Bracha and Cook 1990), probably because it relies on a more abstract pure functional view of OO; maybe also because it was one of the first successful attempts at elucidating inheritance in the paradigm of programming language semantics, when the concept had previously been developed in paradigm of computing systems (Gabriel 2012). Yet, for the same reasons, ***Mixin Inheritance is more fundamental than the other two variants***. It is the simplest kind of inheritance to formalize *given the basis of FP*, in a couple of higher-order functions. Specifications are simple functions, inheritance is just chaining them, and extracting their target computation is just computing their fixpoint. Mixin inheritance also maps directly to the concepts of Modularity and Extensibility I am discussing, and for these reasons I will study it first when presenting a formal semantics of OO (see chapter 5).

Mixins equipped with a binary inheritance operator constitute a monoid (associative with neutral element), and the inheritance structure of a specification is just the flattened list of elementary specifications chained into it, as simple as single inheritance. Mixin inheritance works better at runtime, either with Prototype OO, or for Class OO in a dynamic and somewhat reflective system.

Mixin inheritance is in some way simpler than single inheritance (but only if you understand FP yet are not bound by limitations of most of today's FP type systems), and as expressive as multiple inheritance (arguably slightly more, though not in a practically meaningful way), but is less modular than multiple inheritance because it doesn't automatically handle transitive dependencies but forces developers to handle them manually, effectively making those transitive dependencies part of a specification's interface.

For all these reasons adoption of mixin inheritance remains relatively limited, to languages like StrongTalk (Bak et al. 2002; Bracha and Griswold 1993), Racket (Flatt et al. 2006; Flatt et al. 1998), Newspeak (Bracha et al. 2008), GCL (Bokharouss 2008), Jsonnet (Cunningham 2014), and Nix (Simons 2015). Yet it still has outsized outreach, for just the use of GCL at Google means a large part of the world computing infrastructure is built upon configurations written using mixin inheritance. One may also construe the way C++ handles non-"virtual" repeated superclasses as a form of mixin inheritance with automatic renaming, at which point mixin inheritance is actually very popular, just not well-understood.

3.5.5 False dichotomy between Inheritance and Delegation

Many authors have called "Delegation" the mechanism used by Prototype OO (Hewitt et al. 1979), as distinct from the "inheritance" mechanism of Class OO. However, Lieberman, in one of the papers that popularized this dichotomy (Lieberman 1986), discusses the two joined concepts of Prototype-Delegation vs Class-Inheritance,[13] and explains in detail how classes and their "inheritance" can be expressed as a special case of prototypes and their "delegation", while classes cannot express prototypes. He

[13] Lieberman also explores at length the philosophical underpinnings of these two idea-complexes, which is quite interesting from a historical and psychological point of view, but has little relevance to a technical discussion on the semantics of OO, its proper use or implementation decades later.

gives multiple examples of behaviors expressible with prototypes but not with classes, wherein prototypes enable dynamic extension of individual "objects" (prototypes) at runtime, while classes only allow extension at compile-time, and only for an entire type ("class") of "objects" (elements of the type). But while it is extremely important indeed to understand the distinction and the relationship between the general notion of prototypes from the special case of classes, it only begets confusion to treat Inheritance and Delegation as separate concepts when they have identical concerns of semantics, implementation or performance, whether used for prototypes versus classes, when the word "inheritance" had already been used by "frames" and other "classless" objects, that would later be dubbed "prototypes" in the early 1980s, and it is much more useful to see classes as a special case of prototypes that partake in the very same mechanism of inheritance.[14]

One confounding factor is that of mutable state. Early OO, just like early FP, was usually part of systems with ubiquitous mutable state; prototype inheritance (or "delegation") algorithms thus often explicitly allow or cope with interaction with such state, including mutation and sharing or non-sharing of per-object or per-class variables, and especially tricky, mutation and sharing of a prototype's inheritance structure. However, class systems often had all their inheritance semantics resolved at compile-time, during which there is no interaction with user-visible side-effects, and it doesn't matter whether the compiler does or doesn't itself use mutable state: from the user point of view it is as if it were pure functional and there is no mutation in the inheritance structure or state-sharing structure of classes, at least not without using "magic" reflection primitives. One may then have been tempted then to see Prototype Delegation as intrinsically stateful, and class inheritance as intrinsically pure (though at compile-time).

Yet, recent pure functional Prototype OO systems (Cunningham 2014; Rideau et al. 2021; Simons 2015) prove constructively that prototypes can be pure, and that they use the very same inheritance mechanisms as classes, indeed with classes as a particular

[14]If irrelevant changes in the context are a valid excuse to give an existing concept a new name and get a publication with hundreds of citations based on such a great original discovery, I here dub "ainheritance" the concept of "inheritance" when the name of the entity inheriting from others starts with "a", "binheritance" the concept of "inheritance" when the programmer's console is blue, "cinheritance" the concept of "inheritance" when the programmer is in China, and "sinheritance" the concept of "inheritance" when the specification is not conflated with its target, therefore neither a class nor a prototype. Also "ninheritance" when there is no actual inheritance, and "tinheritance" when it looks like inheritance, but is not real inheritance, just target extension without open recursion through a module context. I am also reserving the namespace for variants of the name starting with a heretofore unused letter, unicode character, or prefix of any kind, and launching the Interplanetary Xinheritance Foundation to auction the namespace away, as well as the related Intergalactic Zelegation Alliance. I am impatiently awaiting my Turing Award, or at least Dahl Nygaard prize, for all these never discussed before original inventions related to OO.

The Lieberman paper deserves its thousands of citations because it is a great paper. However, a lot of citers seem to only fixate on the unfortunate choice of concept delineation and naming by Lieberman, who probably did not anticipate that he would set a bad trend with it. The delineation made sense in the historical context of the Actor team separately implementing prototypes and classes with related yet distinct mechanisms in their ACT1 language (Hewitt et al. 1979), way before they or anyone understood how classes were a special case of prototypes. But too many readers took this historical artifact as an essential distinction, and thereafter focused on studying or tweaking low-level "message passing" mechanisms on a wild goose chase for tricks and features, instead of looking at the big picture of the semantics of inheritance, what it actually is or should be and why, what is or isn't relevant to its semantics. Concept delineation and naming is tremendously important; it can bring clarity, or it can mislead hundreds of researchers into a dead end.

case of prototypes with the usual construction. Meanwhile, old reflective Class OO systems like Lisp and Smalltalk (Gabriel et al. 1991; Kahn 1976; Kay 1993; Kiczales et al. 1991) also support mutable state to modify the inheritance structure at runtime, for the sake of dynamic redefinition of classes at runtime, in what remains semantically a pure functional model once when the structure is set. See how in CLOS you can define methods on generic function `update-instance-for-redefined-class` to control how data is preserved, dropped or transformed when a class is redefined. Mutable state and mutable inheritance structure in particular are therefore clearly an independent issue from prototypes vs classes, though it might not have been obvious at the time. As I introduce formal models of OO, I will start with pure functional models (see chapter 5), and will only discuss the confounding matter of side-effects much later (see section 6.4).[15]

As for which words to keep, the word "inheritance" was used first for the general concept, in a language with "prototypes", KRL (Winograd 1975). The word "delegation" stems from the Actor message-passing model, and is both later and less general, from after the words "inheritance" and "prototypes" were better established, and is strongly connoted to specific implementations using the message-passing paradigm. It also fell out of fashion some time in the 1990s, after JavaScript became a worldwide phenomenon, and (correctly) used the term "inheritance" rather than delegation (as it isn't particularly "message passing", just calling functions). (ECMA-262 1997)

[15]It might be interesting to explain *why* previous authors failed so hard to identify Delegation and Inheritance, when the similarities are frankly obvious, and the relationship between classes and prototypes is well-known to anyone who implemented classes atop prototypes. But lacking direct access to those authors' brains, my explanations must remain speculative.

First, pioneers are eager to conceptualize and present their experiments as original and not just the same concept in a different context. They necessarily have to sell their ideas as historical package deals, before the underlying concepts are clearly identified and separated from each other. They are too close to the matter to tell which of the features they built would be immortalized through the ages as fundamental concepts vs just contingent implementation details soon to be forgotten. In the brief time that publishing about Prototypes was trendy, scientists studying pioneering works may have focused too much on the specifics of Actors, Self, or other successful Prototype language du jour, and failed to properly conceptualize a general notion of Prototype. Unlike the pioneers themselves, they deserve blame for their myopia, and so do the followers who cite and repeat their "findings" without criticism. However this explanation is not specific to the topic at hand, and is valid for every field of knowledge.

Second, and with more specificity to Prototypes, Computer Scientists following the Programming Language (PL) paradigm (Gabriel 2012) might have been incapable of unifying Prototypes and Classes when Delegation happens at runtime while inheritance happens at compile-time: not only does the machinery look very different to users and somewhat different as implementers, written in different languages with different formalisms, but PL people tend to deeply compartmentalize the two. They may have looked at low-level mutable state (omnipresent in any practical language until the mid-2000s) as essential when happening at runtime, when they could clearly conceptualize it away as an implementation detail when happening at compile-time. Systems paradigm people (including the old Lisp, Smalltalk and Self communities) who freely mix or interleave runtime and compile-time in the very same language, might have had no trouble unifying the two across evaluation times, but they tend not to publish articles about PL semantics, and not to be read and understood by PL semanticians when they do.

Revisiting these topics several decades after they were in vogue, and finding their then-treatment lacking, with errors from the time still uncorrected to this day, makes me wonder about what other false ideas I, like most people, assume are true in all the other topics I haven't revisited, whether in Computer Science or not, where I just blindly assume the "experts" to be correct due to Gell-Mann amnesia.

3.6 Epistemological Digression

Many people will inevitably quibble about my definition or characterization of OO. Though a treatise of epistemology is beyond the scope of this book, I can briefly answer the most frequent epistemological questions as follows.

This section is not essential to the formalization of OO in the chapters that follow, and can be skipped. I am aware that my answers may shock and turn off some of my readers. Nevertheless, I believe this section is very relevant to the debate at hand, and worth publishing as is.

If a philosophical disagreement with this section will turn you off from reading subsequent technical chapters, maybe you should skip this section, or only return to it after you read those more technical chapters. If so, you should also be careful never to ask about the philosophical opinions of authors, inventors, colleagues, etc., in your technical field.

3.6.1 Is my definition correct?

> The truth or falsehood of all of man's conclusions, inferences, thought and knowledge rests on the truth or falsehood of his definitions.
>
> Ayn Rand

Yes, my definition is correct: it accurately identifies what people usually mean by those words, and distinguishes situations where they apply from situations where they do not, in the contexts that people care about. People using my definition will be able to make good decisions, whereas those using other definitions will make bad decisions where their definitions differ.

3.6.2 What does it even mean for a definition to be correct?

> "When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."
> "The question is," said Alice, "whether you can make words mean so many different things."
> "The question is," said Humpty Dumpty, "which is to be master—that's all."
>
> Lewis Carroll

Some people will argue that definitions are "just" arbitrary conventions, and that there is therefore no rational criterion of correctness, only arbitrary political power of the strong over the weak, to determine what the definitions of words are or should be.

But no, such a point of view is worse than wrong—it is outright evil. The phenomena that effectively affect people, that they care to name, discuss, think about and act

on, are not arbitrary. Thus the important part of definitions isn't convention at all: it is the structure and understanding of these phenomena, rather than the labels used for them. A correct definition precisely identifies the concepts that are relevant to people's concerns, that help them make better decisions that improve their lives, whereas an incorrect definition misleads them into counterproductive choices. Specifically overriding your reason with power is an act of war against you, and generally overriding all reason with power is the very definition of evil.

3.6.3 Is there an authority on those words?

> Those who need leaders aren't qualified to choose them.
>
> Michael Malice

No, there is no authority on software vocabulary, person or committee, that can decree different words for others to use, or different phenomena for others to care about. People care about a phenomenon currently identified under the moniker OO, and even if some "authority" manages to change the name for it, or to denature the name "OO" not to identify the same phenomenon anymore, then people will keep caring about what they now call OO under a different name, rather than care about whatever those who corrupt the name may want them to.

3.6.4 Shouldn't I just use the same definition as Alan Kay?

> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.
>
> Alan Kay

No, that isn't possible, nor would it be appropriate if it were. Alan Kay coined the expression "Object Oriented Programming" in 1967. Originalists might say everyone must take it to mean whatever He defined It to mean, and sometimes cite him as in the epigraph above.

But neither the above (Kay 2003) nor any of Kay's pronouncement on OO constitutes a precise definition with an objective criteria, if a definition at all.[16] And even if

[16] My interpretation is that the first part of this definition (until the last comma) corresponds to modularity, the ability to think about programs in terms of separate "local" entities each with its own "state-process" wherein interactions only happen through well-delimited interfaces ("messaging"). The second part "extreme late-binding of all things" indirectly references the in-language and extensible aspect of modules: extreme late-binding means that the value of those units of modularity may change at runtime, which means not only dynamic dispatch of method invocation depending on the runtime class of an object, but also the ability to dynamically define, incrementally extend, refine or combine those units in the language. Those units may be first-class prototypes, and even when they are only second-class classes, there is a first-class reflection mechanism to define and modify them. When this extensibility is only available at compile-time, as in the object system of many static languages, then the OOP only happens in the meta-language (as in e.g. C++ templates), or the language lacks complete support for OOP.

Note that Kay didn't immediately adopt Simula's inheritance mechanism in Smalltalk-72 (it wasn't called

he had at some point given a definition, one still should remain skeptical of what Kay, and other pioneers, said, if only to recursively apply the same semantic attention to the definition of the words they used in their definitions. Now, one should certainly pay close attention to what pioneers say, but one should pay even closer attention to what they *do*. The pioneer's authority lies not in precise words, but in inspiring or insightful ones; not in well-rounded neatly-conceptualized theories, but in the discovery of successful new practices that are not yet well understood. Solid theories arise only after lots of experience, filtering, and reformulation.

3.6.5 Shouldn't I just let others define "OO" however they want?

> The opinion of 10,000 men is of no value if none of them know anything about the subject.
>
> Marcus Aurelius

Not at all. Some people are reluctant to fight over the meaning of words, and are ready to cave to popular opinion or spurious authorities when they define and redefine "OO" or any word to have whatever precise or murky meaning. Instead they propose that I should stick to "inheritance" when discussing the field characterized by the use of inheritance.

But it is no good to let an ignorant majority "define" the term "Object-Orientation" to mean what little they know of it—for instance, to pick the most popular elements: Class OO only, always mutable records, only single inheritance or C++ style flavorless "multiple inheritance", only single dispatch, no method combination, etc. Letting those who don't know and don't care define technical words would be knowledge bowing to ignorance; it would be for those who know and care to abdicate their responsibility and follow the masses when they should instead lead them; it would be ceding terrain to the Enemy—snake oil salesmen, chaosmongers, corrupters of language, manipulators, proud spreaders of ignorance, etc.—who if let loose would endlessly destroy the value of language and make clear meaning incommunicable. Beside, if you retreat to "inheritance" in the hope that at least for that term you can get people to agree on a clear unambiguous meaning,[17] you'll find that if you have any success defining a useful term

that yet in Simula, either); but he did adopt it eventually in Smalltalk-76, notably under the push of Larry Tesler (who previously used "slot inheritance" on early desktop publishing applications), and this adoption is what launched OO as a phenomenon. Kay stated adopting single inheritance over multiple inheritance was a compromise (Kay 1993); his team later added multiple inheritance to Smalltalk (Goldstein and Bobrow 1980), but it is unclear that Kay had much to do with that addition, that never became standard. More broadly, Kay didn't endorse any specific inheritance mechanism, and never focused on that part of the design. To Kay it was only a means to an end, which is what Kay called "extreme late binding": the fact that behavior definition happens and takes effect dynamically up to the last moment based on values computed at runtime. Inheritance, the practical means behind the late behavior definition that is late bound, and the precise form it takes, is secondary to Kay; what matters to Kay is the role it plays in enabling dynamic code specialization. But inheritance becomes a primary concern to whoever wants to formalize the concepts behind OO, and must refine the intuitions of a pioneer into codified knowledge after decades of practice. And if other means are found to arguably satisfy Kay's "extreme late binding", then they'll have to be given a name that distinguishes them from what is now called OO.

[17] The term "inheritance" is already corrupted, since Goguen uses it at times to mean refinement (Goguen

that way, the agents of entropy will rush to try to defile it in direct proportion to your success; you will have given up precious lexical real estate for no gain whatsoever, only terrible loss.[18]

3.6.6 So what phenomena count as OO?

> The medium is the message.
>
> Marshall McLuhan

What defines OO is not the metaphors of those who invent, implement, or comment about it as much as the design patterns used by programmers when they write code in an OO language; the interactions they have with computers and with each other; the decision trees that are enabled or disabled when evolving a program into another—these phenomena are what OO is. What programmers do, not what programmers say.

And these phenomena are what is captured by the intra-linguistic extensible modularity as defined above: (a) the ability to "code against an interface" and pass any value of any type that satisfies the interface (modularity, be it following structural or nominative rules), (b) the ability to extend and specialize existing code by creating a new entity that "inherits" the properties of existing entities and only needs specify additions and overrides in their behavior rather than repeat their specifications, wherein each extension can modularly refer to functionality defined in other yet-unapplied extensions; and (c) the fact that these entities and the primitives to define, use and specialize them exist *within* the programming language rather than in an external preprocessing layer.

I contend that the above is what is usually meant by OO, that matches the variety of OO languages and systems without including systems that are decidedly not OO, like Erlang, SML or UML. And whatever clear or murky correspondence between names and concepts others may use, this paradigm is what matters, and what I will call OO—it is what I will discuss in this book, and systematically reduce to elementary concepts.

1992) while claiming to do OO, and others use it to mean the (non-modular) extension of database tables or equivalent. Moreover, the term "inheritance", that originated in KRL, in parallel to the adoption and evolution it saw in the field of OO, also had its evolution in the field of Knowledge Representation, Description Logics, Semantic Web, etc. And there are plenty of further legitimate non-OO uses of the word "inherit", to mean that some entity derives some property from a historical origin, an enclosing context, etc.

[18] Indeed, if you don't know to stand your ground, you will constantly retreat, and be made to use ever more flowery "politically correct" vocabulary as a humiliation ritual before those who will wantonly take your words away to abuse you and thereby assert their dominance over you.

Chapter 4

OO as Internal Extensible Modularity

4.1 Modularity

4.1.1 Division of Labor

> The benefits expected of modular programming are: (1) managerial—development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time.
>
> David Parnas

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, by one or multiple developers over time. The "interface" of each module defines a semi-formal contract between the users and implementers of the module, such that users need only know and understand the interface to enjoy the functionality of the module, whereas the implementers have freedom to build and modify the module internals as long as they satisfy the interface.

4.1.2 First- to Fourth-class, Internal or External

> I object to doing things that computers can do.
>
> Olin Shivers

A few languages offer a builtin notion of modules as *first-class* entities, that can be manipulated as values at runtime. But popular modern programming languages usually only offer *some* builtin notion of modules as *second-class* entities, entities that exist at compile-time but are not available as regular runtime values.[1] Either first-class or second-class entities are considered *internal* to the language, part of its semantics, handled by its processors (compiler, interpreter, type system, etc.).

However, many languages offer no such internal notion of modules. 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 their language's development; only the few that have success and see their codebase grow in size and complexity will generally bother to add a module system.[2]

Now 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 means to express modules as *third-class* entities, automated by tools *outside* the language: a preprocessor, an object file linker, editor macros, "wizards" or LLMs. And even if they somehow don't because they can't use or can't afford to use such automation, developers may achieve modules as *fourth-class* entities, ones that they handle manually, with design patterns, editors, copy-paste, and lots of debugging. Either third-class or fourth-class entities are considered *external* to the language, not part of its semantics, not handled by its processors, yet conceptually present in the minds of the programmers.[3] Thus, programmers will:

- copy and paste sections of code, or prefix identifiers, as poor man's modules;
- preprocess files to concatenate, transform and generate code fragments;
- transclude "include" files to serve as libraries or interfaces to later-linked libraries;
- divide code in files they independently compile then "link" or "load" together;
- orchestrate incremental (re)building of software with utilities such as "make";

[1] 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.

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

[3] Whereas the terms "first-class" and "second-class" are well-established in the literature,

I am hereby introducing the terms "third-class" and "fourth-class", as well as the distinction between "internal" and "external" entities. Of course, depending on where you draw the line for "the language", the very same entity processed by the very same tools may shift between these classes: thus, in C++ classes are second-class entities; but if the language being considered is C, and C++ is seen as an external metalanguage (as was the case in the original implementation `cfront` of C++, a preprocessor that generated C code), then the very same classes are third-class entities for C; and if `cfront` were updated to support templates, classes would be first-class entities for the compile-time C++ template language; and if done by hand as a set of conventions on top of C, they would be fourth-class entities.

- bundle software into "packages" they exchange and distribute with "package managers";
- "containerize" consistent package installations into "images".

When for the sake of "simplicity", "elegance", or ease of development or maintenance, support for modularity is left out of a language, or of an ecosystem around one or multiple languages, this language or ecosystem becomes but the weak kernel of a haphazard collection of tools and design patterns cobbled together to palliate this lack of internal modularity with external modularity. The result inevitably ends up growing increasingly complex, ugly, and hard to use, develop and maintain.

4.1.3 Criterion for Modularity

A design is more modular if it enables developers to cooperate more while coordinating less 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. More ability to code one's part, while requiring less knowledge about other people's parts.

For instance, the object-oriented design of the Common Lisp build system ASDF (Rideau 2014; 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 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 the use of objects (possibly for portability reasons at the time), was notably hard to configure, and resisted several attempts to extend or refactor it.

Note that while external modularity enables cooperation during development, internal modularity also enables cooperation during deployment, with conditional selection and reconfiguration of modules, gradual deployment of features, user-defined composition of modules, automated A/B testing, and more.

For a more complete theory of Modularity, see Rideau (2016).

4.1.4 Historical Modularity Breakthroughs

Programmers these days are very aware of files, and file hierarchies, as units of modularity they have to think about quite often, naming, opening, visiting and closing them in their editors, manipulating them in their source control, etc. Although files as such embody a form of modularity external to most simple programming languages that they use, the more advanced programming languages, that they use most often, tend to have some notion of those files, runtime and/or compile-time representation for those files, and often some correspondence between file names and names of internal module entities that result from compiling those files. In other words, for most languages that matter, files embody modularity internal to programming languages.[4]

[4]Note that while a file may be a unit of modularity, this is not always the case: often, actual modules span multiple files; at other times, there may be multiple modules in a single file. For instance, compiling a C program typically requires each file to be registered into some "build" file (e.g. a `Makefile`) to be

Now, while files go back to the 1950s and hierarchical filesystems to the 1960s, there was an even earlier breakthrough in modularity, so pervasive that programmers use it without even thinking about using it: subroutines, already conceptualized by Ada Lovelace as "operations" (Lovelace 1843), formalized early on by Wilkes et al. in the 1940s used by the earliest programmers in manually assembled code, part of the first programming language Plankalkül, and of FORTRAN II. Subroutines allow programmers to divide bigger problems into smaller ones at a fine grain, that can each be solved by a different programmer, while other programmers only have to know its calling convention.

No less important than subroutines as such yet also taken for granted was the concept of a return address allowing calls to a single copy of a subroutine from multiple sites, and later of a call stack of such return addresses and other caller and callee data. The call stack enabled reentrancy of subroutines, such that a subroutine can be recursively called by a subroutine it itself called; thus, programmers do not have to care that their subroutines should not be reentered while being executed due to a direct or indirect subroutine call. Indeed, subroutines can now be reentered, not just accidentally, but intentionally, to express simpler recursive solutions to problems. The ability to cooperate more with less coordination with programmers from other parts of the software: that's the very definition of modularity.

Sadly, the generalization of call stacks to first-class continuations in the 1970s, and later to delimited control in 1988 and beyond is still not available in most programming languages, eschewing another advance in modularity, wherein programmers could otherwise abstract over the execution of some fragment of code without having to worry about transformations to their control structure, e.g. for the sake of non-deterministic search, robust replaying of code in case of failure, security checks of code behavior, dynamic discovery and management of side-effects, etc.

On a different dimension, separately compiled object files, as implemented by FORTRAN (1956), provided third-class modularity through an external linker. Later developments like typechecked second-class Modules à la Modula-2 (1978), Higher-Order Modules à la ML (1985), typeclasses à la Haskell (or traits in Rust), interfaces à la Java, and much more, provided second-class modularity as part of a language itself. Finally, objects in Smalltalk-72 (even before Smalltalk adopted inheritance in 1976), or first-class modules in ML (Russo 1998), provided first-class modularity.

---

considered as part of the software; a C library also comes with an accompanying "header" file, though there need not be a 1-to-1 correspondence between C source files and C header files (as contrasted with source and interface files in OCaml). A C "module" therefore typically spans multiple files, and some files (headers and build files) can have bits from multiple modules. Of course, having to manually ensure consistency of information between multiple files makes the setup less modular than a language that doesn't require as much maintenance. Then again, the C language also recently adopted some notion of namespace, which, though it doesn't seem used that much in practice, can constitute yet another notion of modules, several of which can be present in a file. At a bigger scale, groups of files in one or multiple directories may constitute a library, and a "package" in a software "distribution" may include one or several libraries, etc. Thus, even within a single language, there can be many notions of modularity at several different scales of software construction, enabling division of labor across time for a single person, for a small team, for a large team, between several teams, etc., each time with different correspondences between modules and files, that are never quite 1-to-1: even "just a file" is not merely a blob of data, but also the meta-data of a filename registered in a broader namespace, and some intended usage.

4.1.5 Modularity and Complexity

Modularity used correctly can tremendously simplify programs and improve their quality, by enabling the solutions to common problems to be solved once by experts at their best, and then shared by everyone, instead of the same problems being solved over and over, often badly, by amateurs or tired experts.

Use of modules also introduces overhead, though, at development time, compile-time and runtime alike: boiler plate to define and use modules, name management to juggle with all the modules, missing simplifications in using overly general solutions that are not specialized enough to the problems at hand, duplication of entities on both sides of each module's interface, extra work to cover the "impedance mismatch" between the representation used by the common solution and that used by the actual problems, etc.

Modularity used incorrectly, with too many module boundaries, module boundaries drawn badly, resolved at the wrong evaluation stage, can increase complexity rather than reduce it: more parts and more crossings between parts cause more overhead; the factoring may fail to bring simplification in how programmers may reason about the system; subtle underdocumented interactions between features, especially involving implicit or mutable state, can increase the cognitive load of using the system; attempts at unifying mismatched concepts only to then re-distinguish them with lots of conditional behavior will cause confusion rather than enlightenment; modules may fail to offer code reuse between different situations; programmers may have to manage large interfaces to achieve small results.[5]

[5] A notable trend in wrongheaded "modularity" is *myopic* designs, that achieve modest 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 to the design, in inter-module glue being made more complex, or in module namespace curation getting more contentious.

For instance, the once vastly overrated and overfunded notions "microkernels" or "microservices" consist in dividing a system in a lot of modules, "servers" or "services" each doing a relatively small task within a larger system, separated by extremely expensive hardware barriers wherein data is marshalled and unmarshalled across processes or across machines. Each service in isolation, that you may focus on, looks and is indeed simpler than the overall software combining those services; but this overall software kept out of focus was made significantly more complex by dividing it into those services. Another myopic design that is much rarer these days is to rewrite a program from a big monolithic heap of code in a higher level language into a lot of subroutines in a lower-level language (sometimes assembly language), boasting how vastly simpler each of these subroutines is to the previous program; yet obviously the sum total of those routines is much more complex to write and maintain than a similarly factored program written in an appropriate higher level language (that indeed the original language might not be).

While myopic designs might offer some small performance improvement in terms of pipelining, locality and load-balancing for very large worksets, they offer no improvement whatsoever in terms of modularity, quite the contrary: (1) The modules must already exist or be made to exist at the language level, before the division into services may be enacted. Far from *providing* a solution to any problem for the programmer, these techniques *require* programmers to already have solved the important problem of module factoring before you can apply them, at which point they hinder rather than help. (2) Given the factoring of a program into modules, these techniques add runtime barriers that do not improve safety compared to compile-time barriers (e.g. using strong types, whether static or dynamic). (3) Yet those runtime barriers vastly decrease performance, and even more so by the compile-time optimizations they prevent (whether builtin or user-defined with e.g. macros). (4) The interface of each module is made larger for having to include a specific marshalling, and the constraints related to being marshalled. This problem can be alleviated by the use of "interface description languages", but these languages tend to be both poorer and more complex than a good programming language's typesystem, and introduce an impedance mismatch of their own. (5) Any alleged

4.1.6 Implementing Modularity

To achieve modularity, the specification for a software computation is divided into independently specified modules. Modules may reference other modules and the subcomputations they define, potentially specified by many different people at different times, including in the future. These references happen through a module context, a data structure with many fields or subfields that refer to each module, submodule, or subcomputation specified therein. When comes time to integrate all the module specifications into a complete computation, then comes the problem of implementing the circularity between the definition of the module context and the definitions of those modules.

With third-class modularity, the module context and modules are resolved before the language processor starts, by whatever preprocessor external to the language generates the program. With second-class internal modularity, they are resolved by some passes of the language-internal processor, before the program starts. In either of the

benefits from dividing in multiple distributed processes, whether in terms of failure-resistance, modularity, safety or performance are wholly lost if and when (as is often the case) persistent data all ends up being centralized in database services, that become a bottleneck for the dataflow. (6) The overall system is not made one bit smaller for being divided in smaller parts; actually, all the artificial process crossings, marshallings and unmarshallings, actually make the overall system noticeably larger in proportion to how small those parts are. Lots of small problems are added at both runtime and compile-time, while no actual problem whatsoever is solved. (7) These designs are thus actually detrimental in direct proportion to how much they are followed, and in proportion to how beneficial they claim to be. They are, technically, a lie. (8) In practice, "successful" microkernels import all services into a monolithic "single server", and successful microservices are eventually successfully rebuilt as vertically integrated single services.

In the end, the technical justification for these misguided designs stem from the inability to think about meta-levels and distinguish between compile-time and runtime organization of code: Modularity is a matter of semantics, as manipulated by programmers at programming-time, and by compilers at compile-time; the runtime barrier crossings of these myopic designs cannot conceivably do anything to help about that, only hinder.

Now, there are sometimes quite valid socio-economical (and not technical) justifications to dividing a program into multiple services: when there are many teams having distinct incentives, feedback loops, responsibilities, service level agreements they are financially accountable for, etc., it makes sense for them to deploy distinct services. Indeed, Conway's Law states that the technical architecture of software follows its business or management architecture.

Finally, it is of note that some systems take a radically opposite approach to modularity, or lack thereof: instead of gratuitous division into ever more counter-productive modules, some prefer wanton avoidance of having to divide code into modules at all. For instance, APL reduces the *need* for modules or even subroutines by being extremely terse, and by shunning abstraction in favor of concrete low-level data representations (on top of its high-level virtual machine rather than of the lower-level model of more popular languages), to the point of replacing many routine names by common idioms—known sequences of combinators. (The many talks by Aaron Hsu at LambdaConf are an amazing glimpse at this style of programming.)

The programmer can then directly express the concepts of the domain being implemented in terms of concrete APL's tables and functions, stripped of all the unnecessary abstractions, until the solution is both so simple and task-specific that there is no need for shared modules. By using his terse combinators to directly manipulate entire tables of data at a time, a competent APLer can find a complete solution to a problem in fewer lines of code than it takes for programmers in other languages just to name the structures, their fields and accessors. Admittedly, there is only so much room in this direction: as the software grows in scope, and the required features grow in intrinsic complexity, there is a point at which it is becomes big to fit wholly in any programmer's mind, then it must be chipped away by moving parts into other modules, notably by reusing common algorithms and data structures from libraries rather than inline specialized versions. But in practice, a lot of problems can be tackled this way by bright enough developers, especially after having been divided into subproblems for socio-economic reasons.

above cases, a whole-program optimizer might fully resolve those modules such that no trace of them is left at runtime. Without a whole-program optimizer, compilers usually generate global linkage tables for the module context, that will be resolved statically by a static linker when producing an executable file, or dynamically by a dynamic loader when loading shared object files, in either case, before any of the programmer's regular code is run (though some hooks may be provided for low-level programmer-specified initialization).

At the other extreme, all third-class or second-class module evaluation may be deferred until runtime, the runtime result being no different than if first-class internal modularity was used, though there might be benefits to keeping modularity compile-time only in terms of program analysis, synthesis or transformation. For local or first-class modules, compilers usually generate some kind of "virtual method dispatch table" (or simply "virtual table" or "dispatch table" or some such) for each modularly defined entity, that, if it cannot resolve statically, will exist as such at runtime. Thus, Haskell typeclasses become "dictionaries" that may or may not be fully inlined. In the case of OO, prototypes are indeed typically represented by such "dispatch table" at runtime—which in Class OO would be the type descriptor for each object, carried at runtime for dynamic dispatch.

In a low-level computation with pointers into mutable memory, the module context is often being accessed through a global variable, or if not global, passed to functions implementing each module as a special context argument, and each field or subfield is initialized through side-effects, often with some static protocol to ensure that they are (hopefully, often all) initialized before they are used. In a high-level computation without mutation, each module is implemented as a function taking the module context as argument, the fields are implemented as functional lenses, and the mutual recursion is achieved using a fixpoint combinator; lazy evaluation may be used as a dynamic protocol to ensure that each field is initialized before it is used.[6] The more

[6]It is hard to ensure initialization before use; a powerful enough language makes it impossible to predict whether that will be the case.

In unsafe languages, your program will happily load nonsensical values from uninitialized bindings, then silently corrupt the entire memory image, dancing a fandango on core, causing a segmentation fault and other low-level failures, or even worse, veering off into arbitrary undefined behavior and yielding life-shatteringly wrong results to unsuspecting users. The symptoms if any are seen long after the invalid use-before-init, making the issue hard to pin-point and debugging extremely difficult unless you have access to time-travel debugging at many levels of abstraction.

In not-so-safe languages, a magic NULL value is used, or for the same semantics just with additional syntactic hurdles, you may be forced to use some option type; or you may have to use (and sometimes provide out of thin air) some default value that will eventually prove wrong. In the end, your program will fail with some exception, which may happen long after the invalid use-before-init, but at least the symptoms will be visible and you'll have a vague idea what happened, just no idea where or when.

Actually safe languages, when they can't prove init-before-use, will maintain and check a special "unbound" marker in the binding cell or next to it, possibly even in a concurrency-protected way if appropriate, to identify which bindings were or weren't initialized already, and eagerly raise an exception immediately at the point of use-before-init, ensuring the programmer can then easily identify where and when the issue is happening, with complete contextual information.

In even safer languages, lazy evaluation automatically ensures that you always have init-before-use, and if the compiler is well done may even detect and properly report finite dependency cycles; you may still experience resource exhaustion if you generate infinite new dynamic dependencies, but so would you in the less safe alternatives if you could reach that point.

Safest languages may require you to statically prove init-before-use, which may be very hard with full

programmers must agree on a protocol to follow, the less modular the approach. In a stateful applicative language, where the various modular definitions are initialized by side-effects, programmers need to follow some rigid protocol that may not be expressive enough to capture the modular dependencies between internal definitions, often leading to indirect solutions like "builder" classes in Java, that stage all the complex computations before the initialization of objects of the actually desired class.

4.2 Extensibility

> Malum est consilium, quod mutari non potest.
> (It is a bad plan that admits of no modification.)
>
> Publius Syrus (1st century BC)

4.2.1 Extending an Entity

Extensibility is the ability to take a software entity and create a new entity that includes all the functionality of the previous entity, and adds new functionality, refines existing functionality, or otherwise modifies the previous entity. Extensibility can be realized inside or outside of a programming language.

4.2.2 First-class to Fourth-class Extensibility

External Extensibility is the ability to extend some software entity by taking its code then reading it, copying it, editing it, processing it, modifying it, (re)building it, etc.

dependent types, or very constraining with a more limited proof system, possibly forcing you to fall back to only the "not-so-safe" alternative. But this technology is onerous and not usable by programmers at large.

Some languages may let you choose between several of these operation modes depending on compilation settings or program annotations.

Interestingly, whichever safety mode is used, programmers have to manually follow some protocol to ensure init-before-use. However the lazy evaluation approach minimizes artificial constraints on such protocol, that when not sensible might force them to fall back to the not-so-safe variant.

The init-before-use issue is well-known and exists outside of OO: it may happen whenever there is mutual recursion between variables or initial elements of data structures. However, we'll see that "open recursion" (Cardelli 1992; Pierce 2002), i.e. the use of operators meant to be the argument of a fixpoint combinator, but also possibly composition before fixpointing, is ubiquitous in OO, wherein partial specifications define methods that use other methods that are yet to be defined in other partial specifications, that may or may not follow any particular protocol for initialization order. Thus we see that the simplest, most natural and safest usable setting for OO is: lazy evaluation.

This may come at a surprise to many who mistakenly believe the essence of OO is in the domain it is commonly applied to (defining mutable data structures and accompanying functions as "classes"), rather than in the semantic mechanism that is being applied to these domains (extensible modular definitions of arbitrary code using inheritance). But this is no surprise to those who are deeply familiar with C++ templates, Jsonnet or Nix, and other systems that allow programmers to directly use unrestricted OO in a more fundamental purely functional setting wherein OO can be leveraged into arbitrary programming and metaprogramming, not just for "classes" stricto sensu. In the next subsubsection, I argue the case from the point of view of modularity rather than initialization safety; however, both can be seen as two aspects of the same argument about semantics.

Extensibility is much easier given source code meant to be thus read and written, but is still possible to reverse-engineer and patch binary code, though the process can be harder, less robust, less flexible and less reusable. Thus, external extensibility is always possible for any software written in any language, though it can be very costly, in the worst case as costly as rewriting the entire extended program from scratch. When done automatically, it's third-class extensibility. When done manually, it's fourth-class extensibility. The automation need not be written in the same language as the software being extended, or as its usual language processor, though either may often be the case.

Internal or Intra-linguistic extensibility, either first-class or second-class, by contrast, is the ability to extend, refine or modify a software entity without having to read, rewrite, or modify any of the previous functionality, indeed without having to know how it works or have access to its source code, only having to write the additions, refinements and modifications. Second-class extensibility happens only at compile-time, and is quite restricted, but enables more compiler optimizations and better developer tooling. First-class extensibility may happen at runtime, and is less restricted than second-class extensibility but more so than external extensibility in terms of what modifications it affordably enables, yet less restricted than either in terms of enabling last-minute customization based on all the information available to the program.

These notions of extensibility are complementary and not opposite, for often to extend a program, you will first extract from the existing code a shared subset that can be used as a library (which is an extra-linguistic extension), and rewrite both the old and new functionality as extensions of that library (which is an intra-linguistic extension), as much as possible at compile-time for efficiency (which is second-class extensibility), yet enough at runtime to cover your needs (which is first-class extensibility).

Finally, as with modularity, the lines between "external" and "internal", or between "second-class" and "first-class", can often be blurred by reflection and "live" interactive environments. But these demarcations are not clear-cut objective features of physical human-computer interactions. They are subjective features of the way humans plan and analyze those interactions. To end-users who won't look inside programs, first-class, second-class, third-class, and if sufficiently helpless even fourth-class, are all the same: things they cannot change. For adventurous hackers willing to look under the hood of the compiler and the operating system, first-class to fourth-class are also all the same: things they can change and automate, even when playing with otherwise "dead" programs on "regular" computing systems. The demarcations are still useful though: division of labor can be facilitated or inhibited by software architecture. A system that empowers an end-user to make the simple changes they need and understand, might prove much more economical than a largely equivalent system that requires a high-agency world expert to make an analogous change.

4.2.3 A Criterion for Extensibility

A design is more extensible if it enables developers to enact more kinds of change through smaller, more local modifications compared to alternative designs that require larger (costlier) rewrites or more global modifications, or that prohibit change (equivalent to making its cost infinite).

Extensibility 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.

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 non-local modifications to the program structure; in Java it involves changes throughout the code—straightforward but numerous—while preserving the local program structure; in Lisp it requires minimal local changes; and in Haskell it requires one local change only. Thus, with respect to this and similar kinds of change, if expected, Haskell is more extensible than Lisp, which is more extensible than Java, which is more extensible than C.

Extensibility does not necessarily mean that a complex addition or refactoring can be done in a single small change. Rather, code evolution can be achieved through many small changes, where the system assists developers by allowing them to focus on only one small change at a time, while the system tracks down the remaining necessary adjustments.

For instance, a rich static type system can often serve as a tool to guide large refactorings by dividing them into manageably small steps—as extra-linguistic code modifications—making the typechecker happy one redefinition at a time after an initial type modification. This example also illustrates how ***Extensibility and Modularity usually happen through meta-linguistic mechanisms rather than linguistic ones***, i.e. through tooling outside the language rather than through expressions inside the language. Even then, internalizing the locus of such extensible modularity within the language enables dynamic extension, runtime code sharing and user-guided specialization as intra-linguistic deployment processes that leverage the result of the extra-linguistic development process.

4.2.4 Historical Extensibility Breakthroughs

While any software is externally extensible by patching the executable binary, the invention of assembly code made it far easier to extend programs, especially with automatic offset calculations from labels. Compilers and interpreters that process source code meant for human consumption and production, along with preprocessors, interactive editors, and all kinds of software tooling, greatly facilitated external extensibility, too. Meanwhile, software that requires configuration through many files that must be carefully and manually kept in synch is less extensible than one configurable through a single file from which all required changes are automatically propagated coherently (whether that configuration is internal or external, first-class to fourth-class). Whenever multiple files must be modified together (such as interface and implementation files in many languages), the true units of modularity (or in this case extensibility) are not the individual files—but each group of files that must be modified in tandem; and sometimes, the units of extensibility are entities that span parts of multiple files, which is even more cumbersome.

Higher-level languages facilitate external extensibility compared to lower-level ones, by enabling programmers to achieve larger effects they desire through smaller, more local changes that cost them less. Thus, FORTRAN enables more code extensibility than assembly, and Haskell more than FORTRAN. Languages with higher-order

functions or object orientation enable greater internal extensibility by allowing the creation of new in-language entities built from existing ones in more powerful ways. To demonstrate how OO enables more extensibility, consider a course or library about increasingly more sophisticated data structures: when using a functional language with simple Hindley-Milner typechecking, each variant requires a near-complete rewrite from scratch of the data structure and all associated functions; but when using OO, each variant can be expressed as a refinement of previous variants with only a few targeted changes.

4.2.5 Extensibility without Modularity

Before delving deeply into how OO brings together extensibility and modularity, it is worth explaining what extensibility without modularity means.

Operating Systems, applications or games sometimes deliver updates via binary patch formats that minimize data transmitted while maximizing changes effected to the software. Such patches embody pure extensibility with total lack of modularity: they are meaningful only when applied to the exact previous version of the software. Text-based diff files can similarly serve as patches for source code, and are somewhat more modular, remaining applicable even in the presence of certain independent changes to the source code, yet are still not very modular overall: they are fragile and operate without respecting any semantic code interface; indeed their power to extend software lies precisely in their not having to respect such interfaces.

The ultimate in extensibility without modularity would be to specify modifications to the software as bytes concatenated at the end of a compressed stream (e.g. using `gzip` or some AI model) of the previous software: a few bits could specify maximally large changes, ones that can reuse large amounts of information from the compressor's model of the software so far; yet those compressed bits would mean nothing outside the exact context of the compressor—maximum extensibility, minimum modularity.

Interestingly, these forms of external extensibility without modularity, while good for distributing software, are totally infeasible for humans to directly create: they vastly increase rather than decrease the cognitive load required to write software. Instead, humans use modular forms of extensibility, such as editing source code as part of an edit-evaluate-debug development loop, until they reach a new version of the software they want to release, after which point they use automated tools to extract from the modularly-achieved new version some non-modular compressed patches.

As for internal extensibility without modularity, UML, co-Algebras or relational modeling, as previously discussed in section 2.7, fail to model OO precisely because the "classes" they specify are mere types: they lack the module context that enables self-reference in actual OO classes. As in-language entities, they can be extended by adding new attributes, but these attributes have to be constants: they may not contain any self-reference to the entity being defined and its attributes.

4.2.6 Extensibility and Complexity

Extensibility of software entities means that variants of these software entities may be reused many times in as many different contexts, so more software can be achieved for

fewer efforts.

Now, external extensibility through editing (as opposed to e.g. preprocessing) means that these entities are "forked", which in turn reintroduces complexity in the maintenance process, as propagating bug fixes and other changes to all the forks becomes all the harder as the number of copies increases. External extensibility through preprocessing, and internal extensibility, do not have this issue, and, if offered along a dimension that matters to users, enable the development of rich reusable libraries that overall decrease the complexity of the software development process.[7]

Note that often, the right way to reuse an existing software entity is not to reuse it as is (internal extensions), but first to refactor the code (external extensions) so that both the old and new code are extensions of some common library (internal extensions). Then the resulting code can be kept simple and easy to reason with. Internal and external extensibility are thus mutual friends, not mutual enemies.

By decreasing not only overall complexity, but the complexity of *incremental* changes, extensibility also supports shorter software development feedback loops, therefore more agile development. By contrast, without extensibility, developers may require more effort before any tangible incremental progress is achieved, leading to loss of direction, of motivation, of support from management and of buy-in from investors and customers. Extensibility can thus be essential to successful software development.

4.2.7 Implementing Extensibility

To enable extensibility, a software entity must be represented in a way that allows semantic manipulation in dimensions meaningful to programmers. By the very nature of programming, any such representation is enough to enable arbitrary extensions, given a universal programming language, though some representations may be simpler to work with than others: they may enable programmers to express concisely and precisely the changes they want, at the correct level of abstraction, detailed enough that they expose the concepts at stake, yet not so detailed that the programmer is drowned in minutiae.

Now, in the case of second-class internal extensibility, or of first-class internal extensibility in a less-than-universal language, representations are no longer equivalent, as only a subset of computable extensions are possible. This kind of limited extensibility is not uncommon in computing systems; the restrictions can help keep the developers safe and simplify the work of language implementers; on the other hand, they may push developers with more advanced needs towards relying external extensibility instead, or picking an altogether different language.

[7] Interestingly, detractors of OO will often count "code reuse" as a bad aspect of OO: they will claim that they appreciate inheritance of interfaces, but not of implementation. Many OO language designers, as of Java or C#, will say the same of multiple inheritance though not of single inheritance. Yet, many practitioners of OO have no trouble reusing and extending libraries of OO classes and prototypes, including libraries that rely heavily on multiple inheritance for implementation.

It is quite likely that users bitten by the complexities yet limitations of multiple inheritance in C++ or Ada may see that it does not bring benefits commensurable with its costs; and it is also quite likely that users fooled by the absurd "inheritance is subtyping" slogan found that code written under this premise does not quite work as advertised. These disgruntled users may then blame OO in general for the failings of these languages to implement OO, or for their believing false slogans and false lessons propagated by bad teachers of OO.

In a pure functional model of extensibility, an extension is a function that takes the original unextended value and returns the modified extended value. In a stateful model of extensibility, an extension can instead be a procedure that takes a mutable reference, table or structure containing a value, and modifies it in-place to extend that value—sometimes using "hot-patches" that were not foreseen by the original programmer.

4.3 Extensible Modularity

> Power Couple: Two individuals that are super heroes by themselves yet when powers combine become an unstoppable, complementary unit.
>
> Urban Dictionary

4.3.1 A Dynamic Duo

Modularity and Extensibility work hand in hand: ***Modularity means you only need to know a small amount of old information to make software progress. Extensibility means you only need to contribute a small amount of new information to make software progress.*** Together they mean that one or many finite-brained developers can better divide work across time and between each other, and make more software progress with a modular and extensible design, tackling larger problems while using less mental power each, compared with using less-modular and less-extensible programming language designs. Together they enable the organic development of cooperative systems that grow with each programmer's needs without getting bogged down in coordination issues.

Early examples of Modularity and Extensibility together that pre-date fully-formed OO include of course classes in Simula 1967 (Dahl and Nygaard 1967), but also precursor breakthroughs like the "locator words" of the Burroughs B5000 (Barton 1961; Lonergan and King 1961), and Ivan Sutherland's Sketchpad's "masters and instances" (Sutherland 1963), that both inspired Kay, or Warren Teitelman's Pilot's ADVISE facility (Teitelman 1966), that was influential at least in the Lisp community and led to method combination in Flavors and CLOS.[8]

4.3.2 Modular Extensible Specifications

A modular extensible specification specifies how to extend a previous value, but in a modular way: the extension is able to use some modular context to refer to values defined and extended by other pieces of code.

At this point, I reach the end of what can be clearly explained while remaining informal, and have to introduce a formal model of modularity, extensibility, and the

[8] I wonder how much the Smalltalk and Interlisp teams did or did not interact at PARC. I can't imagine they didn't, and yet, Interlisp seems to have had more influence on the MIT Lispers than the co-located Smalltalkers.

two together, to further convey the ideas behind OO. The last task to carry informally is therefore a justification of my approach to formalization.

4.4 Why a Minimal Model of First-Class OO using FP?

4.4.1 Why a Formal Model?

The current section was largely appealing to well-established concepts in Computing. Inasmuch as it can be understood, it is only because I expect my readers to be seasoned programmers and scientists familiar with those concepts. My most complex explanations were in terms of functions from a modular context to a value, or from some (original) value to another (extended) value (of essentially the same type). As I try to combine these kinds of functions, the complexity is greater than can be explained away in a few words, and the devil will be in the details.

A formal model allows one to precisely define and discuss the building blocks of OO, in a way that is clear and unambiguous to everyone. This is all the more important when prevailing discourse about OO is full of handwaving, imprecision, ambiguity, confusions, and identical words used with crucially different meanings by different people.

4.4.2 Why a Minimal Model?

> Actually a person does not *really* understand something until teaching it to a *computer*, i.e. expressing it as an algorithm.
>
> Donald Knuth

I seek a *minimal* model because a non-minimal model means there are still concepts that haven't been teased apart from each other, but are confused and confusing.

If I *can* express classes in terms of prototypes, prototypes in terms of specifications, or multiple and single inheritance in terms of mixin inheritance, then I *must* do so, until I reduce OO to its simplest expression, and identify the most fundamental building blocks within it, from which all the usual concepts can be reconstituted, explained, justified, evaluated, generalized, and maybe even improved upon.

4.4.3 Why Functional Programming?

Functional Programming (FP) is a computational model directly related to formal or mathematical logic whereby one can precisely reason about programs, their semantics (what they mean), how they behave, etc. The famous Curry-Howard Correspondence establishes a direct relationship between the terms of the λ-calculus that is the foundation of FP, and the rules of logical deduction. That correspondence can also be extended to cover the Categories of mathematics, and more.

Therefore, in an essential sense, FP is indeed the "simplest" paradigm in which to describe the semantics of OO and other programming paradigms: it is simultaneously

amenable to both computation and reasoning with the least amount of scaffolding for bridging between the two.

4.4.4 Why an Executable Model?

While I could achieve a slightly simpler algorithmic model by using a theoretical variant of the λ-calculus, I and other people would not be able to directly run and test my algorithms without a costly and error-prone layer of translation or interpretation.

A actual programming language that while very close to the λ-calculus comes with both additional features and additional restrictions, will introduce some complexity, and a barrier to entry to people not familiar with this particular language. But it will make it easy for me and my readers to run, to test, to debug, and to interact with the code I offer in this book, and to develop an intuition on how it runs and what it does, and later to extend and to adapt it.

My code will also provide a baseline for implementers who would want to use my ideas, and who may just port my code to their programming language of choice, and be able to debug their port by comparing its behavior to that of the original I provide. They can achieve implement basic OO in two lines of code in any modern programming language, add have a full-featured OO system in a few hundreds of lines of code.

4.4.5 Why First-Class?

The most popular form of OO is second-class, wherein all inheritance happens at compile-time when defining classes. But for some programmers to use OO as a second-class programming construct, language implementers still have to implement OO as a first-class construct within their compilers and other semantic processors. ***Anyone's second-class entities are someone else's first-class entities.*** And you still don't fully understand those entities until you have implemented them, at which point they are first class. Thus, every useful minimal semantic model is always a first-class model, even if "only" for the implementation of a meta-level tool such as a typechecker.

4.4.6 What Precedents?

I was flabbergasted when I first saw basic OO actually implemented in two function definitions, in the Nix standard library (Simons 2015). These two definitions can be ultimately traced in a long indirect line to the pioneering formalization by Bracha and Cook (Bracha and Cook 1990), though the author wasn't aware of the lineage, or indeed even that he was doing OO;[9] however, Nix also implements conflation, a crucial element missing from Bracha and Cook.

[9]Peter Simons, who implemented prototypes as a user-level library in Nix as "extensions", wrote in a private communication that he did not not know anything about their relationship to Prototypes, Mixins or OO, but semi-independently reinvented them and their use, inspired by examples and by discussions with Andres Löh and Conor McBride; the latter two unlike Simons were well-versed in OO literature, though they are usually known to advocate FP over OO.

I will deconstruct and reconstruct this formalization. Then, on top of this foundation, I will be able to add all the usual concepts of OO, developed by their respective authors, whom I will cite.

4.4.7 Why Scheme?

The Algorithmic Language Scheme is a minimal language built around a variant of the applicative λ-calculus, as a dialect in the wider tradition of LISP. It has many implementations, dialects and close cousins, a lot of documentation, modern libraries, decades of established lore, code base, user base, academic recognition. It also has macro systems that allow you to tailor the syntax of the language to your needs.

Some other languages, like ML or Haskell, are closer to the theoretical λ-calculus, but come with builtin typesystems that are way too inexpressive to accept the λ-terms I use. I suspect that systems with dependent types, such as in Rocq, Agda or Lean, are sufficiently expressive, but the types involved might be unwieldy and would make it harder to explain the basic concepts I present. I leave it as an exercise to the reader to port my code to such platforms, and look forward to the result.

Nix, that directly provides λ-calculus with dynamic typing, is lazy, which actually makes the basic concepts simpler. But it would require more care for implementers trying to port such an implementation to most programming contexts that are applicative. Also, Nix is more recent, less well-known, its syntax and semantics less recognizable; the Lindy effect means it will probably disappear sooner than Scheme, making this book harder to read for potential readers across time. Finally, Nix as compared to Scheme, is missing a key feature beyond the basic λ-calculus, that I will use when building multiple inheritance: the ability to test two specifications for equality.

Therefore, I pick Scheme as the best compromise in which to formalize OO.

Chapter 5

Minimal OO

> Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.
>
> Antoine de Saint-Exupéry

Now that I've given an informal explanation of OO and what it is for (Internal Extensible Modularity), I can introduce formal semantics for it, starting with a truly minimal model: (1) No classes, no objects, just specifications and targets. (2) The simplest and most fundamental form of inheritance, mixin inheritance. Mixin inheritance is indeed simplest from the point of view of post-1970s formal logic, though not from the point of view of implementation using mid-1960s computer technology, at which point I'd be using single inheritance indeed.

5.1 Minimal First-Class Extensibility

5.1.1 Extensions as Functions

I will start by formalizing First-Class Extensibility in pure FP, as it will be easier than modularity, and a good warmup. To make clearer what kind of computational objects I am talking about, I will be using semi-formal types as fourth-class entities, i.e. purely as design-patterns to be enforced by humans. I leave a coherent typesystem as an exercise to the reader, though I will offer some guidance and references to relevant literature (see section 6.3).

Now, to extend to some computation returning some value of type `V`, is simply to do some more computation, starting from that value, and returning some extended value of some possibly different type `W`. Thus, in general an "extension" is actually an arbitrary transformation, which in FP, will be modeled as a function of type `V → W`.

However, under a stricter notion of extension, `W` must be the same as `V` or a subtype thereof: you can add or refine type information about the entity being extended, or adjust it in minor ways; you may not invalidate the information specified so far, at least

none of the information encoded in the type. When that is the case, I will then speak of a "strict extension".

Obviously, if types are allowed to be too precise, then any value `v` is, among other things, an element of the singleton type that contains only `v`, at which point the only allowed transformation is a constant non-transformation. Still, in a system in which developers can *explicitly* declare the information they *intend* to be preserved, as a type `V` if any (which funnily is non-trivial if *not* `Any`), it makes sense to require only extensions that strictly respect that type, i.e. functions of type $\mathtt{V} \to \mathtt{V}$, or $\mathtt{W} \subset \mathtt{V} \Rightarrow \mathtt{V} \to \mathtt{W}$ (meaning $\mathtt{V} \to \mathtt{W}$ under the constraint that `W` is a subtype of `V`, for some type `W` to be declared, in which case the function is also of type $\mathtt{V} \to \mathtt{V}$).[1]

5.1.2 Coloring a Point

The prototypical type `V` to (strictly) extend would be the type `Record` for records. Assuming for the moment some syntactic sugar, and postponing discussion of precise semantics, I could define a record as follows:

```
(define point-p (record (x 2) (y 4)))
```

i.e. the variable `point-p` is bound to a record that associates to symbol `x` the number `2` and to symbol `y` the number `4`.

A sample (strict) extension would be the function `paint-blue` below, that extends a given record (lexically bound to `p` within the body of the function) into a record that is a copy of the previous with a new or overriding binding associating to symbol `color` the string `"blue"`:

```
(define (paint-blue p) (extend-record p 'color "blue"))
```

Obviously, if you apply this extension to that value with `(paint-blue point-p)` you obtain a record equal to what you could have directly defined as:

```
(record (x 2) (y 4) (color "blue"))
```

Readers familiar with the literature will recognize the "colored point" example used in many OO papers. Note, however, that in the present example, as contrasted to most such papers, and to further examples in subsequent sections:

(a) I am extending a point *value* rather than a point *type*,

(b) the value is a regular record, and not an "object" by any means, and

(c) indeed I haven't started modeling the modularity aspect of OO yet.

5.1.3 Extending Arbitrary Values

The type `V` of some values being extended could be anything. The possibilities are endless, but here are a few simple real-life examples of strict extensions for some given type:

[1] And even without static types, the pure lazy functional language Jsonnet (Cunningham 2014) allows programmers to specify dynamically checked constraints that objects must satisfy after they are instantiated (if they are, lazily). These constraints can ensure a runtime error is issued when a desired invariant is broken, and cannot be disabled or removed by inheriting objects. They can be used to enforce the same invariants as types could, and richer invariants too, albeit at runtime only and not at compile-time.

- The values could be numbers, and then your extensions could be adding some increment to a previous number, which could be useful to count the price, weight or number of parts in a project being specified.
- The values could be bags of strings, and your extensions could append part identifiers to the list of spare parts or ingredients to order before starting assembly of a physical project.
- The values could be lists of dependencies, where each dependency is a package to build, action to take or node to compute, in a build system, a reactive functional interface or a compiler.

A record (indexed product) is just a common case because it can be used to encode anything. Mathematicians will tell you that products (indexed or not) give a nice "cartesian closed" categorical structure to the set of types for values being extended. What it means in the end is that you can decompose your specifications into elementary aspects that you can combine together in a record of how each aspect is extended.

5.1.4 Applying or Composing Extensions

The ultimate purpose of an extension `ext` is to be applied to some value `val`, which in Scheme syntax is written `(ext val)`.

But interestingly, extensions can be composed, such that from two extensions `ext1` and `ext2` you can extract an extension `(compose ext1 ext2)`, also commonly written `ext1` ∘ `ext2`, that applies `ext1` to the result of applying `ext2` to the argument value. And since I am discussing first-class extensions in Scheme, you can always define the `compose` if not yet defined, as follows, which is an associative operator with the identity function `id` as neutral element:

```
(define compose (λ (ext1 ext2) (λ (val) (ext1 (ext2 val)))))
(define id (λ (val) val))
```

Now if I were discussing second-class extensions in a restricted compile-time language, composition might not be definable, and not expressible unless available as a primitive. That would make extensions a poorer algebra than if they could be composed. With composition, strict extensions for a given type of values are a monoid (and general extensions are a category). Without composition, extensions are just disjointed second-class constants without structure. I will show later that this explains why in a second-class setting, Single inheritance is less expressive than mixin inheritance and multiple inheritance.

5.1.5 Top Value

Now, sooner or later, composed or by itself, the point of an extension is to be applied to some value. When specifying software in terms of extensions, what should the initial base value be, to which extensions are applied? One solution is for users to be required to somehow specify an arbitrary base value, in addition to the extension they use.

A better solution is to identify some default value containing the minimum amount of information for the base type `V` that is being strictly extended. Each extension then

transforms its "inherited" input value into the desired extended output value, possibly refining the type `V` along the way, such that the initial type is the "top" type of this refinement hierarchy. The initial base value is also called a "top value" $\top$, and somewhat depends on what monoidal operation is used to extend it as well as the domain type of values.

- For the type `Record` of records, $\top$ `= record-empty` the empty record.
- For the type `Number` of numbers, $\top$ `= 0` if seen additively, or `1` if seen multiplicatively, or `-`∞ (IEEE floating-point number) seen with `max` as the operator, or `+`∞ with `min`.
- For the type `Pointer` of pointers into a graph of records, $\top$ `= null`, the universal null pointer.[2]
- For the type `Type` of types (in a compiler, at the meta-level), $\top$ `= Any`, the top type ("contains everything, about which you know nothing") that you refine, or the bottom type $\bot$ `= Nothing` ("contains nothing, about which you know everything") that you extend.
- For any function type in a language with partial functions, $\top$ `= abort`, a function that never returns regularly, and instead always aborts regular evaluation and/or throws an error.
- For the type `Lazy` of lazy computations, which would be an appropriate default in a language with lazy evaluation, even and especially a pure functional language with partial functions, such as Haskell or Nix, $\top$ `= (lazy` $\bot$`)` is a universal top value, where $\bot$ `= (abort)` is a computation that never terminates normally. Indeed, `(lazy` $\bot$`)` carries no useful information but can be passed around, and has "negative infinite" information if you try to force, open or dereference it, which is much less than the 0 information provided by a regular null value. In Scheme, one may explicitly use the `delay` primitive to express such laziness, though you must then explicitly `force` the resulting value rather than having the language implicitly force computations whenever needed.
- Last but not least for the type `Any` of arbitrary values of any type, which would be an appropriate default in a language with eager evaluation, any value could do as default, but often there is a more colloquial default. In C, you would use 0 in integer context and the `NULL` pointer in pointer context. In modern C++, similarly but with the `nullptr` instead of `NULL`. In Java, `null`. In Lisp, `NIL`. In Scheme, the false boolean `#f`, or, in some dialects, a special unit value `(void)` (the null value `'()` being traditionally used only for lists). In JavaScript, `undefined`, `null` or an empty record `{}`. In Python, `None`. Various languages may each offer some value that is easier to work with as a default, or some library may offer an

[2] Hoare called his 1965 invention of null his "billion dollar mistake". Now, to Hoare's credit, his invention of classes in the same article, for which he vaguely suggests the semantics of single inheritance as Dahl and Nygaard would implement after his article, yet that he (and they) wrongfully assimilate to subtyping, was a trillion dollar happy mistake. Overall, the effect of his article (Hoare 1965) was probably net vastly positive.

arbitrary value to use as such. Some strongly typed applicative languages will offer no universal such value, though, and then some ad hoc arbitrary value must be provided for each type used.

In the context of extensions for objects, OO languages usually use `Record` as their top type, or some more specific `Object` subtype thereof that carries additional information as instance of Prototype or Class (depending on the language), with some null value or empty object as their top value. Your mileage may vary, but in my examples below, I will be more ambitious as to how widely applicable OO techniques can be, and, using Scheme as my language, I will choose the `Any` type as my top type, and the false boolean `#f` as my top value:

```
(define top #f)
```

5.1.6 Here there is no Y

Looking at the type signature `(V → V) → V` for the process of obtaining a value from a strict extension (and every extension is strict for the top type), one may be tempted to say "I know, this is the type of the fixpoint combinator Y" and propose the use of said combinator.

The idea would indeed work in many cases, as far as extracting a value goes: for any lazy type `V` (and similarly for function types), any extension that would return a useful result applied to `(lazy ⊥)` passing it as argument to the lazy `Y` combinator would also yield a result. And indeed the results will be the same if the extension wholly ignores its argument, as is often the intent in those situations. Typically, you'd compose extensions, with one "to the right" ignoring its argument, overriding any previous value (or lack thereof, as the default default is a bottom computation), and returning some default value that is more useful in context (including not being a bottom computation); further extensions "to the left" then build something useful from that default value.

However, if there is no such overriding extension, then the results would not necessarily be the same between applying the extension or passing it to `Y`. For instance, given the extension `(λ (x) (lazy-cons 1 x))` for some `lazy-cons` function creating a co-inductive stream of computations (as opposed to an inductive list of values as with the regular Scheme `cons`), applying the extension to `(lazy ⊥)` yields a stream you can destructure once, yielding `1` as first value, and "exploding in your face" if you try to destructure the rest. Meanwhile, applying the `Y` combinator yields an infinite stream of 1's.

Then comes the question of which answer is more appropriate. Using the `Y` combinator only applies to functions and lazy values, the latter being isomorphic to nullary functions that always return the same cached result; it doesn't apply and therefore isn't appropriate in the general case of eager values. Meanwhile, applying extensions to a top value is always appropriate; it also corresponds better to the idea of specifying a value by starting from no specific information then refining bit by bit until a final value is reached; it does require identifying a type-dependent top value to start from, but there is an obvious such value for the types where the fixpoint combinator applies; finally, it is easier to understand than a fixpoint combinator and arguably more intuitive.

All in all, the approach of applying extensions to a top value is far superior to the approach of using a fixpoint combinator for the purpose of extracting a value from an extension. Thus, as far as one cares about extensibility: *here, there is no Y*.[3]

5.2 Minimal First-Class Modularity

5.2.1 Modeling Modularity (Overview)

To model Modularity in terms of FP, I need to address several issues, each time with functions:

- First, programmers must be able to define several independent entities. For that, I introduce records, as functions from identifiers to values.
- Second, programmers must be able to use existing modularly-defined entities. For that, I introduce the notions of module context as a record, and modular definition as function from module context to entity.
- Third, programmers must be able to publish the entities they define as part of the modular context that can be used by other programmers. For that, I introduce the notion of module as record, and (open) modular module definition, as function from module context to module.
- Last, programmers need to be able to link together those independent modular definitions into complete programs that end-users can run. For that, I introduce closed modular module definitions, and show how to resolve the open references.

End-users are provided with a "linked" program as a resolved module context from which they can invoke a programmer-defined "main" entry-point with their choice of arguments.

5.2.2 Records

Before I model Modularity as such, I shall delve deeper into the modeling of Records, that are the usual substrate for much of Modularity.

Record Nomenclature I will follow Hoare (Hoare 1965) in calling "record" the concrete representation of data that contains zero, one or many "fields". A typical low-level implementation of records is as consecutive "words" (or "characters") of "computer store" or "storage space", all of them typically mutable. But for the sake of studying semantics rather than performance, this book will discuss higher-level implementations, immutable, and without notion of consecutive words. By contrast, Hoare uses "object" to refer to more abstract entities from the "problem" being "modeled", and an "object" has higher-level "attributes". An object and some or all of its attributes might be simply represented as a record and its fields, but other representations are possible.

[3] With apologies to Primo Levi (Levi 1947).

Other authors at times have used other words for records or other representations of the same concept: struct, structure, data structure, object, table, hash-table, tuple, named tuple, product, indexed product, entity, rows, etc. Their fields in various traditions and contexts may have been called such things as: methods, slots, attributes, properties, members, variables, functions, bindings, entries, items, keys, elements, components, columns, etc. Mapping which terms are used in any given paper to those used in this or another paper is left as an exercise to the reader.

Typing Records I could be content with a simple type `Record`, but then every access would be require some kind of type casting. Instead, I will propose slightly more elaborate types.

Given types `V`, `W`, `X`... for values, `I`, `J`... for identifiers or other indexes, consider records or sets of bindings as values of an index product $\prod$`R = i:I` $\rightarrow$ R_i wherein to each value `i` in the index set `I`, the record will associate a value of type R_i, where `R` is a schema of types, a function from `I` to `Type`.

To simplify this model, a pure functional record of type $\prod$`R` can be seen as an indexed function from the domain `I` of indexes to R_i for each `i`. When invoked on a value not in `I`, the function may return any value, or diverge. To further simplify, and for the sake of modeling records as first-class functions, when using Scheme, for indexes I will use symbols (interned strings) instead of identifiers (that in Scheme are symbols enriched with scoping and provenance information, used for second-class macro-expansion).

For example, where `Number` is the type of numbers and `String` the type of strings, I could have a record type

```
type ∏R = {x: Number, y: Number, color: String}
```

and a point `point-q` of type $\prod$`R` defined as follows:

```
(define point-q
  (record (x 3) (y 4) (color "blue")))
```

Implementing Records I will call *identifier* some large or infinite type of values over which equality or inequality can be decided at runtime by using a suitable primitive, or calling a suitable function. Since I use Scheme for my examples, I will use the symbols for identifiers (that I will later extend with booleans), the `equal?` primitive for testing equality, and the `(if` *`condition then-clause else-clause`*`)` special form for conditionals. In other programming languages that lack symbols as a builtin functionality, they can be implemented as interned strings, or instead of them, uninterned strings or numbers can be used as identifiers. And if you only care about the pure λ-calculus, there are many embeddings and encodings of unary or binary numbers, and lists or trees thereof, that will do.

Programming languages usually already provide some data structure for records with second-class identifiers, or "dictionaries" with first-class identifiers, or have some library that does. For instance, while the core Scheme language has no such data structure, each implementation tends to have its own extension for this purpose, and there are multiple standard extensions for records or hash tables. Many papers and experiments use a linked list of (key, value) pairs, known in the Lisp tradition as an

alist (association list), as a simple implementation for records; alists don't scale, but don't need to in the context of such experiments. Nevertheless to make the semantics of records clear, I will provide a trivial purely functional implementation, that also doesn't scale, but that is even simpler.

The basic reference operator is just function application:

```
(define record-ref (λ (key) (λ (rec) (rec key))))
```

The empty record can be represented as a function that always fails, such as `(λ (_) (error "empty record"))`. Now, the "fail if not present" representation is great when implementing a (notionally) statically typed model. But for a dynamic environment, a "nicer" representation is as a function that always returns a language-wide top value, such as `undefined` in JavaScript. This is especially the case in this book where we don't have space to discuss error handling protocols. But you should use what makes sense in your language. In portable Scheme, I will use:

```
(define record-empty (λ (_) #f))
```

To extend a record with one key-value binding, you can use

```
(define record-cons (λ (key) (λ (val) (λ (rec) (λ (i)
  (if (eqv? key i)
     val
     (rec i)))))))
```

Note how this trivial implementation does not support getting a list of bindings, or removing a binding. Not only does one not need these features to implement OO,[4] they constitute a "reflection" API that if exposed would interfere with various compiler optimizations, the use of which is actively rejected when statically typing records. However, there are other reasons why my implementation is not practical for long-running programs: it leaks space when a binding is overridden, and the time to retrieve a binding is proportional to the total number of bindings (including overridden ones) instead of being logarithmic in the number of visible bindings only as it could be in a pure functional implementation based on balanced trees, or constant time, for a stateful implementation based on either known field offsets within an array, or hashing.[5]

Merging Records Given a list of bindings as pairs of an identifier and a value, you can define a record that maps each identifier to the value in the first binding with that identifier in the list, if any (or else, errors out, diverges, returns null, or does whatever a record does by default). Conversely, given a list of identifiers and a record, you can get a list of bindings as pairs of an identifier and the value that the record maps it to. Most practical implementations of records support extracting from a record the list of identifiers it binds, and such a list of all bindings in the record; but this "reflection" feature is not necessary for most uses of records. Indeed, the trivial implementation I will use, wherein records are functions from identifier to value, doesn't; and some more

[4] I generate HTML for my presentations using exactly this implementation strategy. The Scheme implementation I use has builtin record support, and there are now somewhat portable libraries for records in Scheme; but I made it a point to use a minimal portable object system to show the feasability and practicality of the approach; and this way the code can be directly ported to any language with higher-order functions.

[5] The nitpicky would also account for an extra square root factor due to the limitations of physics (Ernerfeldt 2014).

elaborate implementations will deliberately omit runtime reflection, and only support second-class knowledge of what identifiers are bound in a record.

Finally, given a list of records and for each record a set of identifiers (that may or may not be the set of all identifiers bound by it, possibly extracted via reflection), you can merge the records along those sets of identifiers, by converting in order each record to a list of bindings for its given set of identifiers, appending those lists, and converting the appended result to a record.

5.2.3 Modular definitions

Now I can introduce and model the notion of modular definition: a modular definition is a way for a programmer to specify how to define an entity of some type `E` given some modular context, or just *module context*, of type `C`. The module context contains all the available software entities, that were defined in other modules by other programmers (or even by the same programmer, at different times, who doesn't presently have to hold the details of them in their limited brain). And the simplest way to model a modular definition as a first-class value, is as a function of type `C` → `E`, from module context to specified entity.[6]

Typically, the module context `C` is a set of bindings mapping identifiers to useful values, often functions and constants, whether builtin the language runtime or available in its standard library. Now, I already introduced types for such sets of bindings: record types. And as a language grows in use and scope, the module context will include further libraries from the language's wider ecosystem, themselves seen as sets of bindings of their respective record types, accessed hierarchically from a global module context, that now contains "modules" (records) as well as other values (or segregated from regular values at different levels of the module hierarchy). In any case, the global module context is typically a record of records, etc., and though many languages have special restrictions on modules as second-class entities, for my purpose of modeling the first-class semantics of modularity, I may as well consider that at runtime at least, a module is just a regular record, and so is the global module context, of type `C = ∏R`.

For instance, I could modularly specify a function `ls-sorted` that returns the sorted list of filenames (as strings) in a directory (as a string), from a module context of type `∏R` that provides a function `ls` of type `String → List(String)` and a function `sort` that sorts a list of strings:

```
(define ls-sorted (λ (ctx) (compose (ctx 'sort) (ctx 'ls))))
```

Note how in the above code snippet, I model records as functions from symbol to value, and to extract a binding from a record, one thus calls it with a symbol as single argument. The functions extracted are then chained together, as in the `compose` function I defined earlier (that I could have used if I extracted it from the module context, or could otherwise assume it was a language builtin).

[6] A Haskeller may well interpret "modular" in this book as meaning "in the reader monad of a module context", in that a modular something will be a function from the module context to that something. This is correct, but beware that this is only half the story. The other half is in what it means for something to be a module context, rather than any random bit of context. And we'll see shortly that the answer involves open recursion and fixpoints. Informally, modular means that locally you're dealing with part of a whole, and globally, the whole will be made out of the parts in the end.

5.2.4 Open Modular Definitions

Now, programmers usually do not specify just a single entity of type `E`, but many entities, that they distinguish by associating them to identifiers. That is, they modularly define a *module* of type `∏P`. A modular module definition is thus "just" a function from record to record: the input record is the modular context of type `∏R`, and the output record is the specified module of type `∏P`. I will say that the identifiers bound in `∏R` are *required* by the modular definition (or *referenced* by it), whereas the identifiers bound in `∏P` are *provided* by the modular definition (or *defined* by it).

In general, I call a modular definition "open", inasmuch as some entities may be *required* that are not *provided* by the modular definition, and must be provided by other modular definitions. Now, the notion that the modular definition "provides" entities supposes that these entities will be available, bound to well-known identifiers in the module context. There are many strategies to realize this provision:

- The modular definition can be a paired with a single identifier, under which the entity is intended to be bound in a complete module context.
- The modular definition can provide multiple bindings, wherein the entity defined is a module, to be merged as a record into the complete module context.
- The first strategy above can be adapted to a hierarchical namespace of nested records, by pairing the modular definition with a path in the namespace, i.e. a list of identifiers.
- For multiple bindings, the second strategy can also be paired with a path; or the merging of records can be recursive, at which point a strategy must be devised to determine how deep to recurse, for instance by somehow distinguishing "modules" from regular "records".

In the end, programmers could somehow specify how their modular definition will extend the module context, with whatever merges they want, however deeply nested—which will lead them to modular extensibility. But for now, I will abstract over which strategy is used to assemble many modular definitions together.

5.2.5 Linking Modular Module Definitions: Y

I will call a modular module definition "closed" when, after whatever assembly of individual modular definitions happened, it specifies the global module context of an entire program, wherein every entity required is also provided. A closed modular module definition is thus of type `∏R → ∏R`.

Then comes the question: how can I, from a closed modular module definition, extract the actual value of the module context, of type `∏R`, and thereby realize the program that was modularly specified?

This module realization function I am looking for is of type `(C → C) → C` where `C = ∏R`. Interestingly, I already mentioned a solution: the fixpoint combinator `Y`. And whereas it was the wrong solution to resolve extensions, it is precisely the right solution to resolve modular definitions: the `Y` combinator "ties the knots", links each reference

requiring an entity to the definition providing it, and closes all the open loops. It indeed does the same in a FP context that an object linker does in the lower-level imperative context of executable binaries: link references in open modular definitions to defined values in the closed result.

If there remain identifiers that are required but not provided, there will be an error—or non-termination, or some default value that will not make sense, at runtime, or at compile-time, depending on how sophisticated the exact implementation is (a type-checker might catch the missing requirement, for instance). If there remain identifiers that are provided but not required, and they are not otherwise (meant to) be used via reflection, then a "tree shaker" or global dead code optimizer may eliminate them.

5.2.6 Digression: Scheme and FP

> Purely applicative languages are poorly applicable.
>
> Alan Perlis

Here are two ways in which Scheme departs from the theoretical model of Functional Programming, that affect their suitability to modeling Object Orientation. These discrepancies also apply to many (but not all) other programming languages.

Many Y combinators First, there are many variants to the fixpoint (or fixed-point) combinator Y, and the pure applicative Y combinator you could write in Scheme's pure subset of the λ-calculus is actually quite bad in practice. Here is the applicative Y combinator, expressed in terms of the composition combinator B and the duplication combinator D:[7]

```
(define B (λ (x) (λ (y) (λ (z) (x (y z))))))
(define applicative-D (λ (x) (λ (y) ((x x) y))))
(define applicative-Y (λ (f) (applicative-D ((B f) applicative-D))))
(define applicative-Y-expanded
  (λ (f) ((λ (x) (λ (y) ((x x) y)))
          (λ (x) (f (λ (y) ((x x) y)))))))
```

The Y combinator works by composing the argument function `f` with indefinite copies (duplications) of itself (and accompanying plumbing). In this applicative variant, the first, minor, issue with this combinator is that it only works to compute functions, because the only way to prevent an overly eager evaluation of a computation that would otherwise diverge is to protect this evaluation under a λ. I happen to have chosen a representation of records as functions, such that the applicative Y still directly applies; if not, I may have had to somehow wrap my records in some sort of function, at which point I may as well use the lazy Y below, or switch to representing modular contexts as records of functions, instead of functions implementing or returning records.

[7] A simple way to test the `applicative-Y` combinator, or the subsequent variants `applicative-Y-expanded` and `stateful-Y` is to use it to define the factorial function: `(define fact (applicative-Y (λ (f) (λ (n) (if (<= n 1) n (* n (f (1- n))))))))` and you can then test that e.g. `(fact 6)` returns `720`.

The second, major, issue with the applicative Y is that the pure applicative λ-calculus by itself has no provision for sharing non-fully-reduced computations, only for sharing (fully-reduced) values;[8] therefore the fixpoint computations are duplicated, and any information used along the way will have to be recomputed as many times as computations are duplicated, which can grow exponentially fast as the computation involves deeper sub-computations. In some cases, the eager evaluation may never terminate at all when lazy evaluation would, or not before the end of the universe. And of course, if there are any non-idempotent side effects, they too will be potentially duplicated a large number of times.

There are several potential alternatives to the practically inapplicable applicative Y combinator: (1) a stateful Y combinator, (2) a lazy Y combinator, or (3) a second-class Y combinator.

A stateful Y combinator is what the `letrec` construct of Scheme provides (and also its `letrec*` variant, that the internal `define` expands to): it uses some underlying state mutation to create and initialize a mutable cell that will hold the shared fixpoint value, with the caveat that you should be careful not to access the variable before it was initialized.[9] If the variable is only accessed after it is initialized, and the rest of the program is pure and doesn't capture intermediate continuations with Scheme's famous `call/cc`, the mutation cannot be exposed as a side-effect, and the computation remains overall pure (deterministic, referentially transparent), though not definable in terms of the pure applicative λ-calculus. Note however how in the definition below, `p` still needs be a function, and one must η-convert it into the equivalent but protected `(λ (y) (p y))` before passing it to `f`, to prevent access to the variable `p` before its initialization (and `f` must also be careful not to invoke this protected `p` before returning):

```
(define (stateful-Y f) (letrec ((p (f (λ (y) (p y))))) p))
```

A second solution is to use a lazy Y. In a language like Nix (where `λ (f)` is written `f:`, and `let` like Scheme `letrec` recursively binds the variable in the definition body), you can simply define `Y = f: let p = f p; in p`. In Scheme, using the convention that every argument variable or function result must be protected by `delay`, and one must `force` the delayed reference to extract the result value, you would write:[10]

```
(define (lazy-Y f) (letrec ((p (f (delay p)))) p))
```

[8] You could of course emulate sharing by implementing state monadically, or through a virtual machine interpreter, inside the applicative λ-calculus, and then reimplementing your entire program on top of that richer calculus. But that would be a global transformation, not a local one, and in the end, it's the new stateful paradigm you would be using, not the pure applicative λ-calculus anymore.

[9] A variable won't be accessed before it is used if you're immediately binding the variable to a λ expression, but may happen if you bind the variable to a function application expression, wherein the variable is passed as argument without wrapping it in a λ, or the λ it is wrapped in is called before the evaluation of this expression completes. The Scheme language does not protect you in this case, and, in general, could not protect you without either severely limiting the language expressiveness, or solving the halting problem. Various languages and their implementations, depending on various safety settings they might have or not, may raise a "variable not bound" exception, use a special value such as `#!void` or `null` or `undefined` that is not usually part of the expected type, or access uninitialized memory potentially returning nonsensical results or causing a latter fandango on core, etc.

[10] Again, a simple way to test the lazy Y combinator is to use it to define the factorial function: `(define fact (lazy-Y (λ (f) (λ (n) (if (<= n 1) n (* n ((force f) (1- n))))))))` and you can then test that e.g. `(fact 6)` returns 720. Note that I do without wrapping of `n` in a `delay`, but `f` itself is a delayed function value to fit the calling convention of `lazy-Y`, and I therefore must `force` it before I call it. The subsequent variants of `lazy-Y` can be tested in the same way.

Or, if you want a variant based on combinators:

```
(define lazy-B (λ (x) (λ (y) (λ (z) ((force x) (delay ((force y) z)))))))
(define lazy-D (λ (x) ((force x) x)))
(define lazy-Y-with-combinators
  (λ (f) (lazy-D (delay ((lazy-B f) (delay lazy-D))))))
(define lazy-Y-expanded
  (λ (f) ((λ (x) ((force x) x))
          (delay (λ (x) ((force f) (delay ((force x) x))))))))
```

One advantage of a lazy Y is that evaluation is already protected by the `delay` primitive and thus can apply to any kind of computation, not just to functions; though if you consider that `delay` is no cheaper than a λ and indeed uses a λ underneath, that's not actually a gain, just a semantic shift. What the `delay` does buy you, on the other hand, is sharing of computations before they are evaluated, without duplication of computation costs or side-effects.[11] (Note that `delay` can be easily implemented on top of any stateful applicative language, though a thread-safe variant, if needed, is somewhat trickier to achieve.)

A third solution, often used in programming languages with second-class OO only (or languages in which first-class functions must terminate), is for the `Y` combinator (or its notional equivalent) to only be called at compile-time, and only on modular definitions that abide by some kind of structural restriction that guarantees the existence and well-formedness of a fixpoint, as well as e.g. induction principles to reason about said fixpoint. Also, the compile-time language processor usually doesn't expose any side-effect to the user, such that there is no semantic difference whether its implementation is itself pure or impure, and uses a fixpoint combinator or any other representation for recursion. Since I am interested in first-class semantics for OO, I will ignore this solution in the rest of this book, and leave it as an exercise for the reader.

Function Arity Functional Programming usually is written with unary functions (that take exactly one argument), and to express more than one argument, you "curry" it: you define a function of one argument that returns a function that processes the next argument, etc., and when all the arguments are received you evaluate the desired function body. Then to apply a function to multiple arguments, you apply to the first argument, and apply the function returned to the second argument, etc. The syntax for defining and using such curried functions is somewhat heavy in Scheme, involving a lot of parentheses. By contrast, the usual convention for Functional Programming languages is to do away with these extra parentheses: in FP languages, two consecutive terms is function application, which is left-associative, so that `f x y` is syntactic sugar for `((f x) y)`; and function definition is curried, so that λ `x y . E` is syntactic sugar

[11] Whether wrapped in a thunk, an explicit delay, an implicitly lazy variable, a call-by-name argument, or some other construct, what is interesting is that ultimately the fixpoint combinator iterates indefinitely on a *computation*, and this wrapping is a case of mapping computations into values in an otherwise call-by-value model that requires you to talk about values. In a calculus such as call-by-push-value (Blain Levy 1999), where values and computations live in distinct type universes, the fixpoint combinator would clearly be mapping computations to computations without having to go through the universe of values. Others may say that the fixpoint operation that instantiates prototypes is coinductive, rather than inductive like the definitions of data types.

for λ x . λ y . E.

Thus, there is some syntactic discrepancy that makes code written in the "native" Functional style look ugly and somewhat hard to follow in Scheme. Meanwhile, colloquial or "native" Scheme code may use any number of arguments as function arity, and even variable numbers of arguments, or, in some dialects, optional or keyword arguments, which does not map directly to mathematical variants of Functional Programming; but it is an error to call a function with the wrong number of arguments.

One approach to resolving this discrepancy is to just cope with the syntactic ugliness of unary functions in Scheme, and use them nonetheless, despite Lots of Insipid and Stupid Parentheses.[12]

A second approach is to adopt a more native Scheme style over FP style, with a variety of different function arities, making sure that a function is always called with the correct number of arguments. This approach blends best with the rest of the Scheme ecosystem, but may hurt the eyes of regular FP practitioners, and require extra discipline (or extra debugging) to use.

A third approach is to use Scheme macros to automatically curry function definitions and function applications, such that a function called with insufficient arguments becomes the same function partially applied, whereas a function called with too many arguments becomes a call of the result of calling the function with the correct number of arguments, with the rest of the arguments. Such macros can be written in a few tens of lines of code, though they incur a performance penalty; an efficient variant of these macros that statically optimize function calls could be much larger, and might require some level of symbiosis with the compiler.

I have implemented variants of my minimal OO system in many combinations of the above solutions to these two issues, in Scheme and other languages. For the rest of this book, I will adopt a more "native" Scheme style, assuming `stateful-Y` and using multiple function arities that I will ensure are carefully matched by function callers; yet to keep things simple and portable, I will avoid variable arity with optional arguments, rest arguments or keyword arguments. As a result, the reader should be able both to easily copy and test all the code in this book at their favorite Scheme REPL, and also easily translate it to any other language that sports first-class higher-order functions.

And with these issues settled, I will close this digression and return to rebuilding OO from first principles.

[12] Detractors of the LISP language and its many dialects and derivatives, among which Scheme is prominent, invented the backronym "Lots of Insipid and Stupid Parentheses" to deride its syntax. While Lispers sometimes yearn for terser syntax—and at least one famous Schemer, Aaron Hsu, adopted APL—to them, the parentheses, while a bit verbose, are just a familiar universal syntax that allows them to quickly understand the basic structure of any program or data, even when they are unfamiliar with the syntactic extensions it uses. By contrast, in most "blub" languages, as Lispers call non-Lisp languages, parentheses, beyond function calls, carry the emotional weight of "warning: this expression is complex, and doesn't use the implicit order of operations". Ironically, the syntactic and semantic abstraction powers of Lisp allow for programs that are significantly shorter than their equivalent in a mainstream language like Java, and as a result have fewer parentheses overall, not counting all kinds of brackets. It is therefore not the number of parentheses, but their density, that confuses mainstream programmer, due to unfamiliarity and emotional connotations. Now, it may well be that the same abstraction powers of Lisp make it unsuitable for a majority of programmers incapable of mastering such powers. As an age of AI programmers approaches that will have abstraction powers vastly superior to the current average human programmer, it remains to be seen what kind of syntax they prefer.

5.3 Minimal First-Class Modular Extensibility

5.3.1 Modular Extensions

One can combine the above extensibility and modularity in a minimal meaningful way, as modular extensibility. I will call "modular extension" a modular definition for an extension. Thus, given a module context of type `C` (typically a record with `C = ∏R`), a type `V` for the "inherited" value being extended and `W` for the extended value being "provided", an (open) modular extension is a function of type `C → V → W`. When `W` is the same as `V`, or a subtype thereof, I will call it a strict (open) modular extension. When `W = V = ∏P` for some record type `∏P`, I will call it a modular module extension. When `C = V = W`, I will call it a closed modular extension, which as I will show can be used as a specification for a value of that type.

5.3.2 Composing Modular Extensions

While you could conceivably merge such modular extensions, the more interesting operation is to compose them, or more precisely, to compose each extension under the module context and bound identifier, an operation that for reasons that will soon become obvious, I will call Mixin Inheritance (for modular extensions):

```
(define mix (λ (c p) (λ (s) (λ (t) (c s (p s t))))))
```

The variables `c` and `p` stand for "child" and "parent" specifications, wherein the value "inherited" by the composed function will be extended (right to left, with the usual function-as-prefix syntax) first by `(p s)` then by `(c s)`, where `s` is the module context (also called `self` in many contexts, for reasons that will become obvious)[13] and `t` is the inherited value (also called `super` in the same contexts, for the same reasons).[14] The function can also be written with `compose`, eliding the "super" variable:

[13]The `self` argument is the one involved in open recursion or "late binding"; it embodies the *modular* side of OO. It is called `self` because it is destined to be bound as the fixpoint variable of a fixpoint operator, wherein it refers to the very entity being defined. The name `self` is used in Smalltalk, Scheme, Self, Python, Jsonnet, Nix, many more languages, and in a lot of the literature about OO semantics. In Simula, and after it, in C++, Java, JavaScript or Scala, the `this` keyword is used instead. Note however, that I am currently discussing a variant of Prototype OO, as in Self, Jsonnet, Nix, JavaScript, where the `self` or `this` is indeed the open recursion variable. In Class OO language, the definition being one of a type descriptor, not of a record, the open recursion variable would instead be something like `Self`, `MyType` or `this.type`, though there is even less standardization in this area. See below the discussion of the meaning of "object" in Prototype OO vs Class OO.

[14]The `super` argument refers to the partial value computed so far from *parent* specifications (composed to the right, with the usual composition order); the rightmost seed value of `super` when computing the fixed point is a "base" or "top" value, typically an empty record, possibly enriched with metadata or ancillary fields for runtime support. `super` embodies the *extensible* side of OO, enabling a specification to build on values *inherited* from parent specifications, add aspects to them, override aspects of them, modify aspects of them, etc., in an extension to the computation so far. `super` is the variable or keyword used in Smalltalk and in many languages and object systems after it, such as Java or Python, to access inherited methods. In CLOS you'd use `call-next-method`. In C++ you cannot directly express that concept in general, because you have to name the superclass whose method (or "(virtual) member function") you want to call, so you can't directly write traits that inherit "super" behavior along the class precedence list; but it works if you restrict yourself to single inheritance, or if you use template metaprogramming to arrange to pass a superclass, or list or DAG of superclasses, as argument to your template, and manually reimplement mixin inheritance (Smaragdakis and Batory 2000), or if you're adventurous, multiple inheritance, on top of C++.

```
(define mix (λ (c p) (λ (s) (compose (c s) (p s)))))
```

Modular extensions and their composition have nice algebraic properties. Indeed, modular extensions for a given context form a category, wherein the operation is composition with the `mix` function, and the neutral element `idModExt` is the specification that "extends" any and every value by returning it unchanged, as follows:[15]

```
(define idModExt (λ (s) (λ (t) t)))
```

5.3.3 Closing Modular Extensions

A closed modular extension is a function of type `C → C → C`, i.e. a modular extensible module specification where `C = V = W`. In the common case that `C` is a record, this means that an extension is provided for every identifier required.

As with closed modular module definitions before, the question is: how do you get from such a closed modular module extension to an actual module definition where all the loops are closed, and every identifier is mapped to a value of the expected type? And the way I constructed my model, the answer is simple: first, under the scope of the module context, you apply your extension to the top value for a module context (usually, that's the empty record); then you have reduced your problem to a regular modular module definition `∏R → ∏R`, at which point you only have to compute the fixpoint. I will call this operation instantiation for modular extensions:

```
(define fix (λ (t) (λ (m) (Y (λ (s) ((m s) t))))))
```

In this expression, `t` is the top value for the type being specified (typically the empty record, for records), `m` is the modular extension, and `s` is the fixpoint variable for the module context being computed.

5.3.4 Default and non-default Top Type

Assuming some common top type `Top` and default value `top` in that type (I will use `Any` and `#f` in my example Scheme implementation), I will define the common instantiation operation for modular extensions:

```
(define fixt (fix top))
```

or to inline `fix` in its definition:

```
(define fixt (λ (m) (Y (λ (s) ((m s) top)))))
```

Note that if the language-wide top type is too wide in some context: for instance I chose `Any` as my top type in Scheme, with `#f` as my top value; but you may want to choose the narrower `Record` as your top type, so as to be able define individual methods, with a `empty-record` as default value. Then you can compose your modular extension with a modular extension as follows to the right, that throws away the previous value or

[15] As usual, a change of representation from `p` to `mp = mix p`, with inverse transformation `p = mp idModExt`, would enable use of the regular `compose` function for composition of specifications. Haskellers and developers using similar composition-friendly languages might prefer this kind of representation, the way they like van Laarhoven lenses (O'Connor 2012); yet, Oliveira (Oliveira 2009) or the `Control.Mixin.Mixin` library (part of the `monadiccp` package), instead both use a slightly different representation that compared to mine swaps the order of arguments of the `self` and `super` arguments. I will stick with my representation, also shared by the Nix standard library, as it makes my explanations, and, in later sections, the types of specifications, slightly simpler.

computation (ignores its `super` argument) and returns the new default value regardless of context (ignores its `self` argument; unless that default is extracted from the context):
`(define record-spec (λ (self) (λ (super) empty-record)))`
I could then equivalently define a variant of `fix` specialized for records in any of the following ways:

```
(define fix-record (fix empty-record))
(define fix-record (λ (m) (Y (λ (s) ((m s) empty-record)))))
(define fix-record (λ (m) (fixt (mix m record-spec))))
```

Note that because it ignores its `super` argument and thus throws away any inherited value, the `record-spec` modular extension must appear last, or at least after any modular extension the result of which isn't to be ignored. Why not make `empty-record` the language-wide default? Because the language-wide default will apply not just to the specification of records, but also to the specification of individual fields of each record, and in this more general context, the default value `#f` is possibly more efficient at runtime, and definitely more colloquial—therefore more efficient in the most expensive resource, human-time.

5.3.5 Minimal OO Indeed

The above functions `mix` and `fix` are indeed isomorphic to the theoretical model of OO from Bracha and Cook (Bracha and Cook 1990) and to the actual implementation of "extensions" in nixpkgs (Simons 2015).[16]. This style of inheritance was dubbed "mixin inheritance" by Bracha and Cook;[17] and the two functions, that can easily be ported to any language with first-class functions, are enough to implement a complete object system.

How does one use these inheritance and instantiation functions? By defining, composing and closing modular extensions of type `C → V → V` where `C` is the type of the module context, and `V` that of the value under focus being extended:

[16]My presentation of mixin inheritance is actually slightly more general than what Bracha, Cook or Simons did define, in that my definition is not specialized for records. Indeed, my closed modular extensions work on values of any type; though indeed to fully enjoy modularity, modular extensions work better when the module context is a record, or somehow contains or encodes a record. But my theory also importantly considers not just closed modular extensions, but also the more general open modular extensions, for which it is essential that they universally apply to target values of any type, with a module context of any type. I can therefore claim as my innovation a wider, more general understanding of mixin inheritance, of which there is no evidence in earlier publications.

[17]The name "mixin" originally comes from Flavors (Cannon 1979), inspired by the ice cream offerings at Emack & Bolios (as for the concept itself, it was inspired both by previous attempts at multiple inheritance in KRL (Bobrow and Winograd 1976) or ThingLab (Borning 1977), combined with the ADVISE facility (Teitelman 1966)). However, Flavors offers full multiple inheritance (and was the first system to do it right), whereas the "mixins" of Bracha and Cook are a more rudimentary and more fundamental concept, that does not include automatic linearization of transitive dependencies. Also, "mixins" in Flavors are not distinguished by the language syntax or by the compiler; they are just abstract classes not meant to be instantiated, but to be inherited from (the word "abstract class" didn't exist back then); a mixin in Flavors need not make sense as a base class, and can instead be inherited as part of many different hierarchies. Since the word implies no special processing by the compiler or by the human operator, it can be dispensed with in the original context, and gladly assigned a new, useful, technical meaning. But that doesn't mean the context that made the word superfluous should be forgotten, quite the contrary. I will get back to Flavors when I discuss multiple inheritance.

`(define my-spec (λ (self) (λ (super) body ...)))`
where `self` is the module context, `super` is the inherited value to be extended, and `body ...` is the body of the function, returning the extended value.

In the common case that `V = ∏P`, and with my trivial representation of such records as `∏P = I → P` where `I` is the type of identifiers, a typical modular module extension will look like:
`(define my-spec (λ (self) (λ (super) (λ (method-id) body ...))))`
where `method-id` is the identifier for the method to be looked up, and the body uses `(super method-id)` as a default when no overriding behavior is specified.

Alternatively, this can be abstracted in terms of using a mix of one or multiple calls to this method-defining specification, that specifies a single method with a given `key` as name for a recognized value of `method-id`, and a given open modular extension function `compute-value` that takes the `self` context and the `inherited` value `(super method-id)` as arguments and returns an extended value for the method at `key`:

```
(define method-spec (λ (key) (λ (compute-value)
    (λ (self) (λ (super) (λ (method-id)
      (let ((inherited (super method-id)))
        (if (eqv? key method-id)
          (compute-value self inherited)
          inherited))))))))
```

Note how `method-spec` turns an open modular extension for a value into an open modular extension for a record (that has this value under some key). In this case, the module context `self` is the same, whereas the `super` value for the inner function `compute-value` is the specialized `(super method-id)` value extracted from the record. That's an example of how open modular extensions themselves have a rich algebraic structure, wherein you can combine, compose, decompose, extract, and otherwise operate on open modular extensions to get richer open modular extensions, and eventually build a closed modular extension that you can instantiate.

Now, where performance or space matters, you would use an encoding of records-as-structures instead of records-as-functions. Then, instead of calling the record as a function with an identifier, you would invoke a dereference function with the record as first argument and the identifier as second argument. But with some small overhead, the records-as-functions encoding is perfectly usable for many simple applications.[18] Also, in a more practical implementation, the inherited value in the `method-spec` would be made lazy, or would be wrapped in a thunk, to avoid unneeded computations (that might not even terminate); or for more power, the `compute-value` function would directly take `super` as its second argument, and `(super method-id)` would only be computed in the second branch. In a lazy context, `lazy-method-spec` could also

[18] I notably use this technique to generate all my slides in a pure functional way in Racket (a close cousin of Scheme). Interestingly, I could define a generic specification for slides that indicate where they are in the presentation, highlighting the name of each new section in an otherwise constant list of all sections. That specification uses the `self` context to extract the list of sections in the presentation, including sections not yet defined. It might seem impossible in an eager language, and without side-effects, to import data from slides that will only be defined later into a whichever slide is being defined now; and yet the Y combinator achieves this feat, and although I use the stateful Y for performance, a pure applicative Y also works without too much slowdown, because the substructures I recursively examine remain shallow.

directly use `lazy-record-cons` to add a binding to a record without having to eagerly compute the bound value.

Whichever way simple modular extensions are defined, they can thereafter be composed into larger modular extensions using the `mix` function, and eventually instantiate a target record from a modular extension using the `fix` function. Since I will be using records a lot, I will use the specialized `fix-record` function above. Note that since my targets are records, my minimal object system is closer to Prototype OO than to Class OO, though, as I will show, it doesn't offer "prototypes" per se, or "objects" of any kind.

5.3.6 Minimal Colored Point

I will demonstrate the classic "colored point" example in my Minimal Object System. I can define a modular extension for a point's coordinates as follows:

```
(define coord-spec
  (mix (method-spec 'x (λ (self) (λ (inherited) 2)))
       (method-spec 'y (λ (self) (λ (inherited) 4)))))
```

The modular extension defines two methods `x` and `y`, that respectively return the constant numbers 2 and 4.

I can similarly define a modular extension for a record's `color` field as follows:

```
(define color-spec
  (method-spec 'color (λ (self) (λ (inherited) "blue"))))
```

Indeed, I will check that one can instantiate a point specified by combining the color and coordinate modular extensions above, and that the values for `x` and `color` are then as expected:

```
(define point-p (fix-record (mix color-spec coord-spec)))

(point-p 'x) ;⇒ 2
(point-p 'color) ;⇒ "blue"
```

Consider how `x` is computed. `fix-record` provides the `empty-record` as the top value for mixin composition. Then, mixins are applied under call-by-value evaluation, with right-to-left flow of information across function calls, When querying the composed modular extension for method `x`, the rightmost modular extension, `coord-spec`, is applied first, and matches the key `x`; it is then passed the top value `#f` extracted as a fallback default when trying to read a missing field from `empty-record`; it proceeds to ignore that value, and return 2; that value is then returned unchanged by `color-spec`, since `x` does not match its key `color`. Similarly, the query for method `color` returns the string `"blue"`.

However, this colored point example is actually trivial: there is no collision in method identifiers between the two modular extensions, and thus the two modular extensions commute; and more importantly, the values defined by the modular extensions are constant and exercise neither modularity nor extensibility: their value-computing functions make no use of their `self` or `super` arguments. I will now show more interesting examples.

5.3.7 Minimal Extensibility and Modularity Examples

I will illustrate extensibility with this example wherein the function `add-x-spec` accepts an argument `dx`, and returns a modular extension that overrides method `x` with a new value that adds `dx` to the `inherited` value:

```
(define add-x-spec
  (λ (dx) (method-spec 'x (λ (self) (λ (inherited) (+ dx inherited))))))
```

Now I will illustrate modularity with another example wherein `rho-spec` specifies a new field `rho` bound to the Euclidean distance from the origin of the coordinate system to the point, using the Pythagorean theorem. I assume two functions `sqrt` for the square root (builtin in Scheme) and `sqr` for the square (which could be defined as `(λ (x) (* x x))`). Note how the coordinates `x` and `y` are modularly extracted from the module context `self`, which is the record being defined; and these coordinates are not provided by `rho-spec`, but have to be provided by other modular extensions to be composed with it using `mix`:

```
(define rho-spec
  (method-spec 'rho (λ (self) (λ (inherited)
    (sqrt (+ (sqr (self 'x)) (sqr (self 'y))))))))
```

I can check that the above definitions work, by instantiating the composed modular extensions `(add-x-spec 1)`, `coord-spec` and `rho-spec`, and verifying that the `x` value is indeed `3`: i.e., first (right-to-left) specified to be `2` by `coord-spec`, then incremented by `1` by `(add-x-spec 1)`. Meanwhile `rho` is `5`, as computed by `rho-spec` from the `x` and `y` coordinates:

```
(define point-r (fix-record
   (mix (add-x-spec 1)
     (mix coord-spec
          rho-spec))))

(point-r 'x) ;⇒ 3
(point-r 'rho) ;⇒ 5
```

This demonstrates how modular extensions work and indeed implement the basic design patterns of OO.

Now, note how trying to instantiate `(add-x-spec 1)` or `rho-spec` alone would fail: the former relies on the `super` record to provide a useful inherited value to extend, whereas the latter relies on the `self` context to modularly provide `x` and `y` values. Neither modular extension is meant to stand alone, but instead to be a mixin, in the sense of Flavors—an abstract class, or a trait, members of some programming language ecosystems would say. That not every specification can be successfully instantiated is actually an essential feature of modular extensibility, since the entire point of a specification is to contribute some *partial* information about one small aspect of an overall computation, that in general depends on other aspects being defined by complementary specifications.

5.3.8 Interaction of Modularity and Extensibility

Without extensibility, a modular module specification need never access the identifiers it specifies via the global module context (`self`), since it can more directly access or inline their local definition (though it may then have to explicitly call a fixpoint locally if the definitions are mutually recursive, instead of implicitly relying on a global fixpoint via the module context).

For instance, consider the following modular definition, to be merged with other specifications defining disjoint sets of identifiers. In this definition, the `case` special form of Scheme selects a clause to execute based on which constant, if any, matches its first argument. This definition provides (a) a `start` method that returns the constant 42; (b) a `length` utility function that computes the length of a list, using the open recursion through `self` for its recursion; (c) a `size` method that subtracts the `start` value from the length of a list returned by the `contents` method, while using the `length` method, provided above, for the length computation. The `contents` method is required but not provided; it must be modularly provided by another modular definition.

```
(define my-modular-def (λ (self) (λ (method-id)
   (case method-id
      ((start) 42)
      ((length) (λ (l) (if (null? l) 0 (+ 1 ((self 'length) l)))))
      ((size) (- ((self 'length) (self 'contents)) (self 'start)))
      (else #f)))))
```

Since by my disjointness hypothesis, the global specification for `start`, `length` and `size` will not be overridden, then `(self 'start)` and `(self 'length)` will always be bound to the values locally specified. Therefore, the value 42 may be inlined into the specification for `size`, and a fixpoint combinator or `letrec` can be used to define the `length` function, that can also be inlined in the specification for `size`. The `contents` method, not being provided, must still be queried through open recursion.

```
(define my-modular-def-without-global-recursion
  (let ((_start 42))
    (letrec ((_length (λ (l) (if (null? l) 0 (+ 1 (_length l))))))
      (λ (self) (λ (method-id)
        (case method-id
          ((start) _start)
          ((length) _length)
          ((size) (- (_length (self 'contents)) _start))
          (else #f)))))))
```

By contrast, with extensibility, a modular extensible module specification may usefully require *a value* for a method for which it also provides *an extension* (and not a value). The value received will then be, not that returned by the extension, but the final fully-resolved value bound to this identifier after all the extensions are accounted for. That information obviously cannot be provided locally by the specification, since it involves further extensions that are not knowable in advance, many different variants of which can be instantiated, in the future.[19]

[19] Unless the specification ignores its `super` argument, the value may also depend on an inherited value that is not provided yet. However, using this `super` argument and providing an updated, extended value

Thus, consider the base specification for a parts-tracking protocol below. It provides a `parts` method, returning a list of parts, initialized to the empty list; also a `part-count` method that returns the length of the `parts` list; and it is otherwise pass-through for other methods. The `part-count` method crucially accesses the final value of the `parts` method through the module context `self`, and not the currently available initial empty value:

```
(define base-bill-of-parts
  (λ (self) (λ (super) (λ (method-id)
    (case method-id
      ((parts) '())
      ((part-count) (length (self 'parts)))
      (else (super method-id)))))))
```

You cannot inline the empty list in the call to `(self 'parts)` because the method `parts` can be extended, and indeed such is the very intent and entire point of this `base-bill-of-parts` specification! Even future extensions cannot inline the value they reach, unless they are guaranteed that no further extension will extend the list of parts (a declaration known as "sealing", after Dylan).

The interaction between modularity and extensibility therefore expands the scope of useful opportunities for modularity, compared to modularity without extensibility. ***Programming with modularity and extensibility is more modular than with modularity alone***. This makes sense when you realize that when the software is divided into small modular extensions many of which conspire to define each bigger target, there are more modular entities than when there is only one modular entity for each of these bigger targets. Modular extensibility enables modularity at a finer grain.

There is another important shift between modularity alone and modularity with extensibility, that I quietly smuggled in so far, because it happened naturally when modeling first-class entities using FP. Yet this shift deserves to be explicitly noted, especially since it is not quite natural in other settings such as existed historically during the discovery of OO or still exist today for most programmers: Modularity alone was content with a single global module context that everyone linked into, but ***the whole point of extensibility is that you will have many entities that will be extended in multiple different ways***; therefore when you combine the two, it becomes essential that module contexts be not a single global entity, but many local entities. This shift from singular and global to plural and local is essential for Class OO, and even more so for Prototype OO.

---

from it, is something the specification is explicitly allowed and expected to do. But the specification cannot expect that extended value to be the *final* value that the "effective method" will return. Thus, the problem is not with the inherited `super` argument, but with the fact that the value returned by the current method is itself the `super` value inherited by further methods.

Chapter 6

Rebuilding OO from its Minimal Core

> Well begun is half done.
>
> Aristotle

Now that I have reconstructed a minimal OO system from first principles, I can layer all the usual features from OO languages on top of that core. In this chapter, I will rebuild the most common features from popular OO languages: those so omnipresent that most developers think they are necessary for OO, even though they are just affordances easily added on top of the above core. More advanced and less popular features will follow in subsequent sections.

6.1 Rebuilding Prototype OO

6.1.1 What did I just do?

In the previous chapter, I reconstructed a minimal yet recognizable model of OO from first principles, the principles being modularity, extensibility, and first-class entities. I will shortly extend this model to support more features (at the cost of more lines of code). Yet my "object system" so far has no classes, and indeed no objects at all: instead, like the object system of Yale T Scheme (Adams and Rees 1988), on top of which its windowing system was built, my system is made of target records and their specifications, that can do almost everything that a Prototype object system does, but without either records or specifications being objects or prototypes (see section 3.3.2).

I defined my system in two lines of code, that can be similarly defined in any language that has higher-order functions, even a pure functional language without mutation; and indeed Nix defines its "extension" system similarly (Simons 2015). But there is indeed one extra thing Nix does that my model so far doesn't, wherein Nix has prototype objects and I don't: conflation.

6.1.2 Conflation: Crouching Typecast, Hidden Product

Prototype object systems have a notion of "object", also known as "prototype", that can be used both for computing methods, as with my model's record *targets*, and for composing with other objects through some form of inheritance, as with my model's *specifications*. How can these two very different notions be unified in a single entity? Very simply: by bundling the two together as a pair.

The type for a prototype, `Proto = Spec` $\times$ `Target`, is thus notionally the product of the type `Spec` for a specification and the type `Target` for its target. Giving more precise types for `Spec` and `Target` may involve types for fixpoints, subtyping, existential types, etc. See section 6.3.

However, the notions of specification and target and this notional product all remain unmentioned in the documentation of any OO system that I am aware of. Instead, the product remains implicit, hidden, and the prototype is implicitly typecast to either of its factors depending on context: when calling a method on a prototype, the target is used; when composing prototypes using inheritance, their respective specifications are used, and then the resulting composed specification is wrapped into a pair consisting of the specification and its target.

My implementation below makes this product explicit, where I use the prefix `pproto` to denote a prototype implemented as a pair. I use the `cons` function of Scheme to create a pair, and the functions `car` and `cdr` to extract its respective first and second components; in a more practical implementation, a special kind of tagged pair would be used, so the runtime would know to implicitly dereference the target in the common case, without developers having to painfully maintain the knowledge and explicitly tell the program when to dereference it (most of the time). The function `pproto←spec` is used to define a prototype from a specification, and is used implicitly when composing prototypes using inheritance with the `pproto-mix` function. The function `spec←pproto` extracts the specification from a prototype, so you may inherit from it. The function `target←pproto` extracts the target from a prototype, so you may call methods on it:

```
(define pproto←spec (λ (spec)
  (cons spec (fix-record spec))))
(define spec←pproto (λ (pproto)
  (car pproto)))
(define target←pproto (λ (pproto)
  (cdr pproto)))
(define pproto-mix (λ (child parent)
  (pproto←spec (mix (spec←pproto child) (spec←pproto parent)))))
```

Now, there is a subtle issue with the above implementation: when a target recursively refers to "itself" as per its specification, it sees the target only, and not the conflation of the target and the specification. This is not a problem with second-class OO, or otherwise with a statically staged style of programming where all the specifications are closed before any target is instantiated. But with a more dynamic style of programming where no such clear staging is guaranteed, it is insufficient.[1]

[1] Metaobjects are a typical use case where you don't (in general) have a clean program-wide staging of specification and targets: to determine the meta methods, you partially instantiate the meta part of the objects,

6.1.3 Recursive Conflation

In a dynamic first-class OO language, the conflation of specification and target into a single entity, the prototype, must be recursively seen by the target when instantiating the specification. This is achieved by having the instantiation function compose a "magic" wrapper specification in front of the user-given specification before it takes a fixpoint. Said magic wrapper will wrap any recursive reference to the target into an implicit conflation pair of the specification and the target.[2] Here is an implementation of that idea, wherein I prefix function names with `qproto`:

```
(define qproto-wrapper (λ (spec) (λ (self) (λ (super)
  (cons spec super)))))
(define qproto←spec (λ (spec)
  (fix-record (mix qproto-wrapper spec))))
```

Note how the following functions are essentially unchanged compared to `pproto`:

```
(define spec←qproto (λ (qproto)
  (car qproto)))
(define target←qproto (λ (qproto)
  (cdr qproto)))
(define qproto-mix (λ (child parent)
  (qproto←spec (mix (spec←qproto child) (spec←qproto parent)))))
```

What changed from the previous `pproto` variant was that the `(λ (x) (cons spec x))` extension was moved from outside the fixpoint to inside: If `R` is the parametric type of the reference wrapper (e.g. `R Integer` is the type of a reference to an integer), and `M` is the parametric type of modular extension, also known as a *recursion scheme*, then the type of `pproto` is `R (Y M)`, and that of `qproto` is `Y (R ∘ M)`, so in both cases I have a reference to a recursive data structure that follows the recursion scheme, but in the second case further recursive accesses also use the reference. Note that `Y (R ∘ M) = R (Y (M ∘ R))` and `Y (M ∘ R)` is the type of a raw record that follows the recursion scheme and uses references for recursion, instead of the type of reference to such, i.e. I have in turn `Y (M ∘ R) = M (Y (R ∘ M))`; people interested in low-level memory access might want to privilege this latter `Y (M ∘ R)` representation instead of `Y (R ∘ M)`, which indeed is a notable difference between the OO models of C++ vs Java: C++ makes you deal with data structures, Java with references to data structures.

Now, it is not unusual in computer science for access to records, recursive or not, to be wrapped inside some kind of reference type: pointer into memory, index into a table, key into a database, cryptographic hash into a content-addressed store, location into a file, string or identifier used in a hash-table or tree, etc. In the case of recursive data structures implemented as data in contiguous regions of memory, such level of indirection is inevitable, as there is no way to have a contiguous region of memory of

based on which you can instantiate the rest.

[2] Abadi and Cardelli (1996b) struggle with variants of this problem, and fail to find a solution, because they *want* to keep confusing target and specification even though at some level they can clearly see they are different things. If they had conceptualized the two as being entities that need to be distinguished semantically then explicitly grouped together as a pair, they could have solved the problem and stayed on top of the λ-calculus. Instead, they abandon such attempts, and rebuild their own syntactic theory of a variant of the λ-calculus just for object, an insanely complex *abstraction inversion* (Baker 1992) that won't enlighten OO practitioners, nor make OO interesting to mathematical syntax theoreticians.

some size contain as a strict subset a contiguous region of memory of the same size, as would be required for a data structure to directly include an recursive element of the same type.[3]

This use of a reference wrapper can be seen as an instance of the so-called "Fundamental theorem of software engineering" (Wikipedia 2025a): *We can solve any problem by introducing an extra level of indirection*. But more meaningfully, it can also be seen as the embodiment of the fact that, computationally, ***recursion is not free***. While at some abstract level of pure logic or mathematics, the inner object is of the same type as the outer one, and accessing it is free or constant time, at a more concrete level, recursion involves fetching data from another memory region. In the best case of a simple sequence of data, the sequence can be a contiguous array of memory and this fetching is constant time; in general, this fetching goes through the memory caching hierarchy, the latency of which grows as the square root of the size of the working set (Ernerfeldt 2014).

Importantly, isomorphic as it might be at some abstract level, the reference type is not equal to the type being referenced, and is not a subtype of it. Thus, the necessary reference wrapper extension is crucially not a strict extension with respect to the type being wrapped. This proves that it is a bad idea to require all extensions to always be strict (which would beg the question of which type to be strict for, or lead to the trivial answer that of being strict for the top type, for which all extensions are trivially strict). The reference wrapper, pure isomorphism at one level, yet effectful non-isomorphism at another (requiring access to disk, database, network, credentials, user interface, etc.), also illustrates that one man's purity is another man's side effect (to channel Alan Perlis). For instance, with merkleization, a reference uniquely identifies some pure data structure with a cryptographically secure hash that you can compute in a pure functional way; but dereferencing the hash is only possible if you already know the data based on which to compute and verify the hash, that you indexed into a database that you need some side-effect to consult. Many OO languages have an implicit builtin reference wrapper, as opposed to, e.g., explicit pointers as in C++; but the reference semantics doesn't disappear for having been made implicit.

6.1.4 Conflation for Records

If the target type can be anything, including an atomic value such as small integers, then there's nowhere in it to store the specification, and the conflation of specification and target must necessarily involve such a wrapping as I showed earlier. But if the target type is guaranteed to be a (subtype of) Record, it is possible to do better.

In the Nix extension system, a target is a record (called an attrset in Nix), mapping string keys to arbitrary values, and the modular extension instantiation function `fix'` stores the specification under a "magic" string `"__unfix__"`. The advantage is that casting a prototype (called "extension" in Nix) to its target is a trivial zero-cost identity no-op; the slight disadvantage is that the target must be a record, but that record cannot

[3]Exception: If the only element of a structure is an element of the same structure. At which point it's just an infinite loop to the same element, the type is isomorphic to the unit type, and can be represented as a trivial data structure of width zero. But if there is any other data element that isn't a unit type, direct recursion would mean infinite copies of it, one for each recursive path, which can't fit in finite memory.

use arbitrary keys, and must avoid the magic string as key. As a minor consequence, casting to a specification becomes slightly more expensive (table lookup vs fixed-offset field access), whereas casting to a target (the more common operation by far) becomes slightly cheaper (free vs fixed-offset field access). The semantics are otherwise essentially the same as for my implementation using pairs.

Here is a Scheme implementation of the same idea, where the prefix `rproto` denotes a prototype implemented as a record, and my magic key is `#f`, the boolean false value, instead of some reserved symbol, so it doesn't impede the free use of arbitrary symbols as keys. The function `rproto←spec` is used to define a prototype from a specification, by prepending a special specification `rproto-wrapper` in front that deals with remembering the provided specification; this function is used implicitly when composing prototypes using inheritance with the `rproto-mix` function. The function `spec←rproto` extracts the specification from a prototype, so you may inherit from it; this specifically doesn't include the `rproto-wrapper`, which would notably interfere with mixing. The function `target←rproto` extracts the target from a prototype, so you may call methods on it—it is the identity function, and you can often inline it away.

```
(define rproto-wrapper (λ (spec) (λ (self) (λ (super) (λ (method-id)
  (if method-id (super method-id) spec))))))
(define rproto←spec (λ (spec)
  (fix-record (mix (rproto-wrapper spec) spec))))
(define spec←rproto (λ (rproto)
  (rproto #f)))
(define target←rproto (λ (rproto)
  rproto))
(define rproto-mix (λ (child parent)
  (rproto←spec (mix (spec←rproto child) (spec←rproto parent)))))
```

Once again, some special extension is used in front, that is not strict, and is almost-but-not-quite an isomorphism, and specially memorizes the specification.

6.1.5 Small-Scale Advantages of Conflation: Performance, Sharing

First, note how, if a specification is pure functional, i.e. without side-effects, whether state, non-determinism, I/O, or otherwise, then indeed there is only one target, uniquely specified up to behavioral equality: recomputing the target multiple times will lead to the same result in all contexts. It thus makes sense to consider "the" target for a specification, and to see it as but another aspect of it. Caching the target value next to the specification can then be seen simply as a performance enhancement. Indeed, in case of recursive access to the target, this performance enhancement can grow exponentially with the depth of the recursion, by using a shared computation instead of repeated recomputations (see the related discussion on the applicative Y combinator in section 5.2.6).

If on the other hand, the specification has side-effects (which of course supposes the language has side-effects to begin with), then multiple computations of the target value

will lead to different results, and caching a canonical target value next to the specification is not just a performance enhancement, but a critical semantic feature enabling the sharing of the state and side-effects of a prototype between all its users. Meanwhile, if some users explicitly want to recompute the target, so as to get a fresh state to be modified by its own set of side-effects, they can always clone the prototype, i.e. create a new prototype that uses the same specification. Equivalently, they can create a prototype that inherits from it using as extension the neutral element `(rproto←spec idModExt)`.

Now, plenty of earlier or contemporary Prototype OO languages, from Director and ThingLab to Self and JavaScript and beyond, support mutation yet also offer objects as conflation of two aspects: one for inheritable and instantiatable specification, another one for the instantiated target that holds mutable state and that methods are called against. However, the two aspects might not be as neatly separated in these stateful languages as in pure functional languages, because the mutation of the internals of the specification, in languages that allow it, may interact with the target state and behavior in weird ways. This mutation is not usually idiomatic in production code, but may be heavily relied upon during interactive development, or as part of implementing advanced infrastructure.

Last but not least, if your choice of representation for specifications and targets is such that instantiating a specification may itself issue side-effects such as errors or non-termination or irreversible I/O—then it becomes essential to wrap your target behind lazy evaluation, or, in Scheme, a `delay` form. Thus you may still define prototypes from incomplete erroneous specifications, and use them through inheritance to build larger prototypes, that, when complete, will not have undesired side-effects. Once again, ***Laziness proves essential to OO, even and especially in presence of side-effects***.

6.1.6 Large-Scale Advantage of Conflation: More Modularity

Remarkably, conflation is more modular than the lack thereof: thanks to it, programmers do not have to decide in advance when each part of their configuration is to be extended or instantiated. Indeed, if only using unconflated specifications and targets, one would have to choose, for each definition of each identifier in each (sub)module, whether to use a specification or its target. And when choosing to maintain a specification for the purpose of extensibility, the programmer may still want to explicitly bind an identifier to the target, and another one to the specification, for the sake of state sharing, and sanity, and not having to remember the hard way the "shape" of the nested extensions to replace by the values when instantiating the target. Thus, users would end up doing by hand in an ad hoc way what conflation gives them systematically for free.

Furthermore, users cannot know which of their definitions they themselves, or other users, might later want to extend. A simple service will often be made part of a larger service set, in multiple copies each slightly tweaked; its configuration, complete and directly instantiable as it may be for its original author, will actually be extended, copied, overridden, many times, in a larger configuration, by the maintainers of the larger service set.

The modularity of conflation is already exploited at scale for large software distributions on one machine or many, using GCL, Jsonnet or Nix as (pure functional) Prototype OO languages:

- The Google Control Language GCL (Bokharouss 2008) (née BCL, Borg Control Language), 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).
- Jsonnet (Cunningham 2014), inspired by GCL but cleaned up to use static scoping, has been particularly used to generate configurations for AWS or Kubernetes.
- Nix (Dolstra and Löh 2008) is used not just to configure entire software distributions for Linux or macOS, but also distributed services with NixOps or DisNix.

All three languages have proven the practicality of pure lazy functional prototype objects, with mixin inheritance and conflation of specification and target, as a paradigm to specify configuration and deployment of software on a world-wide scale, each with hundreds of active developers, tens of thousands of active users, and millions of end-users. Despite its minimal semantics and relatively obscure existence, this mixin prototype object model has vastly outsized outreach. It is hard to measure how much of this success is due to the feature of Conflation, yet this feature is arguably essential to the ergonomics of these languages.

6.1.7 Implicit Recognition of Conflation by OO Practitioners

The notion of a *conflation of specification and target*, that I presented, is largely unknown by OO developers, and seems never, ever, to have been made explicit in the literature until I published it (Rideau et al. 2021). And yet, the knowledge of this conflation is necessarily present, if implicit, if not across the OO community, at the very least among OO implementers—or else OO wouldn't be possible at all. Sixty years of crucial information you don't know you know!

Common practitioners of OO have long implicitly recognized the conflated concepts of specification and target. Back in 1979, Flavors (Cannon 1979) introduces the concept of a *mixin* as a flavor meant to be inherited from, but not to be instantiated, by opposition to an instantiatable flavor; however, the nomenclature only stuck in the Lisp community (and even there, flavors yielded to classes though the term mixin stayed). In other communities, the terms of art are *abstract classes* and *concrete classes* (Johnson and Foote 1988): an abstract class is one that is only used for its specification—to inherit from it; a concrete class is one that is only used for its target type—to use its methods to create and process class instances. Experienced practitioners recommend keeping the two kinds of classes separate, and frown at inheriting from a concrete class, or trying to instantiate an abstract class.

Theorists have also long implicitly recognized the conflated concepts when working to develop sound type systems for OO: for instance, Fisher (Fisher 1996) distinguishes `pro` types for objects-as-prototypes and `obj` types for objects-as-records; and

Bruce (Bruce 1996) complains that "the notions of type and class are often confounded in object-oriented programming languages" and there again distinguishes subtyping for a class's target type (which he calls "subtyping") and for its open specification (which he calls "matching"). Yet though they offer correct models for typing OO, both authors fail to distinguish specification and target as syntactically and semantically separate entities in their languages, leading to much extraneous complexity in their respective type systems.

Implementers of stateful object systems at runtime may not have realized the conflation of entities, because they are too focused on low-level mechanisms for "delegation" or "inheritance". By contrast, writers of compilers for languages with second-class Class OO may not have realized the conflation because at their level it's all pure functional specification with no target until runtime. One group of people though, must explicitly deal with the conflation of specification and target embodied as a first-class value: implementers of pure functional prototype systems. Nix (Simons 2015) explicitly remembers the specification by inserting the `__unfix__` attribute into its target records, and Jsonnet (Cunningham 2014) must do something similar under the hood; yet the authors of neither system make note of it in their documentation as a phenomenon worthy of remark. Though they implicitly rediscovered the concept and made it flesh, they failed to realize how momentous the discovery was, and shrugged it off as yet another one of those annoying implementation details they had to face along the way.

Finally, the confusion between target and specification can be seen as a special case of the confusion between object and implementation discussed in (Chiba et al. 2000), wherein you can see the specification as *implementing* the target. But though these authors saw a more general case in a wider theory with far reaching potential, they do not seem to have noticed this common case application.

Thus, through all the confusion of class OO languages so far, both practitioners and theorists have felt the need to effectively distinguish specification and target, yet no one seems to have been able to fully tease apart the concepts up until recently.

6.2 Rebuilding Class OO

6.2.1 A Class is a Prototype for a Type

Having elucidated Prototype OO in the previous sections, including its notion of Object, a.k.a. Prototype, as conflation of Specification and Target, I can now fully elucidate Class OO including its notion of Class: ***A Class is a Prototype for a Type***. Or, to be pedantic, a class is a prototype, the target of which is a *type descriptor*, i.e. a record describing a type together with methods associated with the type.

The target type of a class is usually, but not always, a record type (indexed product, structure, struct, named tuple), at which point the descriptor will also describe its fields; it may also be an enumeration type (indexed sum, enum, variant, discriminated union, tagged union), at which point the descriptor will also describe its alternatives; and while this is less common in Class OO, a class's target could really describe any kind of type: function, number, array, associative array, etc.

The target of a class may be a runtime type descriptor, that describes how to create, recognize, and process elements of the type in the same evaluation environment that the type exists in; or, as is often the case in second-class Class OO, the target may be a compile-time type descriptor, that describes how to generate code for entities of that type to be executed in a further stage of evaluation; or it may be both.

Whichever kind of type descriptors are used, Class OO is but a special case of Prototype OO, wherein a class is a prototype for a type, i.e., the conflation of a modular extension for a type descriptor, and the type descriptor that is the fixpoint of that specification. Thus, when I claimed in section 2.2 that the situation of classes in OO was similar to that of types in FP, I meant it quite literally.

6.2.2 Simple First-Class Type Descriptors

Since I do not wish to introduce a Theory of Compilation in this book in addition to a Theory of OO, I will only illustrate how to build *first-class* Class OO, at runtime, on top of Prototype OO. I will use a technique described by Lieberman (Lieberman 1986), and no doubt broadly similar to how many Class OO systems were implemented on top of Javascript before web browsers ubiquitously adopted ES6 classes (International 2015).

A type descriptor (which in C++ would correspond to a `vtable`) will typically have methods as follows:

- A method `instance-methods` returning a record of instance methods (or, as a runtime optimization that requires more compile-time bookkeeping, encode those object methods directly as methods of the type descriptor).
- For record types, a method `instance-fields` returning a record of field descriptors, each of them a record with fields `name` and `type` (and possibly more).
- Additionally, self-describing records (the default) will have a special field `#t` (the Scheme true boolean) to hold their type descriptor (I could have used the string `"__type__"` if keys had to be strings, using a common convention of surrounding a system-reserved identifier with underscores; but in Scheme I can use a different kind of entity and leave strings entirely for users; my choice of `#t` also rhymes with my previous choice of `#f` (the Scheme false boolean) to hold the specification; plus `#t` has the same letter as the initial of "type").
- In dynamic languages, or static languages that accept subsets of canonical inferred static types as types, a method `element?` returning a predicate that is true if its argument is an element of the type. If the language supports error reporting, a method `validate` returning a unary function that returns its argument unchanged if it is an element and otherwise raises a suitable error. Either method can be derived from the other using sensible defaults. First-class class instances are definitely subsets of whichever underlying data type is used for their representation, and so their type descriptors will usefully have such a method.

- Usually, some kind of method `make-instance` returning a constructor to create a new object instance with some extensible protocol from user-specified arguments. To support indexed sum types, either the `make-instance` method will take a discriminant as first argument, or a method `instance-constructors` could hold a record of several "constructor" functions for each case. Instead of constructors, or in addition to them, a basic `instance-prototype` (or `instance-prototypes`) method could be provided that implements the skeleton of a class instance.

- For languages that support user management of dynamic object lifetime, a `destroy-instance` method could be part of the class, or among the `instance-methods`; or if there can be several kinds of destructors, they could be held by a method `instance-destructors`.

A "class instance" (corresponding to an "object" in Class OO) is a self-describing record, the type descriptor of which can be extracted with the function `type-of` below; to call an "instance method" on it, you use the function "instance-call" with it as first argument, and it calls the method from the instance's type's `instance-methods` record, with the instance as first argument.

```
(define type-of (λ (instance)
  (instance #t)))
(define instance-call (λ (instance) (λ (method-id)
  ((((type-of instance) 'instance-methods) method-id) instance))))
```

Note that, if I use the Nix approach of zero-cost casting to target when the target is a record, then I can use the very same representation for type descriptors, whether they were generated as the fixpoint target of a specification, or directly created as records without such a fixpoint. This kind of representation is notably useful for bootstrapping a Meta-Object Protocol (Kiczales et al. 1991).

As for "class methods" (also known as "static methods" in C++ or Java), they can be regular methods of the type descriptor, or there can be a method `class-methods` in the type descriptor containing a record of them.

6.2.3 Parametric First-Class Type Descriptors

There are two main strategies to represent parametric types: "function of descriptors" vs "descriptor of functions".

In the "function of descriptors" strategy, a parametric type is represented as a function from type descriptor to type descriptor: the function takes a type parameter as input, and returns a type descriptor specialized for that parameters as output. As it computes the specialized descriptor, it can apply various specializations and optimizations, pre-selecting code paths and throwing away cases that do not apply, allowing for slightly better specialized code, at the expense of time and space generating the specialized descriptor. This representation allows for uniform calling conventions for all type descriptors satisfying the resulting monomorphic interface, regardless of whether it was obtained by specializing a parametric type or not. This strategy is notably used during the "monomorphization" phase used within C++ compilers when expanding

templates. It is useful when trying to statically inline away all traces of type descriptors before runtime, but in a dynamic setting requires creation of more descriptors, or some memoization mechanism.

In the "descriptor of functions" strategy, a parametric type is represented as a type descriptor the methods of which may take extra parameters in front, one for the type descriptor of each type parameter. Methods that return non-function values may become functions of one or more type parameters. Thus, the type descriptor for a functor `F` may have a method `map` that takes two type parameters `A` and `B` and transforms an element of `P A` into an element of `P B`. This strategy eliminates the need to heap-allocate a lot of specialized type descriptors; but it requires more bookkeeping, to remember which method of which type descriptor takes how many extra type descriptor parameters.

Both strategies are useful; which to prefer depends on the use case and its tradeoffs. A given program or compiler may use both, and may very well have to: even using the "descriptor-of-functions" strategy, you may still have to generate specialized type descriptors, so that you may pass them to functions that expect their parametric type descriptors to only take `N` type parameters, and are unaware that these were specialized from more general parametric type descriptors with `N + M` type parameters (where `M > 0`). This unawareness may stem from any kind of dynamic typing, or from a choice to avoid generating many copies of the code for each value of `M` as the depth of parametric constructors varies. And even using the "function-of-descriptors" strategy, you may want to maintain collections of functions that each take one or many descriptors as arguments, especially when such collections contain a large number of functions, only one or a few of which are to be used at a time per tuple of descriptor arguments, and this tuple of descriptor arguments itself changes often.

Even fixpoints can be done differently in the two strategies: the "function-of-descriptors" strategy leads to generating descriptors that embed the fixpoints so that clients never need to know they were even fixpoints; whereas the "descriptor-of-functions" strategy leads to descriptors the calling convention of which requires clients to systematically pass the descriptor itself to the methods it contains so as to close the fixpoint loop.

6.2.4 Class-style vs Typeclass-style

Now, there are two common styles for using type descriptors: Class-style and Typeclass-style.

In Class-style, each type element (known in this style as "class instance", "instance", or "object") carries its own type descriptor. To this end, the type descriptor is conflated with the type element in the same way that specifications were conflated with their targets (see section 6.1.3, section 6.1.4). As mentioned before, this can be very cheap when the type elements are records (see section 6.1.4, section 6.2.2): just add a special field with a "magic" key.

In Typeclass-style, by contrast, type descriptors and type elements are kept distinct and separate. There is emphatically no conflation. Type-descriptors are passed "out of band" as extra variables (see second-class "dictionaries" in Haskell (Wadler and Blott 1989), or first-class "interfaces" in "Interface-Passing Style" (Rideau 2012)). This is efficient in another way, because you can pass around a few type descriptors that do

not change within a given algorithm, and not spend time wrapping and unwrapping conflations.

There are many tradeoffs that can make one style preferable to the other, or not:

- When the precise type of the element is known in advance, a runtime type descriptor need not be used in either style, and the compiler can directly generate code for the known type. When a function works on data the precise type for which is *not* known in advance, that's when a runtime type descriptor in either style may help.

- Class-style helps most when each piece of data is always going to be used with the same type, and/or when functions are more "dynamic" and do not always work with the same type of data. Typeclass-style helps most when each piece of data can be used as element of many types, and/or when functions are more "static" and always work with the same type of data. The two styles can be complementary, as the balance of static and dynamic information about functions and data can vary across libraries and within a program.

- Typeclass-style works better when the data is or might not be records, but has homogeneous types, and it is only wasteful to re-extract the same types from each piece of data. Typeclass-style works well for processing a lot of uniform data.

- Class-style works better when data is made of records with heterogeneous types, and functions make control-flow decisions based on the kind of records the users feed them. Class-style works well for processing weakly-structured documents.

- Typeclass-style works better if you can build your type descriptor once per function and use it on a lot of data. Class-style works better if you can build your type descriptor once per piece of data and use it on a lot of functions.

- Typeclass-style works better when the same pieces of data are being reinterpreted as elements of varying types, often enough or at a fine enough grain, that it does not make sense to re-conflate the entire data set with every change of point of view. Class-style works better if type reinterpretations are few and localized.

- There can be many different type descriptors that match any given object, with different methods to work with them from a different point of view, parameterized by administrator-configured or user-specified parameters that vary at runtime. For instance, a given number may be used, in different contexts, with type descriptor that will cause it to be interacted with as a decimal number text, a hexadecimal number text, a position on a slide bar, or a block of varying color intensity; or to be serialized according to some encoding or some other. In a static language like Haskell, `newtype` enables compile-time selection between multiple different points of view on what is, underneath, a "same" low-level data representation; in a language with Prototype OO, "typeclass"-style type descriptors enable the same kind of behavior, sometimes to implement at runtime

second-class typeclasses that are known constants at compile-time, but sometimes for first-class typeclasses that change at runtime based on user interaction, even while the object representation may stay the same.

- Constructors are quite special in "class-style", since regular methods are called by extracting them from an object's type descriptor, but there is not yet an object from which to extract a type descriptor when what you are doing is precisely constructing the first known object of that type (in the given scope at least). Constructors are so special, that some encodings of classes identify a class with "the" constructor function that generates an object of that type, complete with an appropriate type descriptor. This creates an asymmetry between constructors and other methods, that requires special treatment when transforming type descriptors. By contrast, in "typeclass-style", constructors are just regular methods; there can be more than one, using different calling conventions, or creating elements from different subsets or subtypes of the type (disjoint or not). Many typeclass transformations, dualities, etc., become more uniform and simpler when using typeclass-style. Type descriptors in typeclass style, in which constructors are just normal methods, are therefore more natural and easy to use for parametric polymorphism than type descriptors in class style.

- Finally, typeclass-style can be extended more easily and more uniformly than traditional class-style to support APIs involving multiple related types being simultaneously defined in mutually recursive ways: data structures and their indexes, paths or zippers, containers and containees, bipartite graphs, grammars with multiple non-terminals, expressions and types, etc. Instead of distinguishing a main class under which others are nested, or having a hierarchy of "friend" classes indirectly recursing through a global context, typeclass-style treats all classes uniformly, yet can locally encapsulate an entire family of them, potentially infinite. There is, however, a way to retrieve most of these advantages of typeclass-style while remaining in class-style, though only few languages support it: using multi-methods (see below).

There is an isomorphism between class-style and typeclass-style type descriptors, to the point that a simple transformation can wrap objects written in one style and their methods so they follow the calling convention of the other style.[4] Simple metaprograms that enact this transformation have been written in Lisp (Rideau 2012) in the simple case of functions in which the self-type only appears directly as one of the (potentially multiple) arguments or (potentially multiple) return values of a method. Such

[4]To go from Class-style to Typeclass-style, simply extract the type descriptor, class metaobject, vtable, etc., from an object, and make that your type descriptor. For "typeclasses" that are parameterized by many types, use a record that contains each of those descriptors for each of those types as fields.

To go from Typeclass-style to Class-style, you can wrap every value or tuple of values into a record of the values and the typeclass they are supposed to be used with. When invoking typeclass functions, unwrap the values from those records to use them as arguments; and wrap the results back into records that conflate the values and an appropriate typeclass. This is the same trick that we used with recursive conflation. You just need to know which arguments and results of which method to wrap or unwrap.

Note how the values associated with typeclasses can be "naked" primitive values that need not be records, just like the fields of class instances.

automation could be generalized to work with any kind of higher-order function types, where occurrences of the self-type in arbitrary position require suitable wrapping of arguments and return values, in ways similar to how higher-order contracts wrap arguments and return values with checks (Findler and Felleisen 2002). Note how similarly, metaprograms have been written to transform pure objects into mutable objects that store a current pure value, or mutable objects into linear pure objects that become invalid if a use is attempted after mutation.

Thus, programming with either (A1) classes or (A2) typeclasses, wherein objects are either (B1) mutable or (B2) pure, is a matter of style, with some tradeoffs with respect to performance, ease of reasoning, between the four combined styles. You could add more style variants, such as data representation as (C1) a graph of records, or (C2) tables of entities, (D1) dynamically typed, or (D2) statically typed, etc. In the end, programs in one style can be mechanically transformed into programs in another style, and vice versa. Some programs, given the kind of data they are expected to work on at runtime, may be better compiled onto one of those styles, if not directly written in it. But if programmers have to deal with conceptual issues that are more natural in a different style, they may be better off writing their programs in the style that works for them, and pay the price of automatic or manual translation later.

Interestingly, all this section's discussion was about styles for target programs. Type descriptors can be used in any and all of those styles without any OO whatsoever.[5] And OO can be used at runtime or compile-time to specify and generate type descriptors in any and all of those styles, as well as non-type-descriptor targets. OO is completely agnostic as to whether you write in any particular style or another. OO is about specifications for programs and parts of programs. Although it was historically developed in the context of the style (A1, B1, C1, D1) of dynamic graph of mutable classes (Smalltalk, Lisp, JavaScript), or the style (A1, B1, C1, D2) of static graph of mutable classes (Simula, C++, Java), OO can well accommodate any style. I wrote OO in both Typeclass-style (A2) and pure-style (B2) (Rideau 2012, 2020), and I have no doubt you could write OO to manipulate tables (C2) instead of record graphs (C1). There is no reason why you couldn't use OO to specify programs in the (A2, B2, C2, D2)-style of static tables of pure functional typeclasses: this combination would make for a nice hypothetical language to statically specify pure transformations on tables of uniform data, with applications to modeling neural networks, physics engines or 3d video pipelines.

[5] Indeed, Haskell typeclasses are multi-type descriptors with modularity but without inheritance, and so their Rust equivalent called "traits"— not to be confused with Scala "traits", that are single-type descriptors with multiple inheritance. Modules in SML or OCaml can also offer "typeclass-style" type descriptors without modular extensibility through inheritance.

Non-OO class-style type descriptors are also possible. For instance, Gambit Scheme allows users to define new data structures, and to declare the equivalent of methods that specialize how values of those new types will be printed, or tested for equality; these methods are not part of any actual object system capable of inheritance, yet each "structure" record carries the equivalent of a type descriptor field in "class style" so that the system knows which method to use. It is possible to layer an actual OO class system on top of such non-OO "class-style" mechanism, and indeed the Gerbil Scheme object system is built atop Gambit Scheme's non-OO structure facility.

6.2.5 A Class is Second-Class in Most Class OO

In most Class OO languages, the semantics of Class OO are fully evaluated at compile-time, in a "type-level" evaluation stage. The class specifications, and the type descriptors they specify, are second-class entities: they are not available as first-class values subject to arbitrary programming at runtime. Class OO then is a special case of Prototype OO, but only in a restricted second-class language—a reality that is quite obvious when doing template metaprogramming in C++.

Some dynamic languages, such as Lisp, Smalltalk, Ruby or Python, let you programmatically use, create, inspect or modify classes at runtime, using some reflection mechanisms. The regular definition and usage of classes involve dedicated syntax, such that restricting yourself to the language fragment with that syntax and never using reflection would be equivalent to classes being second-class; but the reflection mechanisms ultimately make classes into first-class entities. Indeed, the second-class semantics of classes are often implemented in terms of those first-class mechanisms, that are thus just as powerful, or more so.

Static languages lack such full-powered runtime reflection mechanisms (or they would arguably be dynamic languages indeed). Some lack any runtime reflection at all; but many offer read-only runtime reflection, whereby users can inspect classes created at compile-time, yet not create new ones: such capabilities can notably simplify I/O, property-based testing, debugging, etc. A few static language implementations may even offer limited ability to modify existing classes at runtime, but often void the compiler's warranty for those who use them.

6.2.6 Type-Level Language Restrictions

The language in which second-class classes are defined and composed in static languages is almost never the same as the "base" language in which the programmer specifies first-class computations (e.g. C++, Java, C#), but instead a distinct *type-level language*, deliberately restricted in expressiveness so as to enable static analysis and optimizations, in which the types and the base-level functions operating on them are being modularly and extensibly specified.

Restrictions on the type-level language often 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, especially as features get added over time (as in C++, Java, Haskell) (Grigore 2016).[6] The attempts do usually succeed, however, at making these type-level languages require a completely different mindset from the "base language", 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; this is in stark contrast with the base languages, that

[6] Even the C preprocessor, with annoying rules added to "guarantee" termination, ends up allowing arbitrary metaprogramming in practice.

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).

6.2.7 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 and in practice: most popular OO languages only offer Class OO; and even though the arguably most popular OO language, JavaScript, may have started with Prototype OO only (Eich 1996), people were constantly reimplementing classes on top—and twenty years later, classes were added to the language itself (International 2015).

And yet I will argue that Prototype OO is more fundamental than Class OO: as I demonstrated above, Class OO can be easily expressed in terms of Prototype OO and implemented on top of it, such that inheritance among classes is indeed a special case of inheritance among the underlying prototypes; however the opposite is not possible, since you cannot express Prototype OO's first-class entities and their inheritance in terms of Class OO's second-class entities and their inheritance.

At best, Prototype OO can be implemented on top of those dynamic languages that offer full-powered reflection so that prototypes can be classes; but even then it is unclear how much these mechanisms help, compared to directly implementing prototypes. There could be code sharing between the two; yet trying to fit prototypes on top of classes rather than the other way around is what Henry Baker dubbed an *abstraction inversion* (Baker 1992), i.e. putting the cart before the horse.

6.3 Types for OO

6.3.1 Dynamic Typing

One could just say that objects are of a monomorphic type `Record`, and that all accesses to methods are to be dynamically typed and checked at runtime, with no static safety. Many OO languages, like Smalltalk, Lisp, Ruby, Python, JavaScript, Jsonnet, Nix, etc., adopt this strategy.

Advantages of dynamic typing include the ability to express programs that even the best static typesystems cannot support, especially when state-of-the-art typesystems are too rigid, or not advanced enough in their OO idioms. Programs that involve dependent types, staged computation, metaprogramming, long-lived interactive systems, dynamic code and data schema evolution at runtime, etc., can be and have been written in dynamically typed systems that could not have been written in existing statically typed languages, or would have required reverting to unitypes with extra verbosity.

On the other hand, system-supported static types can bring extra performance and safety, help in refactoring, debugging programs, some forms of type-directed metaprogramming, and more. Even without system enforcement, thinking in terms of types can help understand what programs do or don't and how to write and use them. I have already started to go deeper by describing records as indexed products. Let's see how to model OO with more precise types.

6.3.2 Partial Record Knowledge as Subtyping

In a language with static types, programmers writing extensible modular definitions should be able to specify types for the entities they provide (an extension to) and require (a complete version of), without having to know anything about the types of the many other entities they neither provide nor require: indeed, these other entities may not have been written yet, and by other people, yet will be linked together with his modules into a complete program.

Now, with only modularity, or only extensibility, what's more second-class only, you could contrive a way for the typechecker to always exactly know all the types required, by prohibiting open recursion through the module context, and generating magic projections behind the scenes (and a magic merge during linking). But as I previously showed in section 5.3.8, once you combine modularity and extensibility, what's more first-class, then open recursion through the module context becomes the entire point, and your typesystem must confront it.

Strict extensibility, which consists in monotonically contributing partial knowledge about the computation being built, once translated in the world of types, is subtyping. In the common case of records, a record type contains not just records that exactly bind given identifiers to given types, but also records with additional bound identifiers, and existing identifiers bound to subtypes. Record types form a subtyping hierarchy; subtyping is a partial order among types; and function types monotonically increase with their result types, and decrease with their argument types. Then, when modularly specifying a module extension, the modular type for the module context, that only contains "negative" constraints about types for the identifiers being required, will match the future actual module constraint, that may satisfy many more constraints; meanwhile, the "positive" constraints about types for the identifiers being provided may satisfy many more constraints than those actually required from other modules using it.

6.3.3 The NNOOTT: Naive Non-recursive OO Type Theory

The simplest and most obvious theory for typing OO, that I will dub the Naive Non-recursive Object-Oriented Type Theory (NNOOTT), consists in considering subclassing (a relation between specifications) as the same as subtyping (a relation between targets). Thus, in this theory, a subclass, that extends a class with new fields, is (supposedly) a subtype of the parent "superclass" being extended.

To demonstrate an advanced variant of the NNOOTT, consider extending the Simply Typed Lambda-Calculus (STLC), or some more elaborate but still well-understood variant of the λ-calculus, with primitives for the language's builtin constructs and standard libraries, and (if not already available) a minimal set of features for OO: indexed products for records, subtyping ($\subset$ or $\leqslant$ or in ASCII `<:`) and type intersections ($\cap$). In this NNOOTT variant, a NNOOTT Modular Extension could have a type of the form

```
type NModExt required inherited provided =
  required → inherited → (inherited ∩ provided)
```

A `NModExt` is a type with three parameters, the type `required` of the information required by the modular extension from the module context, the type `inherited` of the information inherited and to be extended, and the type `provided` of the information

provided to extend what is inherited. Note that this type refines the `C → V → V` from section 5.3.5: `inherited` and `provided` each separately refine the value `V` being specified; that value can be anything: it need not be a record at all, and if it is, it can have any shape or type, and need not have the same as the module context. Meanwhile, `required` refines the module context `C`, and is (almost) always some kind of record. The two need not be the same at all, and usually are not for (open) modular extensions, unless and until you're ready to close the recursion, tie the loops and compute a fixpoint.

In Prototype OO, the value inherited holds the methods defined so far by the ancestors; the value provided consists in new methods and specialization of existing methods; the top value is an empty record. In Class OO, that value inherited holds methods and fields defined so far by the ancestors; the value provided consists in the new or specialized fields and methods; the top value is a type descriptor for an empty record type. In both cases, the context required may narrowly define a prototype or class, but may also more broadly define an entire namespace.

The `fix` operator, first takes a top value as a seed, then second takes a specification for a target with the target itself as module context and starting with the top value as a seed, and returns the target fixpoint. The `mix` operator chains two mixins, with the asymmetry that information provided by the parent (parameter `p2` for the second argument) can be used by the child (first argument), but not the other way around.

```
fix : top → NModExt target top target → target
mix : NModExt r1 i1∩p2 p1 → NModExt r2 i2 p2 →
        NModExt r1∩r2 i1∩i2 p1∩p2
```

This model is simple and intuitive, and has good didactic value to explain how inheritance works: given two "mixin" specifications, you can chain them as child and parent; the combined specification requires a context with all the information required from either child or parent; the inherited information must contain all information expected by the parent, and all information expected by the child that isn't provided by the parent; the provided information contains all information provided by either child or parent.

However, this "Naive Non-recursive OO Type Theory", as the name indicates, is a bit naive indeed, and only works in simple non-recursive cases. Yet the NNOOTT is important to understand, both for the simple cases it is good enough to cover, and for its failure modes that tripped so many good programmers into wrongfully trying to equate inheritance and subtyping.

6.3.4 Limits of the NNOOTT

The NNOOTT works well in the non-recursive case, i.e. when the types of fields do not depend on the type of the module context; or, more precisely, when there are no circular "open" references between types being provided by a modular extension, and types it requires from the module context. In his paper on objects as co-algebras, Bart Jacobs characterizes the types for the arguments and results of his methods as being "(constant) sets" (Jacobs 1995),[7] which he elaborates in another paper (Jacobs 1996)

[7] Jacobs is particularly egregious in smuggling this all-important restriction to how his paper fails to address the general and interesting case of OO in a single word, furthermore, in parentheses, at the end of

as meaning «not depending on the "unknown" type X (of self).» This makes his paper inapplicable to most OO, but interestingly, precisely identifies the subset of OO for which inheritance coincides with subtyping, or, to speak more precisely, subtyping of modular extensions coincides with subtyping of their targets.

Indeed, in general, specifications may contain so called "binary methods" that take another value of the same target type as argument, such as in very common comparison functions (e.g. equality or order) or algebraic operations (e.g. addition, multiplication, composition), etc.; and beyond these, they can actually contain arbitrary higher-order functions involving the target type in zero, one or many positions, both "negative" (as an overall argument) or "positive" (as an overall result), or as parameters to type-level functions, "templates", etc. These methods will break the precondition for subclassing being subtyping.

And such methods are not an "advanced" or "anomalous" case, but quintessential. The very first example in the very first paper about actual classes (Dahl and Nygaard

section 2, without any discussion whatsoever as to the momentous significance of that word. A discussion of that significance could in itself have turned this bad paper into a stellar one. Instead, the smuggling of an all-important hypothesis makes the paper misleading at best. His subsequent paper (Jacobs 1996) has the slightly more precise sentence I also quote, and its section 2.1 tries to paper over what it calls "anomalies of inheritance" (actually, the general case), by separating methods into a "core" part where fields are declared, that matter for typing inheritance, and for which his hypothesis applies, and "definitions" that must be reduced to the core part. The conference reviewing committees really dropped the ball on accepting those papers, though that section 2.1 was probably the result of at least one reviewer doing his job right. Did reviewers overall let themselves be impressed by formalism beyond their ability to judge, or were they complicit in the sleight of hand to grant their domain of research a fake mantle of formal mathematical legitimacy? Either way, the field is ripe with bad science, not to mention the outright snake oil of the OO industry in its heyday: The 1990s were a time when IBM would hire comedians to become "evangelists" for their Visual Age Smalltalk technology, soon recycled into Java evangelists. Jacobs is not the only one, and he may even have extenuating circumstances. He may have been ill-inspired by Goguen, whom he cites, who also abuses the terminology from OO to make his own valid but loosely-related application of Category Theory to software specification. He may also have been pressured to make his work "relevant" by publishing in OO conferences, under pains of losing funding, and he may have been happy to find his work welcome even though he didn't try hard, trusting reviewers to send stronger feedback if his work hadn't been fit. The reviewers, unfamiliar with the formalism, may have missed or underestimated the critical consequences of a single word; they may have hoped that further work would lift the limitation. In other times, researchers have been hard pressed to join the bandwagon of Java, Web2, Big Data, Mobile, Blockchain or AI, or whatever trendy topic of the year; and reviewers for the respective relevant conferences may have welcomed newcomers with unfamiliar points of view. Even Barbara Liskov, future Turing Award recipient, was invited to contribute to OO conferences, and quickly dismissed inheritance to focus on her own expertise, which involves modularity without extensibility—and stating her famous "Liskov Substitution Principle" as she did (Liskov 1987); brilliant, though not OO. Are either those who talk and publish what turns out not to be OO at all at OO conferences, or those who invite them to talk and publish, being deliberately misleading? Probably not, yet, the public can be fooled just the same as if dishonesty were meant: though the expert of the day can probably make the difference, the next generation attending or looking through the archives may well get confused as to what OO is or isn't about as they learn from example. At the very least, papers like that make for untrustworthy identification and labeling of domains of knowledge and the concepts that matter. The larger point here being that one should be skeptical of papers, even by some of the greatest scientists (none of Jacobs', Goguen's nor Liskov's expertises are in doubt), even published at some of the most reputable conferences in the field (e.g. OOPSLA, ECOOP), because science is casually corrupted by power and money, and only more cheaply so for the stakes being low. This particular case from thirty years ago is easily corrected in retrospect; its underlying lie was of little consequence then and is of no consequence today; but the system that produced dishonest science hasn't been reformed, and I can but imagine what kind of lies it produces to this day in topics that compared to the semantics of OO are both less objectively arguable, and higher-stake economically and politically.

1967), involves recursive data types: it is a class `linkage` that defines references `suc` and `pred` to the "same" type, that classes can inherit from so that their elements shall be part of a doubly linked list. This example, and any data structure defined using recursion, will defeat the NNOOTT if examined closely. Not only is such recursion a most frequent occurrence, I showed above in section 5.3.8 that while you can eschew support for fixpoints through the module context when considering modularity or extensibility separately, open recursion through module contexts becomes essential when considering them together. In the general and common case in which a class or prototype specification includes self-reference, subtyping and subclassing are very different, a crucial distinction that was first elucidated in (Cook et al. 1989).

Now, the NNOOTT can be "saved" by reserving static typing to non-self-referential methods, whereas any self-reference must dynamically typed: wherever a recursive self-reference to the whole would happen, e.g. in the type of a field, programmers must instead declare the value as being of a dynamic "Any" type, or some other "base" type or class, so that there is no self-reference in the type, and the static typechecker is happy. Thus, when defining a list of elements of type `A`, you could not write the usual recursive formula `List(A) = 1 + A*List(A)` or the fixpoint `List(A) = Y (λ Self . 1 + A*Self)`, and would just write `List(A) = 1 + A*Any`. Similarly, for trees with leaves of type `B`, you couldn't write the recursive formula `Tree(B) = B + List(Tree(B))`, and would instead write just the non-recursive and dynamically typed `Tree(B) = B + List(Any))`.

To compensate for the imprecision of the type system when retrieving an element of the desired self-type, some kind of explicit dereference, typecast (downcast), or coercion is required from the programmer; that operation may be either safe (with a dynamic runtime check), or unsafe (program may silently misbehave at runtime if called with the wrong argument). In some languages, self-reference already has to go through pointer indirection (e.g. in C++), or boxing (e.g. in Haskell, when using a `newtype Fix` generic constructor for fixpoints, while the open modular definition goes into a "recursion scheme"); thus the NNOOTT does not so much introduce an extra indirection step for recursion as it makes an existing indirection step obvious—and makes it dynamically rather than statically typed. In other words, it makes us realize once again that *recursion is not free*.

6.3.5 Why NNOOTT?

The NNOOTT was implicit in the original OO paper (Dahl and Nygaard 1967) as well as in Hoare's seminal paper that inspired it (Hoare 1965).[8] It then proceeded to dominate the type theory of OO until debunked in the late 1980s (Cook et al. 1989). Even after that debunking, it has remained prevalent in popular opinion, and still very

[8]Hoare probably intended subtyping initially indeed for his families of record types; yet subclassing is what he and the Simula authors discovered instead. Such is scientific discovery: if you knew in advance what lied ahead, it would not be a discovery at all. Instead, you set out to discover something, but usually discover something else, that, if actually new, will be surprising. The greater the discovery, the greater the surprise. And you may not realize what you have discovered until analysis is complete much later. The very best discoveries will then seem obvious in retrospect, given the new understanding of the subject matter, and familiarity with it due to its immense success.

active also in academia and industry alike, and continually reinvented even when not explicitly transmitted (AbdelGawad 2014; Cartwright and Moez 2013). And I readily admit it's the first idea I too had when I tried to put types on my modular extensions, as you can see in (Rideau et al. 2021).

The reasons why, despite being inconsistent, the NNOOTT was and remains so popular, not just among the ignorant masses, but even among luminaries in computer science, is well worth examining.

- The NNOOTT directly follows from the confusion between specification and target when conflating them without distinguishing them (section 3.3.2). The absurdity of the theory also follows from the categorical error of equating entities, the specification and its target, that not only are not equivalent, but are not even of the same type. But no one *intended* for "a class" to embody two very distinct semantic entities; quite on the contrary, Hoare, as well as the initial designers of Simula, KRL, Smalltalk, Director, etc., were trying to have a unified concept of "class" or "frame" or "actor", etc. Consequently, the necessity of considering two distinct entities was only fully articulated in the 2020s(!).

- In the 1960s and 1970s, when both OO and type theory were in their infancy, and none of the pioneers of one were familiar with the other, the NNOOTT was a good enough approximation that even top language theorists were fooled. Though the very first example in OO could have disproven the NNOOTT, still it requires careful examination and familiarity with both OO and Type Theory to identify the error, and pioneers had more urgent problems to solve.

- The NNOOTT actually works quite well in the simple "non-recursive" case that I characterized above. In particular, the NNOOTT makes sense enough in the dynamically typed languages that (beside the isolated precursor Simula) first experimented with OO in the 1970s and 1980s, mostly Smalltalk, Lisp and their respective close relatives. In those languages, the "types" sometimes specified for record fields are suggestions in comments, dynamic checks at best, sometimes promises made by the user to the compiler, never static guarantees made by the language; the recursive case was always dynamically typed, as was any non "atomic" value.

- Even in the 1980s and 1990s, theorists and practitioners being mostly disjoint populations, did not realize that they were not talking about precisely the same thing when talking about a "class". Those trained to be careful not to make categorical errors might not have realized that others were doing it in ways that mattered. The few at the intersection may not have noticed the discrepancy, or understood its relevance, when scientific modeling must necessarily make many reasonable approximations all the time. Once again, more urgent issues were on their minds.

- Though the NNOOTT is inconsistent in the general case of OO, as obvious from quite common examples involving recursion, it will logically satisfy ivory tower theorists or charismatic industry pundits who never get to experience cases more

complex than textbook examples, and pay no price for dismissing recursive cases as "anomalies" (Jacobs 1996) when confronted with them. Neither kind owes their success to getting a consistent theory that precisely matches actual practice.

- The false theory will also emotionally satisfy those practitioners and their managers who care more about feeling like they understand rather than actually understanding. This is especially true of the many who have trouble thinking about recursion, as is the case for a majority of novice programmers and vast majority of non-programmers. Even those who can successfully *use* recursion, might not be able to *conceptualize* it, much less criticize a theory of it.

6.3.6 Beyond the NNOOTT

The key to dispelling the "conflation of subtyping and inheritance" (Fisher 1996) or the "notions of type and class [being] often confounded" (Bruce 1996) is indeed first to have dispelled, as I just did previously, the conflation of specification and target. Thereafter, OO semantics becomes simple: by recognizing target and specification as distinct, one can take care to always treat them separately, which is relatively simple, at the low cost of unbundling them apart before processing, and rebundling them together afterwards if needed. Meanwhile, those who insist on treating them as a single entity with a common type only set themselves for tremendous complexity and pain.

That is how I realized that what most people actually mean by "subtyping" in extant literature is *subtyping for the target type of a class* (or target prototype), which is distinct from *subtyping for the specification type of a class* (or prototype), variants of the latter of which Kim Bruce calls "matching" (Bruce et al. 1997). But most people, being confused about the conflation of specification and target, don't conceptualize the distinction, and either try to treat them as if it were the same thing, leading to logical inconsistency hence unsafety and failure; or they build extremely complex calculi to do the right thing despite the confusion. By having a clear concept of the distinction, I can simplify away all the complexity without introducing inconsistency.

One can use the usual rules of subtyping (Cardelli and Wegner 1986) and apply them separately to the types of specifications and their targets, knowing that "subtyping and fixpointing do not commute", or to be more mathematically precise, *fixpointing does not distribute over subtyping*, or said otherwise, ***the fixpoint operator is not monotonic***: If `F` and `G` are parametric types, i.e. type-level functions from `Type` to `Type`, and `F ⊂ G` (where ⊂, sometimes written ⩽ or `<:`, is the standard notation for "is a subtype of", and for elements of `Type → Type` means `∀ t, F t ⊂ G t`), it does not follow that `Y F ⊂ Y G` where `Y` is the fixpoint operator for types.[9]

[9]The widening rules for the types of specification and their fixpoint targets are different; in other words, forgetting a field in a target record, or its some of its precise type information, is not at all the same as forgetting that field or its precise type in its specification (which introduces incompatible behavior with respect to inheritance, since extra fields may be involved as intermediary step in the specification, that must be neither forgotten, nor overridden with fields of incompatible types).

If the two entities are treated as a single one syntactically and semantically, as all OO languages so far have done, then their type system will have to encode in a weird way a pair of subtly different types for each such entity, and the complexity will have to be passed on to the user, with the object, and each of its field having two related but different declared types, and potentially different visibility settings. Doing this right

A more precise view of a modular extension is thus as an entity parameterized by the varying type `self` of the module context (that Bruce calls `MyType` (Bruce 1996; Bruce et al. 1997)). As compared to the previous parametric type `NModExt` that is parametrized by types `r i p`, this parametric type `ModExt` is itself parametrized by parametric types `r i p` that each take the module context type `self` as parameter:[10]

```
type ModExt required inherited provided =
  ∀ self, super : Type
    self ⊂ required self, super ⊂ inherited self ⇒
        self → super → provided self ∩ super
```

Notice how the type `self` of the module context is *recursively* constrained by `self ⊂ required self`), whereas the type `super` of the value in focus being extended is constrained by `super ⊂ inherited self`, and the returning a value is of type `provided self ∩ super`, and there is no direct recursion there (but there can be indirectly if the focus is itself referenced via self somehow). Those familiar with universal quantifiers may also notice how the quantification of `self` and `super`, in absence of reflective capabilities, pretty much forces those functions to be well-behaved with respect to gracefully passing through any call to a method they do not define, override or otherwise handle. Finally, notice how, as with the simpler NNOOTT variant above, the types `self` and `referenced self` refer to the module context, whereas the types `super` and `provided self` refer to some value in focus, that isn't at all the same as the module context for open modular extensions in general.

My two OO primitives then have the following type:

```
fix : ∀ required, inherited, provided : Type → Type, ∀ self, top :
Type,
      self = inherited self ∩ provided self,
      self ⊂ required self,
      top ⊂ inherited self ⇒
        top → ModExt required inherited provided → self
mix : ModExt r1 i1∩d2 p1 → ModExt r2 i2 p2 → ModExt r1∩r2 i1∩i2
p1∩p2
```

involves a lot of complexity, both for the implementers and for the users, at every place that object types are involved, either specified by the user, or display to him. Then again, some languages may do it wrong by trying to have the specification fit the rules of the target (or vice versa), leading to inconsistent rules and consistently annoying errors.

A typical way to record specification and target together is to annotate fields with visibility: `public` (visible in the target) yet possibly with a more specific type in the target than in the specification (to allow for further extensions that diverge from the current target); fields marked `protected` (visible only to extensions of the specification, not in the target); and fields marked `private` (not visible to extensions of the specification, even less so to the target; redundant with just defining a variable in a surrounding `let` scope). I retrieve these familiar notions from C++ and Java just by reasoning from first principles and thinking about distinct but related types for a specification and its target.

Now, my opinion is that it is actually better to fully decouple the types of the target and the specification, even in an "implicit pair" conflating the two: Indeed, not only does that means that types are much simpler, that also mean that intermediate computations, special cases to bootstrap a class hierarchy, transformations done to a record after it was computed as a fixpoint, and records where target and specification are out of sync because of effects somewhere else in the system, etc., can be safely represented and typed, without having to fight the typesystem or the runtime.

[10]The letters `r i p`, by contrast to the `s t a b` commonly used for generalized lenses, suggest the mnemonic slogan: "Generalized lenses can stab, but modular extensions can rip!"

In the `fix` function, I implicitly define a fixpoint `self` via suitable recursive subtyping constraints. I could instead make the last constraint a definition `self = Y (inherited ∩ provided)` and check the two subtyping constraints about `top` and `referenced`. As for the type of `mix`, though it looks identical with `ModExt` as the NNOOTT type previously defined with `NModExt`, there is an important but subtle difference: with `ModExt`, the arguments being intersected are not of kind `Type` as with `NModExt`, but `Type → Type`, where given two parametric types `f` and `g`, the intersection `f∩g` is defined by `(f∩g)(x) = f(x)∩g(x)`. Indeed, the intersection operation is defined polymorphically, and in a mutually recursive way for types, functions over types, etc.

6.3.7 Typing Advantages of OO as Modular Extensions

By defining OO in terms of the λ-calculus, indeed in two definitions `mix` and `fix`, I can do away with the vast complexity of "object calculi" of the 1990s, and use regular, familiar and well-understood concepts of functional programming such as subtyping, bounded parametric types, fixpoints, existential types, etc. No more `self` or `MyType` "pseudo-variables" with complex "matching" rules, just regular variables `self` or `MyType` or however the user wants to name them, that follow regular semantics, as part of regular λ-terms.[11] OO can be defined and studied without the need for ad hoc OO-specific magic, making explanations readily accessible to the public. Indeed, defining OO types in term of LaTeX deduction rules for ad hoc OO primitives is just programming in an informal, bug-ridden metalanguage that few are familiar with, with no tooling, no documentation, no tests, no actual implementation, and indeed no agreed upon syntax much less semantics...[12] the very opposite of the formality the authors affect.

Not having OO-specific magic also means that when I add features to OO, as I will demonstrate in the rest of this book, such as single or multiple inheritance, method combinations, multiple dispatch, etc., I don't have to update the language to use increasingly more complex primitives for declaration and use of prototypes or classes. By contrast, the "ad hoc logic" approach grows in complexity so fast that authors soon may have to stop adding features or find themselves incapable of reasoning about the result because the rules for those "primitives" boggle the mind.[13] Instead, I can let logic and typing rules be as simple as possible, yet construct my object features to be as sophisticated as I want, without a gap in reasoning ability, or inconsistency in the primitives.

[11] Syntactic sugar may of course be provided for optional use, that can automatically be macro-expanded away through local expansion only; but the ability to think directly in terms of the expanded term, with simple familiar universal logic constructs that are not ad hoc but from first principles, is most invaluable.

[12] See Guy Steele's keynote "It's Time for a New Old Language" at the Principles and Practice of Parallel Programming 2017 conference about the "Computer Science Metanotation" found in scientific publications.

[13] Authors of ad hoc OO logic primitives also soon find themselves incapable of fitting a complete specification within the limits of a conference paper, what's more with intelligible explanations, what's more in a way that anyone will read. The approach is too limited to deal even with the features 1979 OO (Cannon 1979), much less those of more modern systems. Meanwhile, readers and users (if any) of systems described with ad hoc primitives have to completely retool their in-brain model at every small change of feature, or introduce misunderstandings and bugs, instead of being able to follow a solidly known logic that doesn't vary with features.

My encoding of OO in terms of "modular extension", functions of the form `mySpec (self : Context, super : Focus) : Focus`, where in the general "open" case, the value under `Focus` is different from the `Context`, is also very versatile by comparison to other encodings, that are typically quite rigid, specialized for classes, and unable to deal with OO features and extensions. Beyond closed specifications for classes, or for more general prototypes, my `ModExt` type can scale down to open specifications for individual methods, or for submethods that partake in method combination; it can scale up to open specifications for groups of mutually defined or nested classes or prototypes, all the way to open or closed specifications for entire ecosystems.

More importantly, my general notion of "modular extension" opens an entire universe of algebraically well-behaved composability in the spectrum from method to ecosystem; the way that submethods are grouped into methods, methods into prototypes, prototypes into classes, classes into libraries, libraries into ecosystems, etc., can follow arbitrary organizational patterns largely orthogonal to OO, that will be shaped the evolving needs of the programmers, yet will at all times benefit from the modularity and extensibility of OO.

OO can be one simple feature orthogonal to many other features (products and sums, scoping, etc.), thereby achieving *reasonability*, i.e. one may easily reason about OO programs this way. Instead, too many languages make "classes" into a be-all, end-all ball of mud of more features than can fit in anyone's head, interacting in sometimes unpredictable ways, thereby making it practically impossible to reason about them, as is the case in languages like C++, Java, C#, etc.

6.3.8 Typing First-Class OO

I am aiming at understanding OO as *first-class* modular extensibility, so I need to identify what kind of types are suitable for that. The hard part is to type *classes*, and more generally specifications wherein the type of the target recursively refers to itself through the open recursion on the module context.

Happily, my construction neatly factors the problem of OO into two related but mostly independent parts: first, understanding the target, and second, understanding its instantiation via fixpoint.

I already discussed in section 6.2 how a class is a prototype for a type descriptor: the target is a record that describes one type and a set of associated functions. The type is described as a table of field descriptors (assuming it's a record type; or a list of variants for a tagged union, if such things are supported; etc.), a table of static methods, a table of object methods, possibly a table of constructors separate from static methods, etc. A type descriptor enables client software to be coded against its interface, i.e. being able to use the underlying data structures and algorithms without having to know the details and internals.

A first-class type descriptor is a record whose type is existentially quantified: (Cardelli and Wegner 1986; Mitchell and Plotkin 1988; Pierce and Turner 1993) as per the Curry–Howard correspondence, it is a witness of the proposition according to which "there is a type `T` that has this interface", where the interface may include field getters (functions of type $\mathtt{T} \to \mathtt{F}$ for some field value `F`), some field setters (functions of type $\mathtt{T} \to \mathtt{F} \to \mathtt{T}$ for a pure linear representation, or $\mathtt{T} \to \mathtt{F} \rightsquigarrow \mathtt{1}$ if I denote by $\rightsquigarrow$

a "function" with side-effects), binary tests (functions of type `T → T → 2`), binary operations (functions of type `T → T → T`), constructors for `T` (functions that create a new value of type `T`, at least some of which without access to a previous value of type `T`), but more generally any number of functions, including higher-order functions, that may include the type `T` zero, one or arbitrarily many times in any position "positive" or "negative", etc. So far, this is the kind of thing you could write with first-class ML modules, which embodies first-class modularity, but not modular extensibility.

Now, if the type system includes subtypes, extensible records, and fixpoints involving open recursion, e.g. based on recursively constrained types (Eifrig et al. 1995b,a), then those first-class module values can be the targets of modular extensions.

And there we have first-class OO capable of expressing classes.

Regarding subtyping, however, note that when modeling a class as a type descriptor, not only is it absolutely not required that a subclass's target should be a subtype of its superclass's target (which would be the NNOOTT above), but it is not required either that a subclass's specification should be a subtype of its superclass's specification. Indeed, adding new variants to a sum type, which makes the extended type a supertype of the previous, is just as important as adding fields to a product type (or specializing its fields), which makes the extended type a subtype of the previous. Typical uses include extending a language grammar (as in (Garrigue 2000)), defining new error cases, specializing the API of some reified protocol, etc. In most statically typed OO languages, that historically mandate the subclass specification type to be a subtype of its superclass specification types, programmers work around this limitation by defining many subclasses of each class, one for each of the actual cases of an implicit variant; but this coping strategy requires defining a lot of subclasses, makes it hard to track whether all cases have been processed; essentially, the case analysis of the sum type is being dynamically rather than statically typed.

Note on Types for Second-Class Class OO Types for second-class classes can be easily deduced from types for first-class classes: A second-class class is "just" a first-class class that happens to be statically known as a compile-time constant, rather than a runtime variable. The existential quantifications of first-class OO and their variable runtime witnesses become unique constant compile-time witnesses, whether global or, for nested classes, scoped. This enables many simplifications and optimizations, such as lambda-lifting (making all classes global objects, modulo class parameters), and monomorphization (statically inlining each among the finite number of cases of compile-time constant parameters to the class), and inlining globally constant classes away.

However, you cannot at all deduce types for first-class classes from types for second-class classes: you cannot uninline constants, cannot unmake simplifying assumptions, cannot generalize from a compile-time constant to a runtime variable. The variable behavior essentially requires generating code that you wouldn't have to generate for a constant. That is why typing first-class prototypes is more general and more useful than only typing second-class classes; and though it may be harder in a way, involving more elaborate logic, it can also be simpler in other ways, involving more uniform concepts.

6.3.9 First-Class OO Beyond Classes

My approach to OO can vastly simplify types for it, because it explicitly decouples concepts that previously people implicitly conflated: not only specifications and their targets, but also modularity and extensibility, fixpoints and types.

By decoupling specifications and targets, I can type them separately, subtype them separately, not have to deal with the extreme complexity of the vain quest of trying to type and subtype them together.

By decoupling modularity and extensibility, I can type not just closed specifications, but also open specifications, which makes everything so much simpler, more composable and decomposable. Individual class, object, method, sub-method specifications, etc., can be typed with some variant of the `C` → `V` → `V` pattern, composed, assembled in products or co-products, etc., with no coupling making the unit of specification the same as the unit of fixpointing.

Finally, with typeclass-style (as in section 6.2.4), the unit of fixpointing need not be a type descriptor; it could be a value without an existential type, or a descriptor for multiple existential types; it could be a descriptor not just for a finite set of types, but even an infinite family of types, etc. In an extreme case, it can even be not just a type descriptor, but the entire ecosystem — a pattern actively used in Jsonnet or Nix (though without formal types).

To build such records, one may in turn make them the targets of modular extensions, such that first-class OO can express much more than mere "classes", especially so than second-class classes of traditional Class OO. First-class OO can directly express sets of cooperating values, types and algorithms parameterized by other values, types and algorithms.

6.4 Stateful OO

6.4.1 Mutability of Fields as Orthogonal to OO

I showed how OO is best explained in terms of pure lazy functional programming, and how mutable state is therefore wholly unnecessary for OO. There have been plenty of pure functional object libraries since at least the 1990s, even for languages that support mutable objects; OO languages that do not support any mutation at all have also existed since at least the 1990s, and practical such languages with wide adoption exist since at least the early 2000s.

Yet, historical OO languages (Lisp, Simula, Smalltalk), just like historical OO-less languages of the same time (FORTRAN, ALGOL, Pascal), were stateful, heavily relying on mutation of variables and record fields. So are the more popular OO and OO-less languages of today, still, though there are now plenty of less-popular "pure (functional)" options. How then does mutation fit in my functional OO paradigm? The very same way it does on top of Functional Programming in general, with or without OO: by adding an implicit (or then again explicit) "store" argument to all (or select) functions, that gets linearly (or "monadically") modified and passed along the semantics of those functions (the linearity, uniqueness or monadicity ensuring that there is one single shared state at a time for all parts of the program). Then, a mutable variable

or record field is just a constant pointer into a mutable cell in that store, a "(mutable) reference".

This approach perfectly models the mutability of object fields, as found in most OO languages. It has the advantages of keeping this concern orthogonal to others, so that indexed products, fixpoints, mutation, visibility rules and subtyping constraints (separately before and after fixpointing), etc., can remain simple independent constructs each with simple reasoning rules, logically separable from each other yet harmoniously combinable together. By contrast, the "solution" found in popular languages like C++ or Java is all too often to introduce a single mother-of-all syntactic and semantic construct of immense complexity, the "class", that frankly not a single person in the world fully understands, and of which scientific papers only dare study but simplified (yet still very complex) models.[14]

Actually, when I remember that in most OO languages, OO is only ever relevant but at compile-time, I realize that of course mutation is orthogonal to OO, even in these languages, nay, especially so in these languages: since OO fragments are wholly evaluated at a time before there is any mutation whatsoever, mutation cannot possibly be part of OO, even though it is otherwise part of these languages. Indeed the compile-time programming model of these languages, if any, is pure lazy functional. Thus, whether fields are mutable or immutable is of precious little concern to the compiler fragment that processes OO: it's just a flag passed to the type checker and code generator after OO is processed away.

6.4.2 Mutability of Inheritance as Code Upgrade

Keeping mutability orthogonal to OO as above works great as long as the fields are mutable, but the inheritance structure of specifications is immutable. Happily, this covers every language with second-class classes (which is most OO languages), but also all everyday uses of OO even in languages with first-class prototypes and classes. Still,

[14]The concept of class is the "Katamari" of semantics: just like in the 2004 game "Katamari Damacy", it is an initially tiny ball that indiscriminately clumps together with everything on its path, growing into a Big Ball of Mud (Foote and Yoder 1997), then a giant chaotic mish mash of miscellaneous protruding features, until it is so large that it collapses under its own weight to become a star—and, eventually, a black hole into which all reasonable meaning disappears never to reappear again. Languages like C++, Java, C#, Scala, have a concept of class so complex that it boggles the mind, and it keeps getting more complex with each release. "C++, like Perl, is a Swiss-Army chainsaw of a programming language. But all the blades are permanently stuck half-open while running full throttle." I much prefer the opposite approach: have a small basis of orthogonal primitives, that follow a few simple rules that can simultaneously fit in a brain, out of which to make large assemblies (but no larger than needed), from which the emerging meaning is always explainable. Certainly, there is a time when the emerging meaning of a large enough assembly has a complexity of its own that cannot be reduced, but this is *intrinsic* complexity. Classes made into humongously complex language primitives introduce *extrinsic* complexity, omnipresent unnecessary parasitic interactions—which utterly defeats the very purpose for which classes were invented: modularity, the ability to clearly think in terms of entities that minimally interact with each other. If a large utility function or macro must exist to define classes, each use of it should clearly reduce to an entity that could have been directly written without the utility, though perhaps in a less concise way, but still can be described in terms of simple primitives, with no more exhibited affordances than needed. Now, at a certain scale, the smaller entities disappear behind the abstraction, and those that were large become the building blocks; but if they were built with bad primitives, the protruding affordances will generate lots of unwanted interactions that break the abstraction, causing the semantic plumbing to leak everywhere, and offering attack vectors for active enemies to enter between the cracks.

there are use cases in which changes to class or prototype hierarchies is actively used by some dynamic OO systems such as Smalltalk or Lisp: to support interactive development, or schema upgrade in long-lived persistent systems. How then to model mutability of the inheritance structure itself, when the specification and targets of prototypes and classes are being updated?

First, I must note that such events are relatively rare, because they involve programmers not only typing, but thinking, which happens millions of times slower than computers process data. Most evaluation of most programs, especially where performance matters, happens in-between two such code upgrades, in large spans of time during which the code is constant. Therefore, the usual semantics that consider inheritance structures as constant still apply for an overwhelming fraction of the time and an overwhelming fraction of objects, even in presence of such mutability. There are indeed critical times when these usual semantics are insufficient, and the actual semantics must be explained; but these usual semantics are not rendered irrelevant by the possibility of dynamic changes to object inheritance.

Second, updates to the inheritance structure of OO specifications can be seen as a special case of code upgrade in a dynamic system. Code upgrade, whether it involves changes to inheritance structure or not, raises many issues such as the atomicity of groups of upgrades with respect to the current thread and concurrent threads, or what happens to calls to updated functions from previous function bodies in frames of the current thread's call stack, how the code upgrades do or do not interfere with optimizations such as inlining or reordering of function calls, how processor caches are invalidated, page permissions are updated, how coherency is achieved across multiple processors on a shared memory system, etc. These issues are not specific to inheritance mutability, and while they deserve a study of their own, the present paper is not the right place for such a discussion. The only popular programming language that fully addresses all these code upgrade issues in its defined semantics is Erlang (Armstrong 2003); however it does not have any OO support, and its approach is not always transposable to other languages.[15] Lacking such deep language support, user support is required to ensure upgrades only happen when the system is "quiescent" (i.e. at rest, so there are no issues with outdated code frames upstack or in concurrent threads) for "Dynamic Software Updating" (Hicks et al. 2001); at that point, the compiler need only guarantee that calls to the upgradable entry points will not have been inlined. Happily, since code upgrade events happen at a much larger timescale than regular evaluation, it is also generally quite acceptable for systems to wait until the right moment that the system is indeed quiescent, after possibly telling its activities to temporarily shutdown, before applying such code upgrades.

Third, I must note how languages such as Smalltalk and Common Lisp include a lot of support for updating class definitions, including well-defined behavior with respect to how objects are updated when their classes change: see for instance the

[15] Erlang will notably kill processes that still use obsolete code from before the current version now being upgraded to the next one. This is possible because Erlang has only very restricted sharing of state between processes, so it can ensure PCLSRing (Bawden 1989) without requiring user cooperation; this is useful because Erlang and its ecosystem have a deep-seated "let it fail" philosophy wherein processes randomly dying is expected as a fact of life, and much infrastructure is provided for restarting failed processes, that developers are expected to use.

protocol around `update-instance-for-redefined-class` in CLOS, the Common Lisp Object System (Bobrow et al. 1988). These facilities allow continuous concurrent processing of data elements with some identity preserved as the code evolves, even as the data associated to these identities evolves with the code. Even then, these languages do not provide precise and portable semantics for how such code upgrade upgrades interfere with code running in other threads, or in frames up the stack from the upgrade, so once again, users must ensure quiescence or may have to deal with spurious incoherence issues, up to possible system corruption.

Lastly, as to providing a semantics for update in inheritance structure, language designers and/or programmers will have to face the question of what to do with previously computed targets when a specification is updated: Should a target once computed be left forever unchanged, now out-of-synch with the (possibly conflated) specification? Should a target be wholly invalidated, losing any local state updates since it was instantiated? Should a target have "direct" properties that override any computation involving inheritance, while "indirect" properties are recomputed from scratch just in case the inheritance structure changed? Should some "indirect" properties be cached, and if so how is this cache invalidated when there are changes? Should a protocol such as `update-instance-for-redefined-class` be invoked to update this state? Should this protocol be invoked in an eager or lazy way (i.e. for all objects right after code update, or on a need basis for each object)? Should a class maintain at all times and at great cost a collection of all its instances, just so this protocol can be eagerly updated once in a rare while? Should some real-time system process such as the garbage collector ensure timely updates across the entire heap even in absence of such explicitly maintained collection? Are children responsible for "deep" validity checks at every use, or do parents make "deep and wide" invalidations at rare modifications, or must parents and children somehow deal with incoherence? There is no one-size-fits-all answer to these questions.

If anything, thinking in terms of objects with identity and mutable state both forces software designers to face these issues, inevitable in interactive or long-lived persistent systems, and provides them with a framework to give coherent answers to these questions. Languages that assume "purity" or lack of code upgrade, thereby deny these issues, and leave their users helpless, forced to reinvent entire frameworks of mutation so they may then live in systems they build on top of these frameworks, rather than directly in the language that denies the issues.

Chapter 7

Inheritance: Mixin, Single, Multiple, or Optimal

> Inheritance of methods encourages modularity by allowing objects that have similar behavior to share code. Objects that have somewhat different behavior can combine the generalized behavior with code that specializes it.
> Multiple inheritance further encourages modularity by allowing object types to be built up from a toolkit of component parts.
>
> David Moon (Moon 1986)

7.1 Mixin Inheritance

7.1.1 The Last Shall Be First

What I implemented in the previous chapters is mixin inheritance (see section 3.5.4): the last discovered and least well-known variant of inheritance. And yet, I already discussed above that object prototypes with mixin inheritance are used to specify software configurations at scale. I further claim that it is the most fundamental variant of inheritance, since I built it in two lines of code, and will proceed to build the other variants on top of it.

7.1.2 Mixin Semantics

I showed above (see section 5.3) that mixin inheritance involves just one type constructor `ModExt` and two functions `fix` and `mix`, repeated here more concisely from above:

```
type ModExt r i p = ∀ s, t : Type . s ⊂ r s, t ⊂ i s ⇒ s → t →
(p s)∩t
```

```
fixt : ∀ r i p : Type → Type, ∀ s, t : Type .
       s = i s ∩ p s, s ⊂ r s, t ⊂ i s ⇒
       t → ModExt r i p → s
mix : ModExt r1 i1∩p2 p1 → ModExt r2 i2 p2 → ModExt r1∩r2 i1∩i2
p1∩p2

(define fixt (λ (m) (Y (λ (s) ((m s) top)))))
(define mix (λ (c p) (λ (r) (compose (c r) (p r)))))
```

7.2 Single Inheritance

7.2.1 Semantics of Single Inheritance

In Single Inheritance, the specifications at stake are open modular definitions, as studied in section 5.2, simpler than the modular extensions of mixin inheritance from section 5.3.[1] Modular definitions take a `self` as open recursion argument and return a record using `self` for self-reference. Unlike modular extensions, they do not take a `super` argument, since they only are inherited from, but don't themselves inherit, at least not anymore: what superclass they did inherit from is a closed choice made in the past, not an open choice to make in the future; it is baked into the modular definition already. The semantics can then be reduced to the following types and functions:

```
type ModDef r p = ∀ s : Type . s ⊂ r s ⇒ s → p s
fixModDef : ModDef p p → Y p
extendModDef : ModExt r1 p2 p1 → ModDef r2 p2 → ModDef r1∩r2 p1∩p2
baseModDef : ModDef (λ (_) Top) (λ (_) Top)

(define fixModDef Y)
(define extendModDef (λ (mext) (λ (parent) (λ (self)
  (mext self (parent self))))))
(define baseModDef (λ (_) top))
```

Note how the type for an open modular definition has two parameters `r` (required) and `p` (provided), but a closed modular definition has the same value for those two parameters. There is no parameter `i` (inherited), just like there was no argument `super`.

I already showed how the instantiation function for a closed modular definition was simply the fixpoint combinator `Y`. The case of extending a modular definition is more interesting. First, I will simply remark that since extending works on open modular

[1] In (Bracha and Cook 1990; Cook 1989), Cook calls "generator" what I call "modular definition", and "wrapper" what I call "modular extension". But those terms are a bit too general, while Cook's limitation to records makes his term assignment a bit not general enough (only closed and specialized for record). Even in the confines of my exploration of OO, I already used the term "wrapper" in a related yet more specific way when discussing wrapping references for recursive conflation in section 6.1.3; and a decade before Cook, Cannon (Cannon 1979) also used a notion of wrapper closer to what I use, in Flavor's predecessor to CLOS `:around` methods (Steele 1990), or in the more general case, to CLOS declarative method combinations. The term "generator" is also too generic, and could describe many concepts in this book, while being overused in other contexts, too. I will thus stick with my expressions "modular definition" and "modular extension" that are not currently in widespread use in computer science, that are harder to confuse, and that I semantically justified by reconstructing their meaning from first principle.

definitions, not just on closed ones like instantiating, the value under focus needs not be the same as the module context. But more remarkably, extension in single inheritance requires you use a modular *extension* in addition to an existing modular definition.

When building a modular definition through successive extensions, an initial known existing modular definition is needed as a base case to those extensions; this role is easily filled by the base modular definition `baseModDef`, that given some modular context, just returns the `top` value. Now the recursive case involves a different kind of entities, modular extensions. But I showed that modular extensions were already sufficient by themselves to define mixin inheritance. Why then ever use single inheritance, since it still requires the entities of mixin inheritance in addition to its own? Might one not as well directly adopt the simpler and more expressive mixin inheritance?

7.2.2 Comparing Mixin- and Single- Inheritance

Mixin Inheritance is Simpler than Single Inheritance, *assuming FP* Assuming knowledge of Functional Programming (FP), the definitions of single inheritance above are slightly more complex than those of mixin inheritance, and noticeably more awkward. And indeed, *if* you have already paid the price of living in a world of functional programming, with higher-order functions and sufficiently expressive types and subtypes and fixpoints, and of thinking in terms of programming language semantics, you might not need single inheritance at all.

But while Functional Programming and its basic concepts including lexical scoping and higher-order functions may be boringly obvious to the average programmer of 2025, they were only fully adopted by mainstream OO programming languages like C++, Java in the 2010s, and slightly earlier for C#, after JavaScript became popular in application development and made FP popular with it, in the 2000s. Back when single inheritance was invented in the 1960s, these were extremely advanced concepts that very few mastered even among language designers. Unlike mixin inheritance, single inheritance does not require any FP, and many languages have or had single inheritance and no FP, including Simula, many Pascal variants, early versions of Ada or Java or Visual Basic.

Neither lexical scoping nor higher-order functions are required for single inheritance because the "modular extension" conceptually present in the extension of a modular definition need never be explicitly realized as a first-class entity: literally using my above recipe to implement a class or prototype definition with single inheritance would involve building a modular extension, then immediately applying it with `extendModDef`, only to forget it right afterwards; but instead, most OO languages would support some special purpose syntax for the definition, and process it by applying the extension to its super specification as it is being parsed, without actually building any independent first-class entity embodying this extension. The semantics of this special purpose syntax are extremely complex to explain without introducing FP concepts, but neither implementors nor users need actually conceptualize that semantics to implement or use it.

Mixin Inheritance is More Expressive than Single Inheritance Single inheritance can be trivially expressed in terms of mixin inheritance: a single inheritance specification is just a list of modular extensions composed with a base generator at one end; you can achieve the same effect by composing your list of modular extensions, and instantiate it as usual with the top value as the seed for inheritance. Single inheritance can be seen as a restrictive style in which to use mixin inheritance, wherein the modular extensions considered as specification will be tagged so they are only ever used but as the second argument (to the right) of the `mix` function, never the first. Meanwhile, those extensions used as the first argument (to the left) of the `mix` function must be constant, defined on the spot, and never reused afterwards. Thus, single inheritance is no more expressive than mixin inheritance.

Conversely, given a language with FP and dynamic types or sufficiently advanced types, you can implement first-class mixin inheritance on top of first-class single inheritance by writing a function that abstracts over which parent specification a specification will inheritance from, as in Racket née PLT Scheme (Flatt et al. 2006; Flatt et al. 1998). In terms of complexity, this construct puts the cart before the horse, since it would be much easier to build mixin inheritance first, then single inheritance on top. Still, this style is possible, and may allow to cheaply leverage and extend existing infrastructure in which single inheritance was already implemented and widely used.

Just like in mixin inheritance, a *target* can thus still be seen as the fixed point of the composition of a list of elementary modular extensions as applied to a top value. However, since modular definitions, not modular extensions, are the specifications, the "native" view of single inheritance is more to see the parent specified in `extend` as a direct super specification, and the transitive supers-of-supers as indirect super specifications; each specification is considered as not just the modular extension it directly contributes, but as the list of all modular extensions directly and indirectly contributed.

Now what if you only have second-class class OO, and your compile-time language lacks sufficiently expressive functions to build mixin inheritance atop single inheritance? Then, mixin inheritance is strictly more expressive (Felleisen 1991) than single inheritance: You can still express single inheritance as a stunted way of using mixin inheritance. But you can't express mixin inheritance on top of single inheritance anymore. Single inheritance only allows you to build a specification as a list of modular extensions to which you can add one more modular extension at a time (as in `cons`), "tethered" to the right. Meanwhile, mixin inheritance allows you to build a specification as a list of modular extensions that you can concatenate with other lists of modular extensions (as in `append`), mixable both left and right. If you have already defined a list of modular extensions, and want to append it in front of another, single inheritance will instead force you to duplicate the definition of each and every of those modular extensions in front of the new base list. See next section for how that makes single inheritance less modular.

Finally, since the two are equivalent in the context of first-class OO with higher-order functions, but different in the more common context of second-class OO without higher-order second-class functions, it makes sense to only speak of single inheritance in a context where the language syntax, static type system, dynamic semantics, or socially enforced coding conventions, or development costs somehow disallow or strongly discourage modular extensions as first-class entities.

Mixin Inheritance is More Modular than Single Inheritance In second-class class OO with single inheritance, each modular extension can only be used once, at the site that it is defined, extending one modular definition's implicit list of modular extensions. By contrast, with mixin inheritance, a modular extension can be defined once and used many times, to extend many different lists of modular extensions.

Thus, should a `WeightedColoredPoint` inherit from `ColoredPoint` and then have to duplicate the functionality from `WeightedPoint`, or should it be the other way around? Single inheritance forces you not only to duplicate the functionality of a class, but also to make a choice each time of only one which will be inherited from to reuse code. Multiply this problem by the number of times you combine duplicated functionality. This limitation can cause a maintenance nightmare: bug fixes and added features must also be duplicated; changes must be carefully propagated everywhere; subtle discrepancies creep in, that cause their own issues, sometimes critical.

When there are many such independent features that are each duplicated onto many classes, the number of duplicated definitions can grow quadratically with the number of desired features, while the potential combinations grows exponentially, requiring users to maintain some arbitrary order in that combination space. Important symmetries are broken, arbitrary choices must be made all over the codebase, and the code is more complex not just to write, but also to think about. Every instance of this duplication is external modularity through copy/paste rather than internal modularity through better language semantics. Overall, single inheritance is much less modular than mixin inheritance, and in that respect, it fails to fulfill the very purpose of inheritance.

Mixin Inheritance is Less Performant than Single Inheritance If single inheritance is more complex, less expressive and less modular than mixin inheritance, is there any reason to ever use it? Yes: the very semantic limitations of single inheritance are what enables a series of performance optimizations.

Using single inheritance, the system can walk the method declarations from base to most specific extension, assign an index number to each declared method, and be confident that every extension to a specification will assign the same index to each and every inherited methods. Similarly for the fields of a class. Method and field lookup with single inheritance can then be as fast as memory access at a fixed offset from the object header or its class descriptor (or "vtable"). And code using this fast strategy for access to methods and fields of a specification is still valid for all descendants of that specification, and can be shared across them.

By contrast, when using mixin inheritance, because the code for a method cannot predict in advance what other modular extensions will have been mixed in before or after the current one, and thus cannot assume any common indexes between the many instances of the prototype or class being specified; in the general case, a hash-table lookup will be necessary to locate any method of element field provided by an instance of the current specification, which is typically ten to a hundred times slower than fixed offset access. Some caching can speed up the common case somewhat, but it will remain noticeably slower than fixed offset access, and caching cannot wholly avoid the general case.

The simplicity of implementation and performance superiority of single inheritance

makes it an attractive feature to provide even on OO systems that otherwise support mixin inheritance or multiple inheritance (that has the same performance issues as mixin inheritance). Thus, Racket's default object system has both single and mixin inheritance, and Common Lisp, Ruby and Scala have both single and multiple inheritance. Users can selectively use single inheritance when they want more performance across all the subclasses of a given class.

7.3 Multiple Inheritance

7.3.1 Correct and Incorrect Semantics for Multiple Inheritance

With multiple inheritance (see section 3.5.3), a specification can declare a list of parent specifications that it inherits from. Each specification may then contribute methods to the overall definition of the target. The list can be empty in which case the specification is a base specification (though many systems add a special system base specification as an implicit last element to any user-specified list), or can be a singleton in which case the inheritance is effectively the same as single inheritance, or it can have many elements in which case the inheritance is actually multiple inheritance.

Now, early OO systems with multiple inheritance (and sadly many later ones still) didn't have a good theory for how to resolve methods when a specification inherited different methods from multiple parents, and didn't provide its own overriding definition (Borning 1977; Curry et al. 1982; Ingalls 1978). This situation was deemed a "conflict" between inherited methods, which would result in an error, at compile-time in the more static systems. Flavors (Cannon 1979) identified the correct solution, that involves cooperation and harmony rather than conflict and chaos. Failing to learn from Flavors, C++ (Stroustrup 1989) and after it Ada not only issue an error like older systems, they also try to force the ancestry DAG into a tree like CommonObjects (Snyder 1986). Self initially tried a weird resolution method along a "sender path" that dives depth first into the first available branch of the inheritance DAG without backtracking (Chambers et al. 1991), but the authors eventually recognized how wrongheaded that was, and reverted to, sadly, the conflict paradigm (Ungar and Smith 2007).[2]

I will focus mainly on explaining the correct, *flavorful* semantics for multiple inheritance, discovered by Flavors, and since then widely but sadly not universally accepted. But I must introduce several concepts before I can offer a suitable formalization; and along the way, I will explain where the *flavorless* dead end of "conflict" stems from.

7.3.2 Specifications as DAGs of Modular Extensions

I will call "inheritance hierarchy", or when the context is clear, "ancestry", the transitive closure of the parent relation, and "ancestor" is an element of this ancestry. With

[2]Like the "visitor pattern" approach to multiple dispatch, the Self's once "sender path" approach to multiple inheritance fails to capture semantics contributed by concurrent branches of a partial order, by eagerly taking the first available branch without backtracking. In the end, like the "conflict" approach to method resolution though in a different way, it violates of the "linearity" property I describe in section 7.3.5, which explains why it cannot be satisfying.

single inheritance, this ancestry was a list. With multiple inheritance, where each specification may inherit from multiple parents, the ancestry of a specification is not a list as it was with single inheritance. It is not a tree, either, because a given ancestor can be reached through many paths. Instead, the ancestry of a specification is a Directed Acyclic Graph (DAG). And the union of all ancestries of all specifications is also a DAG, or which each specification's ancestry is a "suffix" sub-DAG (i.e. closed to further transitive parents, but not to further transitive children), of which the specification is the most specific element.

Note that in this book, I will reserve the word "parent" for a specification another ("child") specification depends on, and the word "super" to the partial target the value that is inherited as argument passed to the child's modular extension. This is consistent with my naming the second argument to my modular extensions `super` (sometimes shortened to `t`, since `s` is taken for the first `self` argument) and the second argument to my `mix` function `parent` (sometimes shortened to `p`). Extant literature tends to confuse specification and target as the same entity "class" or "prototype" without being aware of a conflation, and so confusing "parent" and "super" is par for the course in that literature. My nomenclature also yields distinct terms for "parent" and "ancestor" where the prevailing nomenclature has the slightly confusing "direct super" and "super" (or "direct superclass" and "superclass", in a literature dominated by Class OO).

The ancestor relation can also be viewed as a partial order on specifications (and so can the opposite descendant relation). In the single inheritance case, this relation is a total order over a given specification's ancestry, and the union of all ancestries is a tree, which is a partial order but more restricted than a DAG. I can also try to contrast these structures with that of 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 an ancestor specification (and its own ancestors) can appear multiple times in a specification's tree.

7.3.3 Representing Specifications as DAG Nodes

To represent a specification in multiple inheritance, one will need not just a modular extension, but a record of:

(a) a modular extension, as in mixin inheritance, that contributes an increment to the specification,

(b) an ordered list of parent specifications it inherits from, that specify increments of information on which it depends, and

(c) a tag (unique name, fully qualified path, string, symbol, identifier, number, etc.) to uniquely identify each specification as a node in the inheritance DAG.

Most languages support generating without side-effect some kind of tag for which some builtin comparison operator will test identity with said entity. Many languages also support such identity comparison directly on the specification record, making the tag redundant with the specification's identity. To check specification identity, I will thus use `eq?` in Scheme, but you could use == in Java, === in JavaScript, address equality in C or C++, etc. Implementation of multiple inheritance will also be significantly

sped up if records can be sorted by tag, or by stable address or hash, so that looking up a record entry, merging records, etc., can be done efficiently; but I will leave that as an exercise to the reader. Side-effects could also be used to generate unique identifying numbers for each specification; but note that in the case of second-class OO, those effects would need be available at compile-time. If the language lacks any of the above features, then users can still implement multiple inheritance by manually providing unique names for specifications; but maintaining those unique names is a burden on users that decreases the modularity of the object system, and can become particularly troublesome in nested and computed specifications. Interestingly, the λ-calculus itself crucially lacks the features needed for DAG node identity; a tag must be externally provided, or side-effects (or a monad encoding) are required for a counter.

The type for a multiple inheritance specification would thus look like the following, where `Nat` is the type of natural numbers, `Iota` introduces a finite dependent type of given size, `DependentList` introduces a dependent list, `Tag` is a type of tags giving the specifications an identity as nodes in a DAG, and the `{...}` syntax introduces some kind of record type.

```
type MISpec r i p =
  ∀ r i p : Type → Type .
  ∀ l : Nat .
  ∀ pr pi pp : Iota l → Type → Type .
  r ⊂ Intersection pr,
  i ∩ Intersection pp ⊂ Intersection pi ⇒
  { getModExt : ModExt r i p ;
    parents : DependentList j: (ModExt (pr j) (pi j) (pp j)) ;
    tag : Tag }
```

7.3.4 Difficulty of Method Resolution in Multiple Inheritance

Then comes the question of how to instantiate a multiple inheritance specification into a target. It seems obvious enough that the inheritance DAG of modular extensions should be reduced somehow into a single "effective" modular definition: only then can specifications of large objects and ecosystems be composed from specifications of many smaller objects, methods, etc., such that the effective modular definition for a record of methods is the record of effective modular definitions for the individual methods. What then should be the "super" argument passed to each modular extension, given the place of its specification in the ancestry DAG?

The Diamond Problem One naive approach could be to view the inheritance DAG as some kind of attribute grammar, and compute the (open modular definition for) the super at each node of the DAG as a synthetic attribute,[3] by somehow combining the modular definitions at each of the supers. After all, that's what people used to do with single inheritance: synthesize the modular definition of the child from that of the parent

[3]Beware that what is typically called "child" and "parent" in an attribute grammar is inverted in this case relative to what is "child" and "parent" in the inheritance DAG. For this reason, computing effective modular extensions from ancestor to descendant along the inheritance DAG makes that a synthesized attribute rather than an inherited attribute along the attribute grammar. This can be slightly confusing.

and the child's modular extension. Unhappily, I showed earlier in section 5.2 that there is no general way to combine multiple modular definitions into one, except to keep one and drop the others. Modular extensions can be composed left and right, but modular definitions can only be on the right and on the left must be a modular extension.

The difficulty of synthesizing a modular definition is known as the "diamond problem" (Bracha 1992; Taivalsaari 1996):[4] Consider a specification C with two parents B1 and B2 that both have a common parent A. The contribution from A has already been baked into the modular definitions of each of B1 and B2; therefore trying to keep the modular definitions of both B1 and B2 leads to duplication of what A contributed to each, which can cause too many side-effects, resource explosion, yet possibly still the loss of what the B2 contributed, when the copy of A within B1 reinitializes the method (assuming B2 is computed before B1). Keeping only one of either B1 or B2 loses information from the other. There is no good answer; any data loss increases linearly as diamonds get wider or more numerous; meanwhile any duplication get exponentially worse as diamonds stack, e.g. with E having parents D1 and D2 sharing parent C, and so on. That is why Mesa, Self, C++, Ada, PHP, etc., view multiple distinct methods as a "conflict", and issue an error if an attempt is made to rely on a method definition from specification C's parents; C has to provide a method override, to specify one of its parents to effectively inherit the method from, or to signal an error if the method is called.

The "conflict" approach is internally consistent; but it is probably the single least useful among all possible consistent behaviors:

- The approach drops all available information in case of conflict; users are then forced to otherwise reimplement the functionality of all but at most one of the methods that could have been combined, thereby failing in large part the Criterion for Extensibility (see section 4.2.3).

- Even this reimplementation in general is impractical or at times impossible; the whole point of modularity is that the person doing the extensions is not an expert in the code being extended and vice versa (division of labor); the source code for the module being extended might not even be available for legal reasons, and not be understandable even when it is; even when the code is available, understandable and legally copyable, it may not be affordable to keep up with changes in code from a different project, with people moving at a speed and in a direction incompatible with one's own schedule. This is a big failure for Modularity (see section 4.1.3).

- Users trying to circumvent the broken inheritance mechanism still have to invent their own system to avoid the same exponential duplication problem that the implementers of the OO system have punted on. They are in a worse position to do it because they are mere users; and their solution will involve non-standard coding conventions that will not work across team boundaries. This is another big failure for Modularity (again see section 4.1.3).

[4] Bracha says he didn't invent the term "diamond problem", that must have already circulated in the C++ community; his thesis quotes Bertrand Meyer who talks of "repeated inheritance".

Now, if computing a modular definition from parent modular definitions, conflict detection and picking a winner are the only consistent solutions, and the latter is not much better than the former, less symmetrical, and more prone to wasting hours of programmer time by silently doing the wrong thing. Which means, better behavior has to *not* be simply based on synthesizing a child's modular definition from its parents' modular definition.

Cooperation not Conflict To find a better consistent behavior than conflict requires a reassessment of what better designed attribute than a modular definition should be synthesized from the inheritance DAG if any. Step back.[5] From what can you extract a modular definition, that isn't bothered by diamonds? How about a modular extension? Well, there are these individual modular extensions that you can compose. Ah, but you can't just compose everything with exponential repetitions. How then do you find an ordered list of modular extensions to compose without repetition, and how can you maximize modularity as you determine that list? And, if you step back further—what are all the consistency constraints that this ordered list should satisfy, and how do you know?

7.3.5 Consistency in Method Resolution

Here are important consistency properties for method resolution to follow, also known as constraints on the method resolution algorithm. There is sadly no consistent naming for those properties across literature, so I will propose my own while recalling the names previously used.

Inheritance Order: Consistency with Inheritance A specification's modular extension shall always be composed "to the left" of any of its ancestors', where sequential effects and computation results flow right to left. Thus children may have as preconditions the postconditions of their parents.

Thus, if `method-spec` declares `record-spec` as a parent, every method defined or overridden by the former can safely assume that indeed there will be a properly initialized record into a specific field of which to define or override the value. Overrides will happen after initialization, and will not be cancelled by a duplicate of the initialization. Similarly, if specification adds a part to a design, it can depend on `base-bill-of-parts` as a declared parent, it can be confident that when it registers a part, the part database will already be initialized, and will not be overwritten later.

This property is so fundamental it is respected by all OO languages since Simula (Dahl and Nygaard 1967), and may not have been explicitly named before as distinct from inheritance itself.

Linearity: Conservation of Information The information contributed by each ancestor's modular extension shall be taken into account once and only once. User-specified extensions may drop or duplicate information, but the system-provided algorithms that combine those extensions and are shared by all methods must not.

[5] As Alan Kay said, "Perspective is worth 80 IQ points".

Thanks to this property, a specification for a part as above can declare the base-bill-of-parts as a parent then safely assume that the part database will be initialized before it is used (no ignoring the initialization), and won't be reinitialized again after registration, cancelling the registration (no duplicating the initialization). Each part registered by its respective extension will be counted once and only once, even and especially when contributed by independent specifications that are in no mutual ancestry relation.

The linearity property is not respected by the languages that see "conflict" in independent method specifications as above; instead this linearity property replaces conflict with *cooperation*. Instead of distrust and negative sum games where developers have to fight over which extension will prevail, contributions from others extensions are dropped and must be reimplemented, there can be trust and positive sum games, where developers of each specification contribute their extension to the final result, and all these contributions combine harmoniously into the whole. This property also was not respected by the once "sender path" approach of Self (Chambers et al. 1991; Ungar and Smith 2007). This property would be expressible but not modular and hard to enforce in hypothetical languages that would require users to manually synthesize attributes from the inheritance DAG so as to extract semantics of methods. I am naming this property after the similar notion from "linear logic", wherein the preservation of computational resources corresponds to some operator being "linear" in the mathematical sense of linear algebra.

This property was the groundbreaking innovation of Flavors (Cannon 1979). Flavors' flavor of multiple inheritance, which includes many more innovations, was a vast improvement in paradigm over all its predecessors, and sadly, also over most of its successors.

Linearization: Consistency across Methods Any sequential effects from the ancestor's modular extension should be run in a consistent "Method Resolution Order"[6] across all methods of a given specification that may have such effects. This property, that extends and subsumes the previous two, implies that this order is a *linearization* of the inheritance DAG, i.e. a total ("linear") order that has the partial order of the DAG as a subset.[7] Since CommonLoops (Bobrow et al. 1986), it has been customary to call it the class (or object, for Prototype OO) *precedence list*, a term I will use.[8]

[6]The term and its abbreviation MRO were introduced by Python 2.3 circa 2003, and subsequently adopted by various popular languages including Perl 5.10 circa 2007.

[7]Note how the word "linear" means something very different in the two constraints "linearity" and "linearization": In the first case, the word comes from Linear Logic, and means conservation of information or other resources. In the second case, the word comes from Set Theory and Order Theory, and means a total order where any two elements are comparable, as opposed to a partial order where some elements are incomparable. Ultimately, the word "linear" in Linear Logic is inspired by Linear Algebra, that is connected to Order Theory via Boolean Algebras. And we'll see the two are related in that a way to combine arbitrary black box sequential computations by executing each of them once and only once (linearity) necessarily implies finding a total order (linearization) in which to compose them. Still the same word has very different meanings in the two contexts.

[8]The original Flavors paper just mentions that "the lattice structure is *flattened* into a linear one", and the original source code caches the list in a field called `FLAVOR-DEPENDS-ON-ALL`. The LOOPS manual talks of precedence but not yet of precedence list. The Simula manual has a "prefix sequence" but it only involves single inheritance.

Thanks to this property, methods that marshal ("serialize") and unmarshal ("deserialize") the fields of a class can follow matching orders and actually work together. Methods that acquire and release resources can do it correctly, and avoid deadlock when these resources include holding a mutual exclusion lock. Inconsistency can lead to resource leak, use-before-initialization, use-after-free, deadlock, data corruption, security vulnerability, and other catastrophic failures.

This property was also one of the major innovations of Flavors (Cannon 1979). As I will show, it implies that the semantics of multiple inheritance can be reduced to those of mixin inheritance (though mixin inheritance would only be formalized a decade later). It is the first of the constraints after which C3 (Barrett et al. 1996) is named. Inheritance order and linearity together imply linearization, especially since some methods involve sequential computations, and a uniform behavior is mandated over all methods.

Interestingly, all Class OO languages, even the "flavorless" ones, necessarily have some variant of this property: when they allocate field indexes and initialize instance fields, they too must walk the inheritance DAG in some total order preserving the linearity of slots, initialized in inheritance order. Unhappily, they do not expose this order to the user, and so pay the costs without providing the benefits.[9]

Now, a *valid* concern about linearization is that when two extensions ignore their super argument, and the system puts one in front of the other, the second and everything after the first one is actually ignored, and it might not be obvious which, and there probably should be at least some warning (Snyder 1986), if not an outright error. However, if that were actually a problem practically worth addressing, then you could have a solution similar to that of languages like Java or C++ that have you annotate some methods with a keyword `override` to signify that they modify a previous method; another option would be to have users annotate their methods with an opposite keyword `base` (or deduce it from the method body ignoring the `super` argument), and issue a warning or error if one inherits two different `base` definitions for a method, and tries to call the super method (either through an override, or through the lack thereof). One advantage of this approach is that it does not affect the runtime semantics of methods in Class OO, it only filters code at compile-time (though with prototypes, this "compile-time" might be interleaved with runtime). However, whether such a feature is worth it depends crucially on its costs and benefits. The benefits would be the ability to find and locate bugs that are not easily found and located by existing forms of testing and debugging, which is not obvious. The costs, which are obvious, are that it increases the cost of writing programs, forcing programmers to insert a lot of dummy methods that call or reimplement the "right" base method. Instead, with linearization, the issue of which method to prefer is perfectly under the control of the programmer, thanks to one cheap tool: the local order.

[9] A clever C++ programmer might recover the linearization implicit in object initialization by having all classes in his code base follow the design pattern of constructors computing the effective methods for the class as Flavors would do. Unhappily, "static" member initialization does not rely on linearization, only instance member initialization does; thus object constructors would have to do it the first time an object of the class is instantiated; but the test for this first time would slow down every instantiated a little bit, which defeats the "need for speed" that often motivates the choice of C++. Also, since this design pattern requires active programmer cooperation, it will not work well when extending classes from existing libraries, though this can be worked around in ugly ways if those classes didn't keep crucial functionality "private".

Local Order: Consistency with User-Provided Order The "local (precedence) order" in which users list parents in each specification must be respected: if a parent appears before another in the list of parents local to some specification, then the two will appear in the same relative order (though not necessarily consecutively) in the precedence list.

This property enables users to control the precedence list, and to specify ordering dependencies or tie-breaks that the system might not otherwise detect or choose, including but not limited to compatibility with other systems or previous versions of the code. If users really want to relax ordering dependencies, they can introduce intermediate shim specifications with pass-thru behavior, so that the ordering constraint only concerns the irrelevant shims, while the actual parents are not constrained. This is burdensome, though, and users may prefer to simply adjust the local order of their parents to whichever global order of specifications is mandated by constraints from other parts of the code, despite a very slight decrease in modularity when the ordering is partly an arbitrary choice heuristically made by the linearization algorithm.

This property was first used in New Flavors (Moon 1986), that calls it "local ordering". CommonLoops (Bobrow et al. 1986) adopted it as "local precedence", "local ordering", and "local precedence list". CLOS (Bobrow et al. 1988; Steele 1990) adopts it as "local precedence order". Ducournau et al. speak of "local ordering" or "local precedence order" (Ducournau et al. 1994; Ducournau et al. 1992). C3 says "local precedence order". It is the second of the three eponymous constraints of C3 (Barrett et al. 1996). Among popular "flavorful" languages, Python, Perl, Lisp and Solidity notably respect this constraint, but Ruby and Scala fail to.

Monotonicity: Consistency across Ancestry The "method resolution order" for a child specification should be consistent with the orders from each of its parents: if the precedence list for a parent places one extension before another, it will keep doing so in every child.

This property allows extensions to partake in the same protocols as the specifications being extended. Indeed, lack of this consistency property when the order of the extensions drives the acquisition and release of resources including but not limited to heap space, locks, file descriptors, stack space, time slots, network bandwidth, etc., can cause memory leaks, deadlocks, kernel space leak, memory corruption, or security vulnerabilities instead of deadlocks. By contrast, with this consistency property, developers may not even have to care what kind of resources their parents may be allocating, if any, much less in what order.

This property was first described (Ducournau et al. 1992) then implemented (Ducournau et al. 1994) by Ducournau & al., and is the third of the three constraints after which C3 is named (Barrett et al. 1996). Among popular "flavorful" languages, Python, Perl and Solidity respect this constraint, but Ruby, Scala and Lisp fail to. (Though at least in Common Lisp you can use metaclasses to fix the issue in your code.)

Shape Determinism: Consistency across Equivalent Ancestries Two specifications with equivalent inheritance DAGs (with an isomorphism between them, bijection preserving partial order both ways) will yield equivalent precedence lists, up to the

same isomorphism. Renaming methods or specifications, moving code around, fixing typos, updating method bodies, adding or removing methods, changing filenames and line numbers, etc., will not change the precedence list.

This property enables users to predict the "method resolution order" for a specification, based on the "shape" of its inheritance DAG alone. Unrelated changes to the code will not cause a change in the precedence list, thereby potentially triggering bugs or incompatibilities between code versions. This property can also be seen as generalizing linearization, in that linearization guarantees the same precedence list for all methods within a given closed specification, whereas shape determinism guarantees the same precedence list for all open specifications with equivalent inheritance DAG, which subsumes the previous case, since the methods of a class or prototype are "just" open specifications that have been assembled together into a closed one, with a shared ancestry. Thanks to Shape Determinism, changes made while debugging won't suddenly hide bad behavior, and changes made while refactoring or adding features won't introduce unrelated bad or unexpected behavior.

This property was first described (Ducournau et al. 1992) under the nondescript name "acceptability". It received little attention, maybe because most (all?) popular OO systems already respect it implicitly. The C3 algorithm respects it, but not enough to name it and count it among the constraints it purports to implement (Barrett et al. 1996).[10]

As an alternative to Shape Determinism, you could establish a global ordering all defined classes across a program, e.g. lexicographically by their name or full path, or by assigning a number in some traversal order, or from a hash of their names or definitions, etc. This ordering could then be used by a linearization algorithm as a tie-breaking heuristic to choose which superclass to pick next while computing a precedence list, whenever the constraints otherwise allow multiple solutions. But the instability of such a heuristic when the code changes would lead to many *heisenbugs*.

7.3.6 Computing the Precedence List

Consider a function `compute-precedence-list` that takes a specification featuring multiple inheritance (and possibly more features) and returns a list of specifications, that is a linearization of the specification's ancestry DAG, as per the linearization property above. Further assume that the above returned precedence-list starts with specification itself, followed by its ancestors from most specific to most generic (left to right). This is the convention established both by Flavors' "class precedence list" and, maybe surprisingly, also by Simula's "prefix sequence", though in the case of Simula this convention is contravariant with the order in which the bodies of the "prefix classes" are concatenated into the effective class definition. Most (all?) OO systems seem to have adopted this convention.

Now, the precedence list can be computed simply by walking the DAG depth-first, left-to-right. The original Flavors used a variant of such an algorithm,

[10]There are thus effectively four constraints enforced by C3, just like there are effectively four musketeers as main protagonists in The Three Musketeers (Dumas 1844).

and Ruby still does to this day. Unhappily, this approach fails at respecting either Local Order or Monotonicity.

Another approach is to consider the precedence list a synthesized attribute, and compute a child's precedence list from those of its parents. That's the only reasonable way to ensure monotonicity. However, the naive way to do it, by concatenating the lists then removing duplicates, like LOOPS (Bobrow and Stefik 1983) or after it (though removing from the other end) Scala (Odersky and Zenger 2005), preserves neither Local Order nor Monotonicity. The somewhat more careful algorithm used by CommonLoops (Bobrow et al. 1986) and after it by CLOS (with minor changes) preserves Local Order, but not monotonicity. The slightly complex algorithm by Ducournau et al. (Ducournau et al. 1994), and the latter somewhat simpler C3 algorithm (Barrett et al. 1996; Wikipedia 2021), synthesize the precedence list while preserving all desired properties. C3 was notably adopted by OpenDylan, Python, Raku (Perl), Parrot, Solidity, PGF/TikZ.

I provide in section 7.4.4 below an informal description of my extension to the C3 algorithm, and, in appendix, the complete code.

7.3.7 Mixin Inheritance plus Precedence List

How then can one use this precedence list to extract and instantiate a modular definition from the modular extensions of a specification and its ancestors? By extracting the list of these modular extensions in that order, and composing them as per mixin inheritance:

```
compute-precedence-list : MISpec ? ? ? → DependentList ? (MISpec ?
? ?)
effectiveModExt : MISpec r i p → ModExt r i p
fixMISpec : top → MISpec p top p → p

(define effectiveModExt (λ (mispec)
  (foldr mix idModExt (map getModExt (compute-precedence-list mis-
pec)))))
(define fixMISpec (λ (top) (λ (mispec)
  (fix top (effectiveModExt mispec)))))
```

The `map` function is the standard Scheme function to map a function over a list. The `foldr` function is the standard Scheme function to fold a list with a function (also known as *reduce* in Lisp and many languages after it). The type parameters to `MISpec` in `compute-precedence-list` were left as wildcards above: the precise dependent type, involving existentials for `l pr pi pp` such that every specification in the list can be composed with the reduced composition of the specifications to its right, is left as an exercise to the reader.

I have thus reduced the semantics of multiple inheritance to mixin inheritance (or, in this case equivalently, single inheritance) by way of computing a precedence list.

Complete implementations of prototypes using multiple inheritance in a few tens of lines of code are given in my previous paper using Scheme (Rideau et al. 2021), or in a proof of concept in Nix (Rideau 2021). My production-quality implementation in

Gerbil Scheme (Rideau 2020) including many features and optimizations fits in about a thousand lines of code.

7.3.8 Notes on Types for Multiple Inheritance

As usual, `effectiveModExt` works on open specifications, whereas `fixMISpec` only works on closed specifications. The present formalization's ability to deal with open specification and not just closed ones crucially enables finer granularity for modular code.

Now, note how multiple inheritance relies on subtyping of specifications in a way that single inheritance and mixin inheritance don't: In those simpler variants of inheritance, the programmer controls precisely what are the next modular extensions to be composed with, and so does not need to rely on subtyping; indeed, I showed when introducing wrappers for conflation that sometimes one really wants to use modular extensions that do not follow the usual subtyping constraints (in that case in section 6.1.3, `qproto-wrapper` that wraps the value into a pair). By contrast, with multiple inheritance, a specification only controls the relative order of its ancestors in the precedence list that will be composed, but its modular extension must remain correct when used as part of a descendant specification, in which case other modular extensions may be interleaved with its ancestors as part of a larger precedence list. That is why a multiple inheritance specification's modular extension must always be a strict modular extension (though there can be non-strict wrapper extensions around the precedence list), whereas single inheritance and mixin inheritance can use any kind of modular extension.

Sadly, multiple inheritance often remains unjustly overlooked, summarily dismissed, or left as an exercise to the reader in books that discuss the formalization of programming languages in general and/or OO in particular (Abadi and Cardelli 1996a; Friedman et al. 2008; Khrisnamurthi 2008; Pierce 2002). The wider academic literature is also lacking in proper treatment of types for multiple inheritance, with some notable exceptions like (Allen et al. 2011).

Much of the focus of the literature is on subtyping, with a deemphasis or outright avoidance of fixpoints and self-recursion, leading many authors to confuse subtyping of specification and target. Subtyping is then often studied in the context of single inheritance, which is ironic since subtyping isn't quite as important without multiple inheritance.

More generally, computer science researchers seem largely uninterested in the nature of modularity or extensibility, at best assuming they are purely technical aspects of a language with a fixed formal expression, or else someone else's problem; they have no consideration for how programming language features do or do not affect the social dynamics of interactions between programmers, how much coordination they require or eschew across development teams, or within one programmer's mind. Consequently, they lack any criterion for modularity, and how to compare no inheritance, single inheritance, mixin inheritance and multiple inheritance. Finally, a lot of language designers, industrial or academic, invent some primitives that embodies all the features of a small model of OO; they fail to enlighten in any way by introducing their own ad hoc logic, and still crumble under the complexity of the features they combined

despite being way short of what an advanced OO system can provide. Meanwhile, truly groundbreaking work, such as Flavors, is routinely rejected as obscure, left uncited, or is only cited to quickly dismiss it with no attempt to take its contents seriously.

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.

7.3.9 Comparing Multiple- to Mixin- Inheritance

Since all variants of OO can be expressed simply as first-class concepts in FP, comparing the variants of OO actually requires assuming second-class OO, with a compile-time language significantly weaker than the λ-calculus. The comparison can still inform us about first-class OO in that it tells us how much one can enjoy OO as if it were second-class, versus how much one has to painfully escape the illusion, when using this or that variants. The expressiveness comparison informs us about what some variants automate that has to be done manually under other variants. The modularity comparison informs us about what requires synchronization with other programmers under some variants but not others. Together, they also tell us that some features could be automated in theory yet cannot be so in practice due to lack of synchronization between programmers, due to lack of a better OO variant.

Multiple Inheritance is less Simple than Mixin Inheritance Obviously, multiple inheritance requires the system to implement some extra logic either to handle proper ancestor linearization, or to implement some really bad method resolution algorithm instead. This makes multiple inheritance more complex to implement, and more complex to explain especially if you don't do it properly.

But is that complexity intrinsic to the problem it is solving, or is it extrinsic? In other words, does multiple inheritance solve a problem you would have to face anyway with mixin inheritance, or does it introduce concepts you do not need? And assuming it is a problem you face anyway, does it solve it in the simplest manner?

Multiple Inheritance is as Expressive as Mixin Inheritance Mixin inheritance is clearly not less expressive as multiple inheritance, since every entity that can be written using multiple inheritance can just as well be written using mixin inheritance, by computing the precedence list manually.

Multiple inheritance is as expressive as mixin inheritance, if you restrict yourself to the subset of mixin inheritance where you don't repeat any modular extension in a list you compose: define one multiple inheritance specification without parents per modular extension, and one specification with the list of the previous as parents to combine them. If for some reason you do want to repeat a modular extension, then you may have to duplicate the repeated definitions, though you may factor most code out of the definition into a helper (part of the specification containing the modular extension), and only have duplicate calls to the helper. Strictly speaking, this is still slightly less expressive than mixin inheritance, but this case never seems to happen

in the wild.[11] Also, this slight decrease in expressiveness, if any, does not impact modularity, since the same module that exported a modular extension to use multiple times, would instead export a multiple inheritance specification to use once, that defines a helper that it possibly calls once, but can be thereafter called many times. Therefore I can say that multiple inheritance is as expressive as mixin inheritance in practice.

Multiple Inheritance is more Modular than Mixin Inheritance I will now compare the two variants of inheritance from the point of view of modularity. Multiple inheritance requires somewhat more sophistication than mixin inheritance, adding the cognitive burden of a few more concepts, which at first glance can be seen as detrimental to modularity. Yet, I will argue that inasmuch as modularity matters, these concepts are already relevant, and that multiple inheritance only internalizes modularity issues that would otherwise still be relevant if left as external concepts. Moreover, multiple inheritance automates away a crucial cross-module task that mixin inheritance requires users to handle manually, thereby reducing modularity.

Thus, consider the issue of dependencies between modular extensions. I showed that in practice, the common modular extension `method-spec` depends on `record-spec`, while part specifications in my notional example depend on `base-bill-of-parts`. More generally, a specification may depend on a method having been implemented in an ancestor so that its inherited value may be modified in a wrapper (in this case, the "database" of parts), or, in some languages, just declared so it may be used (though with a proper type system this might not have to be in a parent). Dependency constraints between modular extensions are an ubiquitous phenomenon in mixin inheritance, and these constraints exist even when they are not represented within the language as internal notions of "parent" and "ancestor".

Some clever chaps might suggest to pre-compose each modular extension with all its dependencies, such that when modular extension `B1` depends on `A`, you'd export `B1precomposed = (mix B1 A)` instead of `B1`, and that's what your users would use. Unhappily, that means that if another module `B2` also depends on `A` and exports `B2precomposed = (mix B2 A)`, then users who want to define a class `C` that uses both `B1` and `B2`, will experience the very same diamond problem as when trying to synthesize a modular definition from an attribute grammar view of of multiple inheritance in section 7.3.4: the pre-composed dependencies (`A` in this case) would be duplicated in the mix of `(mix B1precomposed B2precomposed) = (mix (mix B1 A) (mix B2 A))`; these copies would badly interfere, in addition to leading to an exponential resource explosion as you keep pre-composing deeper graphs. Therefore, pre-composing modular extensions is the same non-solution that led to the "conflict" view of multiple inheritance, based on the naive conceptualization of how to generalize single inheritance. Indeed, precomposed modular extensions are essentially the same as modular definitions.

In the end, composing modular extensions is subject to dependency constraints. And these dependency constraints only get more complex and subtle if you want your linearization to respect not just the inheritance order, but also the local precedence or-

[11] If you ever see this repeated usage pattern appear in useful code, or if you have a good argument why it is never actually useful, you should definitely publish a paper about it, and send me a copy.

der, monotonicity of precedence lists. Yet, automatically or manually, the constraints *will* be enforced in the end, or programs *will* fail to run correctly. Multiple inheritance enforces these constraints intra-linguistically, and requires no further communication between programmers. Mixin inheritance requires the programmer to enforce them extra-linguistically, and thus care about, and communicate about, which modular extension depends on which, in which order, including subtler details of local order and monotonicity if their specifications ever drive order-dependent resource management such as with locking. In other words, with multiple inheritance, a specification's dependencies are part of its implementation, but ***with Mixin Inheritance, a specification's dependencies become part of its interface***.

In practice, that means that with mixin inheritance, programmers must not just document the current direct dependencies of their specifications; they must keep it up to date with every indirect dependencies from libraries they transitively depend on. And when a library makes a change to the dependencies of one of its specification, then every single library or program that directly or indirectly depends on that specification, must be suitably updated. Sensitivity to change in transitive dependencies more generally means much *tighter coupling* between the versions of the many software libraries, and fragility of the entire ecosystem, as incompatibilities ripple out, and it becomes hard to find matching sets of libraries that have all the features one needs. Tight coupling is the antithesis of modularity.[12]

One thing that *could* actually help deal with dependencies without multiple inheritance would be a rich enough strong static type system such that the `r`, `i` and `p` parameters (for required, inherited and provided) of my parameterized type `ModExt r i p` can identify whether modular extensions are composed in ways that make sense. This strategy can indeed greatly help in dealing with dependencies of modular extensions. However, it does not fully solve the problem: yes it helps users *check* that their manual solutions are a valid ordering that will eliminate certain classes of runtime errors; and yes it helps struggling users search for such a valid solution faster than random attempts unguided by types, especially if error messages can pinpoint what elements fail to be provided, or are provided in ways incompatible with what other extensions expect. But the user still has to come up with the solution manually to begin with; and the user still has to enforce all the consistency constraints that his application needs

[12] If you want to make the change easy on your transitive users, you may have write and send patches to lots of different libraries and programs that depend on your software. This is actually the kind of activity I engaged in for years, as maintainer of Common Lisp's build system ASDF. This was greatly facilitated by the existence of Quicklisp, a database of all free software repositories using ASDF (thousands of them), and of `cl-test-grid`, a program to automatically test all those repositories, as well as by the fact that most (but by no means all) of these software repositories were on github or similar git hosting webserver. Making a breaking change in ASDF was painful enough as it is, having to slightly fix up to tens of libraries each time, but was overall affordable. If every refactoring of the class hierarchy within ASDF, of which there were several, had broken every repository that uses ASDF, due to every user having to update their precedence list, then these changes might have been one or two orders of magnitude more expensive. This would have been prohibitive, whether the cost is born solely by the maintainer, or distributed over the entire community. By contrast, my experience with the OCaml and Haskell ecosystems is that their strong static types without lenient subtyping creates very tight coupling between specific versions of libraries, with costly rippling of changes. What results is then "DLL hell", as it becomes hard to find coherent sets of inter-compatible libraries that cover all the needs of your program, and to keep them up-to-date when one library requires a bug fix, even though there might be a library to cover each of your needs.

between the solutions he comes up with for each and every specification; and the user still has to synchronize with authors of related other packages so they too do all those tasks correctly, and fight those who don't see the point because they don't personally need the same level of consistency.

Interestingly, single inheritance doesn't have the above modularity issue of mixin inheritance, since every specification already comes with all its ancestors, so that users of a specification don't have to worry about changes in its dependencies. However, single inheritance avoids this issue only by eschewing all the benefits of mixin and multiple inheritance. The above issue is only an obstacle to kinds of modularity that single inheritance can never provide to begin with, and not to uses of OO wherein programmers export specifications that should only be used as the pre-composed rightmost base to further extensions. Therefore not having this modularity issue is actually a symptom of single inheritance being less modular rather than more modular than mixin inheritance.

All in all, ***Multiple Inheritance is more modular than Mixin Inheritance***, that is more modular than single inheritance, that is more modular than no inheritance; the modularity issues one experiences with one kind of inheritance are still superior to the lack of modularity issues one experiences with the kinds of inheritance lacking the modularity about which to have issues to begin with.

Multiple Inheritance is slightly less Performant than Mixin Inheritance Compared to mixin inheritance, multiple inheritance involves this extra step of computing a specification's precedence list. The linearization algorithm has a worst case complexity O(dn), where d is the number of parents and n the total number of ancestors. Yet, in practice the extra step is run at compile-time for second-class OO, and does not affect runtime; moreover, it is often quite fast in practice because most OO hierarchies are shallow.[13]

Multiple inheritance otherwise involves the same runtime performance issues as mixin inheritance compared to single inheritance (see section 7.2.2): in general, method or field access requires a hash-table lookup instead of a fixed-offset array lookup, which is typically 10-100 times slower.

Now, a lot of work has been done to improve the performance of multiple inheritance, through static method resolution when possible, and otherwise through caching (Bobrow et al. 1986). But these improvements introduce complexity, and caching increases memory pressure and still incurs a small runtime overhead even when success-

[13] A study of Java projects on GitHub (Prykhodko et al. 2021) found that the vast majority of classes had fewer than 5 ancestors, including the base class `Object`. But that is a language with single inheritance. A survey I ran on all Common Lisp classes defined by all projects in Quicklisp 2025-06-22 (minus a few that couldn't be loaded with the rest) using `ql-test/did` shows that the 82% of the 18939 classes have only 1 parent, 99% have 3 or fewer, the most has 61, `MNAS-GRAPH::<GRAPH-ATTRIBUTES>`; 90% have 5 or fewer non-trivial ancestors, 99% have 16 or fewer, the most has 63, `DREI:DREI-GADGET-PANE`. (By non-trivial, I mean that I count neither the class itself, nor the 3 system classes shared by all user-defined classes.) As for the 2841 structs that use single inheritance (and not usually to define many methods, just for data fields), 63% had no non-trivial ancestor, 86% has 0 or 1, 96% had 2 or fewer, 99% had 3 or fewer, 99.9% had 4 or fewer, 1 had 5, 1 had 6, `ORG.SHIRAKUMO.BMP::BITMAPV5INFOHEADER`. As can be seen, multiple inheritance leads to a very different style of programming, with more done in "traits" or "mixins", than when only single inheritance is available.

ful at avoiding the full cost of the general case, while not eliminating the much slower behavior in case of cache miss. For all these reasons, many performance-conscious programmers prefer to use or implement single inheritance when offered the choice.

7.4 Optimal Inheritance: Single and Multiple Inheritance Together

7.4.1 State of the Art in Mixing Single and Multiple Inheritance

With all these variants of inheritance and their tradeoffs naturally comes a question: Is there some optimal inheritance, that is equally or more expressive, modular and performant than any other variant? It would have to be at least as expressive as mixin inheritance and multiple inheritance, as modular as multiple inheritance, and as performant as single inheritance.

Now, as far back as 1979, Lisp offered both single inheritance with its `structs`, and multiple inheritance with its `classes` (nées flavors). Since 1988 or so, the Common Lisp Object System (a.k.a. CLOS) (Bobrow et al. 1988; Steele 1990) even offered a way to interface uniformly with either structs or classes, using generic functions and metaclasses. Programmers could develop software with the flexibility of classes, then when their design was stable, declare their classes to be structs underneath, for an extra boost in performance. However, CLOS has this limitation, that structs and classes constitute disjoint hierarchies: a class cannot extend a struct, and a struct cannot extend a class. Thus, before you can declare a class to actually be a struct underneath, you must make sure to eliminate any trace of multiple inheritance in all its ancestry, both the classes that it extends, and those that extend it, and declare them as structs, too, thereby forfeiting use of multiple inheritance anywhere in that hierarchy, and losing any modularity benefit you might have enjoyed.[14]

Since then, Ruby (1995) and Scala 2 (2004) have done better, wherein "single inheritance" structs and "multiple inheritance" classes can extend each other—except that Ruby calls these entities respectively "classes" and "modules" whereas Scala, following the Smalltalk tradition, calls them respectively "classes" and "traits" (Odersky and Zenger 2005).[15] Ruby and Scala combine the two forms of inheritance in ways

[14] Another limitation of structs and classes in Common Lisp is that for historical reasons, the default syntax to define and use structs is very different (and much simpler) from the CLOS syntax to use and define objects. You can use the explicitly use the CLOS syntax to define structs by specifying an appropriate metaclass `structure-class` as opposed to `standard-class` for the standard objects of CLOS; however, the resulting syntax is more burdensome than either plain struct or plain CLOS syntax. This syntax discrepancy creates another barrier to refactoring of code between structs and classes. Yet this syntactic barrier remains minor compared to the semantic barrier of having to forfeit multiple inheritance in an entire class hierarchy.

Now, one could also conceivably use Common Lisp metaclasses (Kiczales et al. 1991) to re-create arbitrary user-defined inheritance mechanisms, including my Optimal Inheritance below. The semantics of it would be relatively easy to re-create. However, it might still be hard to do it in a way that the underlying implementation actually benefits from the optimization opportunities of single inheritance, at least not in a portable way.

[15] It is bad enough that "class" denotes entities with multiple inheritance in Lisp or Python, but specifically entities with single inheritance in Smalltalk, Ruby or Scala. It doesn't help that the Scala documentation is not consistent about that naming, and uses the word "class" ambiguously, sometimes to mean suffix specifi-

broadly similar to each other and to the optimal inheritance that I propose below (Ruby did so about 10 years earlier than Scala 2, but without an academic publication to cite, though also without static types). However, Ruby uses a variant of the Flavors algorithm (depth first traversal), and Scala a variant of the LOOPS algorithm (concatenation of precedence lists then removal of duplicates), and neither respects Local Order nor Monotonicity, making them less modular than they could be, and sub-optimal.[16]

Note that Scala 2 further requires the user to explicitly merge the "suffixes" by hand, and include the most specific suffix ancestor as the semantically last parent of a specification;[17] Scala 3 by contrast relaxes this restriction, and, like Ruby, will automatically merge the "suffixes" and identify the most specific suffix ancestor (or issue an error if there is an incompatibility in suffixes).[18]

7.4.2 The Key to Single Inheritance Performance

In this section, I will use the respective words "struct" or "class" as per the Lisp tradition, to denote specifications that respectively do or do not abide by the constraints of single inheritance (with according performance benefits). My discussion trivially generalizes beyond specifications for type descriptors conflated with their targets to any specification; only the inheritance structure of my specifications matters to this discussion.

As seen in section 7.2.2, what enables the optimizations of single inheritance is that the indexes to the fields and methods of a specification's target are also the indexes

---

cations only, sometimes to mean "class-or-trait", specifications both infix and suffix. To further confuse terminology, a C++ `struct` is just a regular class, that always supports (flavorless) multiple inheritance—there are no "single inheritance entities" in C++; the only difference is that classes defined with the `struct` keyword have all their members public by default rather than private as with the `class` keyword, which makes the `struct` statement backward compatible with the equivalent statement in C. Thus there is a "struct/class" distinction in C++, but as a minor syntactic feature that has nothing to do with either single or multiple inheritance. And to make things even more confusing, "multiple inheritance" in C++ is not what it is in Ruby, Scala, Lisp, Python and other flavorful languages; instead it's a mix of flavorlesss "conflict" inheritance (for "virtual" classes), and weird path-renamed duplicated inheritance à la CommonObjects (for the non "virtual") that tries hard to fit the square peg of a multiple inheritance DAG into the round hole of a tree. Anyway, the conclusion here once again is that the word "class" is utterly useless at conveying precise connotations about inheritance outside the context of a specific language.

[16] Interestingly, and although this change and its rationale are not explained in Scala documentation or papers, Scala removes duplicates from the beginning of the concatenate list when LOOPS removes them from the end. This change makes the algorithm preserve the longest suffix possible, which crucially matters for the Scala "single inheritance" fragment; LOOPS instead preserves the longest prefix possible, which serves no purpose, and sacrifices the suffix, when preserving the suffix could have helped with opportunistic optimizations even in the absence of suffix guarantees. Note that Ruby and my C4 algorithm also preserve the longest suffix possible, which matters for the same reasons.

[17] I say semantically last, as Scala, per its documentation, keeps precedence lists in the usual most-specific-first order. However, syntactically, Scala requires users to specify parents in the opposite most-specific-last order, so your suffix parent (a "class" in Scala) must be syntactically specified *first* in Scala 2. As an exception, the most specific suffix ancestor need not be explicitly specified if it is the top class `Object`.

[18] I was unable to find any trace anywhere in the Scala 3.3 documentation of this slight change in syntax and semantics, its precise behavior, design rationale, and implementation timeline; and the Scala team declined to answer my inquiries to this regard. Nevertheless, this is clearly an improvement, that makes Scala 3 as easy to use as Ruby or Gerbil Scheme in this regard: by comparison, Scala 2 was being less modular, in requiring users to do extra work and make the "most specific class ancestor" a part of a trait's interface, rather than only of its implementation.

of the same fields and methods in the targets of its extensions. These indexes are computed by walking the specification's ancestry from least specific to most specific ancestor. In the terminology of multiple inheritance, this is a walk along the reverse of the precedence list. And for the walks to yield the same results, after putting those lists in the usual order, can be stated as the following property, that I will call the *suffix property*: ***the precedence list of a struct is a suffix of that of every descendant of it***.

Now this is semantic constraint, not a syntactic one, and it can very well be expressed and enforced in a system that has multiple inheritance. Thus it turns out that indeed, a struct can inherit from a class, and a class from a struct, as long as this property holds: the optimizations of single inheritance are still valid, even though structs partake in multiple inheritance! Interestingly, the ancestry of a struct then is not a linear (total) order anymore: a few classes may be interspersed between two structs in the precedence list. However, the subset of this ancestry restricted to structs, is a linear (total) order, as every struct in a given specification's ancestry is either ancestor or descendant of every other struct in that same ancestry. Thus structs are still in single inheritance with respect to each other, even though they are part of multiple inheritance in their relationship with classes.

Since the suffix property is the thing that matters, I will name "suffix" the generalization of structs (in Lisp lingo, or classes in Smalltalk lingo) from classes (here meaning prototypes for type descriptor) to prototypes (for any target) and arbitrary specifications (without conflation). By contrast I will call "infix" the specifications that are explicitly not declared as suffix by the programmer, and just say specification (or prototype if conflated with target, or class if furthermore the target is a type descriptor) when it can be either infix or suffix. Thus, in Lisp lingo, a "struct" is a suffix specification, a "class" is a specification and a "mixin" is an infix specification. In Smalltalk lingo, a "class" is a suffix specification, and a "trait" is an infix specification.[19]

7.4.3 Best of Both Worlds

An inheritance system that integrates multiple inheritance and single inheritance in the same hierarchy using the suffix property, yet also respects the consistency constraints of multiple inheritance, can best existing inheritance systems. Suffix specifications (such as Lisp structs) can inherit from infix specifications (such as Lisp classes), and vice versa, in an expressive and modular multiple inheritance hierarchy DAG. Yet suf-

[19]Interestingly, my "suffix" is the same as the "prefix" of Simula. Simula calls "prefix" a superclass, precisely because its single inherited behavior comes before that of the class in left to right evaluation order of its code concatenation semantics. But in multiple inheritance, modular extensions are composed right-to-left, and in a tradition that goes back to Flavors (and maybe beyond), the precedence list is also kept in that order. And so my "suffix" actually means the same as Simula's "prefix". Now, since Simula only has single inheritance, all its classes are "prefix" (i.e. my "suffix"). by contrast, in a multiple inheritance system, regular classes are infix, and their precedence list, while an ordered sublist of an descendant's precedence list, is not necessarily at the end of it, and is not necessarily contiguously embedded. It can also be confusing that Simula calls "prefix sequence" the list of superclasses that it keeps in the same order as the precedence list of Flavors and its successors, from most specific to least specific, which is opposite to the order of "prefixing". Finally, a "final" class in Java or C++ could be called "prefix" by symmetry, because its precedence list is the prefix of that of any transitive subclass (of which there is only one, itself); but that would be only introduce more terminological confusion, without bringing any useful insight, for this prefix property, while true, is not actionable. It is better to leave suffix and prefix as twisted synonyms.

fix specifications being guaranteed that their precedence list is the suffix of any descendant's precedence list, methods and fields from these specifications will enjoy the performance of single inheritance; fast code using fixed-offset access to these can be shared across all descendant specifications. I call that *Optimal Inheritance*.[20]

Note that while suffix specifications with respect to each other are in a "single inheritance" hierarchy as guaranteed by the suffix property, being in such a hierarchy is not enough to guarantee the suffix property; and suffix specifications are still in a "multiple inheritance" hierarchy with other specifications. Thus when "combining single and multiple inheritance", it is not exactly "single inheritance" that is preserved and combined, but the more important struct suffix property. The crucial conceptual shift was to move away from the syntactic constraint on building a class, and instead focus on the semantic constraint on the invariants for descendants to respect, that themselves enable various optimizations. It may be a surprising conclusion to the debate between proponents of multiple inheritance and of single inheritance that in the end, single inheritance did matter in a way, but it was not exactly single inheritance as such that mattered, rather it was the suffix property implicit in single inheritance. The debate was not framed properly, and a suitable reframing solves the problem hopefully to everyone's satisfaction.

In 2024, Gerbil Scheme (Vyzovitis 2016) similarly modernized its object system by unifying its single inheritance and multiple inheritance hierarchies so its "struct"s and "class"es (named in the Lisp tradition) may extend each other. The result ended up largely equivalent to the classes and modules or traits of Ruby or Scala, except that Gerbil Scheme respects all the consistency constraints of the C3 algorithm, that it further extends to support suffixes, in what I call the C4 algorithm. It is the first and so far only implementation of *optimal inheritance*.[21]

7.4.4 C4, or C3 Extended

The authors of C3 (Barrett et al. 1996; Wikipedia 2021), after Ducournau et al. (Ducournau et al. 1994; Ducournau et al. 1992), crucially frame the problem of ancestry linearization in terms of constraints between the precedence list of a specification and those of its ancestors: notably, the "monotonicity" constraint states that the precedence list of an ancestor must be an ordered subset of that of the specification, though its elements need not be consecutive. I define my own algorithm C4 as an extension of C3, that in addition to the constraints of C3, also respects the *suffix property* for specifications that are declared as suffixes. This means that an optimal inheritance specification, or `OISpec` by extending the `MISpec` of multiple inheritance with a new field `suffix?` of type `Boolean`, that tells whether or not the specification requires all

[20] Optionally, the specification DAG is made slightly simpler with the empty specification, declared suffix, as the implicit ancestor of all specifications, and the empty record specification, declared suffix, as the implicit ancestor of all record specifications. But this only actually helps if the system allows for the after-the-fact definition of multimethods, "protocols" or "typeclasses" on arbitrary such specification.

[21] Interestingly, Ruby, Scala and Gerbil Scheme seem to each have independently reinvented variants of the same general design based on the suffix property. This is a good symptom that this is a universally important design. However, only with Gerbil Scheme was the suffix property formalized and was the C3 algorithm used.

its descendants to have its precedence list as a suffix of theirs. (In a practical implementation, you could add more flags, for instance to determine if the specification is sealed, i.e. allows no further extensions.)

I give a complete Scheme implementation of C4 in the appendix, but informally, the algorithm is as follows, where the steps tagged with (C4) are those added to the C3 algorithm (remove them for plain C3):

1. **Extract parent precedence lists**: Get the precedence lists of each parent, preserving the declared Local Order.

2. **Split step**: Split each precedence list into:

   - **Prefix**: a prefix precedence list containing only infix specifications, up to (but not including) the most specific suffix specification (if any), and
   - **Suffix**: a suffix precedence list from that most specific suffix specification to the end (the empty list if no such suffix specification).

   If using singly linked lists, keep your prefixes in reverse order for later convenience. (In plain C3: everything is in the prefixes; the suffixes are empty.)

3. **Suffix merge step**: Merge all suffix lists into a single merged suffix list. The suffix property requires these lists to be in total order: given any two suffix lists, one must be a suffix of the other. If not, raise an incompatibility error. The merged suffix is the longest of all suffix lists.[22]

4. **Local Order support step**: add the local order (list of parents) to the end of the list of prefixes; (keep it in the same order as those prefix lists, reversed if need be;) call the elements of those lists candidates, the list candidate lists, and this list the candidate list list.

5. **Cleanup Step**:

   - Build a hash-table mapping elements of the merged suffix list to their distance from the tail of the list.
   - For each candidate list, remove redundant suffix specifications from its end:
     - Start from the tail of the candidate list.[23]
     - if the list is empty, discard it and go to the next list.
     - otherwise, consider the next element, and look it up in the above hash-table;

[22] Note that instead of working on the lists, you can work only on their heads, that are "suffix specifications", and skip over infix specifications. To speed things up, each specification could also have a field to remember its "most specific suffix ancestor". And for a space-time tradeoff, you can also keep the "suffix ancestry" of these suffix specifications in a vector you remember instead of just remembering the most specific one, allowing for O(1) checks for subtyping among suffix specifications.

[23] You may already have gotten that list in reverse order at the previous step; or you can reverse it now; or if you keep your precedence lists in arrays, just iterate from the end.

- if it was present in the merged suffix list, with an increased distance from its tail, then remove it from the candidate list and go to the next element;
- if it was present in the merged suffix list, but the distance from its tail decreased, throw an error, the precedence lists are not compatible;
- if it was not present in the merged suffix list, then keep it and the rest of the elements, now in most specific to least specific order (again, if the candidate lists were reversed), and process the next candidate list;

6. **C3 merge on cleaned prefixes**:

   - Create a hash-table for ancestor counts, mapping ancestors to integers, initially 0.
   - For each ancestor, count the number of times it appears in the merged suffix list and in the candidate lists; however, do not count the head element of each candidate list.
   - Repeatedly, and until all the lists are empty, identify the next winning candidate in the candidate list list:
     - The winning candidate is the first head of a candidate list (in order) that has a count of zero in the ancestor count table.
     - If no candidate won, throw an error.
     - If one candidate won, add it to the tail of the merged prefix. Then, for each candidate list of which it was the head:
       * pop it off the candidate list;
       * if the rest of the candidate list is empty, remove it from the candidate lists;
       * otherwise the candidate list is not empty, promote its next element as head, and decrement its count of the new head in the ancestor count table.

7. **Join Step**: Append the merged prefix and the merged suffix.

8. **Return Step**: Return the resulting list; also the most specific suffix.

Note that the C3 algorithm as published by (Barrett et al. 1996), has complexity $O(d^2n^2)$ where d is the number of parents, n of ancestors, because of the naive way it does a linear membership scan in the tails of the lists for each parent.[24] But by maintaining a single hash-table of counts on all tails, I can keep the complexity O(dn),

[24] A worst-case example is, two parameters d and n, to find a linear order of size n, more or less evenly divided in d segments of size n/d, and for each segment head S_i, define a specification T_i that S_i as its single parent; Make the local precedence order the T_i sorted from shortest to longest ancestry. The T_i serve to defeat the local order property in allowing the S_i to be put on the tails list in this pessimal order. Then each candidate $\Theta(n)$ times will be consulted an amortized $\Theta(d)$ times, each time causing (a, without optimization) an amortized $\Theta(d)$ searches through lists of amortized size $\Theta(n)$, or (b, with optimization) a $\Theta(1)$ table search followed by an amortized $\Theta(d)$ table maintenance adjustment. Worst case is indeed $\Theta(d^2n^2)$ for the unoptimized algorithm, $\Theta(dn)$ for the optimized one.

a quadratic improvement, that bring the cost of C3 or C4 back down to the levels of less consistent algorithms such as used in Ruby or Scala. Unhappily, it looks like at least the Python and Perl implementations are missing this crucial optimization; it might not matter too much because their d and n values are I suspect even lower than in Common Lisp.[25]

There is also a space vs time tradeoff to check subtyping of suffixes, by using a vector (O(1) time, O(k) space per struct, where k is the inheritance depth the largest struct at stake) instead of a linked lists (O(k) time, O(1) space per struct). However, if you skip the interstitial infix specifications, suffix hierarchies usually remain shallow,[26] so it's a bit moot what to optimize for.

7.4.5 C3 Tie-Breaking Heuristic

The constraints of C3 or C4 do not in general suffice to uniquely determine how to merge precedence lists: there are cases with multiple solutions satisfying all the constraints. This is actually a feature, one that allows for additional constraints to be added.[27] at which point the linearization algorithm must use some heuristic to pick which candidate linearization to use.

The C3 algorithm (and after it C4) adopts the heuristics that, when faced with a choice, it will pick for next leftmost element the candidate that appears leftmost in the concatenation of precedence lists. I *believe* (but haven't proved) that this is also equivalent to picking for next rightmost element the candidate that appears rightmost in that concatenation.[28] Importantly, I also believe (but again, haven't proved) this heuristic maximizes the opportunity for a specification's precedence list to share a longer suffix with its parents, thereby maximally enabling in practice the optimizations of single inheritance even when specifications are not explicitly declared "suffix".

One interesting property of the C3 heuristic is that, even if the language does not explicitly support suffix specifications, by following the same discipline that Scala 2 forces upon you, of always placing the most-specific suffix specification last in each specification's local precedence order (removing any of its now redundant ancestors for that order, if necessary), you will obtain the same result if the language explicitly supported suffix specifications using the C4 algorithm. Thus, even without explicit lan-

[25] As mentioned in a previous note, in loading almost all of Quicklisp 2025-06-22, I found d⩽3 99% of the time, d=61 max, n⩽5 90% of the time, n⩽19 99% of the time, n=66 max.

Now at these common sizes, a linear scan might actually be faster than a hash-table lookup. However, a good "hash-table" implementation might avoid hashing altogether and fallback to linear scan when the size of the table is small enough (say less than 20 or so), all while preserving the usual API, so users are seamlessly upgraded to hashing when their tables grow large enough, and behavior remains O(1).

[26] As mentioned in a previous note, in loading almost all of Quicklisp 2025-06-22, I found that k⩽4 in 99.9% of cases, k=6 max. Note though that the pressure on struct inheritance is less for Lisp programs than say Java programs, since in Lisp, inheritance for behavior purpose is usually done through classes not structs. That said, if you could mix both kinds of inheritance, the same would probably still be said of structs, with behavior moved to classes that structs inherit from.

[27] For instance, some classes could be tagged as "base" classes for their respective aspects (like our `base-bill-of-parts` in section 5.3.8), and we could require base classes to be treated before others. This could be generalized as assigning some "higher" partial order among groups of classes (metaclasses), that has higher priority than the regular order, or then again "lower" orders, etc.

[28] Exercise: prove or disprove this equivalence.

guage support for suffix specifications, you could enjoy most of the optimizations of de facto single inheritance if the implementation were clever enough to opportunistically take advantage of them. This is interestingly a property shared by the Ruby and Scala algorithm, but not by the original LOOPS algorithm that Scala tweaked. This seems to be an important property for a tie-break heuristic, that should probably be formalized and added to the constraints of C4.[29]

[29] I haven't thought hard enough to say what other interesting properties, if any, are sufficient to fully determine the tie-break heuristic of C3, or of a better variant that would also respect the hard constraints of C3. I also leave that problem as an exercise to the reader.

Chapter 8

Extending the Scope of OO

> The psychological profile [of a programmer] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large. When you're writing a program, you're saying, 'Add one to the counter,' but you know why you're adding one to the counter. You can step back and see a picture of the way a process is moving. Computer scientists see things simultaneously at the low level and the high level.
>
> Donald Knuth

8.1 Optics for OO

8.1.1 Focused Specifications

Before I may revisit familiar features of advanced OO systems such as accessors, method combinations or multimethods, I must once again introduce some new elementary concept that will much simplify their formalization: *focused* specifications.

A specification, whether modular definitions, modular extensions, multiple inheritance specifications, optimal inheritance specifications, etc., can be *focused* by enriching it (via conflation or explicit product) with two access paths: a path from the top of the program state to the module context being referenced, and a path from the top of the program state to the method being extended. Thus, instead of being "free floating", your specification will be "located" within the context of the greater program state. Furthermore, to keep formalizing OO features in terms of pure functional semantics, these access paths I will formalize as functional *lenses* (see below).

And before I discuss new features, I will start with showing how focused specifications can simplify the formalization of individual classes or prototypes within an ecosystem, or of regular methods within a prototype. Having to introduce lenses means the notion of focused specification was overkill and a distraction in the formalism of

previous chapters, when handwaving the relationship between open and closed modular extensions was good enough. But since I am going to pay the price anyway for the sake of explaining advanced features, I may as well enjoy the benefits for basic features as well.

8.1.2 Short Recap on Lenses

A lens (Foster et al. 2007; Meijer et al. 1991; O'Connor 2012; Pickering et al. 2017) is the pure Functional Programming take on what in stateful languages would typically be a C pointer, ML reference, Lisp Machine locative, Common Lisp place, etc.: a way to pin-point some location to inspect and modify within the wider program's state. A lens is determined by a "view", function from a "source" to a "focus"; and an "update", function from a change to the focused data to a change of the wider state from "source" to an updated "target".[1] As a function from source to focus and back, it can thus also be seen as generalizing paths of fields and accessors, e.g. field `bar` of the 3rd element of field `foo` of the entry with key `(42, "baz")` within a table.

Monomorphic Lens A monomorphic lens (or simple lens), can be seen as a pair of a view function `s → a`, and an update function `(a → a) → s → s`. The view function allows you to get a current "inner" value under focus, of type `a`, from the "outer" context, of type `s`. The update function allows you to see how a local change in the "inner" value under focus transforms the "outer" context being focused:

```
type MonoLens s a =
       { view : s → a ; update : (a → a) → s → s }
```

Polymorphic Lens A polymorphic lens (or "stabby" lens), of type `PolyLens s t a b`, generalizes the above: you still have a view function `s → a` to extract an inner value from the outer context, but your update function now has type `(a → b) → s → t`, so that the inner change in value can involve different input and output types, and so can the outer change in context. But the starting points of the update function are the same as the start and end points of the the view function, so that you are updating the same thing you are viewing. Monomorphic lenses are a special case or polymorphic lenses.

```
type PolyLens s t a b =
       { view : s → a ; update : (a → b) → s → t }
type MonoLens s a = PolyLens s s a a
```

[1] In more stateful languages, a more popular view is that of a pair of a "getter" and a "setter"; this maps well to lower-level primitives of stateful languages. But these variants do not compose well in a pure functional setting, where composability is a good sign of good design, while lack of composability is a strong "smell" of bad design. In a pure functional setting, the "getter" part is fine (same as view), but the "setter" part drops crucial information about the flow of information, and does not compose well, instead requiring the flow of information to be be awkwardly moved to other parts of the program to be reinjected into the setter. By contrast, the "update" API trivially composes. In the Haskell lens libraries, the "update" function is instead called "over"; maybe because it "applies an update *over* a change in focus"; maybe also because the word "update" was taken by other functions; it's not a great name. We'll stick to "update".

Skew Lens A skew lens (or "ripsjq" lens, pronounced "rip sick"), yet generalized the above: now the types for the update are not required to be the same as those for the view, so you don't have to be looking exactly at the change you're experiencing. The view `s → r` goes from an outer context `s` to an inner context `r` (where "r" is for required, and "s" is just the next letter), and the update goes from an extension `i → p` to `j → q` (where "i" is for inherited, "p" is for provided, and "j" and "q" are just the next letters). Polymorphic lenses are a special case of skew lenses.

```
type SkewLens r i p s j q =
       { view : s → r ; update : (i → p) → j → q }
type PolyLens s t a b = SkewLens a a b s s t
```

View and Update We can also give separate types fo View and Update:

```
type View r s = s → r
type Update i p j q = (i → p) → j → q
type SkewLens r i p s j q = { view : View r s ; update : Update i p j q } }
```

Getter and Setter There are cases when one may prefer the familiar view of lenses as involving a getter and a setter, instead of a view and an update. The getter and the view are the same thing, so that's easy. On the other hand, the setter and the update are slightly harder.

```
type Setter s t b = b → s → t
lensOfGetterSetter : View a s → Setter s t b → PolyLens s t a b
(define lensOfGetterSetter (λ (get) (λ (set)
  ((makeLens get)
     (λ (f) (λ (s) (set (f (g s))))))))))
setterOfLens : PolyLens s t a b → Setter s t b
(define setterOfLens (λ (l)
  (λ (b) (λ (s) ((l 'update) (λ (a) b))))))
```

Note how you need a matching getter and setter to achieve a polymorphic lens. To achieve a skew lens, you would need two getters and a setter: one getter for the view, another getter and a setter for the update; the two getters needn't match, but the second getter and the setter must.

Composing Lenses We can compose view, update and lenses as follows, with the obvious identity lens:

```
composeView : View s t → View r s → View r t
(define composeView (λ (v) (λ (w)
  (compose w v))))

composeUpdate : Update i p j q → Update j q k r → Update i p k r
(define composeUpdate (λ (f) (λ (g)
  (compose f g))))

makeLens : View r s → Update i p j q → SkewLens r i p s j q
(define makeLens (λ (v) (λ (u)
```

```
  (extend-record 'view v
    (extend-record 'update u
      record-empty)))))

composeLens : SkewLens s j q ss jj qq → SkewLens r i p s j q →
                SkewLens r i p ss jj qq
(define composeLens (λ (l) (λ (k)
  ((makeLens (composeView (l 'view) (k 'view)))
    (composeUpdate (l 'update) (k 'update))))))

idLens : SkewLens r i p r i p
(define idLens
  ((makeLens id) id))
```

You'll notice that `composeView` is just `compose` with flipped arguments, and `composeUpdate` is just `compose`. `composeLens` just composes each component with the proper function.[2] Views, Updates, Lenses are categories, wherein composition is associative, and identities are neutral elements.

Field Lens Given some record representation, a view for a field of identifier key `k` is just a function that returns the field value `r.k` for given as argument the record `r`, whereas an update gives you a change in record given a change for that field. More sophisticated representations will have more sophisticated lenses, but here is what it looks like in my trivial representation of records as functions from identifiers to value, where `r.k = (r 'k)`:

```
(define fieldView (λ (key) (λ (s)
  (s key))))
(define fieldUpdate (λ (key) (λ (f) (λ (s)
  (extend-record s key (f (s key)))))))
(define fieldLens (λ (key)
  ((makeLens (fieldView key)) (fieldUpdate key))))
```

To access the subfield `bar` of the field `foo` of an object `x`, you can apply `(compose-Lens (fieldLens 'foo) (fieldLens 'bar))` to `x`. Note that the order of lenses is covariant with the usual notation `x.foo.bar`. A little bit of syntactic sugar could help you achieve a similar notation, too; but we are deliberately avoiding syntactic sugar in this book.

A `(fieldLens key)` can be a simple lens of type `MonoLens s a` when applied to a record of type `s` that has a field `key` of type `a`. But it can also be used as a polymorphic lens or skew lens, where you view the field `key` of your context and also modify the field `key` of the value you extend, but the two need not be the same, and the modification need not preserve types.

[2] As usual, you can represent your lenses such that you can compose them with the regular `compose` function, by pre-applying the `composeLens` function to them. Haskellers use a further condensed representation as a single composable function that some claim is more efficient, "van Laarhoven" lenses, but I will avoid it for the sake of clarity.

8.1.3 Focusing a Modular Extension

From Sick to Ripped You may have notice that I used the same letters `r i p` to parameterize a `SkewLens` (plus their successors) as to parameterize a `ModExt`. This is not a coincidence. You can focus a modular extension by looking at it through a matching skew lens:

```
skewModext : SkewLens r i p s j q → ModExt r i p → ModExt s j q
(define skewModExt (λ (l) (λ (m)
  (compose (l 'update)
    (compose m (l 'view))))))
```

Thus with a `SkewLens r i p s j q`, we can change the continuation for a modular extension from expecting `s j q` (pronounced "sick") to expecting `r i p` (pronounced "rip").

Metaphors for Modular Extensions and Skew Lenses A modular extension can be conceived as a *sensactor*: it has a sensor, the input from the module context, and an actuator, the output of an extension to the value under focus.

A single skew lens can change both the module context, and the extension focus. A `SkewLens r i p s j q` can transform an inner `ModExt r i p` into an outer `ModExt s j q`. As always, note that in general, `r` (required, the module context) is largely independent from `i p` (inherited and provided, the extension focus). They only coincide just before the end of the specification, to obtain a closed modular definition you can resolve with a single use of the Y combinator.

One way to think of a skew lens is as similar to the pair of a periscope through with a submarine pilot may look outside, and the wheel or yoke through which they may pilot their boat: the submarine pilot is the sensactor, and the skew lens transforms the I/O loop of the pilot into the I/O loop of the submarine. Similarly, one may consider the laparoscope that a surgeon may use, and the separate instrument with which he will operate the patient: the "skew lens" transforms the surgeon I/O into the instrument I/O.

It is a common case that the actuator is within the frame of the sensor, such that the sensactor may observe not only what they are modifying, but also a wider context. In formal words, there are two polymorphic lenses `l` and `k` such that the skew lens view is `(l 'view)`, and the skew lens update is `((composeLens k l) 'update)`, further specializing the previous view. But that is not necessary in the general case. Just like competent painters can put paint on canvas while looking at their model, not at their hand, the "skew lens" does not imply that the hand is seen by the eye, that the actuator is somehow visible in the frame of the sensor. The actuator may involve a bigger extension focus (heating the whole room, not just the thermometer), or an extension focus completely disjoint from the context being observed (like a caricaturist reproducing features observed in his model, but in an exaggerated style).

Now, inasmuch as you consider those function types as a model for stateful mutation with in-memory pointers, that means that the "read pointer" `r` and the "(re(ad)-write) pointer" `p` (with its "initial value" `i`) are independent; in a mutable variant of a skew lens, you will have two pointers, not just one as with monomorphic or polymorphic lens.

Focused Specification A *focused specification* will be the datum of a skew lens and a specification. Above, the specification was a modular extension; but in general, it may as well be a modular definition, a multiple inheritance specification, an optimal inheritance specification, etc., depending on the kind of specification you're interested in.

The skew lens says what the specification is modifying exactly, relative to a known place—typically the entire ecosystem, but potentially any other place you may currently be looking at.

8.1.4 Adjusting Context and Focus

Adjusting both together A monomorphic lens, or simple lens, can refocus a closed specification focus into another closed specification focus, such that a local closed specification `ClosedSpec a` can be turned into a global closed specification for the complete ecosystem, `ClosedSpec ES`. Thus, when specifying a value `foo.bar` in the ecosystem, you will use `(composeLens (fieldLens 'Foo) (fieldLens 'Bar))` as your `SpecFocus`.

As is a theme in this book, though, and which was never discussed before in the OO literature, the interesting entities are the *open* specifications, not just the closed ones. This is what makes skew lenses interesting. If you considered only closed specifications, you would only consider monomorphic lenses (or maybe polymorphic lenses when you specifically want to distinguish types for the ecosystem before and after extension). But then, you wouldn't be able to formalize the advanced notions we are going to discuss in the rest of this chapter.

Adjusting the Extension Focus Given a focus on a specification one can focus on a specific method of that specification by further adjusting the extension focus using `u = (fieldUpdate key)` where `key` is the identifier for the method. Thus, `(composeLens (composeLens (fieldLens 'foo) (fieldLens 'bar)) (updateOnlyLens (fieldUpdate 'baz)))` will let you specify a method `baz` for the specification under `foo.bar` in the ecosystem, where:

```
updateOnlyLens : Update i p j q → SkewLens r i p r j q
(define updateOnlyLens (λ (u)
  ((makeLens id) u)))
```

More generally, given a lens `l` to focus on the specification, and a lens update `u` to refocus on just the extension focus, the lens `(composeLens l (updateOnlyLens u))` will up focus on the method at `u` of the specification at `l`. This is a common case when specifying a sub-method in a methods (more on that coming), a method in a specification, a specification in a library, a library in the ecosystem—all while keeping the broader entity as context when specifying the narrower one.

Broadening the Focus At times, you may want to make the focus broader than the module context. Then, you can use a lens with "negative focal length": instead of narrowing the focus to some subset of it, it broadens the focus. Thus you can zoom out

rather than only zoom in. Zooming out can take you back to were you were previously, or to a completely different place.

For instance, given a broader context `c : s`, and a lens `l : MonoLens s a`, then the following "reverse lens" broadens the focus, by completing the "rest" of the reverse focus with data from the context. Beware though that if you use the update more than once, you will always get answers completed with data from the same non updated context. If you want to update the context each time, you have to reverse the lens with the updated context every time.

```
reverseView : s → MonoLens s a → View a s
reverseUpdate : s → MonoLens s a → Update a s a s
reverseLens : s → MonoLens s a → MonoLens a s
(define reverseView (λ (s) (λ (l)
  (λ (a) (((setterOfLens l) s) a)))))
(define reverseUpdate (λ (s) (λ (l) (λ (a) (λ (f)
  ((l 'view) (f ((reverseView s) l) a)))))))
(define reverseLens (λ (s) (λ (l)
  ((makeView ((reverseView s) l)) ((reverseUpdate s) l)))))
```

Adjusting the Context The module context contains the data based on which a modular extension may compute its extension. Sometimes, you may want to narrow the context, to match the already narrowed extension focus; or to broaden the context to the next broader entity; or to locally override some configuration in the context; or to locally instrument some of the context entities (e.g. for debugging, testing, profiling performance, etc.); or just to switch the context to something different for any reason. You can also use the getter to narrow the context in terms of visibility, access rights, or some other system of permissions or capabilities, so the extension may only invoke safe primitives, or primitives that were suitably wrapped in a "security proxy" for safety. Or you can focus the context on one object to specify extensions for another object that somehow "mirrors" or reacts to the object in context, or logs its history, backs up or persists its data, does resource accounting, etc.

To adjust the context without adjusting the extension focus, use:

```
viewOnlyLens : View r s → SkewLens r i p s i p
(define updateOnlyLens (λ (v)
  ((makeLens v) id)))
```

8.1.5 Optics for Prototypes and Classes

So far the only primitive lens I showed was the field lens. Here are two kinds of lenses that are essential to deal with prototypes and classes.

Prototype Methods I'll assume for now that prototypes are records implemented with the `rproto` encoding from section 6.1.4. Then, if you have a lens `l` to focus onto a prototype, you may further focus on the prototype's specification by further composing `l` with the following lens:

```
(define rprotoSpecView spec←rproto)
(define rprotoSpecUpdate rproto←spec)
(define rprotoSpecLens ((makeLens rprotoView) rprotoUpdate))
```

The entire point of `rproto` is that the target view is `id`. However, what the target update should be is an interesting question: if you update some fields of the target, then the target will not be in synch with the specification anymore. You could try to have an update function that arbitrary changes the specification to be a function that constantly returns the current state of the record. Or you could erase the magic slot for the prototype. Or you could try and be careful about which fields are being updated, and modify the prototype just for those fields. Or, if your language supports error reporting, you could issue an error if someone tries to update the target in any way other than by updating the specification. There are many options, none of them universally correct. Rather, a standard library might offer all these options to programmers, and we'll soon see which are most liked by programmers.

Class Instance Methods Inasmuch as classes are prototypes, the way to deal with methods on a class are the same as for prototypes. To define more refined lenses, I'll further assume the encoding of section 6.2.2.

XXX EDIT HERE XXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Prototypes, including classes, can thus be defined incrementally, by assembling or composing plenty of such focused specifications; and using lenses and sensactors to programmatically select each time how and where they fit in the bigger picture. You can give types to method specifications independently from any specific surrounding prototype or class specification. You can reuse these specification as part of multiple prototype specifications. You can transform them, store and transmit them, compose them, decompose them, recompose them, as first class objects.

8.1.6 Methods on Class Elements

8.2 Method Combinations

8.2.1 Win-Win

Another fantastic contribution by Flavors (Cannon 1979) is Method Combinations: the idea that the many methods declared in partial specifications are each to contribute partial information that will be harmoniously combined (mixed in), rather than complete information that have to compete with other conflicting methods that contradict it, the winners erasing the losers. Win-win interactions rather win-lose, that was a revolution that made multiple inheritance sensible when it otherwise wasn't.

I will quickly present the refined method combinations from CLOS (Pitman 1996; Steele 1990), rather than the more limited method combinations of the original Flavors.

8.3 Multiple Dispatch

8.4 Dynamic Dispatch

Kin vs type. (Allen et al. 2011)

Chapter 9

Implementing Objects

> Metaobject protocols also disprove the adage that adding more flexibility to a programming language reduces its performance.
>
> Kiczales, des Rivières, Bobrow

9.1 Representing Records

9.1.1 Records as Records

So far we encoded Records as opaque functions with some kind of identifier as input argument, and returning a value as output. This strategy works, and is portable to any language with Higher-Order Functions. But, (1) it isn't efficient, and, (2) it leaks space.

Meanwhile, the name "record" itself, as per Hoare (1965), suggests a low-level representation in terms of consecutive words of memory. And so, I will cons

A better strategy is possible.

It is possible to do better... but to actually efficient, I'll have to write non-portable code. the code won't be trivially portable. Your LLMs will help you.

Records as records? Now we're talking.

9.1.2 Lazy Records

9.2 Meta-Object Protocols

Chapter 10

Conclusion

10.1 Scientific Contributions

> Early in life I had to choose between honest arrogance and hypocritical humility. I chose honest arrogance and have seen no occasion to change.
>
> Frank LLoyd Wright

Here is the part of this book where I actually do the bragging, with a list of never-done-before feats I achieved in its book or the work that immediately preceded it:

10.1.1 OO is Internal Modular Extensibility

I rebuilt Object-Orientation (OO) from First Principles, offering an explanation of how the basic mechanisms of OO directly stem from Modularity, Extensibility, and Internality. The equations of Bracha and Cook (1990) were not arbitrary axioms, but necessary theorems, that I could further simplify and generalize.

10.1.2 My Theory of OO is Constructive

I built minimal OO in two lines of code directly relating principles to code; my code is both simpler and more general than the formulas from the 1990s; it can be used not just as a "semantic model", but as a practical implementation usable in actual applications. A few more lines gave you recursive conflation; yet a few more tens of lines give you multiple inheritance. None of it is just theory, none of it is magic, none of it is ad hoc, it's all justified. And it's portable to any language with higher-order functions.

10.1.3 Precise Characterization of those Principles

So I may derive OO from them, I first gave novel and precise, though informal, characterizations of Modularity and Extensibility, based on objective criteria, when familiar

notions previously have often invoked a lot but never defined well. For Internality, I extended the familiar but not always understood notions of "first-class" and "second-class" with new notions of "third-class" and "fourth-class".

10.1.4 Demarcation of OO and non-OO

Crucially, my Theory of OO explains not only what OO is, but also what it is not. This is important to debunk honest mistakes, incompetent errors, ignorant claims, and outright frauds, that any successful activity attracts, harming actual or potential OO practitioners. OO is not C++, OO is not based on Classes, OO is not imperative, OO is not about "encapsulation", OO is not opposed to Functional Programming (FP), OO is not about message passing, OO is not a data model, OO is not rewrite logic.

10.1.5 OO is naturally Pure Lazy FP

Remarkably, and contrary to popular belief, I proved the natural paradigm for OO is Pure Lazy Functional Programming. This is the very opposite of the eager imperative model that almost everyone associates to OO, indeed used by currently popular static Class OO languages. Yet in these languages, OO only happens at compile-time, indeed in a pure lazy dynamic functional programming language, though often a severely stunted one.

10.1.6 Conflation of Specification and Target is All-Important in OO

I elucidated the concept of conflation, latent in all OO since Hoare (1965), necessarily though implicitly addressed by each and every one of my predecessors, yet never a single time once explicitly documented before. Shame on them. Before making this concept explicit, the semantics of objects was extremely complex and ad hoc. After making it explicit, the semantics of objects is astoundingly simple, it just involves a regular use of the simplest of recursion operators, the fixpoint. Plus a pair to bundle specification and target together.

10.1.7 Open Modular Extensions are the Fundamental Concept of OO

Compared to previous theories that only consider *closed* modular extensions (where the extension focus coincides with the module context), or worse, closed modular definitions (with no notion of extension), my *open* modular extensions *vastly* simplify OO, by enabling:

- Simpler more universal types with no ad hoc construct, just plain recursive subtypes
- Combining, composing and decomposing open specifications with rich algebraic tools

- Using optics to zoom semantics at all scales, down from individual method declarations up to entire ecosystems of mutually recursive prototypes

10.1.8 Flavorful Multiple Inheritance is Most Modular

I explained why flavorful multiple inheritance with local order and monotonicity is more expressive and more modular than the alternatives, be it less consistent flavorful multiple inheritance, flavorless multiple inheritance, mixin inheritance, single inheritance, or no inheritance. I also explained why so many great computer scientists got stuck into the "conflict" view of multiple inheritance and how and why the "harmonious combination" view is so much better.

10.1.9 There is an Optimal Inheritance

I implemented a new variant of inheritance. It is more than just combining previous ideas, such as tucking C3 onto the design of Ruby or Scala, though even that, or just my optimization of C3 from $O(d^2n^2)$ to $O(dn)$, would have been a (modest) contribution: I also showed that the design is necessary to fulfill a higher purpose of optimality; and not arbitrary, not just a clever hack made necessary for backward compatibility with existing infrastructure. This optimal inheritance is now part of Gerbil Scheme; you can easily port my code to add it to your own language.

10.2 Why Bragging Matters

10.2.1 Spreading the New Ideas

> Don't have good ideas if you aren't willing to be responsible for them.
>
> Alan Perlis

What's the use of having good ideas and writing about them if the readers don't even notice?

Writing this book won't bring me fame and status. But if it can convince future programming language implementers to add OO to their language, and add the best variant of it rather than the pathetic variants that are mainstream today, then my work will not have been in vain. By providing a clear list, I am making sure they know where to start from, and what not to forget. Readers, ***the list of achievements above is a list of opportunities for you to improve the software you use and build***.

I actually started the list of achievements above back when this book was supposed to be a short paper to submit to a conference or journal. Reviewers are overwhelmed with papers to review, most of them of bad quality, hiding vacuity under a heap of verbiage. They are not paid, and it is draining to say no, even to outright bad papers, and even more so to papers that have some good in it, but that are not yet worth the reader's time. An author has to make extra effort to make his contributions clear, even

though it's hard on him, even though some authors will give up before they make their paper publishable, and their original contributions are then sadly lost. I thank the rejecting reviewer who once told me to make my claims clearer.

10.2.2 Spreading Coherent Theories

> *Let theory guide your observations*, but till your reputation is well established, be sparing in publishing theory. It makes persons doubt your observations.
>
> Charles Darwin to a young botanist

My mentor Jacques Pitrat once told me that when a French researcher has three ideas, he writes one paper. when an American researcher has one idea, he writes three papers. If you want your paper published in an american conference, you need not only use the american language, as explicitly demanded, but also the american style, as tacitly required; otherwise, the reviewers won't be able to understand you, and even with the best intentions, they will reject your paper. So split your paper in nine parts, and submit each of them independently.

However, some ideas are not worth much by themselves, if they make sense at all. Some ideas only become valuable because of the other ideas that follow. Some other ideas don't make sense without all the ideas that precede. What is the value of debunking bad ideas about OO and saying mean words about a lot of good scientists? Yet how can I even explain the basic principles of OO if the readers misinterpret what I say because they are confused by those bad ideas about OO? And what is the value of those basic principles if they have no correspondance to actual code? But how could I derive a minimal model from the principles without having examined them? etc.

Each chapter of this book is worth something but only so much by itself, each of my innovations may only seem so modest by itself. Trying to publish them separately would be a lot of effort, each article spending most of its time recapitulating knowledge and fighting false ideas before it could establish a result the utility of which would be far from obvious. Yet together, they build a solid coherent Theory of OO that I hope you'll agree is compelling.

10.3 OO in the age of AI

AIs will hit complexity walls too, and need to factor code in good ways to cooperate on larger, better, safer software.

Annotated Bibliography

> If we knew what it was we were doing, it would not be called research, would it?
>
> Albert Einstein

I absolutely loathe bibliographies that are just a wall of references, without any guidance as to what matters about each work referred. What was good about it? What was bad? What made it relevant to the text that cites them? What should I pay attention to? What are the notable achievements, and especially the underrated gems, in that work? What traps did the author fall into that the reader should be wary of? What historical context makes that work important? What relevant words have changed meanings since this text was written? What does the citing author think of the cited work, and how does that differ from the general opinion of said works? Should I read the paper or not? If not, what should I know of it? If yes, what need I know to make the best of the reading?

As an author, I need to have those notes anyway, for my own use, because there are far too many of those articles for me to remember details about each. And many of those papers I read, I definitely want to forget, to make space for more worthy readings; yet not without making a note somewhere, so I never have to re-read those forsaken papers, even if need to revisit the topic—which I definitely do many times over while writing. I think it is my duty as an author to share those notes with my readers. Thus I am making sure that each reference below comes with useful notes. And I invite other authors to do as much.

My notes are opinionated. They skip over aspects of the cited works that I didn't care about, or at least not insofar as this book is concerned. They also lay judgement upon some of the authors. These judgements are summary. They are based on very limited information about the authors and their context. Strong opinions, weakly held. I'll be happy to be proven wrong, or to be shown nuances I missed. But those are still the best opinions I could make, and still worth sharing. ***You should always judge a book by its cover***: that's the only information you have before you may decide to even open the book. But you should be ready to revise the judgement after you get more information.

Most of the books and papers listed in this bibliography are available online for free. For some of them, I only list the DOI (Digital Object Identifier) that identifies them,

from which you cannot access the resource without paying some "legal" monopoly middleman. Yet I trust my readers to locate free copies if they try hard enough.

Martín Abadi and Luca Cardelli. *A Theory of Objects*. 1996a. doi:10.1007/978-1-4419-8598-9 . Abadi and Cardelli (A&C) are well known and well respected authors, and this classic book of theirs was long presented as the gold standard of what it means to formalize OO. A student at the time the book was published, I was dreaming one day I'd be able to understand it and write papers half as good as I was told A&C's were. But their book (and papers) left me stumped, and so I thought the problem was with me. Re-reading it now that I do understand OO, I realize the problem was with A&C all along. A&C hold false beliefs about OO, then deploy their vast intelligence using fancy logical semantics in doomed attempts to find models that satisfy their inconsistent beliefs. Each time, they can explain why it doesn't quite work—then go find a more sophisticated model that of course will also fail—instead of questioning their premises. A sad waste of brilliant minds—for both authors and readers who after those lengthy constructions believe they understand OO better rather than worse.

Martín Abadi and Luca Cardelli. A Theory of Primitive Objects: Untyped and First-Order Systems. *Information and Computation* 125(2), pp. 78–102, 1996b. doi:10.1006/inco.1996.0024 . A&C fall into all the common traps of OO: they confuse specification and target, therefore firmly believe in the NNOOTT; they try to use modular definitions and fail to understand modular extensions, therefore can't apprehend mixin and multiple inheritance. They deploy all their brilliance to seek ever more sophisticated models that will push back the point at which they confront the contradictions in their beliefs; they won't abandon their beliefs so they abandon the models. Their "Self-Application Semantics" turns a lambda into a primitive "sigma" that inverts basic logic so they can avoid obvious contradiction. Their "Recursive Record Semantics" is perfectly correct for targets, but of course doesn't apply to specifications, and A&C can't imagine that the two need to be solved separately before they can be joined. Their "Generator Semantics" is on the right track, and A&C can almost articulate the distinction of target ("record") and specification ("generator"), but once again find the solution problematic (they are missing our "recursive conflation"). Then they go in and out of some Pierce "split-self" semantics (see Pierce and Turner 1993) and note that it works well for Class OO but move on rather than dig into the interesting fine reasons why. Finally their "solution" is to implement objects on top of the same syntactic rewrite logic paradigm as the λ-calculus is usually presented in. A low-level implementation, just for a mathematical "virtual machine" foreign to anyone but logicians. "Cool. So what?" Low-level object implementations in Lisp, BCPL or C existed for 2 decades already. Their calculus won't help OO practitioners understand semantics, and won't help logicians appreciate OO. We already knew such feat was possible, since rewrite logic is a universal subtratum. Instead, the important semantic details worth discussing and arguing get lost in the details of a tedious construction. So nobody learns anything.

Moez A AbdelGawad. A domain-theoretic model of nominally-typed object-oriented programming. *Electronic Notes in Theoretical Computer Science* 301, pp. 3–19, 2014. doi:10.1016/j.entcs.2014.01.002 The author falls deep into the NNOOTT without realizing it. It is more sad than funny to see a young researcher "discover" sixty years later as if they were new the naive ideas that our forefathers had when they first stumbled upon OO, and dismiss without truly understanding it the research that he is citing. He is first responsible for his own failing, but his publications also reflect poorly on his advisors.

Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Program*. 2nd edition. MIT Press, 1996. doi:10.5555/547755 . This classic book, first published in 1984, inspired generations of computer scientists. It is somewhat outdated in its style or its treatment of some topics, but many of its insights are timeless. Even its footnotes alone are gold. The author do build some notion of "objects" as message-handling functions, but this only deals with the "encapsulation" of state (modularity) aspect of OO, and the authors explicitly avoid the topic inheritance.

Norman Adams and Jonathan Rees. Object-Oriented Programming in Scheme. In *Proc. Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pp. 277–288. Association for Computing Machinery, New York, NY, USA, 1988. doi:10.1145/62678.62720 . This paper expounds the object system of Yale T Scheme, actually implemented circa 1982, and notably used for its GUI: class-less, object-less (what they call "object" is any value), it is a sort of Prototype OO, but without conflation of targets (that they call instances) and specifications (that they call components). Their virtual machine merges instances and functions: an instance is a function with multiple entry points (one per method) including a default one, used when just calling it as a function. A function stricto sensu is an object with a single method. T instances as such implement single-inheritance, through a rudimentary protocol of delegation; however "components" provide mixin inheritance (before the name was invented). Impressive, but sadly largely unknown outside Lisp circles.

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*, pp. 973–992, 2011. doi:10.1145/2048066.2048140 . A great paper that shows how to properly type multiple dispatch and multiple inheritance together, with a very interesting concept of runtime kin vs compile-time types. And yet, at some point the paper becomes too formal and I admit I don't follow it anymore to tell what is missing or what is there that I'm missing. One day when I get serious about typesystems, I shall surely re-read this paper and try to fully understand it. In the meantime, I have complete trust in Guy Steele, and will therefore assume that the paper correctly achieves what it says it did, and provides a constructive proof that indeed the problem of typing multimethods is solved.

Joe Armstrong. Making reliable distributed systems in the presence of software errors. PhD dissertation, Royal Institute of Technology, Stockholm, Sweden, 2003. `https://nbn-resolving.org/urn:nbn:se:kth:diva-3658` . Joe gives an overview of Erlang, the great Concurrency-Oriented Programming Language of which he was one of the main authors. Actors taken seriously into production. Section 3.8 "Dynamic Code Change" describes support for hot code upgrade. Only 2 versions of the code allowed; at the 3rd, processes using the 1st are killed. I _suppose_ that means there are again only 2 versions left, and only 1 after all processes are fully updated. That's good enough great in Erlang because failure is always a valid fallback, but may might not be enough in less resilient systems. The semantics of upgrade is well-defined: calls to module-qualified name are dynamic, to unqualified name are static. Tail-calls are recommended but not mandated.

Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hölzle. Mixins in Strongtalk. In *Proc. ECOOP 2002 Workshop on Inheritance*, 2002. `https://bracha.org/mixins-paper.pdf` . This paper describes the typesystem of Strongtalk, a dialect of Smalltalk with mixins and optional types, allowing for efficient compilation of code in cases that matter. The techniques described notably had a powerful influence on subsequent implementations of the JVM. This is a very nice paper with great practical applications. However, its type theory is the NNOOTT, checking method types with no regard for the open recursion in the types that involve self-reference. Yet, that is alright if those recursive references to objects of the "same" type are dynamically typechecked, as we suppose they are, since the system being optionally typed obviously supports such checks. But the authors never explicitly discuss that. Their discussions of related work conflates mixin inheritance and multiple inheritance, which is alright if you're only interested in the types of mixins as they are in this paper; but that too could and should have been made explicit. The authors thus seem to have a lot of implicit knowledge of the issues that do and do not matter, but will leave unsuspecting readers ignorant of those issues.

Henry G. Baker. Critique of DIN Kernel Lisp definition version 1.2. *LISP and Symbolic Computation* 4, pp. 371–398, 1992. doi:10.1007/BF01807180 . A brilliantly insightful critique of a programming language design. Introduces the notion of "abstraction inversion", wherein a low-level notion is implemented on top of a high-level notion, rather than the other way around, creating inefficient software, and programmers thinking at the wrong level of abstraction.

Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A Monotonic Superclass Linearization for Dylan. In *Proc. OOPSLA*, pp. 69–82. Association for Computing Machinery, New York, NY, USA, 1996. doi:10.1145/236337.236343 . This paper introduces the C3 superclass linearization algorithm, providing a simple practical solution to the problem first solved by Ducournau 1994: a multiple inheritance Linearization algorithm that satisfies the Monotonicity property as well as the Local Precedence Order (and, without saying it, Shape Determinism). The best way to do Multiple Inheritance.

R. S. Barton. A New Approach to the Functional Design of a Digital Computer. In *Proc. Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), pp. 393–396. Association for Computing Machinery, New York, NY, USA, 1961. doi:10.1145/1460690.1460736 . This paper shows how the author architected the Burroughs B5000 to support the execution of high-level languages (notably ALGOL 60). It is hard to read because it is dated; what was obvious then isn't now, while many ideas that were new then seem obvious now. Still, Alan Kay cites the B5000 as a breakthrough, and I believe him.

Alan Bawden. PCLSRing: Keeping Process State Modular. MIT, 1989. `http://fare.tunes.org/tmp/emergent/pclsr.htm` . An amazing paper by Alan Bawden, on how the ITS operating system managed to interrupting processes in a way that "a process couldn't be caught with its pants down", i.e. maintaining some important system invariants, when a stopped processes is observed by other processes, even though the invariants may have been temporarily broken by the system at the precise moment that the timer interrupt stopped the execution. If anything, a big part of my (incomplete) PhD thesis was a generalization of this paper.

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*, pp. 228–243. Springer, 1999. doi:10.1007/3-540-48959-2_17 . A wow-good paper, that explains the duality between computations and values, call-by-name and call-by-value, etc., with practical implications on the design and implementation of programming languages.

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 . This paper is a chapter of the Common Lisp specification then in the process of being standardized. The introduction is very short, and most of the paper is a description of the API, primitive by primitive. There it no presentation of the concepts. But the paper shows that CLOS was already mature in 1988 (the standard was finalized in 1994).

Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *Proc. OOPSLA*, 21(11), pp. 17–29. Association for Computing Machinery, New York, NY, USA, 1986. doi:10.1145/960112.28700 . Excellent paper. This reprint of a 1985 report presents CommonLoops, an evolution from LOOPS that integrates lessons from New Flavors, from which it takes linearization and generic functions, that it extends to support multimethods. As in LOOPS, there are metaclasses (that control single or multiple inheritance, slot accessors). It has a form of call-next-method named run-super. Method calls are compiled with various optimizations including local and global method caches. Introduces class variables, init-forms in slot descriptions instead of access-time default values as in LOOPS, slot properties, "active values" (kind of method combination for slot accessors?). More nice design discussions and comparisons. Symbols are all lowercase.

Daniel G. Bobrow and Terry Winograd. An Overview of KRL, A Knowledge Representation Language. Stanford Artificial Intelligence Laboratory, AIM-293, 1976. `https://exhibits.stanford.edu/stanford-pubs/catalog/pf235fy3176` . This paper has multiple inheritance (without using the word) as multiple "descriptors" each providing a "perspective" on a given "object" or "prototype", and "inheritance of properties", meant as a declarative system of data on which operations are then performed, rather than as code as such (though coding is possible).

Daniel Gureasko Bobrow and Mark Stefik. *The LOOPS manual*. 10. Xerox Corporation Palo Alto, CA, 1983. . LOOPS has multiple inheritance. The authors cite the right Flavors passage on _modular_ interactions between orthogonal issues. But instead of linearization, they instead offer tools for methods to manually call methods from one or all parents, or from a named ancestor. On the other hand, LOOPS also includes a rich slot access protocol, a metaclass protocol, and a knowledge base with a rule-oriented programming engine. Interestingly, the paper has a mix of UPPERCASE primitives, CamelCase functions and objects and classes, camelCase variables.

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 H Borning and Daniel HH Ingalls. Multiple inheritance in Smalltalk-80. In *Proc. Proceedings at the National Conference on AI*, 82, pp. 234–237, 1982. `http://www.laputan.org/pub/papers/MI-Borning-Ingalls.PDF`

Alan Hamilton Borning. ThingLab— an Object-Oriented System for Building Simulations using Constraints. In *Proc. 5th International Conference on Artificial Intelligence*, pp. 497–498, 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 . ThingLab is an extension to Smalltalk that supports Prototype OO, and integrates it with the Class OO of Smalltalk. On top of that, Borning builds a constraint-based graphic system that generalizes Sketchpad, and can be used to build simulations. ThingLab also implements multiple inheritance, although the flavorless conflict kind (ThingLab predates Flavors).

Alan Hamilton Borning. Classes Versus Prototypes in Object-Oriented Languages. In *Proc. 1986 Fall Joint Computer Conference*, pp. 36–40, 1986. doi:10.5555/324493.324538

Gilad Bracha. The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. PhD dissertation, University of Utah, 1992. http://www.bracha.org/jigsaw.pdf

Gilad Bracha, Peter Ahe, Vassili Bykov, Yaron Kashai, and Eliot Miranda. The Newspeak Programming Platform. *Cadence Design Systems*, 2008. https://bracha.org/newspeak.pdf

Gilad Bracha and William Cook. Mixin-Based Inheritance. In *Proc. OOPLSA/ECOOP*, 25(10), pp. 303–311. Association for Computing Machinery, New York, NY, USA, 1990. doi:10.1145/97945.97982

Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proc. the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, 28(10), pp. 215–230. Association for Computing Machinery, New York, NY, USA, 1993. doi:10.1145/165854.165893 . A good paper about Strongtalk, a very influential system for the many practical implementation techniques it spearheaded, that were widely adopted thereafter by the JVM. Yet, from a theoretical point of view... their typesystem is the NNOOTT (which is OK if you're going to dynamically check recursive references). The biggest problem with this paper is that it is too short and covers too much terrain to give more than an overview of this very interesting system.

Kim B Bruce. Typing in Object-Oriented Languages: Achieving Expressiveness and Safety. *Unpublished, June*, 1996. https://www.academia.edu/2617908/Typing_in_object_oriented_languages_Achieving_expressibility_and_safety . This paper includes a good explanation of the problems with the NNOOTT. It also is somewhat aware of the issue of conflation: "The notions of type and class are often confounded in object-oriented programming languages" p. 5. But it still chooses an approach with an ad hoc class primitive, in which the open recursion type is accessible via the special type variable MyType, and subtyping before the fixpoint is called "matching" (though this (correct) technique is not explained in those terms).

Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good "match" for object-oriented languages. In Mehmet Akşit and Satoshi Matsuoka (Ed.), *Proc. ECOOP'97 — Object-Oriented Programming*, pp. 104–127. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. doi:10.1007/BFb0053376

Howard Cannon. Flavors: A Non-Hierarchical Approach to Object-Oriented Programming. 1979. https://www.softwarepreservation.org/projects/LISP/MIT/nnnfla1-20040122.pdf . Cannon's groundbreaking work unexpectedly solved the problem of multiple inheritance, by harmoniously combining the methods in a way that generalizes Teitelman's ADVISE facility. The paper is full of ideas and genius insights, even though it is obviously unpolished, and written by an inexperienced young man. Cannon's software and his report were widely circulated at MIT and in the Lisp community, but unhappily the report was never officially published. The MIT Lisp Machine Manual from 1981 on (3rd edition and later) includes Flavors. The first academic publication on Flavors would be Moon 1986's paper on New Flavors. An October 1980 CADR backup tape shows Flavors was implemented as a short 1428 line program. It has linearization by a two-pass depth-first search algorithm, where parents are called "components" that a flavor "depends-on", but there are also other flavors it "includes" that go at the end to act as defaults, which by default includes at the end the system-defined vanilla-flavor. A mixin (or mix-in) is a flavor meant to be combined-in but not instantiated. The Flavors paper describes the Smalltalk "hierarchical"; discusses the conflict view of multiple inheritance (from Borning, looking at his bibliography) as "a scheme called _multiple superclasses_", and how it fails to support _modular_ interactions between orthogonal issues. and itself "non-hierarchical" by comparison, with flavors instead of superclasses, so "instead of a hierarchy, more of a lattice structure is formed". While 20-year old Cannon's underlying concepts and arguments show his genius, his incorrect wording shows his then lack of education: the DAG of ancestors (words he doesn't use) is still a hierarchy (just a partial rather than total order), and not a lattice (which requires existence and unicity of least upper and least lower bounds). Finally, note that Flavors as such offers no call-next-method. The default method combination, :DAEMON, supports only one primary method, lots of wrappers (macros with :around powers), and side-effects from :before and :after methods to combine methods; but other method-combinations can combine things (:LIST :INVERSE-LIST :PROGN :AND :OR), and you could create your own method-combination that returns a function to compose.

Luca Cardelli. *Extensible records in a pure calculus of subtyping*. Digital. Systems Research Center, 1992. https://bitsavers.trailing-edge.com/pdf/dec/tech_reports/SRC-RR-81.pdf

Luca Cardelli and Peter Wegner. *On understanding types, data abstraction, and polymorphism*. Brown University. Department of Computer Science, 1986. http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf

Robert Cartwright and A AbdelGawad Moez. Inheritance IS Subtyping. In *Proc. The 25th Nordic Workshop on Programming Theory (NWPT)*, 2013. https://cs.rice.edu/~javaplt/papers/Inheritance.pdf . Lots of handwaving for what is in the but the NNOOTT combined with a lack of appreciation for the many previous works that went beyond it, some of which is cited. Refers to AbdelGawad 2014 for actual content.

C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proc. OOPSLA*, pp. 49–70. Association for Computing Machinery, New York, NY, USA, 1989. doi:10.1145/74877.74884

Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen (Ed.), *Proc. ECOOP*, pp. 33–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992. http://www.laputan.org/pub/papers/cecil-ecoop-92.pdf

Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF. In *Proc. Lisp and Symbolic Computation*, 1991. `https://bibliography.selflanguage.org/parents-shared-parts.html` . The authors present an erstwhile design that the Self team had for multiple inheritance. The paper explicitly shuns linearization and method combination from CLOS, in the name of "simplicity", though without a theory of what makes things simple. even more so when they have to code against the kluginess of it all. Because they have a low-level mutable data model, they have to deal with dynamic inheritance, and look for low-level graph traversal strategies, and end up adopting a rule of following a “sender path tiebreaking rule” unique to Self when choosing which super method to call next. (In a later paper they explain they abandoned this idea as causing more problems than it solved, falling back to the “conflict” view of multiple inheritance). Mention rename in Eiffel as an alternative, static way to deal with “unordered” inheritance (by contrast with the flavorful approach they call “ordered”). Authors note that C++ linearizes superclasses for constructors and destructors, just not for regular methods. Issues due to their design: Self mixins must be parentless to avoid their general parents overriding the mixed class's specific slots. “Backtracking”, i.e. lack of a stack of senders as above. “considering altering Self's lookup rule to always search children before their ancestors” They also support cycles but say it's too complex to explain how in the paper. All in all, authors claim to strive for “simplicity”, but shun theory, so end up doing _vastly_ complex things in the end.

Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding Confusion in Metacircularity: The Meta-Helix. *Lecture Notes in Computer Science*, 2000. doi:10.1007/3-540-60954-7_49 . Great paper about a missing piece in Meta-Object Protocols (therefore even more so with systems lacking a MOP): the ability to distinguish an object from the underlying object implementing it. There’s obviously a useful conflation in practice; yet at crucial times when implementing some features, you need to distinguish the two (and so on, in a tower of implementation). The paper thus introduces an explicit implemented-by relationship to avoid confusion between metalevels when extending implementations, e.g. adding shadow slots for history, implementing persistence, etc.

Paul Chiusano and Rúnar Bjarnason. *Functional Programming in Scala*. Manning Publications, 2014. doi:10.5555/2688794 . The authors, core contributors to the ScalaZ library, show how to write pure functional programs in Scala. ScalaZ uses multiple inheritance to encode mathematical relationships between abstractions (like “every monad is a functor”) in the style of Haskell typeclasses, not to build traditional stateful imperative object-oriented class hierarchies.

W. Cook and J. Palsberg. A Denotational Semantics of Inheritance and Its Correctness. *Information and Computation* 114, pp. 329–350, 1994. `https://www.cs.utexas.edu/~wcook/papers/ic94/ic94.pdf`

William Cook. On Understanding Data Abstraction, Revisited. In *Proc. Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pp. 557–572. Association for Computing Machinery, New York, NY, USA, 2009. doi:10.1145/1640089.1640133 . Cook in 3.1 notes the OO (re)presentation eschews the existential type, hiding it under a higher-order function interface, but that's false in general—the existential type is there for compatibility between values you can build and modify. 3.2 claims with reference to his thesis that inheritance is orthogonal to OO (!) but misses that what makes method invocation "dynamic binding" isn't the records of higher-order functions, but the open recursion on self and the sharing of methods between multiple selves thanks to inheritance. He also identifies classes as their constructor... a nice implementation trick in simple OO languages that does not cover the general case of OO. 3.3 Cook wrongfully calls "autognosis" (a word that implies runtime reflection) the criterion what Bracha calls "Pure OO", i.e. having all access go through interfaces. Cook wholly skips CLOS, and Prototypes. Sad. In the end, does not bring much lights to the topic, beyond the obvious. Cook understands modularity, but not extensibility.

William Cook. A Proposal for Simplified, Modern Definitions of "Object" and "Object Oriented". 2012. `https://wcook.blogspot.com/2012/07/proposal-for-simplified-modern.html` . Cook delivers lots of precise insight in OO, that show a deep familiarity with the subject that few ever possessed, and can much enlighten his readers. Then, at crucial times, in the most sophisticated way, he totally misses the point. For instance, he claims that "dynamic dispatch of operations is the essential characteristic of objects". But no, all that means is that the object is _first-class_, which was the topic of his previous section, yet he failed to notice. And being first-class is not specific to OO. Maybe Cook's note on "dynamic dispatch" tells us something important about most Class OO languages: that despite inheritance happening on second-class entities (classes), dispatch still happens on first-class entities (objects). Yet Haskell typeclasses show that need not be always the case (though in Haskell you can explicitly use Data.Typeable class constraint to enable runtime dispatch). Now, when Cook claims that inheritance is "useful but not absolutely essential" for OO: that's where he loses track of the plot. As for 2012 JavaScript (before classes), it _was_ OO already, because it had inheritance. What makes OO is inheritance, not existentially-quantified types. Cook claims that "inheritance can also be used for... extending ML-style modules". Did he even try??? I did, it doesn't type. "Delegation is the dynamic analog of inheritance, which is usually defined statically." So close to the truth: "delegation", i.e. inheritance among first-class entities (see our section on delegation), is the one and only inheritance that matters, it's just that most people somehow only use it at the type-level for ultra restricted metaprogramming. The paragraph on "classes" is so right it what little it says, and so wrong for being so short—there's an entire topic deserving to be deepened there! It's a shame that Cook never took Prototypes seriously except as a short counter-example. Telling: he never even says the word!

William Cook and Jens Palsberg. A Denotational Semantics of Inheritance and its Correctness. *ACM Sigplan Notices* 24(10), pp. 433–443, 1989. doi:10.1145/74877.74922

William R Cook, Walter Hill, and Peter S Canning. Inheritance is not subtyping. In *Proc. POPL*, pp. 125–135, 1989. doi:10.1145/96709.96721

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

William R. Cook. Object-Oriented Programming versus Abstract Data Types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg (Ed.), *Proc. Foundations of Object-Oriented Languages*, pp. 151–178. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. doi:10.1007/BFb0019443 . This article has a good historical overview of OO. Cook really understands modularity as a goal of OO but not extensibility; hence his not appreciating inheritance. He distinguishes Procedural Data Abstractions (PDA) organized around constructors (for classes, etc.) as in Smalltalk vs Abstract Data Types (ADT) organized around observations (pattern-matching for data types) as in CLU. In 5.1 he says Simula was precursor to both styles. Cook almost notices conflation: "a class definition is both a constructor of objects and a type". And in 5.4 Cook is baffled by CLOS and its multimethods that break his model of PDA vs ADT.

J. Costa, Amilcar Sernadas, and Cristina Sernadas. Object inheritance beyond subtyping. *Acta Informatica* 31, pp. 5–26, 1994. doi:10.1007/BF01178920 . This is yet another paper that claims to model OO using Category Theory, but actually models the NNOOTT instead. The authors seem clueless about what they do, and provide no warning. They also try to redefine "inheritance" as refinement, except that when it's "non-monotonic", you only have a partial map, and anything goes (you said nothing). Shameful.

Dave Cunningham. Jsonnet. (accessed 2021-03-11), 2014. `https://jsonnet.org` . Dave took the Google Configuration Language (GCL), reduced it to its core concepts, rightfully replaced dynamic scoping by lexical scoping, and used a syntax in between JavaScript and Python, resulting in a beautiful pure functional lazy dynamic language with builtin first-class Prototype OO. Semantically equivalent to Nix, with a nicer syntax, but little tooling and a small ecosystem (Dave sadly never got much support from his then employer Google). Learning Jsonnet then Nix was key to making me understand OO at long last.

Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. Traits: An approach to multiple-inheritance subclassing. *ACM SIGOA Newsletter* 3(1–2), pp. 1–9, 1982. doi:10.1145/966873.806468

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 . SIMULA is an extension of Algol for "quasi-parallel processing". In 1966, it doesn't have classes yet, but has some variant of what MIT people will later call "Actors", will asynchronous message passing between independently running entities.

Ole-Johan Dahl and Kristen Nygaard. *Class and subclass declarations*. 1967. `https://www.ub.uio.no/fag/naturvitenskap-teknologi/informatikk/faglig/dns/dokumenter/classandsubclass1967.pdf` . THE breakthrough paper that leads to OO, though we can debate whether it is itself OO as such; not in the modern form made popular by Smalltalk, at least. Dahl and Nygaard implement classes for the first time. They do not have the word "inheritance", but the concept is there somehow. A class may have a "prefix class" and "subclasses" of which it is the prefix class. Its ancestry is its "prefix sequence", weirdly kept in most- to least- specific order. The keyword "this" is introduced that C++ will popularize. Because a prefix class may also include suffix code, the "body" for subclasses can be "split" with the keyword "inner" that pinpoints where to put the body of subclasses (by default, at the end). The successor BETA will also follow this approach. As in Hoare 1965, "objects" have "attributes".

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*, pp. 114–121. Association for Computing Machinery, New York, NY, USA, 1975. doi:10.1145/800027.808431

Ken Dickey. Scheming with Objects. *AI Expert* 7(10), pp. 24–33, 1992. http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/scheme/oop/yasos/swob.txt

Eelco Dolstra and Andres Löh. NixOS: A purely functional Linux distribution. In *Proc. the 13th ACM SIGPLAN international conference on Functional programming*, pp. 367–378, 2008. https://edolstra.github.io/pubs/nixos-jfp-final.pdf

R. Ducournau, M. Habib, M. Huchard, and M. L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proc. Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Language, and Applications*, OOPSLA '94, pp. 164–175. Association for Computing Machinery, New York, NY, USA, 1994. doi:10.1145/191080.191110

Roland Ducournau, Michel Habib, Marianne Huchard, and Marie-Laure Mugnier. Monotonic conflict resolution mechanisms for inheritance. *ACM SIGPLAN Notices* 27(10), pp. 16–24, 1992. https://www.researchgate.net/publication/234827636_Monotonic_conflict_resolution_mechanisms_for_inheritance . A great paper about flavorful multiple inheritance, the properties one may expect the linearization algorithm to satisfy, and why they matter. The authors notably introduce the heretofore unknown property of monotonicity. Problems with their proposed algorithm, it uses a global rather than local tie break rule, and they do a "best effort" rather than issue an error out if rules are incompatible. See their 1994 paper and the C3 paper for improvements.

Alexandre Dumas. *Les Trois Mousquetaires*. 1844. https://fr.wikisource.org/wiki/Les_Trois_Mousquetaires . This novel, despite being named "The Three Musketeers", famously has four musketeers as protagonists. Admittedly, the main protagonist isn't one yet when he meets (and duels!) the other three.

ECMA-262. ECMAScript: A general purpose cross-platform programming language. 1997. https://ecma-international.org/publications-and-standards/standards/ecma-262/

Brendan Eich. JavaScript Language Specification. 1996. https://archives.ecma-international.org/1996/TC39/96-002.pdf The ECMA/TC39/96/2 is the very first Draft version of the JavaScript Language Specification. There was documentation on webpages in 1995, but no official specification yet. This document mentions neither "inheritance" nor "delegation" about its objects. Notably, the introduction says "JavaScript borrows most of its syntax from Java, but also inherits from Awk and Perl, with some indirect influence from Self in its object system."

Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound Polymorphic Type Inference for Objects. In *Proc. Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pp. 169–184, 1995a. doi:10.1145/217838.217858 . This paper presents I-LOOP, a language with classes, macro-reduced to I-SOOP, with type inference. AFAICT, the authors are the first to use polymorphic recursively constrained types, which is the Right Thing™ to type OO, what more with type inference.

Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science* 1, pp. 132–153, 1995b. https://pl.cs.jhu.edu/projects/type-constraints/papers/type-inference-for-recursively-constrained-types-and-its-application-to-oop.pdf . This paper presents I-SOOP, a simple language with polymorphic recursively constrained types, for which they present a type inference algorithm, and that is the Right Thing™ on top of which to type OO.

Emil Ernerfeldt. The Myth of RAM. (accessed 2025-12-01), 2014. https://www.ilikebigbits.com/2014_04_21_myth_of_ram_1.html . These blog posts explain why physics constraints random memory access to be actually in $O(\sqrt{N})$ rather than O(1), where N is the size of the working set. The "memory hierarchy" of caches, RAM, disk, remote storage, etc., makes for discrete jumps in latency as you exceed local capacity. The ultimate physical limit is ultimately a square root rather than cubic root to avoid collapsing into a blackhole.

Matthias Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming* 17(1), pp. 35–75, 1991. doi:10.1016/0167-6423(91)90036-W . A beautiful paper on what it means for a programming language to be more expressive than another. It is far from the last word on the topic, but it offers a simple, objective formal criterion for the expressibility of one language by another.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proc. Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pp. 48–59, 2002. https://users.cs.northwestern.edu/~robby/publications/papers/ho-contracts-icfp2002.pdf

Kathleen Shanahan Fisher. Type Systems for object-oriented programming languages. PhD dissertation, #f, 1996. https://www.researchgate.net/publication/2828333_Type_Systems_For_Object-Oriented_Programming_Languages

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*, pp. 270–289, 2006. https://www2.ccs.neu.edu/racket/pubs/asplas06-fff.pdf

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. 1998. doi:10.1145/268946.268961

Brian Foote and Joseph W. Yoder. Big Ball of Mud. In *Proc. PLoP*, 1997. http://www.laputan.org/mud/

Nate Foster, M. Greenwald, J. T. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 2007. https://www.cis.upenn.edu/~bcpierce/papers/lenses-full.pdf

Daniel P. Friedman, Christopher T. Haynes, and Mitchell Wand. *Essentials of Programming Languages*. 3rd edition. Massachusetts Institute of Technology, USA, 2008. doi:10.5555/1378240 . EOPL is a great classic about Programming Languages. However, its chapter on OO only has classes with single inheritance. EOPL barely touches multiple inheritance, "powerful, but ... problematic", in exercise 9.26, and prototypes in exercise 9.27-9.29. Disappointing on the OO front.

Richard P Gabriel, Jon L White, and Daniel G Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM* 34(9), pp. 29–38, 1991. https://www.researchgate.net/publication/220422719_CLOS_Integrating_Object-Oriented_and_Functional_Programming

Richard P. Gabriel. The Structure of a Programming Language Revolution. In *Proc. Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 195–214, 2012. https://www.dreamsongs.com/Files/Incommensurability.pdf . A magnificent paper that discusses how different people think about software (and everything else) in often incommensurable paradigms. In the illustrative example, old timers and Lispers approach programming with a "system" paradigm, whereas the academic and industrial mainstream since sometime in the 1980s has a very different "language" paradigm. Interestingly, the paper it rightfully excoriates for its authors' paradigmatic misunderstanding of OO due to their language paradigm, is also one of the best papers on the semantics of OO, the Bracha & Cook 1990. I suspect it was Cook who wrote the more paradigmatically foreign parts of the paper, for I personally know Bracha to very much follow the system paradigm (see his beautiful work on Newspeak). He still wrongly believes mixin inheritance to be more modular than multiple inheritance, though.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994. https://hillside.net/elements-of-reusable-object-oriented-software-book . The classic "Gang of Four" (GoF) book on Design Patterns. Much loved, much hated. To quote Rich Hickey, «Patterns mean "I have run out of language."» I.e. patterns are using fourth class entities because you can't (or otherwise fail to) afford automation and abstraction. Which is a bad thing to happen, but sometimes it does and then you're glad you have patterns.

Jacques Garrigue. Code reuse through polymorphic variants. In *Proc. Workshop on Foundations of Software Engineering*, 13, 2000. https://www.math.nagoya-u.ac.jp/~garrigue/papers/fose2000.html . Open Recursion, etc., is not just for extensible products, but also extensible variants. The requirement for the subclass's type to be a subtype is wrong-headed, even at the specification level, what more at the target level.

GitHub. The top programming languages. (accessed 2024-01-04), 2022. https://octoverse.github.com/2022/top-programming-languages

Joseph A Goguen. Sheaf Semantics for Concurrent Interacting Objects. *Mathematical Structures in Computer Science* 2(2), 1992. doi:10.1017/S0960129500001420

Ira P. Goldstein and Daniel G. Bobrow. Extending Object Oriented Programming in Smalltalk. In *Proc. Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pp. 75–81. Association for Computing Machinery, New York, NY, USA, 1980. doi:10.1145/800087.802792

Radu Grigore. Java Generics are Turing Complete. 2016. https://arxiv.org/abs/1605.05274

Carl Hewitt, Giuseppe Attardi, and Henry Lieberman. Security and Modularity in Message Passing. MIT Artificial Intelligence Laboratory, AI-WP-180, 1979. https://dspace.mit.edu/handle/1721.1/41147 . 16 years before Rees 1995, the authors use scoping as a security mechanism in their Actor system ACT1. They also implement both Prototype OO ("delegation") and Class OO ("inheritance"). Prototypes "increas[e] the modularity of the system"; classes allow "sharing of descriptions common to many actors". Actors have both When an actor delegates message handling to another, messages it does not handle itself result in a call to the other actor with a method "Buck_pass" that reifies the message and its context. Cites Simula, Smalltalk and Director (Kahn) "developed jointly with Kahn".

Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic Software Updating. In *Proc. Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pp. 13–23. Association for Computing Machinery, New York, NY, USA, 2001. doi:10.1145/378795.378798

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` . This seminal paper introduces classes and subclasses, and many other notions now familiar and ubiquitous, including his infamous "million dollar mistake", null. Hoare calls "objects" and "attributes" high-level entities being modeled, "classes" their types, "records" and "fields" the low-level representations of the previous as "consecutive words of computer store", and "record classes" their types. "References" are typed pointers, and "null" is a special reference for when a correspondence is partial rather than total. "Discrimination" is a form of type-safe pattern-matching between subclasses. Hoare considers recursive data structures, yet does not anticipate that for them, subclassing will differ from subtyping.

Daniel H. H. Ingalls. The Smalltalk-76 programming system design and implementation. In *Proc. the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pp. 9–16. Association for Computing Machinery, New York, NY, USA, 1978. doi:10.1145/512760.512762

Ecma International. *ECMAScript 2015 Language Specification*. 6th edition. Geneva, 2015. `http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf` . ECMAScript is the official name of the standard everyone more or less follows for JavaScript. This 2015 version introduces ECMAScript 6, that includes many improvements over previous versions, including a builtin notion of classes, when the only form of OO previously was prototypes.

Bart Jacobs. Objects and Classes, Co-Algebraically. In *Proc. Object Orientation with Parallelism and Persistence*, pp. 83–103, 1995. doi:10.5555/261075.261479 . Cook positively mentioned Jacobs in private email, and cites him in Cook 2012, as justification why OO doesn't need inheritance. But this paper is actually not OO at all, but relational modeling with Category Theory. In the last paragraph of section 2, Jacobs says that "Ai, Bi, Ci are (constant) sets." All the negation of actual OO is concentrated in a single word, what more in parentheses, lost in the middle of the article, that restricts classes to the non-recursive subset where the NNOOTT applies. Wow. But that doesn't require malice, only ignorance. Jacobs cites Goguen, who was also using Category Theory; but Goguen was using it to do rewrite logic and software specification with the trappings of OO. Jacobs does just relational data modeling.

Bart Jacobs. Inheritance and Cofree Constructions. In *Proc. European Conference on Object-Oriented Programming*, 1996. doi:10.1007/BFb0053063 . Jacbos claims to formalize inheritance, but actually goes full NNOOTT. At least, he understands what he is doing, for in the second paragraph of section 2, he explicitly disallows open recursion through the module context: «where A and B are constant sets, not depending on the "unknown" type X (of self)». But he doesn't understand why that's a wrong thing to do when you claim you're doing OO.

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/`

Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of object-oriented programming* 1(2), pp. 22–35, 1988. `http://laputan.org/drc/drc.html`

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*, UODIGS '76, pp. 37–43. Association for Computing Machinery, New York, NY, USA, 1976. doi:10.1145/1024273.1024278

Kenneth Michael Kahn. Creation of computer animation from story descriptions. PhD dissertation, MIT, 1979a. https://dspace.mit.edu/handle/1721.1/16012 . This paper is about the higher-level system Ani, that generates turtle graphics plans on top of the actor-based object system Director (see AIM-482b). While Ani is a constraint solver based on a knowledge representation of the film being directed: very crude 2d film with polygons as "performers" that move toward, away from, around or between each other, at various speeds appropriate for the action, as would humans on a stage. Kahn cites Bobrow & Winograd, as well as Kay, Hewitt, Lieberman, and many more (Licklider, Catmull!)

Kenneth Michael Kahn. Director Guide. MIT, AIM-482b, 1979b. https://dspace.mit.edu/handle/1721.1/6302 . This paper, revised from a previous 1978 edition, presents Director, the earliest Lisp OO system as such, where inheritance exists in the context of programming and not just knowledge representation from which it evolved (see Kahn 1976). Dirctor by then had been extracted from Kahn's wider animation project built atop it. Director implements single inheritance by message passing on top of an Actor system. It has an early form of call-next-method called continue-asking (p.40). The Actor team would later adapt Kahn's system, see Hewitt 1979. Kahn cites Smalltalk and Actors as an influence for Director, but not KRL (Kahn does cite KRL in other papers).

Samuel N. Kamin. Inheritance in Smalltalk-80: A Denotational Definition. In *Proc. POPL*, 1988. doi:10.1145/73560.73567 . With Reddy 1988, one of the very first formal semantics for OO, with a simplified static model of Smalltalk-80 with single inheritance. Kamin uses syntactic rewrite rules, that he directly translated into a running ML program. This paper deserves much praise for being a first of its genre, translating a foreign paradigm in the vocabulary of FP, and being constructive with an actual running implementation. In a way, the paper does too much by trying to have a realistic subset of Smalltalk, and too little by spending too little time explaining its dense model of inheritance, indeed as the fixpoint of the hierarchy of class definitions. But that's OK, because Reddy 1988 is a very complementary paper that explains the concepts behind Kamin.

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. Clarification of "object-oriented" (email). 2003. https://www.purl.org/stefan_ram/pub/doc_kay_oop_en

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

Alan C. Kay. The Early History of Smalltalk. In *History of programming languages—II* 28(3), pp. 511–598. Association for Computing Machinery, New York, NY, USA, 1993. doi:10.1145/155360.155364 . Great read about Smalltalk. Inheritance is discussed in section 11.5.1: Simula-style inheritance was deliberately left out of Kay's work on FLEX and early Smalltalk. Kay credits Larry Tesler (slot inheritance), Henry Lieberman (delegation), and Bobrow and Winograd's KRL (Frames), for the invention of Smalltalk inheritance.

Shriram Khrisnamurthi. *Programming Languages: Application and Interpretation*. 1st edition. Massachusetts Institute of Technology, USA, 2008. `https://plai.org/` . This is a great book about Programming Languages in general, and I love SK. SK introduces objects as closures, and even prototypes, in chapter 10. But one thing he just doesn't get is multiple inheritance (10.3.3). In 10.3.5 he introduces mixins as a mechanism on top of first-class single-inheritance classes, which certainly works, but to me feels like putting the cart before the ox. Great stuff outside OO, and even the OO is decent, but in the end, like many academics, SK doesn't grok OO.

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

Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. 2005. `https://arxiv.org/abs/cs/0509027`

Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. 229, 1987. doi:10.7146/dpb.v16i229.7578 . Beta is the successor of Simula. It unifies classes, procedures, functions and types as a single abstraction mechanism called "patterns", that may be akin to prototypes. It uses the same concatenation semantics as Simula for inheritance, with the infamous "inner" keyword wherein parents specify where the child's code will go, rather than the child controlling how it makes use of the parent-provided value. In the end, each can be expressed in terms of the other one, but it seems to me that the one with the better ergonomics did win. The paper is hard to read. The authors are in their own world, and fail to try to reduce their semantics to something simple in terms of logic, or anything understandable by others.

Julia L. Lawall and Daniel P. Friedman. Embedding the Self Language in Scheme. Computer Science Department, Indiana University, 1989. `https://legacy.cs.indiana.edu/ftp/techreports/TR276.pdf`

Primo Levi. *Se questo è un uomo*. De Silva, 1947. This book impressed me when I read it as a young man (in a French translation), and the words "Hier ist kein 'Warum'!" (Here there is no 'Why'!) still haunt me. A 1959 translation to English by Stuart Woolf was published as "If this is a man", or in later editions "Survival in Auschwitz".

Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proc. OOPLSA*, pp. 214–223, 1986. `https://web.media.mit.edu/~lieber/Lieberary/OOP/Delegation/Delegation.html`

Barbara Liskov. Data Abstraction and Hierarchy. In *Proc. OOPSLA '87*. ACM, 1987. doi:10.1145/62138.62141 . In her invited talk at OOPSLA, Liskov in section 3 promptly dismissed inheritance of implementation as "violating encapsulation", a quick hack that doesn't fit her semantic model focused on subtyping. She obviously does not understand the purpose and semantics on inheritance, falls deep into the NNOOTT, and with inheritance of implementation dismisses any actual OO. And yet, the same paper first introduces a variant of her now famous "Liskov substitution principle". So, is it a bad paper? Was it wrong to invite her at OOPSLA? Not at all. She is a great influential researcher who deserved her Turing Award. But was her talk good about OO? Not at all either.

William M. Lonergan and Paul A. King. Design of the B5000 System. *Datamation* 7(5), pp. 28–32, 1961. https://people.eecs.berkeley.edu/~kubitron/courses/cs252-S12/handouts/papers/b5000.pdf . The paper presents the Burroughs B5000 computer architecture, one of the sources of inspiration cited by Alan Kay. It features: a push-down stack (with "right-hand" Polish notation that prefigures FORTH!); "independence of addressing" (memory position-independence for both code and data) through a "program reference table" (relocatable table for indirect addressing, prefigures not just global linker tables, but also an object's virtual dispatch table).

Augusta Ada Lovelace. Notes [on translation of Luigi Federico Menebrae's paper on Babbage's Analytical Engine]. In *Richard Taylor's Scientific Memoirs* pp. 666–731, 1843. https://www.google.com/books/edition/Scientific_Memoirs_Selected_from_the_Tra/qsY-AAAAYAAJ?gbpv=1&pg=PA666 . This paper is the verily the foundation of all Computer Science, which should rightfully be called by the name Lovelace baptized it: "the science of operations". Ada Augusta King, Countess of Lovelace, translates to English an article (in French) about the Babbage Analytical Engine, by Luigi Menebrae, who would later become Prime Minister of newly-unified Italy; and her translator's notes are fantastic. Only her initials appear (A.A.L, at one point mistyped A.L.L.), because a woman, especially a noble one, shall not write, especially not on such lowly topics. But my, her understanding of "operations", the general notion of pure computation, is better than that of most computer scientists of today. "First the symbols of _operation_ are frequently _also_ the symbols of the _results_ of operations. We say that symbols are apt to have both a _retrospective_ and a _prospective_ signification. [...]" She groks both that there is a conflation of multiple meanings, and that it is good, yet can lead to confusion if not made explicit: past vs future, passive vs active, values vs continuations. She also anticipates symbolic computing: "symbolical" vs "numerical" "results" or "data" (outputs or inputs); and the equivalence of code and data: "the symbols of _numerical magnitude_, are frequently _also_ the symbols of _operations_". She understands that programmability (she lacks the word) makes the Analytical engine vastly superior to the finite-configuration Difference Engine, indeed makes it "the executive right hand of abstract algebra". She also confronts the problem that a variable may be bound to multiple values during an execution, and proposes an elegant solution, using a superscript to the left of a variable to number each successive binding, turning them into actual mathematical variables again. Way more advanced than most computer scientists even today. Prescient.

Yuri I. Manin. *Mathematics as Metaphor*. American Mathematical Society, 2007. doi:10.1093/philmat/nkn013 . This edition includes a nice Foreword by Freeman Dyson.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proc. Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pp. 124–144. Springer-Verlag, Berlin, Heidelberg, 1991. doi:10.1007/3540543961_7

John C Mitchell and Gordon D Plotkin. Abstract Types have Existential Type. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10(3), pp. 470–502, 1988. doi:10.1145/44501.45065

Eugenio Moggi. Notions of computation and monads. *Information and computation* 93(1), pp. 55–92, 1991. doi:10.1016/0890-5401(91)90052-4

David A. Moon. Object-Oriented Programming with Flavors. *SIGPLAN Not.* 21(11), pp. 1–8, 1986. doi:10.1145/960112.28698

Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real World Haskell*. O'Reilly Media, 2008. doi:10.5555/1523280 . This book demonstrates how Haskell by 2008 had grown into a practical language usable to write "Real World" applications.

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

Bruno C. d. S. Oliveira. The Different Aspects of Monads and Mixins. 2009. https://www.cs.ox.ac.uk/publications/publication2965-abstract.html

Russell O'Connor. Polymorphic Update with van Laarhoven Lenses. (accessed 2025-12-01), 2012. http://r6.ca/blog/20120623T104901Z.html Classic essay on functional lenses.

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 . A truly classic paper on Modularity.

Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor Optics: Modular Data Accessors. *The Art, Science, and Engineering of Programming* 1(2), 2017. doi:10.22152/programming-journal.org/2017/1/7

Benjamin C. Pierce. *Types and Programming Languages*. 1st edition. MIT Press, 2002. https://www.cis.upenn.edu/~bcpierce/tapl/ . TAPL treats objects in chapters 18, 19, 24, 32. 18 has a gradual approach to implementing not-quite-objects, and adding features until the reader has actually implemented objects. It's clever, but a bit long-winded. See my note on Pierce & Turner 1993 regarding letting the user provide a rep type. 32.10 Pierce does discuss the actual solution, recursive data types, but leaves it as an exercise after pointing to the bibliography. Pierce is technically masterful, but my impression is that he doesn't understand the purpose of OO, because it is a foreign paradigm (see Gabriel 2012); and so not only he cannot drill into the essence of it, he spends more time on the non-essentials (imperative programming), and leaves the actually interesting essential (recursive types) out of his text, using a shortcut (his rep type) to survive without it.

Benjamin C. Pierce and David N. Turner. Simple Type-Theoretic Foundations for Object-Oriented Programming. 1993. https://www.cambridge.org/core/journals/journal-of-functional-programming/article/simple-typetheoretic-foundations-for-objectoriented-programming/5C18E2E055B028F7214FBB183701830E

Kent Pitman. Common Lisp HyperSpec. (accessed 2026-01-06), 1996. https://www.lispworks.com/documentation/HyperSpec/Front/index.htm . The Common Lisp HyperSpec, or CLHS, is a carefully digitized version of the ANSI Common Lisp standard, the final draft of which was approved by X3J13 on December 8, 1994. It is fully clickable, includes the (non-binding, but enlightening) "issue writeups" that explain some of the design decisions, etc. Objects are described in chapter 7. Importantly, the document is meant as a reference, and not as a tutorial. You will have a hard time learning how to use Common Lisp in general or CLOS in particular by reading this document. But you can read tutorials, look at plenty of working examples in free software libraries, and then make sense of the finer details by reading the CLHS. Practical Common Lisp, the Common Lisp Cookbook, (or I'm told the OOP in CL by Keene) might be better to start learning, or CLtL2, AMOP.

Sergiy Prykhodko, Natalia Prykhodko, and Smykodub Tatiana. A Statistical Evaluation of The Depth of Inheritance Tree Metric for Open-Source Applications Developed in Java. *Foundations of Computing and Decision Sciences* 46, pp. 159–172, 2021. doi:10.2478/fcds-2021-0011

Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996. doi:10.1017/CBO9781139172974 . This classic book will demystify the implementation of languages in the Lisp or Scheme family. Great for theory and practice alike.

Uday Reddy. Objects as Closures - Abstract Semantics of Object Oriented Languages. In *Proc. ACM Symposium on LISP and Functional Programming*, pp. 289–297, 1988. doi:10.1145/62678.62721 . This paper is a great sequel and complement to Kamin 1988. Where Kamin implemented a complete workable subset of Smalltalk-80, Reddy instead decomposes the concepts of OO into simpler concepts that he gradually extends. Reddy doesn't provide running code, though he says Kamin implemented Reddy's variant as well as Kamin's own. Together a great pair of paper.

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*, pp. 114–122, 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, USA, 1995. doi:10.5555/921818 . Fantastic classic paper, Rees's thesis introduces his "W7" security kernel, leveraging scoping as a natural mechanism for object capabilities based security.

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. ASDF 3, or Why Lisp is Now an Acceptable Scripting Language. In *Proc. European Lisp Symposium*, 2014. `https://github.com/fare/asdf3-2013`

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. Climbing Up the Semantic Tower — at Runtime. In *Proc. Off the Beaten Track Workshop at POPL*, 2018a. `https://github.com/fare/climbing`

François-René Rideau. Reconciling Semantics and Reflection. 2018b. `https://bit.ly/FarePhD` Unpublished because undefended. First draft completed in 2018, yet still not in a publishable state as of 2025.

François-René Rideau. Gerbil-POO. (accessed 2021-04-06), 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`

Claudio V. Russo. Types for Modules. PhD dissertation, University of Edinburgh, 1998. `https://era.ed.ac.uk/handle/1842/385` LFCS Thesis ECS-LFCS-98-389

Lee Salzman and Jonathan Aldrich. Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model. In *Proc. ECOOP*, 2005. `https://www.cs.cmu.edu/~aldrich/papers/ecoop05pmd.pdf`

Peter Simons. Nixpkgs fixed-points library. 2015. https://github.com/NixOS/nixpkgs/blob/master/lib/fixed-points.nix . Nix implements prototype OO in two functions. Impressive. Seeing that after having used Jsonnet (that implements an equivalent object system the hard way in C++) is what made OO click for me. Peter doesn't call his extension system "OO" and didn't know about Prototype OO when he wrote that (but had been influenced by others who did); but his extensions are used exactly like objects are in Jsonnet, for sensibly the same purposes—configuring complex systems in a modular extensible way.

Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. In Greg Butler and Stan Jarzabek (Ed.), *Proc. International Symposium on Generative and Component-Based Software Engineering*, pp. 164–178. Springer. Berlin, Heidelberg, 2000. doi:10.1007/3-540-44815-2_12 . This paper very nice paper implements proper mixin inheritance on top of C++ using templates. Note that it does not try to implement ancestry linearization, and therefore does not implement flavorful multiple inheritance; though such linearization could probably be done on top of their mixin inheritance.

Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proc. Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '86, pp. 38–45. Association for Computing Machinery, New York, NY, USA, 1986. doi:10.1145/28697.28702

James L Stansfield. COMEX: A Support System for a Commodities Expert. MIT, AIM-423, 1977. https://dspace.mit.edu/handle/1721.1/6281 . An early expert system for Commodities, written in Lisp, and organizing data in a knowledge base of frames. The paper is the earliest one to mention single and multiple inheritance, p.12 and p.13, about the frame representation of the COMEX data.

Guy Steele. *Common Lisp: The Language, 2nd edition*. Digital Press, USA, 1990. doi:10.5555/1098646 . This book, known as CLtL2, presents a snapshot of the work of X3J13, the committee that was standardizing ANSI Common Lisp. While there are discrepancies between the contents of this book and the actual standard, a lot of the material is already there in essentially its final form. CLtL2 contains many features that were not included in the standard due to lack of consensus; nevertheless, many implementations also adopted these features in some form. Still, the standard is the actual reference document. On the other hand, the standard is dry, whereas CLtL2 is much more didactic. CLtL2 is thus a better introduction to the concepts, though it is not authoritative. There was also a first edition of CLtL, back in 1984, but the language evolved a lot between the two editions, and the first edition came years before the Object System was designed. Objects are described in chapter 28 of CLtL2.

Bjarne Stroustrup. Multiple inheritance for C++. *Computing Systems* 2(4), pp. 367–395, 1989. https://www.usenix.org/legacy/publications/compsystems/1989/fall_stroustrup.pdf

Ivan E. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In *Proc. Proceedings of the May 21-23, 1963, Spring Joint Computer Conference*, AFIPS '63 (Spring), pp. 329–346. Association for Computing Machinery, New York, NY, USA, 1963. doi:10.1145/1461551.1461591

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

Warren Teitelman. PILOT: a step toward man-computer symbiosis. PhD dissertation, MIT, 1966. https://dspace.mit.edu/bitstream/handle/1721.1/6905/AITR-221.pdf . This thesis is very rich in ideas, but the one that matters most for OO is Teitelman's introduction of ADVISE, a facility a function with new behavior: advices to be run before or after the function, or somehow wrapping it, that you add or remove dynamically. This would later inspire Cannon 1979's method combinations.

David Ungar and Randall B. Smith. Self. In *Proc. Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III. Association for Computing Machinery, New York, NY, USA, 2007. doi:10.1145/1238844.1238853 . Section 4.2 notably describes the failure of their "sender path" attempt at multiple inheritance, after which they revert to the "conflict" paradigm.

Dimitris Vyzovitis. Gerbil Scheme. (accessed 2025-04-06), 2016. https://cons.io/

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL*, pp. 60–76. ACM, 1989. doi:10.1145/75277.75283

Daniel Weinreb and David Moon. Lisp Machine Manual. 3rd edition, 1981. http://bitsavers.org/pdf/mit/cadr/chinual_3rdEd_Mar81.pdf . Also known as the "Chinual", this legendary book explains the interface of the Lisp Machine, then commercialized under license by Symbolics, Inc., the first .com. Chapter 20 presents how to use Flavors, what are its concepts and its API; but it does not include all the rationale from the Flavors paper.

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

Wikipedia. Fundamental theorem of software engineering. 2025a. https://en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering

Wikipedia. Prototype-based programming. 2025b. https://en.wikipedia.org/wiki/Prototype-based_programming

Terry Winograd. Frame Representations and the Declarative/Procedural Controversy. In *Representation and Understanding* Daniel G. Bobrow and Allan Collins (Ed.), pp. 185–210. Morgan Kaufmann, San Diego, 1975. doi:10.1016/B978-0-12-108550-6.50012-4 . This paper has so far as I can tell the earliest use of "inheritance" in a context that directly applicable to OO. Winograd also inspired Kahn and Borning for their Prototype OO systems, and worked with Bobrow whose team was next to Kay's, etc. The word "inherit" does appears in other papers from MIT AI researchers at the time (e.g. Hewitt), but it is unclear who talks about it first and gets it from where. Winograd talks about "inheritance of properties", p. 197., to compute properties of a frame based on its hierarchy. Frame hierarchies are a form of multiple inheritance, though that word isn't coined. The exact resolution algorithm is not explained. Frames are not quite data, not quite code, but something in between, "knowledge representation"; Lisp, being a system more than a programming language, blurs the lines. The use of the word "inheritance" is not fully formal, though. Indeed, I can see at least four kinds of ways of naming things (beware though I'm making up this nomenclature on the spot): (1) evocation: neither name nor concept owns the other, yet an allusion is made; (2) distinction: the name distinguishes the concept from others, but isn't owned by it; (3) identification: name and concept own each other, though the limits of the concept may not be clear; (4) definition: the essence of the concept being named is finally well understood. For "inheritance", evocation may have happened in Minsky 1974, distinction in Winograd 1975, and identification around 1976 quite possibly by Bobrow or Kay (and Kay later says Larry Tesler insisted on implementing "slot inheritance" the way he had done in previous desktop publishing projects). Informal practical definition would happen in the early 1980s, and more formal theoretical definitions around 1990 with e.g. Cook & Bracha, refining their immediate predecessors like Kamin, Reddy, Cook.