# Categories for PL

Notes from CS7480 (Fall 2025)

*Steven Holtzen and John M. Li*

*with Shubh Agrawal, Jialu Bao, Bex Golovanov, Mingtong Lin, Joseph Rotella, and Michael Zhang*

MAY 26, 2026

# *Contents*

# *Preface*

This preface is written by Steven.

These are notes from a course on category theory for programming languages researchers. I would say that these notes, and this course, are developed in anger: I have been forced to learn and understand a bit about category theory due to its inevitable usefulness. I would not describe myself as a category theorist, and this class is quite unlike any I've ever tried to teach before. My goal in this class is to tease out what I find inevitable about category theory, and present it in the way that makes the most sense to me. Of course, this isn't possible without John, who is the one who really understands everything here.

Many programming languages researchers and students have encountered category theory at one point or another. Probably most see it for the first time when they are learning Haskell, when they see words like "monad" and "functor." Perhaps others who are more mathematically inclined see it when they are learning about the lambda calculus and hear about Scott, Smyth, and Plotkin's famous developments of the model theory of the untyped lambda calculus [34]. You're here because you probably saw category theory somewhere and wondered why it was needed, or perhaps were skeptical that it was really necessary.

For me, the first time I saw category theory appear in my own research that forced me to stop and understand it was Heunen et al.'s quasi-Borel spaces paper.[1] This was during my second year of grad school, and I have to admit I found it extremely intimidating. This paper was a terrifying combination of (1) seeming very important to my immediate research, and (2) being utterly and completely incomprehensible. In a nutshell, this paper introduced a denotational semantics for a higher-order probabilistic programming language with continuous probability distributions called *quasi-Borel spaces* (QBS). I was – and still am! – a probabilistic programming languages researcher, so this result seemed pivotal: I like functions! However, at the time, it wasn't even obvious to me *why this was a hard thing to do in the first place!* If you can't understand the motivation for a problem, good luck understanding its solution.

[1] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2017

The core challenge, helpfully provided in the second paragraph of the paper, is:

> "Programs in these languages may combine higher-order functions and continuous distributions, or even define a probability distribution on functions. But the standard measure theoretic formalization of probability theory does not handle higher-order functions well, as the category of measurable spaces is not cartesian closed." [14]

This made *absolutely no sense to me*. The only citation is to a paper from 1961, called "Borel structures for function spaces" [3], which was equally (if not even more!) incomprehensible than the current paper I was trying to understand. I was out of my depth and had to move on, and I did not understand this paper for many years.

At the time the QBS paper seemed like it came like a bolt from the blue and really shook my confidence in my own ability to do research, but over the past 8 years I've come to realize that, in some sense, this seemingly tour-de-force alien paper was an almost inevitable combination of simple ideas. It turns out that the idea for quasi-Borel spaces was born not in probability but in a seemingly totally disparate corner of mathematics: differential geometry and topology. Last year I was having a conversation with Max New about QBS, who told me that the setup had occurred to him in grad school, but he wasn't motivated to fully develop it due to its seemingly straightforward relationship to diffeological spaces: these results are so similar, structurally, that to a trained expert the monstrous QBS paper almost seems *too trivial!* This is our first answer to the question of *why category theory?* – it makes precise the analogies that exist between disparate areas of mathematics, allowing you to solve different problems using very similar machinery.

If this was the only time I encountered category theory I would not have become invested enough in it to attempt to teach a course on it. But, it continues to appear. My subsequent encounters with category theory were driven primarily by my PhD. student John, who is helping me teach this class and write these notes. This story begins with Lilac [22], a probabilistic separation logic that we developed during the first two years of John's PhD. Before we get into that we need to briefly discuss separation logics – I promise we'll get to category theory.

Separation logics are logics that describe resources. They are widely used within programming languages to model how resources like memory flow around programs. Concretely, consider the following C program:

```
void foo(int* x, int* y) {
  for(int i = 0; i < 3; i++) {
    *x += *y;
  }
```

```
}
```

One might look at `foo` and assume that it is very inefficiently written: it could be optimized to simply add `3*y` to `x`. But, *what if `x` and `y` point to the same location in memory* (i.e., they are aliases)? Then, this optimization is no longer valid, since `y` is also mutating on each iteration of the loop. Since the C compiler must be pessimistic, it cannot perform this optimization.

Separation logics give us a language for describing memory ownership so that we can tell the compiler that these pointers do not alias. If we want this optimization to be valid, then `foo` must have a *precondition* that asserts that the two arguments must not be aliases of each other. In separation logic, this precondition can be written as:

$$[\mathtt{x} \mapsto -] * [\mathtt{y} \mapsto -]$$

The $*$ is called the *separating conjunction*: it states that the two propositions $[\mathtt{x} \mapsto -]$ (read "`x` points to some location") and $[\mathtt{y} \mapsto -]$ hold of *disjoint* portions of the heap, and hence `x` and `y` cannot be aliases of one another.

However, memory is not the only kind of resource in a program: *randomness* is also a natural kind of resource. In probabilistic programs, we might like to express propositions like:

$$[x \sim \mathtt{Bern}\ 1/2] * [y \sim \mathtt{Bern}\ 1/2]$$

This would express that two variables $x$ and $y$ are *independent* Bernoulli random variables. Beyond probability, it's clear that programs might have many other notions of resources, like network sockets, mutex locks, etc. Must we develop *from-scratch* separation logics for each of them?

This leads us to the second answer to the *why category theory?* question: it makes certain choices that seem ad-hoc *forced*. We will see how there is a categorical description of the separating conjunction that can be *calculated* using something called the "Day convolution" [11, 6]. If your interpretation of $*$ arises from this construction, then you get *for free* that it satisfies many other well-formedness requirements for separating conjunction. This shows how category theory can bring clarity to this proliferation of interactions by highlighting which decisions you've made are actually *new* and which are *consequences*: it forces your hand, which is very useful when you don't know where your hand is supposed to go.

We are foreshadowing here, but we will return to the Day convolution later on in the course once we've worked up to it.

At this point, I'm convinced some familiarity with category theory is a worthwhile investment for anyone doing research in programming languages, especially those working with exotic effects like probability or those working with program logics. I also have some

hubris: I believe that it is not so hard to understand, especially if one is sufficiently motivated by some of the applications that we will attempt to highlight as we go. My target audience for this class is past me from 2017 seeing an important paper in their area that feels terribly out of reach because of its fluent application of categorical ideas. I hope that we can present some of these ideas clearly enough for you to see the inevitability, rather than be drowned by the unintelligability, of these elegant ideas.

# 1
# *Why Categories?*

One of the most common places one encounters category theory is in semantics. To illustrate why categories appear here and what they are typically used for, we'll consider a concrete example that every programmer has encountered: we want to establish which operations change the behavior of our programs and which are irrelevant.

Consider a tiny programming language Calc consisting of let-bindings, pairs, and Booleans:

$$M, N ::= \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N \mid x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ O \mid \langle M, N\rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M$$
$$A, B ::= A \times B \mid \mathsf{Bool} \qquad \text{(Calc)}$$

Calc has a simple type system:

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash M : \mathsf{Bool} \qquad \Gamma \vdash N : A \qquad \Gamma \vdash O : A}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ O : A} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \times B}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{fst}\ M : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{snd}\ M : B}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N : B}$$

Here's a simple transformation one might want to justify: two let-bindings can be interchanged if their definitions do not mention each other.

$$\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ \Big(\mathsf{let}\ y\ \mathsf{be}\ N\ \mathsf{in}\ \langle x, y\rangle\Big)$$
$$\stackrel{?}{=}$$
$$\mathsf{let}\ y\ \mathsf{be}\ N\ \mathsf{in}\ \Big(\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ \langle x, y\rangle\Big)$$

Intuitively, this equation holds because CALC programs have no side effects. We can justify this intuition by proving that these two programs have the same meaning. For this, we need to give a semantics to CALC programs.

## 1.1 *A Denotational Semantics for* CALC

The first step in designing a semantics is to choose a **metalanguage** for stating it. The metalanguage should be a formal language that is unambiguous to describe and use, because its meaning will be taken for granted when we interpret programs. For a simple language like CALC, a natural choice is the language of sets and functions.

Using this metalanguage we can give a **denotational semantics** to CALC programs that describes the meaning of syntactic programs using our choice of metalanguage. Types will be interpreted as sets:

$$[\![\mathsf{Bool}]\!] = \{0, 1\} \tag{1.1}$$

$$[\![A \times B]\!] = [\![A]\!] \times [\![B]\!] \tag{1.2}$$

The Boolean type denotes the two-element set $\{0, 1\}$, and the product type $A \times B$ denotes the Cartesian product of the denotations of $A$ and $B$. Using this interpretation of types, we can give an interpretation to typing contexts:

$$[\![\bullet]\!] = \{\star\}, \qquad [\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!]$$

The empty context $\bullet$ is interpreted as the singleton set $\{\star\}$, and context extension by Cartesian product. Taken together, this interpretation defines $[\![\Gamma]\!]$ to be the set of *substitutions of shape* $\Gamma$, whose elements are functions that map variables $x$ in $\Gamma$ to elements of $[\![\Gamma(x)]\!]$.

Using this interpretation of types and contexts, we can interpret CALC terms $\Gamma \vdash M : A$ as functions $[\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$:

Notational note: we use "$\gamma \mapsto v$" to mean the function that takes each substitution $\gamma$ to value $v$, and $\pi_1$ and $\pi_2$ for the two projections out of a Cartesian product. As usual, we also use $\pi_x$ to magically pick out the component of an environment $\gamma$ that corresponds to a given variable $x$.

$$\begin{aligned}
[\![\mathsf{true}]\!] &= \gamma \mapsto 1 \\
[\![\mathsf{false}]\!] &= \gamma \mapsto 0 \\
[\![\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ O]\!] &= \gamma \mapsto \begin{cases} [\![N]\!]\,\gamma, & [\![M]\!]\,\gamma = 1 \\ [\![O]\!]\,\gamma, & [\![M]\!]\,\gamma = 0 \end{cases} \\
[\![\langle M, N \rangle]\!] &= \gamma \mapsto ([\![M]\!]\,\gamma, [\![N]\!]\,\gamma) \\
[\![\mathsf{fst}\ M]\!] &= \gamma \mapsto \pi_1([\![M]\!]\,\gamma) \\
[\![\mathsf{snd}\ M]\!] &= \gamma \mapsto \pi_2([\![M]\!]\,\gamma) \\
[\![x]\!] &= \gamma \mapsto \pi_x(\gamma) \\
[\![\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N]\!] &= \gamma \mapsto [\![N]\!]\,\gamma[x \mapsto [\![M]\!]\,\gamma]
\end{aligned}$$

We can use this semantics to validate the reordering from earlier, as both side of the equation denote the same function.

$$[\![\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ (\mathsf{let}\ y\ \mathsf{be}\ N\ \mathsf{in}\ \langle x, y\rangle)]\!] = \gamma \mapsto ([\![M]\!]\,\gamma, [\![N]\!]\,\gamma) = [\![\mathsf{let}\ y\ \mathsf{be}\ N\ \mathsf{in}\ (\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ \langle x, y\rangle)]\!]$$

But, notice that there are subtleties! We are relying on our metalanguage's notion of equality of functions here to determine equality. Two functions $f, g : A \to B$ are considered equal if they are *extensionally equal*, meaning that for all $a \in A$, it is the case that $f(a) = g(a)$. If we carefully unpacked these two semantics, we would see that the functions $[\![M]\!]$ and $[\![N]\!]$ are composed in different orders. Under the semantics of sets, this order of composition does not matter since it does not change the input-output behavior.

## 1.2 *The Need for Generality*

To sum up what just happened: in order to prove a natural program equivalence involving reorder of let-bindings, we gave a denotational semantics to Calc in terms of sets and functions. Then, a property of the metalanguage – extensional equality of functions – directly implied that the rewriting was semantics-preserving.

This style of denotational reasoning worked quite well for validating program equivalences, which begs the question: *how well does it generalize* to other kinds of programming languages and analyses? There are several interesting dimensions along which we may want to generalize this kind of argument:

1. To languages with more interesting kinds of effects, such as randomness, nondeterminism, or mutable state;
2. To languages with more interesting features, such as higher-order functions or more interesting type systems.

We will see that it is actually quite inconvenient to use sets and functions as a metalangauge, which will be a central motivation our development of category theory as an alternative and more flexible metalangauge for giving semantics to programs (and other things, like program logics). But before we can see this, let us first see an example of each of these kinds of generalization.

## 1.3 *Generalization 1: Probabilistic Effects*

Let's see an example of generalizing to languages with an interesting probabilistic effect. Consider the following little *probabilistic programming language* called TinyPPL that extends Calc with a primitive flip that generates random Booleans:

$$M, N ::= \cdots \mid \mathsf{flip} \qquad \text{(TinyPPL)}$$

Probabilistic programming languages denote probability distributions. For instance, here is the denotation of flip:

$$[\![\mathsf{flip}]\!] = [\![\mathsf{Bool}]\!] \to [0,1]$$
$$[\![\mathsf{flip}]\!] = b \mapsto \begin{cases} 1/2 & \text{if } b = \mathsf{true} \\ 1/2 & \text{if } b = \mathsf{false} \\ 0 & \text{otherwise.} \end{cases}$$

We are again interested in proving natural equivalences such as the one we saw earlier where let-bindings can be reordered (recall that $N$ does not refer to $x$):

$$\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ \Big(\mathsf{let}\ y\ \mathsf{be}\ N\ \mathsf{in}\ \langle x, y\rangle\Big)$$
$$\stackrel{?}{=}$$
$$\mathsf{let}\ y\ \mathsf{be}\ N\ \mathsf{in}\ \Big(\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ \langle x, y\rangle\Big)$$

Intuitively, it seems like this ought to be a semantics-preserving rewrite for a very similar reason to the fact that an identical rewrite is true for Calc. But, TinyPPL does not have an identical denotation, and so it seems like it requires a from-scratch proof to establish that this rewriting is sound.

Clearly this reordering is a general phenomenon we would like to study across many languages with different semantic interpretations. Category theory gives us a name for property of a semantic domain: it is called **commutativity**. Moreover, category theory gives us language for describing properties of a wide array of effects: probability is an example of a **monad**, and this monad is commutative and hence such rewritings are valid for *all commutative monads*.[1]

[1] This commutativity property was studied first by [35]; it follows essentially the order of summation can be interchanged. In the continuous case, this fact is true by Fubini's theorem.

## 1.4 *Generalization 2: Higher-order functions*

Functional languages have more than let-bindings and pairs. For instance, OCaml has higher-order functions, sum-types, and pattern-matching. These features can be difficult to specify, and category theory gives us a canonical way of designing these specifications.[2]

[2] Indeed, the C in OCaml stands for "categorical" [8].

We will illustrate this principle by higher-order functions:

$$\begin{aligned} A, B &::= \mathsf{Unit} \mid A \times B \mid A \to B \\ M, N &::= \langle\rangle \mid \langle M, N\rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \mid \lambda\, x.\, M \mid M\, N \mid x \end{aligned} \qquad \text{(STLC)}$$

The **function type** $A \to B$ denotes functions from $A$ to $B$. Functions are introduced using lambda abstraction rule $\lambda\, x.\, M$, defines a function with free variable $x$. Functions are eliminated with the lambda application rule $M\, N$, which calls the function $M$ with argument $N$.

Following our original setup with CALC, we can give a denotational semantics to STLC by associating each type with a set and each term with a function between sets. For functions and function applications, this looks like:

$$[\![\lambda\, x.\, M]\!] = \gamma \mapsto (v \mapsto [\![M]\!]\,(\gamma[x \mapsto v]))$$
$$[\![M\, N]\!] = \gamma \mapsto ([\![M]\!]\,\gamma)([\![N]\!]\,\gamma)$$

You should notice that it is not obvious that this interpretation is valid: it remains to be argued that $(v \mapsto [\![M]\!]\,(\gamma[x \mapsto v]))$ is itself representable as an element of a set. This is true because it is possible to represent the set of all functions between two sets as a set: a function can be represented as its *graph* (i.e., a set of input-output pairs), which is itself a set.

Again, following our previous development from CALC to TINYPPL, we wonder: is it similarly possible to represent first-class functions in the denotation of a probabilistic programming language? It's not at all obvious, at first, what shape a first-class function in a probabilistic programming language should have and whether it's possible to represent such an object. In discrete probability it is possible to represent first-class functions, but in continuous probability it is not at all so straightforward: this was the problem being studied by [14].

The crucial idea is that category theory will give us a general characterization of what it means for a semantic domain to support a first-class function: these will be called *exponential objects*, and categories that have exponential objects are called *Cartesian closed* and are capable of giving denotations to simply-typed lambda calculi.

## 1.5 Conclusion

This invites a question: *what sort of structure must a mathematical object have in order to be used to interpret a programming language*? Among many other things, category theory gives us very satisfying answers to this question: we will see that certain kinds of mathematical structure admit properties like "higher order functions", a meaning of "natural numbers", "logical truth," and the "existence of fixed points". Category theory gives us a formal language for describing what kind of structure a mathematical setting has. But before we get there, we'll see more examples of interpreting programs.

# 2
# *Categories*

As a metalanguage, sets and functions work best as a model of *purely functional* programming, where a program is thought of as a black box that produces a uniquely determined output value for each input. But real programs are rarely purely functional: programs in the wild regularly manipulate resources (pointers, files, locks), interact with their environment (input and output), and use complex control flow constructs (exceptions, coroutines).

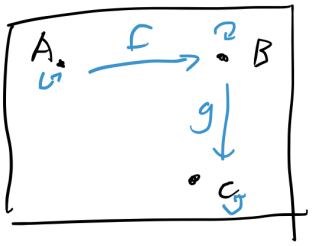
Category theory provides a framework for *swapping out the ambient metalanguage*: it provides techniques for systematically replacing sets and functions with constructs that are more naturally equipped to model realistic programming features. Each category embodies a choice of metalanguage. Sets and functions live in a *category of sets* Set, which we will encounter later on. By swapping out Set with other categories, we gain access to other metalanguages that are better adapted to modelling programming language features:

- Traditional domain theory, of untyped lambda calculus and recursive functional programming such as Haskell, lives in a *category of domains*. As a metalanguage, it supports arbitrary recursive definitions, and recursive types.
- A lot of modern semantics takes place in a category of *step-indexed sets* which as a metalanguage supports *very* recursive types.[1] We will meet this category later, when we discuss presheaves.
- Programs that manipulate resources live in categories of *resourceful sets*, which are used to define modern separation logics. We will meet these categories when we discuss the Day convolution.
- Ordinary continuous probability lives in a *category of measurable spaces*, and its higher-order generalization lives in *quasi-Borel spaces*. We may get to these categories if time permits.

[1] For instance, the type $A = \blacktriangleright \wp(A)$ that is isomorphic to its own power set (up to the so-called "later modality" $\blacktriangleright$).

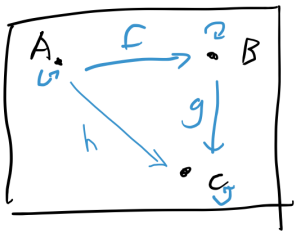
The concept of "category" abstracts over all of these examples. First, a category is defined on a set of **objects**: these interpret types and

typing contexts. Objects are related to each other by **morphisms** $A \xrightarrow{f} B$, or sometimes just $f$ for short; these interpret terms. Each morphism $A \xrightarrow{f} B$ is associated to a **domain** $A$ and a **codomain** $B$. Here is a **picture of a category**[2] with some objects and morphisms in it:




[2] A *picture of a category* shows objects as dots and morphisms as arrows, typically drawn with a box.

Morphisms can be combined via *composition*, which pastes together morphisms that are incident to each other to form new ones. Composing $g$ after $f$ is denoted $g \circ f$, or sometimes more tersely as $gf$. The ordering is a bit confusing; I like to read $\circ$ as "after". Pictorially, we can paste together edges $f$ and $g$ to get an edge $h$:




Categories satisfy several requirements that force them to behave nicely:

- Every object has a self-loop called the **identity morphism**, for an object $A$, we will denote its identity morphism as $\mathsf{id}_A$. This morphism must behave like an identity: for any morphism $A \xrightarrow{f} B$, it must be the case that $\mathsf{id}_B \circ f = f = f \circ \mathsf{id}_A$.
- Composition is **associative**, meaning that for any three morphisms $A \xrightarrow{f} B$, $B \xrightarrow{g} C$, $C \xrightarrow{h} D$, it is the case that $(f \circ g) \circ h = f \circ (g \circ h)$.

Summing up,[3]

**Definition 1: Category**

A category is a tuple $(O, M, \mathsf{dom}, \mathsf{cod}, \mathsf{id}, \mathsf{comp})$ where

- $O$ is a set whose elements are called "objects"
- $M$ is a set whose elements are called "morphisms"

[3] This definition of a category is one of many possible definitions. You may have seen different ones. This definition is used by [9]; different ones are used by [26].

- $\mathsf{dom} : M \to O$
- $\mathsf{cod} : M \to O$
- $\mathsf{id} : O \to M$
- $\mathsf{comp} : \{(f,g) \in M \times M \mid \mathsf{dom}(f) = \mathsf{cod}(g)\} \to M$

such that

- For all $f$ and $g$ in $M$ such that $\mathsf{dom}(f) = \mathsf{cod}(g)$ it holds that

$$\mathsf{dom}(\mathsf{comp}(f,g)) = \mathsf{dom}(g) \quad \text{and} \quad \mathsf{cod}(\mathsf{comp}(f,g)) = \mathsf{cod}(f)$$

- For all $X$ in $O$ it holds that $\mathsf{dom}(\mathsf{id}_X) = \mathsf{cod}(\mathsf{id}_X) = X$
- For all $f, g, h$ in $M$ such that $\mathsf{dom}(f) = \mathsf{cod}(g)$ and $\mathsf{dom}(g) = \mathsf{cod}(h)$, it holds that

$$\mathsf{comp}(f, \mathsf{comp}(g,h)) = \mathsf{comp}(\mathsf{comp}(f,g), h)$$

- For all $f$ in $M$ it holds that

$$\mathsf{comp}(f, \mathsf{id}_{\mathsf{dom}(f)}) = f = \mathsf{comp}(\mathsf{id}_{\mathsf{cod}(f)}, f)$$

This official definition is a bit unwieldy, so in practice we will use a lot of shorthand. We say two morphisms $f, g$ are *composable* if $\mathsf{dom}(f) = \mathsf{cod}(g)$. The composition $\mathsf{comp}(f,g)$ of two composable morphisms will be written $f \circ g$. If we ever write $f \circ g$, it will be tacitly assumed that $f$ and $g$ are composable. Finally, we sometimes omit the subscript from $\mathsf{id}_X$ when $X$ is clear from context. With these abbreviations in mind, the final two laws in the above definition read more comfortably as:

- $f \circ (g \circ h) = (f \circ g) \circ h$
- $f \circ \mathsf{id} = f = \mathsf{id} \circ f$.

## 2.1 *Our first metalanguage*

With a precise definition of category in hand, we are now ready to meet our first "metalanguage".

**Definition 2: the category FinSet of finite sets**

Let FinSet be the category $(O, M, \mathsf{dom}, \mathsf{cod}, \circ, \mathsf{id})$ where

- $O$ is the set of finite sets.
- $M$ is the set of tuples $(A, f, B)$ where $A$ and $B$ are finite sets and $f$ is a function from $A$ to $B$.
- dom is the function $\mathsf{dom}(A, f, B) = A$.

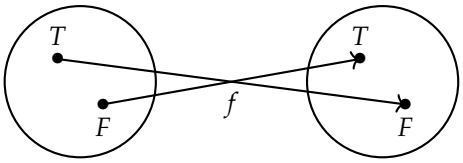
Sticklers may raise an eyebrow at the phrase "set of finite sets" in this definition. Strictly speaking, the set of finite sets does not form a set. But it's not that big of a deal: many workarounds are available. For instance, one could consider just finite subsets of $\mathbb{N}$, or the set of *hereditarily finite sets* (defined to be the union $\bigcup_{n \in \mathbb{N}} \wp^n(\varnothing)$).

- cod is the function $\mathsf{dom}(A, f, B) = B$.
- $\circ$ is the function $(B, f, C) \circ (A, g, B) = (A, f \circ g, C)$ where $f \circ g$ is the ordinary function composition of $f$ and $g$, well defined because the domain of $f$ matches the codomain of $g$.
- $\mathsf{id}_A$ is $(A, i, A)$ where $i$ is the identity function on $A$.

The category laws are satisfied because function composition is associative and composing a function with the identity function does nothing.

Again, shorthand helps to make this definition palatable: we often conflate the tuple $(A, f, B)$ the underlying function $f$ when working with morphisms of FinSet.

Mathematically, FinSet provides a categorical home for finite set theory. As a metalanguage, FinSet embodies *finite purely functional programming*. For instance, the following is a graph of the negation function $f : \{T, F\} \to \{T, F\}$ in the category FinSet, which could serve as a denotation of a program like $x : \mathsf{Bool} \vdash \mathsf{if}\ x\ \mathsf{then\ false\ else\ true} : \mathsf{Bool}$.




## 2.2 *Gaining intuition for categories*

The definition of category is highly abstract, and categories in the wild can look quite different from FinSet. In time, we will see how this level of abstraction allows for clean axiomatizations of the various kinds of "features" that a metalanguage may support, in the form of so-called "universal constructions." But in order to work at this level of abstraction, it will help to have in mind a few concrete examples. Broadly speaking, there are three different classes of example that are handy to have in the back of one's mind when thinking about categories.

### 2.2.1 *Categories as graphs*

The first perspective, perhaps most comfortable to computer scientists, is that a category is a kind of graph. Objects are vertices and morphisms are edges. On top of this, categories come with a composition operation that pastes two incident edges together to form a third, and identity morphisms $\mathsf{id}_A$ that form self-loops centered at each vertex $A$. The category laws then say that pasting of edges is an

associative operation, and that pasting with an identity morphism does nothing. This provides nice visual intuition, and a large stock of examples. In fact,

**Construction 2.2.1.** Every directed multigraph can be turned into a category.

*Proof.* Let $G$ be a directed multigraph, encoded as a tuple $(V, E, \mathsf{src}, \mathsf{trg})$ where $V$ is the set of vertices, $E$ is the set of edges, and $\mathsf{src}, \mathsf{trg}$ are functions $E \to V$ that send each edge to its source and target vertices respectively. The graph $G$ can be used to define a category

$$\mathsf{paths}(G) = (O, M, \mathsf{dom}, \mathsf{cod}, \circ, \mathsf{id})$$

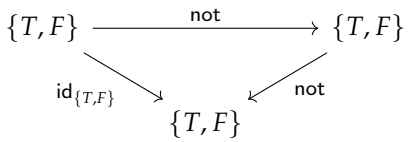
where

- $O = V$
- $M$ is the set of paths in $G$, which are tuples $(v_s, [e_1, \ldots, e_n], v_t)$ where $v_s, v_t$ are vertices and $e_1, \ldots, e_n$ is a list of edges satisfying $\mathsf{src}(e_1) = v_s$ and $\mathsf{trg}(e_n) = v_t$ and $\mathsf{trg}(e_i) = \mathsf{src}(e_{i+1})$ for all $1 \leq i < n$.
- $\mathsf{dom} : M \to O$ is the function $\mathsf{dom}(v_s, [e_1, \ldots, e_n], v_t) = v_s$
- $\mathsf{cod} : M \to O$ is the function $\mathsf{cod}(v_s, [e_1, \ldots, e_n], v_t) = v_t$
- $\circ$ concatenates paths: $(v_t, [f_1, \ldots, f_n], v_u) \circ (v_s, [e_1, \ldots, e_m], v_t) = (v_s, [e_1, \ldots, e_m, f_1, \ldots, f_n], v_u)$. Note that this is well-defined because the two paths $\vec{f}$ and $\vec{e}$ line up tip-to-tail. Note also that $\vec{e}$ comes before $\vec{f}$ in the concatenated path, due to the reversal of order of arguments to $\circ$.
- $\mathsf{id}_v$ is the empty path $(v, [], v)$ starting and ending at $v$.

The category laws are satisfied because concatenation of paths is associative and concatenating a path with the empty path does nothing. □

**Note 2.2.2.** It is essential that morphisms are *paths* in this construction, and not edges! Otherwise, it would be unclear how to define composition and identity.

A key difference between categories and graphs is that two morphisms in a category that come from different "paths" may nonetheless be equal. For example, the following two "paths" through FinSet are actually the same, because negating a Boolean twice is the same as doing nothing:



This relationship between the paths $\mathsf{not} \circ \mathsf{not}$ and $\mathsf{id}_{\{T,F\}}$ is often conveyed by saying that the triangle above is "filled in", or "commutes". The intuition being that one can then "continuously deform" $\mathsf{not} \circ \mathsf{not}$ into $\mathsf{id}_{\{T,F\}}$, by "dragging" it through the filled in triangle.

This idea is a handy visual tool for organizing categorical information and for conducting proofs.

### 2.2.2 *Categories as orders*

A special case of a multigraph is a graph, where there is *at most one* arrow between any given pair of vertices. This leads us to the second class of important examples of a category:

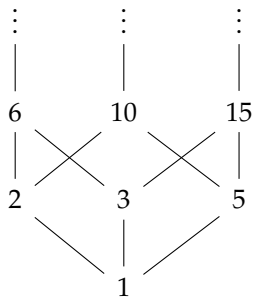
**Definition 3: preorder**

A preorder is a pair $(X, \preceq)$ where $X$ is a set and $\preceq$ is a binary relation on $X$ that is reflexive ($x \preceq x$ for all $x$ in $X$) and transitive (if $x \preceq y$ and $y \preceq z$ then $x \preceq z$).

Preorders are often fruitfully drawn as *Hasse diagrams*. A simple example is the set $\mathbb{N}$ of natural numbers, ordered by $m \preceq n$ if $m$ is less than or equal to $n$. We can draw this preorder as follows:


Hasse diagrams are read bottom-up, with the least elements on the bottom and the greatest elements on the top. An undirected edge is drawn between elements with no elements between them (this is why we have no edge from 0 to 2 in the above figure). A more visually interesting preorder on $\mathbb{N}$ is to define $m \preceq n$ if $m$ is a *divisor* of $n$.

This preorder looks less like a straight line and more like a lattice:




Preorders don't have to be discrete either: the real numbers $\mathbb{R}$, for instance, form a preorder with $x \preceq y$ if $x$ is less than or equal to $y$. This is a little harder to draw.

A preorder that arises frequently in PL is the preorder of *run-time environments*. It is common to model these environments as partial functions $\rho : \mathsf{Var} \rightharpoonup \mathsf{Val}$, assigning to each program variable $x \in \mathsf{Var}$ a concrete value $\rho(x)$. These substitutions form a preorder, with $\rho \preceq \rho'$ if $\rho$ is a subset of $\rho'$, in the sense that all the variables in $\rho$ are in $\rho'$ and mapped to the same values.

Preorders form a category:

**Construction 2.2.3.** Every preorder can be turned into a category.

*Proof.* For any preorder $(X, \preceq)$, there is a category

$$\mathsf{order}(X, \preceq) = (O, M, \mathsf{dom}, \mathsf{cod}, \circ, \mathsf{id})$$

where

- $O = X$.
- $M = (\preceq)$ (recall that a binary relation like $\preceq$ can be thought of as a set of pairs, with $(x, y) \in (\preceq)$ if and only if $x \preceq y$).
- $\mathsf{dom}$ is the function $\mathsf{dom}(x, y) = x$.
- $\mathsf{cod}$ is the function $\mathsf{cod}(x, y) = y$.
- $\circ$ is the function defined by $(y, z) \circ (x, y) = (x, z)$. Note that we may assume that the two arguments to $\circ$ share a component $y$ in this definition because of the assumption that $(y, z)$ and $(x, y)$ are composable. Note also that the composite $(x, z)$ is indeed well-defined this is well defined by the transitivity of $\preceq$.
- $\mathsf{id}_x$ is the pair $(x, x)$, which forms a valid morphpism by reflexivity of $\preceq$.

The category laws are satisfied automatically, as there can only be at most one morphism between any two objects.[4] □

[4] Pause: why is this the case?

The intuitive import of Proposition 2.2.3 is that every category can be thought of as a kind of preorder.[5] where the existence of a morphism $A \xrightarrow{f} B$ expresses that $A$ is in some sense "less than or equal to" $B$. Thinking of categories in this way leads to interesting questions like:

[5] *Reminder*: not all categories directly correspond to some preorder construction, since there can be multiple morphisms between objects in a general category. A category with at most 1 morphism between any 2 objects is called a *thin category*, and it does correspond to some preorder.

- Does a category have a "smallest" or "largest" object? For instance, the smallest runtime environment is the empty environment $\emptyset$ with no variables in it.
- Does a category have "ascending sequences"? For example,

$$\emptyset \subseteq \{x \mapsto A\} \subseteq \{x \mapsto A, y \mapsto B\} \subseteq \dots \tag{2.1}$$

  is an ascending sequence in the preorder of runtime environments.
- Does a category have "limit" objects, like suprema or infima? For instance, you probably know from calculus that there limits in $\mathbb{R}$: for example, the sequence $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16} \cdots$ approaches 0 in the limit. A very interesting question is: which categories "have limits", in the sense that such sequences always converge to some object in the category?

  An example of a situation where there is no limit is the sequence in Eq. (2.1). The set of environments grows and grows, and since there is no notion of a "greatest environment", there is no limit to this sequence.
- Given two categories, is there a notion of "monotone map" between them?

Each of the above observations is foreshadowing a notion of a "universal construction" in a category, which we will explore in the following lectures.

### 2.2.3 *Categories as algebras*

Whereas preorders are like multigraphs where there is at most one *edge* between any two vertices, the opposite extreme is a multigraph with exactly one *vertex*, so that all edges are self-loops:


(2.2)

Categories of this shape form our final class of important example: a special kind of algebraic structure called a *monoid*.

**Definition 4: monoid**

A monoid is a tuple $(X, \bullet, e)$ where $\bullet$ is a binary operation on $X$ (a function $X \times X \to X$) that is associative ($x \bullet (y \bullet z) = (x \bullet y) \bullet z$) and has $e$ as a unit ($x \bullet e = x$ and $e \bullet x = x$).

A simple example of a monoid is $(\mathbb{N}, +, 0)$, the monoid of natural numbers under addition. This forms a monoid because addition of natural numbers is associative and has zero as a unit.

An example that is closer to PL comes from the semantics of imperative programs. It is common to interpret an imperative program as a function $S \to S$, where $S$ is a set of memory states. For example, the program

$$x := x + 1;$$

can be interpreted as a function that sends a memory state $s$ to a new memory state $s'$ in which the value of $x$ has been incremented by 1. The functions $S \to S$ form a monoid $(S \to S, \bullet, \mathsf{id})$, where the monoid operation is given by $f \bullet g = g \circ f$ (note the reversal!), corresponding to the sequencing of the programs $f$ and $g$, and the identity element is the identity function on $S$, corresponding to the empty program that does nothing.

Algebraic identities capture program rewrites for this imperative language. For example, if $f$ is the interpretation of $x := 0$ and $g$ is the interpretation of $y := 1$, and $x$ and $y$ are distinct program variables, then the program equation

$$\begin{pmatrix} x := 0 \\ y := 1 \end{pmatrix} \equiv \begin{pmatrix} y := 1 \\ x := 0 \end{pmatrix} \tag{2.3}$$

can be expressed as the algebraic identity $g \circ f = f \circ g$.

**Construction 2.2.4.** Every monoid can be turned into a category.

*Proof.* For any monoid $(X, \bullet, e)$ there is a category

$$\mathsf{mon}(X, \bullet, e) = (O, M, \mathsf{dom}, \mathsf{cod}, \circ, \mathsf{id})$$

where

- $O = \{\star\}$, a singleton set with a single dedicated element $\star$.
- $M = X$
- $\mathsf{dom}(M) = \star$
- $\mathsf{cod}(M) = \star$
- Composition is defined by $x \circ y = x \bullet y$

- $\mathsf{id}_\star = e$

The category laws follow from the associativity of $(\bullet)$ and that it has $e$ as a unit. □

**Note 2.2.5.** It is very important that the set of objects $O$ is a singleton set $\{\star\}$. A natural thought one might have is that the set of objects should be the set $X$ of elements of the monoid. But this in fact does not work: what would be the morphisms?

There is actually a good reason why the set of objects is the seemingly-arbitrary singleton set $\{\star\}$ in this construction. It comes from the fact that a monoid is what is called a *single-sorted* algebraic structure; hence $\{\star\}$ is a set with a single element.

The intuitive import of Proposition 2.2.4 is that every category can be thought of as a kind of algebraic structure, where composition is like "multiplication" and identity morphisms are like "multiplicative units".

The algebraic perspective leads to questions like: given an equation between morphisms containing some unknowns, are there zero, one, or infinitely many solutions? When can I "cancel" a morphism from both sides of an equation (e.g., from $fg = fh$ deduce $g = h$, or from $fh = gh$ deduce $f = g$)? When does a morphism have an "inverse"? When can a morphism $f$ be "factored" into two pieces, $f = gh$?

# 3
# *Universal Constructions I: Products*

We've been hinting at the utility of knowing that categories have certain special objects in them. For example, if we want to give a category that lets us validate equations of programming languages, that category will need to be able to represent the many features that programs might have: products, sums, functions, etc. Here we will introduce the essential idea of a **universal construction** that characterizes when categories contain these special objects.

## *3.1 Terminal objects & the unit type*

Recall that when designing categorical semantics for programming languages we associate types and typing contexts with objects in the category and terms with morphisms. Concretely, if we have a well-typed term $\Gamma \vdash e : A$, its interpretation $[\![\Gamma \vdash e : A]\!]$ is a morphism $[\![\Gamma]\!] \xrightarrow{[\![e]\!]} [\![A]\!]$. Previously, we showed how FinSet can be used to give an interpretation to Calc programs by associating types like the unit type Unit with special sets, like a set containing only a single element $\{\star\}$. But, what if we want to see if categories other than FinSet can give a reasonable interpretation for Unit?

To proceed, let's pick a particular object $[\![\text{Unit}]\!]$ in a category $\mathcal{C}$ and explore what properties it needs to have in order to be "Unit-like":

- **Fact 1, Introduction rules**: It must be possible in any typing context to produce a value of type Unit. This tells us that for any context $\Gamma$, there must exist a $\mathcal{C}$-morphism $[\![\Gamma]\!] \to [\![\text{Unit}]\!]$. For example, it must be the case that $[\![\Gamma]\!] \xrightarrow{[\![\langle\rangle]\!]} [\![\text{Unit}]\!]$.

- **Fact 2, Equational laws**: Since Calc has no effects, it is the case that if $\Gamma \vdash e : \text{Unit}$, then it is indistinguishable from the term $\langle\rangle$. This is called **eta equality**, and we write this as $e \equiv \langle\rangle$.[1] Eta equality is an equational law that characterizes how a Unit term must behave. Now for an insight: *these equational laws are translated*

[1] The symbol "$\equiv$" means "is equationally equal to".

*into morphism equalities*. It must be that, for any morphism $[\![\Gamma]\!] \xrightarrow{[\![e]\!]} [\![\mathsf{Unit}]\!]$, it is the case that $[\![e]\!] = [\![\langle\rangle]\!]$. In other words, the morphism $[\![\Gamma]\!] \xrightarrow{[\![\langle\rangle]\!]} \mathsf{Unit}$ must be unique.

These two properties can be packaged up into a nice concise definition:

**Definition 5: Terminal object**

Let $T$ be an object in a category $\mathcal{C}$. An object $T$ in $\mathcal{C}$ is called a terminal object if

- for every object $\Gamma$ in $\mathcal{C}$ there is a morphism $\Gamma \xrightarrow{\langle\rangle} T$;
- for every morphism $\Gamma \xrightarrow{f} T$ it holds that $f = \langle\rangle$.

Let's pause to remark on the three categorical interpretations of terminal objects $T$:

- Graphically, an object is terminal if (1) there is an arrow from every other object to it, and (2) it "thins all parallel arrows", meaning that if we have a situation like the following for and morphisms $X \xrightarrow{f} T, X \xrightarrow{g} T$:

$$\begin{array}{ccc} & T & \\ {\scriptstyle f}\nearrow & & \nwarrow{\scriptstyle g} \\ & X & \end{array},$$

  then we know that $f = g$. In other words, there is a unique morphism $\Gamma \to T$ for every $\Gamma$:

$$A \xrightarrow{\exists!} T$$

- The order-theoretic perspective is that the terminal object is the "greatest object" in the category.
- The algebraic perspective says that, for every object $\Gamma$, the equation $f = f$ has a unique solution in the unknown $\Gamma \xrightarrow{f} T$.

We say a category **has terminal objects** if there is an object in the category that is a terminal object. This is our first example of a **universal construction**, which are special objects in categories characterized by their morphisms into them. We will see how universal constructions can be used to give interpretations to all the type formers we have seen so far. But, we can already see their utility: whether or not a particular category admits certain universal constructions will immediately inform us of that category's suitability for modeling certain language properties.

### 3.1.1 FinSet *has terminal objects, isomorphisms*

We saw in previous lectures that we could use the singleton set $\{\star\}$ to give an interpretation to Unit in FinSet. We can easily prove that $\{\star\}$ is a terminal object:

**Theorem 3.1.1.** The singleton set $\{\star\}$ is terminal in FinSet.

*Proof.*
- Existence: The map $\_ \mapsto \{\star\}$ has the required type.
- Uniqueness: Suppose we have two morphisms $A \xrightarrow{f} \{\star\}$ and $A \xrightarrow{g} \{\star\}$. Then by function extensionality we immediately have that $f = g$ as set-functions, and so they are equal as FinSet morphisms.

□

Now you can observe the above proof and see that it immediately works for *any* choice of singleton set: does this mean that FinSet has *many terminal objects?* Yes, and no. *Yes*, in the sense that there are indeed many valid choices of terminal object; but *no*, in the sense that the choice itself does not (*should* not!) actually matter. We capture this by the notion of isomorphism:

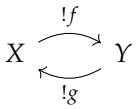
**Definition 6: Isomorphism of objects**

Two objects $A$ and $B$ in a category $\mathcal{C}$ are isomorphic if there exist morphisms $A \xrightarrow{f} B$ and $B \xrightarrow{g} A$ such that (1) $g \circ f = \mathrm{id}_A$ and (2) $f \circ g = \mathrm{id}_B$.

We can immediately see that any two singleton sets in FinSet are isomorphic: this proof is straightforward. What might be surprising is that this fact is true for all terminal objects:

**Theorem 3.1.2.** Terminal objects are unique up to isomorphism.

*Proof.* Let $X$ and $Y$ be terminal objects in some category $\mathcal{C}$. Then there exists a pair of unique morphisms between them:




By the fact that $X$ and $Y$ are terminal, they must also have unique identity morphisms $\mathrm{id}_X$ and $\mathrm{id}_Y$. Then by the uniqueness of identity we have that $g \circ f = \mathrm{id}_X$ and $f \circ g = \mathrm{id}_Y$. □

This provides some evidence that the choice of terminal object does not matter. But are we done? Suppose we had two terminal objects $T$ and $T'$. By the above argument, we have that these two objects are isomorphic. But now another potential choice arises: what

if there are multiple different *isomorphisms* between $T$ and $T'$? Then it could be that the choice to use $T$ instead of $T'$ may implicitly come with a choice of specific isomorphism as well.

Luckily, terminal objects satisfy a stronger property which ensures that one's choice is terminal object is truly irrelevant.

**Proposition 3.1.3.** Terminal objects are unique up to *unique* isomorphism. That is, if $T$ and $T'$ are both terminal, then there is exactly one isomorphism $(f : T \to T', g : T' \to T)$ between $T$ and $T'$.

*Proof.* Both $f$ and $g$, as constructed in the proof of Theorem 3.1.2, are the only possible morphisms $T \to T'$ and $T' \to T$ respectively. Thus they together form the only isomorphism between $T$ and $T'$. □

Later on, when we have the language for it, we will show that all universal constructions are unique up to unique isomorphism in this sense, and hence free of non-canonical choices.

## 3.2 *Products*

Next we want to interpret product types $A \times B$. As before with the unit type, the typing rules for products forces the existence of certain morphisms, and the equational theory of products forces equalities of certain morphisms. The introduction rule that describes how to create new products is:

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \text{ (T-PAIR)}$$

The elimination rules that break apart products are:

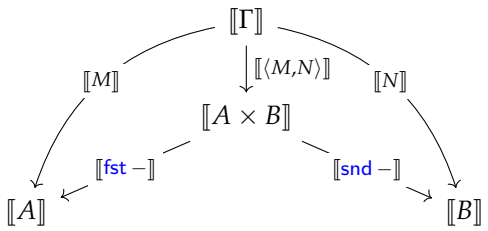
$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{fst}\, M : A} \text{ (T-FST)} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{snd}\, M : B} \text{ (T-SND)}$$

These rules imply the existence of several morphisms that characterize what it means for for a special object $[\![A \times B]\!]$ to be a product. Reading off from the rules:

- The T-PAIR rules says that if there are two morphisms $[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![A]\!]$ and $[\![\Gamma]\!] \xrightarrow{[\![N]\!]} [\![B]\!]$, then there is a morphism $[\![\Gamma]\!] \xrightarrow{[\![\langle M,N \rangle]\!]} [\![A \times B]\!]$.
- The T-FST rule says that if there is a morphism $[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![A \times B]\!]$, then there is a morphism $[\![\Gamma]\!] \xrightarrow{[\![\mathsf{fst}\, -]\!]} [\![A]\!]$.[2]
- The T-SND rule says that if there is a morphism $[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![A \times B]\!]$, then there is a morphism $[\![\Gamma]\!] \xrightarrow{[\![\mathsf{snd}\, -]\!]} [\![B]\!]$.

[2] Note: We are naming these morphisms *suggestively*. The typing rules only enforce the *existence* of certain morphisms: they don't tell us anything about how they behave.

We can package these morphisms up into the following diagram:




(3.1)

Next, the equational theory of products will assert certain equalities of morphisms. We begin with the *β-rules* that give the equational behavior of elimination after introduction:

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \mathsf{fst}\,\langle M, N\rangle \equiv M : A}\ (\beta_1^\times) \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \mathsf{snd}\,\langle M, N\rangle \equiv N : B}\ (\beta_2^\times)$$

Translating these equations into morphism equalities:

- $\beta_1$ enforces that $[\![M]\!] = [\![\mathsf{fst}\ -]\!] \circ [\![\langle M, N\rangle]\!]$ in (3.1)
- $\beta_2$ enforces that $[\![N]\!] = [\![\mathsf{snd}\ -]\!] \circ [\![\langle M, N\rangle]\!]$ in (3.1)

Hence, the $\beta$-rules together enforce that the diagram in 3.1 commutes. Next, we consider the second part of the equational theory of products that describes the behavior of introduction after elimination, sometimes called the *η-rules*:

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M \equiv \langle \mathsf{fst}\,M, \mathsf{snd}\,M\rangle : A \times B}\ (\eta^\times)$$

This law says asserts the extensional equality of any two pair-formation operations. Translated into morphisms, it says that any morphism of the form $[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![A \times B]\!]$ is is equal to a canonical morphism $[\![\Gamma]\!] \xrightarrow{[\![\langle \mathsf{fst}\,M, \mathsf{snd}\,M\rangle]\!]} [\![A \times B]\!]$. Put another way: there is a unique morphism $[\![\Gamma]\!] \to [\![A \times B]\!]$ making the above diagram **commute**.[3]

[3] Formally, a diagram **commutes** if every face of the diagram is "filled in", meaning that all paths beginning and ending in the same place are equal as morphisms.

### 3.2.1 *Packaging up intuition into a definition*

Now, as in the case of terminal objects, we can give a more generic description of products that isn't so specific to the case of equational theories of programs:

**Definition 7: Product**

Let $A$ and $B$ be two objects of a category $\mathcal{C}$. A *product of $A$ and $B$* is a tuple $(P, \pi_A, \pi_B)$ where

- $P$ is an object of $\mathcal{C}$.
- $\pi_A$ is a morphism from $P$ to $A$, called the *projection onto A*.
- $\pi_B$ is a morphism from $P$ to $B$, called the *projection onto B*.

such that

- For any object $\Gamma$ and any two morphisms $f : \Gamma \to A$ and $g : \Gamma \to B$, there exists a morphism $\langle f, g \rangle : \Gamma \to P$ such that the following equations hold:

$$\pi_A \circ \langle f, g \rangle = f \tag{3.2}$$
$$\pi_B \circ \langle f, g \rangle = g \tag{3.3}$$

- The following equation holds for any morphism $h : \Gamma \to P$.

$$h = \langle \pi_A \circ h, \pi_B \circ h \rangle \tag{3.4}$$

**Proposition 3.2.1.** Let $A$ and $B$ be finite sets. The tuple $(A \times B, \pi_A, \pi_B)$ is a product in FinSet, where $\pi_A$ is the morphism $(A \times B, \pi_A, A)$ and $\pi_B$ is the morphism $(A \times B, \pi_B, B)$.

**Proposition 3.2.2.** Let $A$ and $B$ be two objects of a category $\mathcal{C}$. A tuple $(P, \pi_A, \pi_B)$ is a product of $A$ and $B$ if and only if the following **universal property** holds: for any object $\Gamma$ and any two morphisms $f : \Gamma \to A$ and $g : \Gamma \to B$, there exists a unique morphism $h : \Gamma \to P$ such that $\pi_A \circ h = f$ and $\pi_B \circ h = g$.

*Proof.* Suppose $(P, \pi_A, \pi_B)$ is a product of $A$ and $B$. By definition of product, we have for any $\Gamma$ and morphisms $f : \Gamma \to A$ and $g : \Gamma \to B$ that the morphism $\langle f, g \rangle : \Gamma \to P$ satisfies $\pi_A \circ \langle f, g \rangle = f$ and $\pi_A \circ \langle f, g \rangle = g$. This morphism is the unique such, because if any morphism $h : \Gamma \to P$ satisfies $\pi_A \circ h = f$ and $\pi_B \circ h = g$ then it must be that

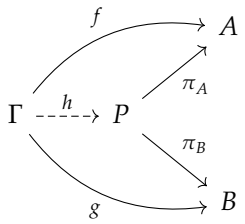
$$h = \langle \pi_A \circ h, \pi_B \circ h \rangle = \langle f, g \rangle. \tag{3.5}$$

Conversely, if $(P, \pi_A, \pi_B)$ is any tuple satisfying the above universal property, then one can define $\langle -, - \rangle$ as the function that sends each pair of morphisms $f : \Gamma \to A$ and $g : \Gamma \to B$ to the unique $h$ such that $\pi_A \circ h = f$ and $\pi_A \circ h = g$. □

Algebraically, the universal property can be formulated as saying the following system of equations has exactly one solution in the unknown $h : \Gamma \to P$.

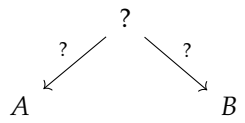
$$\pi_A \circ h = f \tag{3.6}$$
$$\pi_B \circ h = g \tag{3.7}$$

Diagramatically, it can be formulated as saying there exists a unique dashed arrow that fills in all the triangles in the following picture.




(3.8)

Order-theoretically, a product of $A$ and $B$ is the "greatest diagram" of the following shape, where the question marks are filled in with concrete objects and morphisms of $\mathcal{C}$:



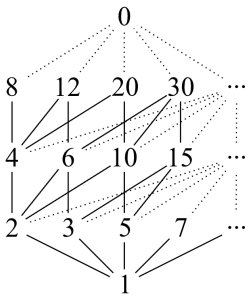

We will make this perspective more precise in Section 3.6.

Finally, as is the case with terminal objects, we say a category **has products** if for any two objects $A$ and $B$ in a category there exists an object $A \times B$ satisfying the universal property of objects.

## 3.3 *Products in a category that is a preorder*

Universal properties serve as a powerful tool for identifying analogies between mathematical objects. The essence of a product is made clear by (3.1): it is a means for placing two pieces of data "side by side," along with ways of extracting those two pieces of data after they've been packaged up without losing any information about those pieces of data. When viewed through the lens of universal properties, we start to see how general this idea of taking a product of two pieces of data can be.

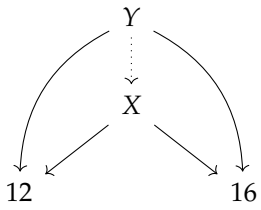
Let's see an example of a category that has an interesting and somewhat surprising example of products: the category of natural numbers ordered by divisibility. Natural numbers ordered by divisibility forms a preorder visualized by the following Hasse diagram:



The set $(\mathbb{N}, \preceq)$ forms a preorder. So, what do products look like in $(\mathbb{N}, \preceq)$ considered as a category? Let's consider the concrete instance of the product of 12 and 16. Let's consider the three possible interpretations of the product of 12 and 16 – let's call it $X$ – to understand what it means here:

In what way is this behaving like an intersection?

- The graph-theoretic picture places the product in a diagram. Suppose there is some natural number $Y$ such that $Y \preceq 12$ and $Y \preceq 16$. Then, the graphical depiction of the product $X$ asserts the existence of a unique morphism $Y \to X$:




- The order theoretic interpretation tells us that $X$ is the greatest element that is less than both 12 and 16 in the partial order – i.e., it is the greatest lower bound. In the case of this particular partial order, this object has a special name: it is the greatest common divisor.

Finally, similar to terminal objects, we say that a category $\mathcal{C}$ **has products** if for any two objects $A$ and $B$ in $\mathcal{C}$ there exists an object $A \times B$ satisfying the universal property of products.

## 3.4 *Getting comfortable working with products*

Though we arrived at the definition of product through the $\beta$ and $\eta$ laws for product types, the universal property is taken as primary as it is the one the generalizes to other type formers. This property has three crucial parts. Let $\Gamma, A, B$ be objects and $f : \Gamma \to A$ and $g : \Gamma \to B$. Then the following three facts hold:

1. There exists a morphism $h : \Gamma \to P$.
2. The morphism $h$ satisfies $\pi_1 \circ h = f$ and $\pi_2 \circ h = g$.
3. The morphism $h$ is the *unique* morphism satisfying property (2). This is made precise as follows: if any morphism $h' : \Gamma \to P$ satisfies $\pi_1 \circ h = f$ and $\pi_2 \circ h = g$, then $h' = h$.

Taken together, these three bullet points say that the special morphism $h$ is *defined by* the property that $\pi_1 \circ h = f$ and $\pi_2 \circ h = g$. A lot of proofs involving this universal property perform equational reasoning on the special morphisms $h$. Here is an example:

**Proposition 3.4.1.** The following holds for all $p : X \to Y$ and $f : Y \to Z$ and $g : Y \to W$ and products $(P, \pi_1, \pi_2)$ of $Z$ and $W$.

$$\begin{aligned}&(\text{the unique } h \text{ such that } \pi_1 \circ h = f \text{ and } \pi_2 \circ h = g) \circ p \\ &= (\text{the unique } h \text{ such that } \pi_1 \circ h = f \circ p \text{ and } \pi_2 \circ h = g \circ p)\end{aligned}$$

Another example:

**Proposition 3.4.2.** Let $(P, \pi_1, \pi_2)$ be a product of $A$ and $B$ in a category $\mathcal{C}$. Then

$$(\text{the unique } h \text{ such that } \pi_1 \circ h = \pi_1 \text{ and } \pi_2 \circ h = \pi_2) = \mathsf{id}_P$$

It gets a little tiresome to write "the unique $h$ such that $\pi_1 \circ h = f$ and $\pi_2 \circ h = g$" everywhere. So category theorists use a convenient notational abbreviation for this: the angle-bracketed expression $\langle f, g \rangle$.[4] With this notation, the above propositions read as the following algebraic identities:

$$\langle \pi_1, \pi_2 \rangle = \mathsf{id} \tag{3.9}$$

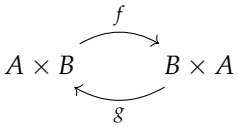
$$\langle f, g \rangle \circ p = \langle f \circ p, g \circ p \rangle \tag{3.10}$$

[4] Note that this notational convention hides some data: in particular, it hides the codomain of $\langle f, g \rangle$. In practice this notational convention is quite useful because, as we will show later (in Theorem 3.6.1), products are unique up to unique isomorphism, so hiding which particular choice of product object is utilized is prudent. Throughout category theory this sort of careful notation can be used (and confused!) to simplify arguments, but it's always important to know why a notational choice is valid.

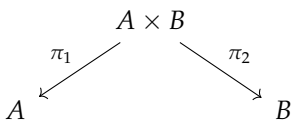
These algebraic identities can then be used to prove more elaborate facts about products.

**Proposition 3.4.3.** Let $\mathcal{C}$ be a category with products. Then $A \times B \cong B \times A$ for any objects $A, B$ of $\mathcal{C}$.

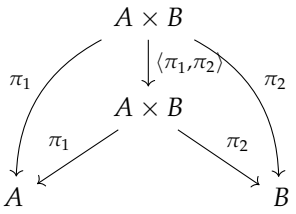
*Proof.* Since this is our first interesting proof we will give a fully diagrammatic argument. To show an isomorphism of objects we must find two morphisms:



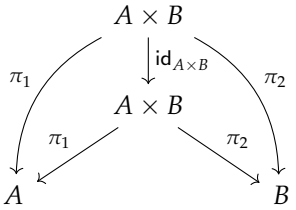

such that $g \circ f = \mathrm{id}_{A \times B}, f \circ g = \mathrm{id}_{B \times A}$. We only have two ways of showing that two morphisms are equal to one another: by reading equations off commuting diagrams, or by showing that two morphisms satisfy the same universal property. Let's first show that $\mathrm{id}_{A \times B}$ satisfies the same universal property as two morphisms $g \circ f$. We will do this very slowly. Let's start with the universal property of the product $A \times B$:



The universal property for products says that, for any object $X$ of the category, if it has morphisms $X \to A$ and $X \to B$, then there is a unique morphism $X \xrightarrow{h} A \times B$ such that the diagram commutes. Our proof will proceed by carefully picking different objects for $X$. First, we pick $X = A \times B$:



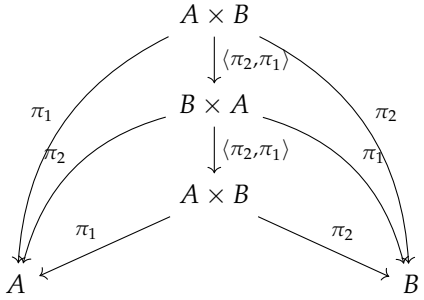

*But,* there is another morphism that satisfies this same universal property: the identity morphism!




(3.11)

Note how $\mathrm{id}_{A\times B}$ also makes this diagram commute. This establishes that $\mathrm{id}_{A\times B} = \langle \pi_1, \pi_2 \rangle$ (i.e., it is a graphical proof of Eq. (3.9))

Now, let's find morphisms $g \circ f$ that satisfy the same universal property as $\mathrm{id}_{A\times B}$. Now we can do this by constructing a chain of products:




(3.12)

Look closely: in the above diagram, $\langle \pi_2, \pi_1 \rangle \circ \langle \pi_1, \pi_2 \rangle$ satisfies the same universal property as $\mathrm{id}_{A\times B}$ in Eq. (3.11). So, by the universal property of products, we have that $\langle \pi_2, \pi_1 \rangle \circ \langle \pi_1, \pi_2 \rangle = \mathrm{id}_{A\times B}$. So, we've found our $f$ and $g$. The other half of the isomorphism can be shown by picking the other ordering of products in the sequence of diagrams. □

Next, here is a theorem with a familiar shape, if one thinks of products like the familiar notion of product of integers and terminal objects as behaving like the product unit:

**Proposition 3.4.4.** Let $\mathcal{C}$ be a category with products and a terminal object. Then $1 \times A \cong A$ for any object $A$ of $\mathcal{C}$.

**Definition 8: Ternary product**

Let $A, B, C$ be objects of a category $\mathcal{C}$. A *ternary product* of $A, B, C$ is a tuple $(P, \pi_A, \pi_B, \pi_C)$ where $\pi_A : P \to A$ and $\pi_B : P \to B$ and $\pi_C : P \to C$ and the following universal property holds: for any object $\Gamma$ and morphisms $f : \Gamma \to A$ and $g : \Gamma \to B$ and $h : \Gamma \to C$, there exists a unique morphism $k : \Gamma \to P$ such that $\pi_A \circ k = f$ and $\pi_B \circ k = g$ and $\pi_C \circ k = h$.

**Proposition 3.4.5.** Let $A, B, C$ be objects in a category $\mathcal{C}$ with products. The tuple $(A \times (B \times C), \pi_A, \pi_B \circ \pi_{B\times C}, \pi_C \circ \pi_{B\times C})$ is a ternary product of $A, B, C$.

**Definition 9: Finite products**

A category has **finite products** if it has products (Def 7) and a terminal object (Def 5).

## *3.5 Initial objects & duality*

The categorical perspective can push you towards new ideas you may not have thought of. Here's an example of this: given a category $\mathcal{C}$, thought of as a graph, one can do a natural thing: *flip the direction of all arrows in the opposite direction*. This new category is called the **opposite category**, and is written $\mathcal{C}^{\text{op}}$. Relationships between properties of $\mathcal{C}$ and properties of its opposite category $\mathcal{C}^{\text{op}}$ are broadly referred to as **dualities**.

Dualities can be a source of free ideas. For instance, let's consider the dual notion of a terminal object where we flip the direction of the arrows:

**Definition 10: Initial object**

Let $I$ be an object in a category $\mathcal{C}$. An object $I$ in $\mathcal{C}$ is called an initial object if, for every other object $A$ in $\mathcal{C}$ there exists a unique morphism $I \to C$. Graphically,

$$I \xrightarrow{\exists!} A$$

Note that an object that is terminal in $\mathcal{C}$ must be initial in $\mathcal{C}^{\text{op}}$ and vice versa: this is what characterizes the terminal object as dual to the initial object.

Now that we have a "free" universal construction, it might be natural to ask: *is there an interesting type that this special initial object corresponds to*? Indeed there is: the initial object corresponds to the Void type that has no inhabitants. It has the following typing rules:

$$\frac{M : \text{Void}}{\Gamma \vdash \texttt{absurd}\ M : A}$$

Intuitively, the Void type represents logical absurdity or "invalid program state". Some programming languages have a construct for working with void types and absurdity: Haskell has the void, which is eliminated with `absurd`.

## 3.6 *Products as Terminal Objects (∗)*

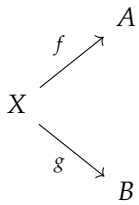
Sections marked with ∗ are optional and won't be covered in class.

The diagrammatic form suggests an order-based intuition for products: the product $(P, p, q)$ is in some sense the "largest" tuple of that shape, with its universal property showing that any other tuple of the same shape is "less than or equal" to it by virtue of the existence of the dashed morphism.

**Definition 11: Category of rooted spans**

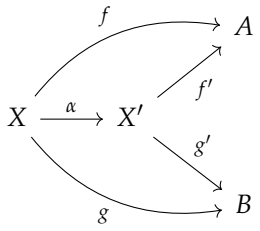
Let $A$ and $B$ be objects of a category $\mathcal{C}$. The category of *spans in $\mathcal{C}$ rooted at $A$ and $B$*, written $\mathsf{Span}_{\mathcal{C}}(A, B)$, is the category whose

- Objects are tuples $(X, f, g)$ where $X$ is an object of $\mathcal{C}$, $f : X \to A$, and $g : X \to B$. Pictorially:




- A morphism is a tuple $((X, f, g), \alpha, (X', f', g'))$ where $(X, f, g)$ and $(X', f', g')$ are objects and $\alpha : X \to X'$ is a morphism of $\mathcal{C}$

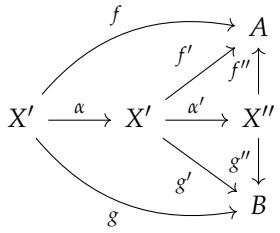
such that $f' \circ \alpha = f$ and $g' \circ \alpha = g$. Pictorially:




- Composition is defined by

$$((X', f', g'), \alpha', (X'', f'', g'')) \circ ((X, f, g), \alpha, (X', f', g')) \quad (3.13)$$
$$= ((X, f, g), \alpha' \circ \alpha, (X'', f'', g'')). \quad (3.14)$$

  Pictorially:




(3.15)

- The identity morphism at $(X, f, g)$ is $((X, f, g), \mathsf{id}_X, (X, f, g))$.

The associativity and identity laws in $\mathsf{Span}_{\mathcal{C}}(A, B)$ are inherited from the corresponding laws for in $\mathcal{C}$.

This perhaps tortured-looking category serves a valuable purpose: it makes precise the sense in which a product $(P, p, q)$ is the "largest" among such tuples.

**Proposition 3.6.1.** Let $A$ and $B$ be objects of a category $\mathcal{C}$. A tuple $(P, p, q)$ is a product of $A$ and $B$ if and only if it is a terminal object of $\mathsf{Span}_{\mathcal{C}}(A, B)$.

To unpack what this is saying, from the perspective of categories as metalanguages: a product for $A$ and $B$ in the metalanguage embodied by $\mathcal{C}$ is a unit type for the metalanguage embodied by $\mathsf{Span}_{\mathcal{C}}(A, B)$. Some of the power of category theory can be seen here:

- Constructions on categories can be used to quickly build nontrivial "metalanguages" from old. (What would $\mathsf{Span}_{\mathcal{C}}(A, B)$ even look like as a traditional language?)

- Category theory "eats itself": the categorical notion of terminal object sheds light on the categorical notion of product, if one knows to look at the right category.

To illustrate the benefits of this perspective, we get a proof that products are suitably unique just like terminal objects are, simply because they *are terminal objects*:

**Proposition 3.6.2.** Products are unique up to unique isomorphism.

*Proof.* Products are terminal objects, and terminal objects are unique up to unique isomorphism.[5] □

[5] Phew! What a nice short proof!

# 4
# *Universal Constructions II: Exponents*

Now, let's move on to the next major addition to our library of language features: first-class functions and function types. Let's recall the rules for a language with function types. First, we have the rule for making terms of function type:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda\, x.\, M : A \to B}\ (\text{T-Lam})$$

Next, we have the elimination rule for using terms of function type:

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}\ (\text{T-App})$$

Finally, we have the $\beta$ and $\eta$ laws, which state that applying a function corresponds to substitution and that every term of function type is equivalent to a $\lambda$-expression.

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda\, x.\, M)\, N \equiv M[N/x] : B}\ (\beta^{\to})$$

$$\frac{\Gamma \vdash M : A \to B}{\Gamma \vdash M \equiv \lambda\, x.\, M\, x : A \to B}\ (\eta^{\to})$$

Our development here will mirror the development so far of product types and unit types: we will identify a special object in a category, characterized by some morphism equalities and a universal property, such that it validates the equational theory of $\lambda$-expressions.

Let's designate a special object in the category to denote the interpretation of function types. This object will be called the **exponential object**:

$$[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]} \tag{4.1}$$

Intuitively, this exponential object must summarize all the morphisms from $A$ to $B$. Like product and unit types, this special object will be uniquely characterized by a universal property. Let's work up to what that universal property is. We can start by filling in our semantics for $\lambda$-formation, which we know must look like:

$$[\![\Gamma \vdash \lambda\, x : A.\, M : A \to B]\!] : [\![\Gamma]\!] \to [\![B]\!]^{[\![A]\!]} \tag{4.2}$$

We are gathering clues: the above tells us that there must exist some morphism from $[\![\Gamma]\!]$ into the special object $[\![B]\!]^{[\![A]\!]}$. Let's suggestively call this morphism $[\![\Gamma]\!] \xrightarrow{\lambda [\![M]\!]} [\![B]\!]^{[\![A]\!]}$:

$$[\![\Gamma \vdash \lambda\, x : A.\, M : B]\!] = [\![\Gamma]\!] \xrightarrow{\lambda [\![M]\!]} [\![B]\!]^{[\![A]\!]} \tag{4.3}$$

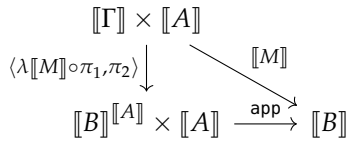
The purpose of the the morphism $\lambda\, [\![M]\!]$ is to compute the closure of $M$: it must package up the body of the lambda $M$ into some representation for which it can later be called. Now, to interpret application, we need another special morphism called app that denotes applying a closure to its argument:

$$[\![B]\!]^{[\![A]\!]} \times [\![A]\!] \xrightarrow{\mathsf{app}} [\![B]\!] \tag{4.4}$$

With this special morphism we can give an interpretation to application:

$$\left[\!\!\left[ \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B} \right]\!\!\right] = [\![\Gamma]\!] \xrightarrow{\langle \lambda [\![M]\!], \mathrm{id} \rangle} [\![B]\!]^{[\![A]\!]} \times [\![\Gamma]\!] \xrightarrow{\langle \pi_1, [\![N]\!] \circ \pi_2 \rangle} [\![B]\!]^{[\![A]\!]} \times [\![A]\!] \xrightarrow{\mathsf{app}} [\![B]\!] \tag{4.5}$$

Now, the closure morphism $\lambda\, [\![M]\!]$ and the application morphism app need to satisfy the equational laws. We can package up these laws into a diagram that must commute:




Let's unpack this diagram. First, the morphism $[\![\Gamma]\!] \times [\![A]\!] \xrightarrow{[\![M]\!]} [\![B]\!]$ is the morphism that gives the semantics of the open term $\Gamma, x : A \vdash M : B$. The diagram says that $[\![M]\!]$ should commute with first forming a closure and then applying the closure to the same argument. Finally, the $\eta$ laws assert the uniqueness of the closure-formation morphism $\lambda\, [\![M]\!]$. Now we can package up all this intuition into a definition of a universal property:

**Definition 12: Universal property of exponential object**

Let $A$ and $B$ be objects of a category $\mathcal{C}$ with products. An *exponential of $B$ by $A$* is a pair $(B^A, \mathsf{app})$ consisting of an object $B^A$ of $\mathcal{C}$ and a morphism app from $B^A \times A$ to $B$ such that the following holds: for any object $\Gamma$ and any morphism $f : \Gamma \times A \to B$, there exists a unique morphism $\lambda f : \Gamma \to B^A$ such that $\mathsf{app} \circ \langle \lambda f \circ \pi_1, \pi_2 \rangle = f$.

A category **has exponents** if for any two objects $A$ and $B$ there is an exponential of $A$ by $B$. Categories that have exponents get a special name: they are called **Cartesian-closed categories**.

## 4.1 Examples of exponential objects

### 4.1.1 Exponents in FinSet

Let's begin as usual by seeing examples of universal properties in categories that we're familiar with. We'll begin with the category of finite sets FinSet. For two finite sets $A$ and $B$, the exponential $B^A$ is an object that represents the set of all functions from $A$ to $B$. For instance, suppose $A = \{a, b\}$, $B = \{c, d\}$, and let $\Gamma = \{\star\}$. Then, there are 4 possible functions between these two sets:

$$
\begin{aligned}
f_1(\gamma, x) &= c \\
f_2(\gamma, x) &= \begin{cases} c & \text{if } x = a \\ d & \text{if } x = b \end{cases} \\
f_3(\gamma, x) &= \begin{cases} d & \text{if } x = a \\ c & \text{if } x = b \end{cases} \\
f_4(\gamma, x) &= d
\end{aligned}
$$

The crucial idea is that this collection of 4 morphisms can itself be represented as a finite set. Recall that the *graph* of a function $f : A \to B$ is the set of pairs $(a, b)$ where $a$ is the input and $b$ is the output of $f$. Then, the exponential object $B^A$ is the *finite set of all graphs of functions from $A$ to $B$*. So, for instance, the set $\{c, d\}^{\{a,b\}}$ is:

$$
\Big\{ \{(a, c), (b, c)\}, \{(a, d), (b, d)\}, \\
\{(a, c), (b, d)\}, \{(a, d), (b, c)\} \Big\}
$$

Using this, we can define the closure $\lambda f$ as mapping a function to its graph. For example:

Note that in this example, since $\Gamma = \{\star\}$, the argument $\gamma$ is ignored in this definition. If $\Gamma$ were more interesting, it may be the case that different graphs are returned for different contexts.

$$\lambda f_1(\gamma) = \{(a,c),(b,c)\}$$
$$\lambda f_2(\gamma) = \{(a,c),(b,d)\}$$
$$\lambda f_3(\gamma) = \{(a,d),(b,c)\}$$
$$\lambda f_3(\gamma) = \{(a,d),(b,d)\}$$

Next, an exponential object is paired with a morphism app that calls the closure on an argument. We can implement the function app by picking the entry of the graph that corresponds to the argument. For example, $\mathsf{app} \circ \langle \lambda f_1, a \rangle = c$.

The exponential object must satisfy the property that $\mathsf{app} \circ \langle \lambda f, A \rangle = f$. This is surely true, because applying the graph of a function to an argument $a$ must agree with calling $f$ on $a$. Finally, uniqueness follows by function extensionality: any $\lambda f$ that satisfies this property must behave identically to the graph of $f$. Summing up:

**Proposition 4.1.1.** FinSet is Cartesian-closed.

*Proof.* We have already shown that FinSet has products (the Cartesian product of finite sets). Let $A, B$ be finite sets. Then, we need to show that for any finite set $\Gamma$ with morphism $\Gamma \times A \xrightarrow{f} B$, there is a unique morphism $\lambda f$ such that the following diagram commutes:

$$\begin{array}{ccc} \Gamma \times A & & \\ {\scriptstyle \langle \lambda f \circ \pi_1, \mathsf{id} \circ \pi_2 \rangle}\downarrow & \searrow^{f} & \\ B^A \times A & \xrightarrow{\mathsf{app}} & B \end{array}$$

To satisfy this, we will define $B^A$ to be set of all graphs of functions $A \to B$. Then, we define $\mathsf{app}(f, a)$ to pick the component of the graph $f$ that corresponds to $a \in A$.

Now let's show this choice validates the universal property. Let $\Gamma$ be any set and $f$ be a function $\Gamma \times A \to B$. Then, there exists a function $\lambda f : \Gamma \to B^A$ such that the diagram commutes: it's the function that sends each $\gamma \in \Gamma$ to the graph $\{(a,b) \mid f(\gamma, a) = b\}$ representing the "partial application" of $f$ to $\gamma$. Observe that, by function extensionality, $\lambda f$ is the unique function making the diagram commute, completing the proof. □

### 4.1.2 *Exponents in* Formula

Recall from Assignment 1 the category Formula whose objects are logical formulae whose morphisms denote formula entailment. Formulae are described by the following grammar:

$$\varphi, \psi ::= x \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\psi \tag{4.6}$$

Fix a global finite set of variable names $\Omega$, and assume $x \in \Omega$. Then, we can give a semantics to Boolean formulae in terms of substitutions. A substitution $\gamma$ is a function $\Omega \to \{\top, \bot\}$ that maps each variable to a truth value. Then, we can define the semantics of formulae inductively:

$$[\![x]\!](\gamma) = \gamma(x)$$

$$[\![\varphi \wedge \psi]\!](\gamma) = \begin{cases} \top & \text{if } [\![\varphi]\!](\gamma) = \top, [\![\psi]\!](\gamma) = \top \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\varphi \vee \psi]\!](\gamma) = \begin{cases} \bot & \text{if } [\![\varphi]\!](\gamma) = \bot, [\![\psi]\!](\gamma) = \bot \\ \top & \text{otherwise} \end{cases}$$

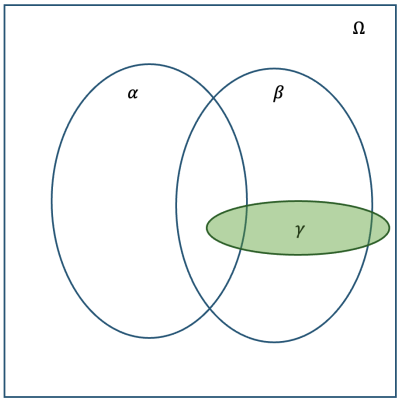
$$[\![\neg\varphi]\!](\gamma) = \begin{cases} \top & \text{if } [\![\varphi]\!](\gamma) = \bot \\ \bot & \text{if } [\![\varphi]\!](\gamma) = \top \end{cases}$$

The *set of models*, written $\mathsf{Mods}(\varphi)$, is the set of all substitutions that evaluate to $\top$, i.e. $\mathsf{Mods}(\varphi) = \{\gamma \mid [\![\varphi]\!](\gamma) = \top\}$. We say a formula $\varphi$ *semantically entails* a formula $\psi$, written $\varphi \models \psi$, if $\mathsf{Mods}(\varphi) \subseteq \mathsf{Mods}(\psi)$. The category $\mathsf{Formula}$ is the category whose objects are Boolean formula and whose morphisms denote formula entailment:

- Let the set of objects $O$ be the set of all syntactic Boolean formulae generated by the above grammar.
- The set of morphisms $M$ consists of pairs $(\varphi, \psi)$ where $\varphi \models \psi$.
- Define $\mathsf{dom}(\varphi, \psi) = \varphi, \mathsf{cod}(\varphi, \psi) = \psi$
- Identity is defined by $\mathsf{id}(\phi) = (\phi, \phi)$
- Composition is defined by $\mathsf{comp}((\psi, \xi), (\varphi, \psi)) = (\varphi, \xi)$

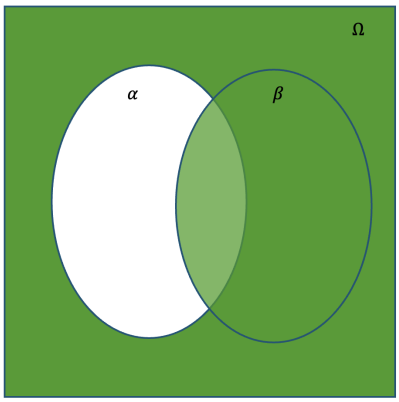
In the assignment you showed that this category has products, and that products denoted logical conjunction of formulae. Now, let's investigate what exponential objects mean in this category.

Suppose $\alpha$ and $\beta$ are logical formulae, and let $\gamma$ be some other logical formula such that $\gamma \wedge \alpha \models \beta$. Pictorially, we can draw this relationship using a Venn diagram:




Let $[\![\alpha]\!]$ denote the set of all models of $\alpha$. To show that $\alpha \wedge \gamma \models \beta$, it must be the case that the $[\![\alpha]\!] \cap [\![\gamma]\!] \subseteq [\![\beta]\!]$. This relationship clearly holds in the above figure.

Viewed from an order-theoretic perspective, the exponential object $\beta^\alpha$ is the *greatest subset* of $\Omega$ such that $[\![\alpha]\!] \cap [\![\beta^\alpha]\!] \subseteq [\![\beta]\!]$. We can draw this subset in green:




Now the question: can this set of models be represented as a logical formula? Indeed it can: it corresponds with the logical formula $\neg\alpha \vee \beta$, also known as the (classical) logical implication $\alpha \Rightarrow \beta$.

So, we have a candidate exponential object. Now, let's check that it satisfies the universal property: we must show that for any formulae $\alpha$ and $\beta$, for any formula $\gamma$ with morphism $\alpha \times \gamma \to \beta$, there exists an object $\neg\alpha \vee \beta$ such that:

1. There is a morphism $\gamma \to \neg\alpha \vee \beta$. This morphism represents *logical weakening of* $\gamma$.

2. There is a morphism $(\neg\alpha \vee \beta) \times \alpha \to \beta$. This morphism represents *modus ponens*.

First we prove (1):

*Proof.* We need to show that if $\alpha \wedge \gamma \models \beta$, then $\gamma \models \neg\alpha \vee \beta$. Suppose by contradiction that $[\![\gamma]\!] \not\subseteq [\![\neg\alpha \vee \beta]\!]$. Then, there exists an $x \in \Omega$ such that $x \in [\![\gamma]\!]$ and $x \notin [\![\neg\alpha \vee \beta]\!]$. By De Morgan's law we have that $x \in [\![\alpha \wedge \neg\beta]\!]$. This shows that $x \in [\![\gamma \wedge \alpha]\!]$ and $x \notin [\![\beta]\!]$, a contradiction. □

Point (2) is a well-known fact about propositional logic called modus ponens, which we won't prove here. Since this is a thin category it is not necessary to check uniqueness. This establishes that Formula has exponential objects and hence is a Cartesian-closed category.

### 4.1.3 *An example from topology*

Category theory is a bridge between many branches of mathematics, and we can gain significant insight into categorical definitions by viewing them through the lens of different mathematical subfields. Many important developments in category theory stemmed from topology, so we explore some of the basics of topology here.

In the previous section we drew a picture of the exponential as a Venn diagram. The exponential was then the largest shaded region $C$ satisfying the constraint $A \cap C \subseteq B$. This spatial intuition can be made formal.

**Definition 13: Open set**

A subset $U$ of of the real numbers $\mathbb{R}$ is **open** if it is a (possibly infinite) union of open intervals, where an open interval $(a, b)$ is the collection of real numbers between $a$ and $b$ excluding $a$ and $b$, i.e. $\{r \mid a < r < b\}$.

An equivalent way to define an open set is a subset $U$ of the reals satisfying that, for any $x \in U$, an $\epsilon$-ball around $x$ is also contained in $U$. I.e., there exists an $\epsilon > 0$ such that $(x - \epsilon, x + \epsilon) \subseteq U$.

The set of all open subsets of $\mathbb{R}$, written $\mathcal{O}(\mathbb{R})$, forms a partial order under subset inclusion, and hence also a thin category (Construction 2.2.3). As a category, $\mathcal{O}(\mathbb{R})$ has finite products: the terminal object is the largest open subset of $\mathbb{R}$, namely the whole space $\mathbb{R}$, and the product of two opens subsets $U$ and $V$ is their intersection $U \cap V$.

**Proposition 4.1.2.** $\mathcal{O}(\mathbb{R})$ is Cartesian closed.

*Proof.* Following the intuition that the exponential of two objects $U$ and $V$ is the largest object $W$ such that $U \cap W \subseteq V$, we define

$$V^U = \text{the largest open subset } W \text{ of } \mathbb{R} \text{ such that } U \cap W \subseteq V \tag{4.7}$$

$$= \text{the largest open subset } W \text{ of } \mathbb{R} \text{ contained in } (\mathbb{R} \setminus U) \cup V. \tag{4.8}$$

This largest open subset has a name: it is called the **interior** of $(\mathbb{R} \setminus U) \cup V$, written $\mathsf{int}((\mathbb{R} \setminus U) \cup V)$. Intuitively, taking the interior of a set is a way of "forcing" it to be open, by deleting all of its boundary points. For example, the interior of a closed interval is the corresponding open interval: $\mathsf{int}[0,1] = (0,1)$. From this perspective, the subset $\mathsf{int}((\mathbb{R} \setminus U) \cup V)$ defining the exponential $V^U$ is the largest open contained in $(\mathbb{R} \setminus U) \cup V$, which corresponds to the Venn diagram (4.1.2). of $(\mathbb{R} \setminus W) \cup V$. □

As an example, let us compute the exponential where $U = (-\infty, 0)$ and $V = \varnothing$. In this case,

$$V^U = \mathsf{int}([0,\infty) \cup \varnothing) = \mathsf{int}[0,\infty) = (0,\infty). \tag{4.9}$$

Note how, in the last step, the use of the interior operator "rounds out" the closed interval $[0,\infty]$ into the open interval $(0,\infty)$.

Thinking of the exponential $\varnothing^U$ as a "complement" or "negation" of $U$, this says that the complement of $(-\infty,0)$ in $\mathcal{O}(\mathbb{R})$ is $(0,\infty)$. Note that $\mathbb{R}$ is *not* the union of $U$ and its complement:

$$U \cup \varnothing^U = (-\infty,0) \cup (0,\infty) = \mathbb{R} \setminus \{0\} \neq \mathbb{R}$$

What this demonstrates is that the law of excluded middle fails in $\mathcal{O}(\mathbb{R})$: the open subsets of $\mathbb{R}$ form a model of intuitionistic propositional logic that refutes $A \vee \neg A = \top$. This is a somewhat surprising bridge between geometry and logic, which we will explore later if we have time.

### *4.1.4 A non-example: natural numbers and divisibility*

It's worth exploring examples of categories that *do not* have exponential objects. A good example of this is the category $(\mathbb{N}, \preceq)$, the category of natural numbers with divisibility relations as morphisms.

Let's see why this is the case. Concretely, we will show that 2 and 3 do not have an exponential.

Suppose that there *was* an exponential $k$. Then we would have two things.

1. A morphism $k \times 2 \xrightarrow{\mathsf{app}} 3$.
2. For any $k'$ and any morphism $k' \times 2 \to 3$, there exists a unique morphism $k' \to k$ such that a triangle commutes.

Unpacking point (1), we have that $\gcd(k,2)$ divides 3 (because product in this category is greatest common divisor and morphisms are divisibility relations). Unpacking point (2), we have that for any $k'$ such that $\gcd(k',2)$ divides 3, it holds that $k'$ divides $k$. The fact that the morphism $k' \to k$ is the unique one making a given triangle

commute does not mean much here because our category is thin: the moment we know a morphism $k' \to k$ exists, we also know it must be the unique such morphism.

What do these two conditions tell us about the supposed exponential object $k$? Intuitively, they say that $k$ is the "largest number $k$ such that $\gcd(k, 2)$ divides 3". It is suspicious that there would be a largest such number: $\gcd(k, 2) = \gcd(3k, 2)$ for any $k$, so if $\gcd(k, 2)$ divides 3 then so does $\gcd(3k, 2)$. In short, we can always keep increasing $k$ by multiplying it by 3, so there cannot be a largest such.

We can turn this intuitive argument into a rigorous one as follows. We will exhibit an counterexample to point (2) above. Set $k' = 3k$. We have that $\gcd(3k, 2) = \gcd(k, 2)$, and $\gcd(k, 2)$ divides 3, so $\gcd(3k, 2)$ divides 3 too. This fulfills the precondition of point (2). The conclusion of point (2) then tells us that $3k$ divides $k$. But this is impossible.

Since we derived a contradiction starting from an arbitrary choice of $k$ satisfying (1) and (2), this argument shows there can be no $k$ satisfying the universal property of the exponential $3^2$, and hence that $(\mathbb{N}, \preceq)$ is not Cartesian closed.

# 5
# *From Categories to Languages*

Consider a variant of CALC from Chapter 1, with a unit type and parameterized by an collection of base types $b$ and constants $c$.

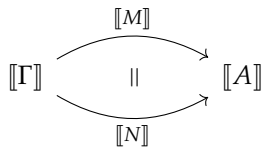
$$\begin{aligned} M, N &::= \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N \mid x \mid \langle\rangle \mid \langle M, N\rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \\ A, B &::= A \times B \mid \mathsf{Unit} \mid b \end{aligned} \tag{CALC}$$

The only goal of this chapter is to prove the following.

**Theorem 5.0.1.** CALC can be interpreted in any category with finite products (Definition 9), provided suitable interpretations of each base type $b$ as an object and each constant $c$ as a morphism.

*Proof.* Let $\mathcal{C}$ be a category with products and a terminal object. We will build an interpretation function $[\![-]\!]$ that maps each of the judgments of CALC into $\mathcal{C}$:

- Types $A$ will become objects $[\![A]\!]$ of $\mathcal{C}$.
- Contexts $\Gamma$ will become objects $[\![\Gamma]\!]$ of $\mathcal{C}$.
- Derivations of $\Gamma \vdash M : A$ will become morphisms $[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![A]\!]$.
- Derivations of $\Gamma \vdash M \equiv N : A$ will become equalities:




Each of these will be defined by induction.[1]

[1] Warning: this is going to take a while.

First, let us interpret CALC types. We assume given the interpretation of base types, with each base type $b$ denoting some chosen object $[\![b]\!]$ of $\mathcal{C}$. This interpretation is bootstrapped into one for all types by induction:

$$[\![\mathsf{Unit}]\!] = 1 \tag{5.1}$$

$$[\![A \times B]\!] = [\![A]\!] \times [\![B]\!] \tag{5.2}$$

These equations look just like (1.1), the interpretation of types given in Section 1.1. But their meaning is very different. Whereas before the operator $\times$ was used to denote the Cartesian product of finite sets, here it denotes the categorical product of the objects $[\![A]\!]$ and $[\![B]\!]$ of an arbitrary category $\mathcal{C}$. Similarly, 1 denotes the terminal object in $\mathcal{C}$, not the singleton set $\{\star\}$.

The interpretation of CALC contexts is similarly straightforward. The empty context is interpreted as the terminal object and context extension is interpreted via product—again, we are using the same symbols for the abstract categorical definitions rather than the set-theoretic ones.

$$[\![\bullet]\!] = 1 \tag{5.3}$$

$$[\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!] \tag{5.4}$$

Next up we have the interpretation of typing derivations. This is where things start getting interesting. We have one case per typing rule. As a warm-up, here is the interpretation of the typing rule for $\langle\rangle$.

$$\left[\!\!\left[ \frac{}{\Gamma \vdash \langle\rangle : \mathsf{Unit}} \right]\!\!\right] = \langle\rangle_{[\![\Gamma]\!]} \tag{5.5}$$

In words: the interpretation of $\langle\rangle$ with respect to typing context $\Gamma$ is the unique morphism $\langle\rangle_{[\![\Gamma]\!]}$ from $[\![\Gamma]\!]$ to $[\![\langle\rangle]\!]$, guaranteed to exist because $[\![\langle\rangle]\!]$ is a terminal object of $\mathcal{C}$.

The interpretation of $\langle\rangle$ is special because its typing rule has no premises. More generally, the interpretations of the other typing rules will rely inductively on interpretations of premises. For instance, here is the interpretation of the typing rule for $\mathsf{fst}\, M$.

$$\left[\!\!\left[ \frac{\begin{array}{c}\vdots\\ \Gamma \vdash M : A \times B\end{array}}{\Gamma \vdash \mathsf{fst}\, M : A} \right]\!\!\right] = \pi_1 \circ \left[\!\!\left[ \begin{array}{c}\vdots\\ \Gamma \vdash M : A \times B\end{array} \right]\!\!\right] \tag{5.6}$$

[2] In words: to interpret a typing derivation for $\mathsf{fst}\, M$ as shown, first interpret the subderivation establishing $\Gamma \vdash M : A \times B$ to obtain a morphism $[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![A \times B]\!]$, and then compose this morphism with the projection $[\![A \times B]\!] \xrightarrow{\pi_1} [\![A]\!]$, guaranteed to exist because $[\![A \times B]\!]$ is a product of $[\![A]\!]$ and $[\![B]\!]$.

[2] **todo**: Really wish I knew how to fix the spacing here.

The typing rule for $\mathsf{snd}\, M$ is interpreted analogously.

$$\left[\!\!\left[ \frac{\begin{array}{c}\vdots\\ \Gamma \vdash M : A \times B\end{array}}{\Gamma \vdash \mathsf{snd}\, M : B} \right]\!\!\right] = \pi_2 \circ \left[\!\!\left[ \begin{array}{c}\vdots\\ \Gamma \vdash M : A \times B\end{array} \right]\!\!\right] \tag{5.7}$$

The pair formation rule is interpreted using the universal property of product.

$$\left[\!\!\left[ \dfrac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \vdash M : A & \Gamma \vdash N : B\end{array}}{\Gamma \vdash \langle M, N \rangle : A \times B} \right]\!\!\right] = \left\langle \left[\!\!\left[ \dfrac{\vdots}{\Gamma \vdash M : A} \right]\!\!\right], \left[\!\!\left[ \dfrac{\vdots}{\Gamma \vdash N : B} \right]\!\!\right] \right\rangle \tag{5.8}$$

Finally we have the typing rules for variables and let-bindings. We have saved these rules for last because they highlight some interesting points regarding the categorical interpretation of operations on the typing context $\Gamma$.

First let us see how to interpret variables. Because typing contexts $\Gamma$ are interpreted as iterated products, variables are interpreted as projections:

$$\left[\!\!\left[ \dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \right]\!\!\right] = \pi_x \tag{5.9}$$

In words: $\pi_x$ is the canonical projection $[\![\Gamma]\!] \to [\![A]\!]$ that extracts the "$x$th component" out of the iterated product used to define $[\![\Gamma]\!]$.

You may find this intepretation of variables unsatisfying: what exactly is this "canonical" projection, and what does it mean to take the "$x$th component"? Here is one way to make this precise. The trick is to reformulate the typing rule for variables in terms of two more elementary rules:

$$\dfrac{}{\Gamma, x : A \vdash x : A}\ \textsc{Hit} \qquad\qquad \dfrac{\Gamma \vdash x : A}{\Gamma, y : B \vdash x : A}\ \textsc{Miss}$$

Together these rules can be thought of as defining a procedure for checking whether a given binding $(x : A)$ lives in a given typing context $\Gamma$. The rule Hit covers the case where $(x : A)$ matches the variable right at the end of the typing context. The rule Miss covers the case where it doesn't and one has to keep looking further into $\Gamma$.

This reformulation of the variable rule allows for a precise definition of the handwavy $\pi_x$.

$$\left[\!\!\left[ \dfrac{}{\Gamma, x : A \vdash x : A}\ (\textsc{Hit}) \right]\!\!\right] = \pi_2 \tag{5.10}$$

$$\left[\!\!\left[ \dfrac{\vdots}{\dfrac{\Gamma \vdash x : A}{\Gamma, y : B \vdash x : A}\ (\textsc{Miss})} \right]\!\!\right] = \left[\!\!\left[ \dfrac{\vdots}{\Gamma \vdash x : A} \right]\!\!\right] \circ \pi_1 \tag{5.11}$$

Using these two rules, we can see that $\pi_x$ is actually a composite consisting of a string of $\pi_1$s followed by a $\pi_2$, with the length of this string depending on where $x$ appears in $\Gamma$. For instance,

$$\left[\!\!\left[ \frac{\vdots}{x:A, y:B, z:C \vdash x:A} \right]\!\!\right] \tag{5.12}$$

$$= \left[\!\!\left[ \frac{\vdots}{x:A, y:B \vdash x:A} \right]\!\!\right] \circ \pi_1 \tag{5.13}$$

$$= \left[\!\!\left[ \frac{\vdots}{x:A \vdash x:A} \right]\!\!\right] \circ \pi_1 \circ \pi_1 \tag{5.14}$$

$$= \pi_2 \circ \pi_1 \circ \pi_1. \tag{5.15}$$

Note that there are two $\pi_1$s in this composite. This corresponds to the fact that the de Bruijn index of $x$ in the context $x:A, y:B, z:C$ is two.[3]

[3] You are now a third of the way through the proof of Theorem 5.0.1.

With variables out of the way, let us now turn to let-bindings. These also illustrate an interesting interaction between operations on the typing context and categorical products.

$$\left[\!\!\left[ \frac{\dfrac{\vdots}{\Gamma \vdash M:A} \qquad \dfrac{\vdots}{\Gamma, x:A \vdash N:B}}{\Gamma \vdash \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N : B} \right]\!\!\right]$$

$$= \left[\!\!\left[ \frac{\vdots}{\Gamma, x:A \vdash N:B} \right]\!\!\right] \circ \left\langle \mathsf{id}_{[\![\Gamma]\!]}, \left[\!\!\left[ \frac{\vdots}{\Gamma \vdash M:A} \right]\!\!\right] \right\rangle$$

Let's break this down a bit. By induction, we have that $M$ denotes a morphism $[\![\Gamma]\!] \to [\![A]\!]$ and $N$ denotes a morphism $[\![\Gamma, x:A]\!] = [\![\Gamma]\!] \times [\![A]\!] \to [\![B]\!]$. The interpretation of $\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N$ composes these together. The twist is that, because the variables in $\Gamma$ are shared between $M$ and $N$, the object $[\![\Gamma]\!]$ must be plumbed around as well. This is accomplished by the morphism $\left\langle \mathsf{id}_{[\![\Gamma]\!]}, [\![\Gamma \vdash M:A]\!] \right\rangle$ guaranteed to exist by the universal property of products.

In practice, it's fairly laborious to write down the interpretation function explicitly as a function on derivations as we have done above. So people tend to shorten the more verbose $\left[\!\!\left[ \frac{\vdots}{\Gamma \vdash M:A} \right]\!\!\right]$ to just $[\![M]\!]$. Written this way, the foregoing discussion can be sum-

marized in terms of the following equations.

$$\begin{aligned}
[\![\langle\rangle]\!] &= \langle\rangle \\
[\![\mathsf{fst}\, M]\!] &= \pi_1 \circ [\![M]\!] \\
[\![\mathsf{snd}\, M]\!] &= \pi_2 \circ [\![M]\!] \\
[\![\langle M, N\rangle]\!] &= \langle [\![M]\!], [\![N]\!]\rangle \\
[\![x]\!] &= \pi_x \\
[\![\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N]\!] &= [\![N]\!] \circ \langle \mathsf{id}_{[\![\Gamma]\!]}, [\![M]\!]\rangle
\end{aligned}$$

This kind of equational definition is what you will typically encounter in a paper that uses categorical semantics.[4]

We have now seen how to interpret the types, typing contexts, and typing derivations of Calc. Only one task remains: interpret derivations of $\Gamma \vdash M \equiv N : A$. There are five groups of rules to interpret.

1. First there are the rules stating that $\equiv$ is an equivalence relation on well-typed terms.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M \equiv M : A}\ \textsc{Refl} \qquad \frac{\Gamma \vdash M \equiv N : A}{\Gamma \vdash N \equiv M : A}\ \textsc{Sym}$$

$$\frac{\Gamma \vdash M \equiv N : A \qquad \Gamma \vdash N \equiv O : A}{\Gamma \vdash M \equiv O : A}\ \textsc{Trans}$$

2. Next there are the *congruence rules*, so called because they state that $\equiv$ is a congruence for each of the Calc term formers.[5]

$$\frac{\Gamma \vdash M \equiv M' : A \qquad \Gamma \vdash N \equiv N' : B}{\Gamma \vdash \langle M, N\rangle \equiv \langle M', N'\rangle : A \times B}\ \textsc{Cong-Pair}$$

$$\frac{\Gamma \vdash M \equiv M' : A \times B}{\Gamma \vdash \mathsf{fst}\, M \equiv \mathsf{fst}\, M' : A}\ \textsc{Cong-Fst}$$

$$\frac{\Gamma \vdash M \equiv M' : A \times B}{\Gamma \vdash \mathsf{snd}\, M \equiv \mathsf{snd}\, M' : A}\ \textsc{Cong-Snd}$$

$$\frac{\Gamma \vdash M \equiv M' : A \qquad \Gamma, x : A \vdash N \equiv N' : B}{\Gamma \vdash (\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N) \equiv (\mathsf{let}\ x\ \mathsf{be}\ M'\ \mathsf{in}\ N') : B}\ \textsc{Cong-Let}$$

3. Then there are the $\beta$ *laws* that describe what happens when one applies an introduction form for a given type former followed by an elimination form.

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \mathsf{fst}\, \langle M, N\rangle \equiv M : A}\ \beta_1^{\times} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \mathsf{fst}\, \langle M, N\rangle \equiv N : B}\ \beta_2^{\times}$$

[4] In some cases it is desirable to have an interpretation function that operates directly on terms, rather than on derivations, as derivations can contain extraneous information not present in a term. In these cases a tricky issue comes up: for $[\![-]\!]$ to be a function on terms, it must be the case that the denotation of a term does not depend on the precise way in which it is proved to be well-typed. Checking this *coherence condition* involves showing that, for any term $M$, the denotations of all typing derivations for $M$ are equal [33]. In our case Calc satisfies the property that every term has at most one typing derivation when each bound variable is annotated with its type, so we won't worry too much about this.

[5] In general, an equivalence relation $\sim$ on a set $X$ is a *congruence* with respect to a function $f : X \to X$ if $x \sim x'$ implies $f(x) \sim f(x')$.

4. Then there are the $\eta$ *laws* that describe what happens when one applies an elimination form followed by an introduction form.

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \langle \mathsf{fst}\, M, \mathsf{snd}\, M \rangle \equiv M : A \times B}\ \eta^{\times} \qquad \frac{\Gamma \vdash M : \mathsf{Unit}}{\Gamma \vdash \langle\rangle \equiv M : \mathsf{Unit}}\ \eta^{\mathsf{Unit}}$$

5. Last but not least, there is the law that let-binding is equivalent to substitution.

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash (\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N) \equiv N[M/x] : B}\ \textsc{Let-Sub}$$

Let us tackle each of these groups one by one.

The rules in group (1) are easy to interpret: they correspond directly to the reflexivity, symmetry, and transitivity of equality of morphisms. For instance, the Sym rule holds because, if $[\![M]\!] = [\![N]\!]$ for some well-typed terms $M$ and $N$, then also $[\![N]\!] = [\![M]\!]$.[6]

The rules in group (2) are also relatively easy. In each case the proof boils down to showing that a given operation on morphisms respects equality. For instance, the rule Cong-Pair boils down to the following true statement: if $[\![M]\!] = [\![M']\!]$ and $[\![N]\!] = [\![N']\!]$ then $\langle [\![M]\!], [\![N]\!] \rangle = \langle [\![M']\!], [\![N']\!] \rangle$. The other cases are similarly straightforward.

The rules in group (3) follow from the definition of categorical product. The law $\beta_1^{\times}$ boils down to showing that $\pi_1 \circ \langle [\![M]\!], [\![N]\!] \rangle = [\![M]\!]$. The law $\beta_2^{\times}$ is analogous.

The rules in group (4) follow from the definitions of categorical product and terminal object. First, the law $\eta^{\mathsf{Unit}}$ follows from the fact that any two morphisms into a terminal object are equal. The law $\eta^{\times}$ boils down to showing that, for any morphism $[\![M]\!] : [\![\Gamma]\!] \to [\![A]\!] \times [\![B]\!]$, it is the case that $[\![M]\!] = \langle \pi_1 \circ [\![M]\!], \pi_2 \circ [\![M]\!] \rangle$. This follows from a uniqueness argument. By definition, $\langle \pi_1 \circ [\![M]\!], \pi_2 \circ [\![M]\!] \rangle$ is the unique morphism $f$ satisfying $\pi_1 \circ f = \pi_1 \circ [\![M]\!]$ and $\pi_2 \circ f = \pi_2 \circ [\![N]\!]$. But $[\![M]\!]$ satisfies this same property. Hence $[\![M]\!] = f = \langle \pi_1 \circ [\![M]\!], \pi_2 \circ [\![M]\!] \rangle$.

All that's left is Let-Sub—the rule stating that let-binding is equivalent to substitution. Validating this rule boils down to showing the following:

**Lemma 5.0.2** ("Let is substitution")**.** For all contexts $\Gamma$, types $A$, $B$ and terms $M$ and $N$ such that $\Gamma \vdash M : A$ and $\Gamma, x : A \vdash N : B$, it holds that $[\![N[M/x]]\!] = [\![N]\!] \circ \langle \mathsf{id}_{[\![\Gamma]\!]}, [\![M]\!] \rangle$.

This lemma essentially relates the syntactic operation of substituting of $M$ into $N$ with the semantic operation of composing the morphism $[\![N]\!]$ with $\langle \mathsf{id}_{[\![\Gamma]\!]}, [\![M]\!] \rangle$.

[6] The stickler may wonder how we know that $M$ and $N$ are well-typed here, since all we have in the premise of Sym is that $M \equiv N$. It turns out that $\equiv$ satisfies the following property: if $\Gamma \vdash M \equiv N : A$, then $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$. This fact can be established by induction on derivations of $\Gamma \vdash M \equiv N : A$, and we won't belabor it. A slicker way of maintaining this well-typedness invariant throughout the definition of $\equiv$ is to make well-typedness a *presupposition* of the judgment $\Gamma \vdash M \equiv N : A$ [27].

A natural proof strategy to try here is to proceed by induction on (a typing derivation for) $N$, as the substitution operation $N[M/x]$ is defined recursively on it. But this approach runs into a problem: the induction hypothesis one gets is not strong enough. The problematic case is when $N = (\text{let } x' \text{ be } N_1 \text{ in } N_2)$. In this case, we have that

1. $\Gamma, x : A \vdash N_1 : A'$ for some type $A'$
2. $\Gamma, x : A, x' : A' \vdash N_2 : B$

and the goal is to show that

$$[\![(\text{let } x' \text{ be } N_1 \text{ in } N_2)[M/x]]\!] = [\![\text{let } x' \text{ be } N_1 \text{ in } N_2]\!] \circ \langle \mathsf{id}_{[\![\Gamma]\!]}, [\![M]\!] \rangle. \tag{5.16}$$

By the definition of substitution, we have that

$$(\text{let } x' \text{ be } N_1 \text{ in } N_2)[M/x] = (\text{let } x' \text{ be } N_1[M/x] \text{ in } N_2[M/x]).$$

At first glance, this equation looks awfully provable from the above assumptions,[7] by applying the induction hypothesis to the terms $N_1[M/x]$ and $N_2[M/x]$ and then performing some algebraic manipulations. But a closer look will reveal that $N_2[M/x]$ actually does not fit the shape required for the induction hypothesis to apply. This is because our induction hypothesis only applies to substitutions of $M$ into terms well-typed in context $\Gamma, x : A$. On the other hand, $N_2[M/x]$ is a substitution of $\Gamma \vdash M : A$ into $\Gamma, x : A, x' : A' \vdash N_2 : B$. This extra red entry in the typing context does not match the shape of our induction hypothesis.

[7] You are now two thirds of the way through the proof of Theorem 5.0.1.

What has happened here is that the statement of Lemma 5.0.2 applies only to terms of the form $N[M/x]$ where $x$ is the right-most variable in the typing context used to type $N$. But while performing the substitution $N[M/x]$, we will recursively substitute $M$ into subterms where $x$ is *not* the right-most variable anymore, due to the presence of let-bindings. Thus there is a mismatch between the recursive structure of the substitution function and the shape of our inductive argument.

We can remedy this by strengthening Lemma 5.0.2 to the following statement, which applies to terms $N[M/x]$ where $x$ may occur anywhere in the context.

**Lemma 5.0.3** (Strengthening 5.0.2)**.** For all contexts $\Gamma$ and $\Delta$, types $A$, and terms $M$ and $N$ such that $\Gamma \vdash M : A$ and $\Gamma, x : A, \Delta \vdash N : B$, it holds that $[\![N[M/x]]\!] = [\![N]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle$.

The statement of Lemma 5.0.3 requires some clarification.

First, it involves morphisms $[\![\Gamma, \Delta]\!] \xrightarrow{\pi_{[\![\Gamma]\!]}} [\![\Gamma]\!]$ and $[\![\Gamma, \Delta]\!] \xrightarrow{\pi_{[\![\Delta]\!]}} [\![\Delta]\!]$. These morphisms are defined by the following fact, which can be

proved by induction on $\Delta$: for all $\Gamma$ and $\Delta$ it holds that $[\![\Gamma, \Delta]\!]$ is a product of $[\![\Gamma]\!]$ and $[\![\Delta]\!]$. The morphisms $\pi_{[\![\Gamma]\!]}$ and $\pi_{[\![\Delta]\!]}$ are the projections out of this product.

Second, we have used a ternary angle-bracket operator to form the morphism $\langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle$. This notation is justified by the following fact, which also can be proved by induction on $\Delta$: for all $\Gamma$ and $A$ and $\Delta$ it holds that $[\![\Gamma, x : A, \Delta]\!]$ is a ternary product of $[\![\Gamma]\!]$, $[\![A]\!]$, and $[\![\Delta]\!]$. The ternary angle-bracket then denotes the unique morphism given by the universal property of this ternary product.

We are now ready to prove Lemma 5.0.3.

*Proof of Lemma 5.0.3.* By induction on the derivation of $\Gamma, x : A, \Delta \vdash N : A$. We start with the trickiest case:

- Suppose $N = (\mathsf{let}\ x' \ \mathsf{be}\ N_1\ \mathsf{in}\ N_2)$ for some $\Gamma, x : A, \Delta \vdash N_1 : A'$ and some $\Gamma, x : A, \Delta, x' : A' \vdash N_2 : B$. Then both the left- and right-hand sides of the desired equation simplify to the same expression. On the one hand we have

$$
\begin{aligned}
&[\![(\mathsf{let}\ x' \ \mathsf{be}\ N_1\ \mathsf{in}\ N_2)[M/x]]\!] \\
&= [\![\mathsf{let}\ x' \ \mathsf{be}\ N_1[M/x]\ \mathsf{in}\ N_2[M/x]]\!] \\
&= [\![N_2[M/x]]\!] \circ \langle \pi_{[\![\Gamma, x : A, \Delta]\!]}, [\![N_1[M/x]]\!] \rangle \\
&\overset{\text{IH}}{=} ([\![N_2]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta, x' : A']\!]} \rangle) \circ \langle \mathsf{id}_{[\![\Gamma, x : A, \Delta]\!]}, [\![N_1[M/x]]\!] \rangle \\
&\overset{\text{IH}}{=} ([\![N_2]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta, x' : A']\!]} \rangle) \circ p \\
&\qquad \text{where } p = \langle \mathsf{id}_{[\![\Gamma, x : A, \Delta]\!]}, [\![N_1]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle \rangle \\
&= ([\![N_2]\!] \circ \langle \pi_{[\![\Gamma]\!]} \circ p, [\![M]\!] \circ \pi_{[\![\Gamma]\!]} \circ p, \pi_{[\![\Delta, x' : A'']\!]} \circ p \rangle) \\
&= ([\![N_2]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta, x' : A']\!]} \circ p \rangle) \\
&= ([\![N_2]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \langle \pi_{[\![\Delta]\!]}, [\![N_1]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle \rangle \rangle
\end{aligned}
$$

  and on the other hand we have

$$
\begin{aligned}
&[\![\mathsf{let}\ x' \ \mathsf{be}\ N_1\ \mathsf{in}\ N_2]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle \\
&= [\![N_2]\!] \circ \langle \mathsf{id}_{[\![\Gamma, x : A, \Delta]\!]}, [\![N_1]\!] \rangle \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle \\
&= [\![N_2]\!] \circ \langle \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle, [\![N_1]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle \rangle.
\end{aligned}
$$

  The equality of these two morphisms boils down to the equality of

$$\langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \langle \pi_{[\![\Delta]\!]}, [\![N_1]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle \rangle \rangle$$

  and

$$\langle \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle, [\![N_1]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle \rangle.$$

  This equality follows from a uniqueness argument: both of these morphisms are the unique $f : [\![\Gamma, \Delta]\!] \to [\![\Gamma, x : A, \Delta, x' : A']\!]$ satisfying the following four properties:

1. $\pi_{[\![\Gamma]\!]} \circ f = \pi_{[\![\Gamma]\!]}$
2. $\pi_{[\![A]\!]} \circ f = [\![M]\!] \circ \pi_{[\![\Gamma]\!]}$
3. $\pi_{[\![\Delta]\!]} \circ f = [\![M]\!] \circ \pi_{[\![\Delta]\!]}$
4. $\pi_{[\![A']\!]} \circ f = [\![N_1]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle$

This completes the proof of the case $N = (\mathsf{let}\ x'\ \mathsf{be}\ N_1\ \mathsf{in}\ N_2)$.

The other cases are much easier.

- Suppose $N = \langle\rangle$. Then

$$\begin{aligned}[\![\langle\rangle[M/x]]\!] = [\![\langle\rangle]\!] = \langle\rangle &= \langle\rangle \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle \\ &= [\![\langle\rangle]\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle.\end{aligned}$$

- Suppose $N = \mathsf{fst}\ N'$ for some $\Gamma, x : A, \Delta \vdash \mathsf{fst}\ N'$. Then

$$\begin{aligned}[\![(\mathsf{fst}\ N')[M/x]]\!] = [\![\mathsf{fst}\ (N'[M/x])]\!] &= \pi_1 \circ [\![N'[M/x]]\!] \\ &\overset{\text{IH}}{=} \pi_1 \circ [\![N']\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle \\ &= [\![\mathsf{fst}\ N']\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle.\end{aligned}$$

- Case $N = \mathsf{snd}\ N'$ is similar.

- Suppose $N = \langle N_1, N_2 \rangle$ for some $B, N_1, N_2$ such that $B = B_1 \times B_2$ and $\Gamma, x : A, \Delta \vdash N_1 : B_1$ and $\Gamma, x : A, \Delta \vdash N_2 : B_2$. Then

$$\begin{aligned}&[\![\langle N_1, N_2 \rangle[M/x]]\!] \\ &= [\![\langle N_1[M/x], N_2[M/x] \rangle]\!] \\ &= \langle [\![N_1[M/x]]\!], [\![N_2[M/x]]\!] \rangle \\ &\overset{\text{IH}}{=} \langle [\![N_1]\!] \circ p, [\![N_2]\!] \circ p \rangle \text{ where } p = \langle \pi_{[\![\Gamma]\!]}, [\![\Gamma \vdash M : A]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle \\ &= \langle [\![N_1]\!], [\![N_2]\!] \rangle \circ p \\ &= [\![\langle N_1, N_2 \rangle]\!] \circ p\end{aligned}$$

- Suppose $N = x$. For this case, it will be helpful to disambiguate the ambient typing context under which a term is interpreted:

$$\begin{aligned}&[\![\Gamma, \Delta \vdash x[M/x] : A]\!] \\ &= [\![\Gamma, \Delta \vdash M : A]\!] \\ &\overset{(*)}{=} [\![\Gamma \vdash M : A]\!] \circ \pi_{[\![\Gamma]\!]} \\ &= \pi_x \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle\end{aligned}$$

The step marked $(*)$ can be proved as a separate lemma, by induction on typing derivations.

- Suppose $N = x'$ for some $x' \neq x$ such that $\Gamma, x : A, \Delta \vdash x' : A'$. Then

$$\begin{aligned}
&[\![\Gamma, \Delta \vdash x'[M/x] : A']\!] \\
&= [\![\Gamma, \Delta \vdash x' : A']\!] \\
&= \pi_{x'} \\
&= [\![\Gamma, x : A, \Delta \vdash x' : A']\!] \circ \langle \pi_{[\![\Gamma]\!]}, [\![M]\!] \circ \pi_{[\![\Gamma]\!]}, \pi_{[\![\Delta]\!]} \rangle.
\end{aligned}$$

This closes the induction, completing the proof of Lemma 5.0.3. □

Lemma 5.0.2 now follows as a corollary, by setting $\Delta = \bullet$. This in turn establishes the validity of the equational rule LET-SUB, which completes the interpretation of all of the equational axioms of CALC and hence concludes the proof of Theorem 5.0.1.[8] □

[8] You heard that right—Theorem 5.0.1 is proved. You are free! Well done if you made it this far.

## 5.1 *Working in internal languages*

Whew! That was quite a long proof. You may be wondering: what have we gained by spending so much energy on proving Theorem 5.0.1? From the PL perspective, Theorem 5.0.1 is not super exciting—CALC is a toy programming language. But Theorem 5.0.1 is extremely helpful for doing category theory. This is because it provides a convenient language for working with an arbitrary category that has finite products.

Let's see an example of this. Recall the divisor lattice of natural numbers $(\mathbb{N}, \preceq)$ extended with a terminal object (the number 0). We showed previously that this category has products, and that these products can be interpreted as greatest common divisors. So, Theorem 5.0.1 applies to this category, and we can use it to prove the following seemingly non-trivial fact about greatest common divisors:

**Proposition 5.1.1.** Let $X, Y, Z$ be natural numbers. Then, $\mathsf{gcd}(\mathsf{gcd}(X, Y), Z)$ divides $\mathsf{gcd}(Y, Z)$.

*Proof.* Recall that in this category, the $\mathsf{gcd}(X, Y)$ is the categorical product $X \times Y$, and divisibility corresponds to the existence of a morphism. So, the proof strategy is to use Theorem 5.0.1 to prove the existence of a morphism of the correct type.

First, we need a way in CALC of discussing arbitrary objects in a category. We do this by extending our grammar of CALC types with type formers for three arbitrary objects in the category:

$$A ::= \cdots \mid X \mid Y \mid Z$$

Then, we define the denotation of these three new types to be $[\![X]\!] = X, [\![Y]\!] = Y, [\![Z]\!] = Z$. Now, the proof amounts to showing that

there exists a morphism $(X \times Y) \times Z \to (Y, Z)$ using Theorem 5.0.1, which means we have to give a term of type $(X \times Y) \times Z \to Y \times Z$. Such a term is quite easy to find if you've ever programmed in a functional programming language:

$$p : (X \times Y) \times Z \vdash \langle \mathsf{fst}\,(\mathsf{snd}\,p), \mathsf{snd}\,p \rangle : Y \times Z$$

This completes the proof. □

Intuitively, Theorem 5.0.1 gives us license to reason about categorical products like how we are familiar reasoning about them in functional programs: we can project out of them, compose them with other projects, etc. A more elaborate application of this theorem is to establish equalities of certain morphisms using the equational theory of CALC:

**Proposition 5.1.2.** Let $\mathcal{C}$ be a category with products. Then $X \times Y \cong Y \times X$ for any objects $X, Y$ of $\mathcal{C}$.

*Proof.* The following argument will hold for any choice of objects $X$ and $Y$ in the category $\mathcal{C}$. For each such pair, extend the grammar of CALC types with new base types:

$$A ::= \cdots \mid \mathsf{X} \mid \mathsf{Y}.$$

Interpret these new types as the objects $X$ and $Y$ of the category $\mathcal{C}$:

$$[\![\mathsf{X}]\!] = X$$
$$[\![\mathsf{Y}]\!] = Y$$

Now define $p : \mathsf{X} \times \mathsf{Y} \vdash M : \mathsf{Y} \times \mathsf{X}$ by $M := \langle \mathsf{snd}\,p, \mathsf{fst}\,p \rangle$ and similarly define $q : \mathsf{Y} \times \mathsf{X} \vdash N : \mathsf{X} \times \mathsf{Y}$ by $N := \langle \mathsf{snd}\,q, \mathsf{fst}\,q \rangle$. By Theorem 5.0.1, we have morphisms $[\![M]\!] : X \times Y \to Y \times X$ and $[\![N]\!] : Y \times X \to X \times Y$. Simple equational reasoning then shows that

$$p : \mathsf{X} \times \mathsf{Y} \vdash N[M/q] \equiv p : \mathsf{X} \times \mathsf{Y} \tag{5.17}$$
$$q : \mathsf{Y} \times \mathsf{X} \vdash M[N/p] \equiv q : \mathsf{Y} \times \mathsf{X} \tag{5.18}$$

which by Theorem 5.0.1 implies $[\![N]\!] \circ [\![M]\!] = \mathsf{id}_{X \times Y}$ and $[\![M]\!] \circ [\![N]\!] = \mathsf{id}_{Y \times X}$ as needed to show $X \times Y \cong Y \times X$.[9] □

This style of proof is broadly referred to was *working in the internal language of a category*. You may have heard of other internal languages for categories; perhaps the most well-known are string diagrams, which we may return to later.

In the proof of Proposition 5.1.2, we extended CALC with new base types $\mathsf{X}$ and $\mathsf{Y}$ to model the objects $X$ and $Y$ of the category $\mathcal{C}$. We could give CALC access to other objects in $\mathcal{C}$ by extending it further.

[9] Those that are familiar with the Curry-Howard correspondence may notice something here: equalities in the internal language are proofs of equalities in the model category, so proving equalities in the model category feels like manipulating proof terms.

Taking this idea to its extreme, we could extend CALC with *all* objects in $\mathcal{C}$, and also all morphisms. This extension to CALC is known as the **internal language** of $\mathcal{C}$, and programming in it is known as "working internal to $\mathcal{C}$".

**Definition 14: Internal language for finite products**

Let $\mathcal{C}$ be a category with finite products. The **internal language** of $\mathcal{C}$ is the syntactic extension to CALC that adds the following:

- One primitive type $\mathsf{X}$ for each object $X$ of $\mathcal{C}$.
- One term former $\mathsf{f}(M)$ for each morphism $X \xrightarrow{f} Y$, with typing rule:

$$\frac{\Gamma \vdash M : \mathsf{X}}{\Gamma \vdash \mathsf{f}(M) : \mathsf{Y}}$$

- For each object $X$ of $\mathcal{C}$, an equational law:

$$\frac{\Gamma \vdash M : \mathsf{X}}{\Gamma \vdash \mathsf{id}(M) \equiv M : \mathsf{X}}$$

  Here $\mathsf{id}$ denotes the term former corresponding to $\mathsf{id}_X$.
- For all morphisms $f : Y \to Z$ and $g : X \to Y$ and $h : X \to Z$ in $\mathcal{C}$ such that $f \circ g = h$, an equational law:

$$\frac{\Gamma \vdash M : \mathsf{X}}{\Gamma \vdash \mathsf{f}(\mathsf{g}(M)) \equiv \mathsf{h}(M) : \mathsf{Z}}$$

**Proposition 5.1.3.** Any category $\mathcal{C}$ with finite products is a model of its own internal language in the evident way.

Internal languages are what formally make categories behave like "metalanguages" for PL. Metalanguage features correspond to additional structure: as we progress through the course, we will encounter universal constructions that model richer languages with features such as records, pattern matching and structural recursion, higher-order functions, and refinement types.

## 5.2 *CCCs are models of the simply-typed lambda calculus*

Cartesian closed categories give us all the structure that we need to interpret simply-typed lambda calculus programs. Recall the lan-

guage STLC:

$$\begin{aligned} A, B &::= \mathsf{Unit} \mid A \times B \mid A \to B \\ M, N &::= \langle\rangle \mid \langle M, N\rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \mid \lambda\, x.\, M \mid M\, N \mid x \end{aligned} \tag{STLC}$$

We give the following interpretation of STLC in a Cartesian closed category $\mathcal{C}$. First we interpret types:

$$\begin{aligned} [\![\mathsf{Unit}]\!] &= \mathbf{1} \\ [\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\ [\![A \to B]\!] &= [\![B]\!]^{[\![A]\!]} \end{aligned}$$

Then we interpret terms as morphisms:

$$\begin{aligned} [\![\langle\rangle]\!] &= \langle\rangle \\ [\![\mathsf{fst}\, M]\!] &= \pi_1 \circ [\![M]\!] \\ [\![\mathsf{snd}\, M]\!] &= \pi_2 \circ [\![M]\!] \\ [\![\langle M, N\rangle]\!] &= \langle [\![M]\!], [\![N]\!] \rangle \\ [\![x]\!] &= \pi_x \\ [\![\lambda x.M]\!] &= \lambda\, [\![M]\!] \\ [\![M\ N]\!] &= \mathsf{app} \circ \langle [\![M]\!], [\![N]\!] \rangle \end{aligned}$$

These morphisms satisfy the equational theory of the simply-typed lambda calculus:

**Theorem 5.2.1.** Every Cartesian closed category is a model of the simply-typed lambda calculus.

We'll spare you the details; if curious, check out some of the standard references [20, 9].

# 6
# *Yoneda I: Internal vs. external*

This section marks the start of a long journey that ends in the statement and proof of the Yoneda lemma. As with any important concept in category theory, the Yoneda lemma can be understood through a variety of perspectives. In these notes we will explore a particular perspective that feels most relevant to PL: the Yoneda lemma is a means of connecting the "internal" to the "external".

Consider, for example, the case of product. In the preceding sections, we have given you the definition of product that is most commonly found in standard textbooks on category theory. This definition has an "internal" character, because it defines product in a category $\mathcal{C}$ purely in terms of other objects and morphisms *inside* of $\mathcal{C}$.

But there is another way to define product. Think back to Theorem 5.0.1, which showed how to interpret a simple programming language in an arbitrary category $\mathcal{C}$ with finite products. The proof of Theorem 5.0.1 gave a recipe for interpreting each of the judgments of programming language in $\mathcal{C}$. Following this recipe, each of the typing rules became *functions on morphisms*. For example, the typing rule

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{fst}\, M : A}\ \textsc{T-Fst}$$

for projecting out the first component of a pair was interpreted as a function

$$\{\text{morphisms } [\![\Gamma]\!] \text{ to } [\![A \times B]\!]\} \longrightarrow \{\text{morphisms } [\![\Gamma]\!] \text{ to } [\![A]\!]\},$$

namely the function that sends interpretations of $[\![M]\!]$ to interpretations of $[\![\mathsf{fst}\, M]\!]$.

It turns out that we can promote this recipe into a *definition* for product. In contrast to the textbook definition in terms of objects and morphisms of $\mathcal{C}$, this definition characterizes the product in terms of *functions* between *sets* of morphisms. This gives it an "external"

character, because it defines the product in terms of set-theoretic objects (sets and functions) that live *outside* of $\mathcal{C}$.

First, we must make precise what we mean by this "external" definition of product that defines product in terms of functions on morphisms. To formalize this idea of "maps between morphisms", it's useful to have the following definition that we've put off for a long time.

**Definition 15: Hom-set**

Let $X$ and $Y$ be objects of a category $\mathcal{C}$. The *hom-set* of $\mathcal{C}$ at $X$ and $Y$, written $\mathcal{C}(X, Y)$, is the set of morphisms from $X$ to $Y$:

$$\mathcal{C}(X, Y) := \{f \mid \mathsf{dom}(f) = X, \mathsf{cod}(f) = Y\}.$$

The word "hom" is an abbreviation of "homomorphism", and the name hom-set comes from the fact many of the original motivating examples of categories had morphisms that were homomorphisms between algebraic structures.

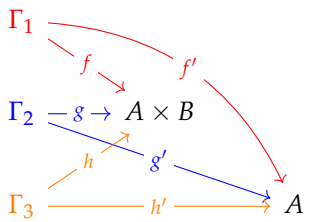
For example,

- In the category of finite sets and set functions FinSet, the hom-set $\mathsf{FinSet}(\{a, b\}, \{1, 2, 3\})$ is the set of all functions from $\{a, b\}$ to $\{1, 2, 3\}$.
- Hom-sets can be empty. In the divisibility category $(\mathbb{N}, \preceq)$, the hom-set $\mathbb{N}(3, 4) = \emptyset$, while $\mathbb{N}(3, 6) = \{\top\}$.

This definition lets us state formally what it means for an inference rule like T-Fst to denote a function from morphisms to morphisms. We shall say that the interpretation of T-Fst is a family of functions $(\mathsf{Fst}_\Gamma)_{\Gamma \in \mathcal{C}}$, indexed by the ambient context $\Gamma$, of the following type:

$$\mathsf{Fst}_\Gamma : \mathcal{C}(\Gamma, A \times B) \to \mathcal{C}(\Gamma, A)$$

We can visualize this family of functions as follows:




(6.1)

...

As an example of working with this function, we have that $\mathsf{Fst}_{\Gamma_1}(f) = f'$.

In addition to the basic type shown above, we will also require that Fst be *natural* in $\Gamma$. Intuitively, what this means is that Fst is in a sense "polymorphic in $\Gamma$". Just as polymorphic functions in System F respect "change of representation" for the type variables that they are

polymorphic over [32], we will require that the family of functions $(\mathsf{Fst}_\Gamma)_{\Gamma \in \mathsf{Ob}(\mathcal{C})}$ respect "change of $\Gamma$". Concretely, this means that if any input $f$ as shown in the picture above is changed into $f \circ s$ via some morphism $s : \Gamma' \to \Gamma$, it must be the case that $\mathsf{Fst}_\Gamma(f)$ is changed into $\mathsf{Fst}_{\Gamma'}(f \circ s)$ via $s$ too. Formally, the following *naturality condition* must hold for any such $f$ and $s$:

We will save a formal abstract definition of naturality for later on.

$$\mathsf{Fst}_\Gamma(f) \circ s = \mathsf{Fst}_{\Gamma'}(f \circ s)$$

Another intution for this condition comes from the requirement that the interpretation of a programming language be *stable under substitution*. For instance, we know that $\mathsf{fst}\, M$ is invariant under substitution for some (possibly multi-way) substitution $s$:

$$(\mathsf{fst}\, M)[s] \equiv \mathsf{fst}\, (M[s])$$

This program equation corresponds directly to the naturality condition given above.

Applying the above analysis to the typing rule T-Snd gives an analogous function for extracting the second component of a pair:

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{snd}\, M : B} \rightsquigarrow \mathsf{Snd}_\Gamma : \mathcal{C}(\Gamma, A \times B) \to \mathcal{C}(\Gamma, B)$$

$$\text{such that } \forall s : \Gamma' \to \Gamma, \mathsf{Snd}_\Gamma(f) \circ s = \mathsf{Snd}_\Gamma(f \circ s)$$

The rule T-Pair for forming pairs becomes a function that takes in two morphisms and bundles them together into a morphism into the product:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \rightsquigarrow \mathsf{Pair}_\Gamma : \mathcal{C}(\Gamma, A) \times \mathcal{C}(\Gamma, B) \to \mathcal{C}(\Gamma, A \times B)$$

$$\text{such that } \forall s : \Gamma' \to \Gamma, \mathsf{Pair}_\Gamma(f, g) \circ s = \mathsf{Pair}_\Gamma(f \circ s, g \circ s)$$

Finally, the $\beta$ and $\eta$ laws become equations between nested applications of the functions Fst, Snd, and Pair.

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \mathsf{fst}\, \langle M, N \rangle \equiv M} \rightsquigarrow \mathsf{Fst}_\Gamma(\mathsf{Pair}_\Gamma(f, g)) = f$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \mathsf{snd}\, \langle M, N \rangle \equiv N} \rightsquigarrow \mathsf{Snd}_\Gamma(\mathsf{Pair}_\Gamma(f, g)) = g$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M \equiv \langle \mathsf{fst}\, M, \mathsf{snd}\, M \rangle \equiv N} \rightsquigarrow \mathsf{Pair}_\Gamma(\mathsf{Fst}_\Gamma(f), \mathsf{Snd}_\Gamma(f)) = f$$

Packaging up all of these constraints into a definition,[1]

[1] We have chosen to name this concept "PL-product" as it was derived from the typing rules for product types in a programming language.

**Definition 16: PL-product**

Let $A$ and $B$ be objects of a category $\mathcal{C}$. A *PL-product* of $A$ and $B$ consists of:

- An object $A \times B$
- $\mathcal{C}$-indexed functions:

$$\begin{aligned}\mathsf{Fst}_\Gamma &: \mathcal{C}(\Gamma, A \times B) \to \mathcal{C}(\Gamma, A)\\ \mathsf{Snd}_\Gamma &: \mathcal{C}(\Gamma, A \times B) \to \mathcal{C}(\Gamma, B)\\ \mathsf{Pair}_\Gamma &: \mathcal{C}(\Gamma, A) \times \mathcal{C}(\Gamma, B) \to \mathcal{C}(\Gamma, A \times B)\end{aligned}$$

satisfying the necessary properties to be $\mathcal{C}$-indexed, i.e.:

- Fst, Snd, and Pair "respect substitution": for all $\Gamma' \xrightarrow{s} \Gamma$,

$$\begin{aligned}\mathsf{Fst}_\Gamma(f) \circ s &= \mathsf{Fst}_{\Gamma'}(f \circ s) && \text{for all } \Gamma \xrightarrow{f} A \times B\\ \mathsf{Snd}_\Gamma(f) \circ s &= \mathsf{Snd}_{\Gamma'}(f \circ s) && \text{for all } \Gamma \xrightarrow{f} A \times B\\ \mathsf{Pair}_\Gamma(f, g) \circ s &= \mathsf{Pair}_{\Gamma'}(f \circ s, g \circ s) && \text{for all } \Gamma \xrightarrow{f} A \text{ and } \Gamma \xrightarrow{g} B\end{aligned}$$

- Analogs of the $\beta$ and $\eta$ laws hold:

$$\begin{aligned}\mathsf{Fst}_\Gamma(\mathsf{Pair}_\Gamma(f, g)) &= f && \text{for all } \Gamma \xrightarrow{f} A \text{ and } \Gamma \xrightarrow{g} B\\ \mathsf{Snd}_\Gamma(\mathsf{Pair}_\Gamma(f, g)) &= g && \text{for all } \Gamma \xrightarrow{f} A \text{ and } \Gamma \xrightarrow{g} B\\ \mathsf{Pair}_\Gamma(\mathsf{Fst}_\Gamma(f), \mathsf{Snd}_\Gamma(f)) &= f && \text{for all } \Gamma \xrightarrow{f} A \times B\end{aligned}$$

Now we can state an important theorem.

**Theorem 6.0.1.** PL-products are equivalent to ordinary products.

This theorem breaks down into multiple components.

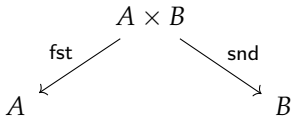
1. Every PL-product can be turned into a categorical product.
2. Every categorical product can be turned into a PL product.
3. Constructions (1) and (2) round-trip to the identity.

The proof tackles each piece separately.

**Construction 6.0.2.** Every PL product of $A$ and $B$ can be turned into a categorical product.[2]

*Proof.* Suppose you have $(A \times B, \mathsf{Fst}_\Gamma, \mathsf{Snd}_\Gamma, \mathsf{Pair}_\Gamma)$. Remember, $\mathsf{Fst}_\Gamma$ (and all these other functions) behave like polymorphic functions, and we need to find a diagram like:

[2] In general, we will use the word "construction" to signal a lemma whose proof details are going to be relevant later. In this case, we will need the precise definition of how PL products are turned into categorical ones later on, to prove that the two constructions we have done round-trip to the identity.




How do we get a morphism fst? The key is that Fst is polymorphic in $\Gamma$, so we can plug in $A \times B$ there to get something of the required type:

$$\mathsf{Fst}_{A\times B} : \mathcal{C}(A \times B, A \times B) \to \mathcal{C}(A \times B, A)$$

Now we need to get a morphism of the required type of fst. From here, we can play "type tetris": we call $\mathsf{Fst}_{A\times B}$ with a morphism that meets the type constraint, the identity!

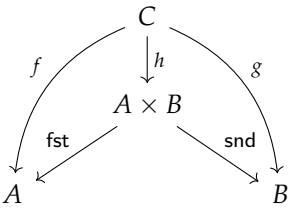
$$\mathsf{Fst}_{A\times B}(\mathrm{id}_{A\times B}) =: \mathsf{fst}$$

We can do the same thing to get the morphism snd.

$$\mathsf{Snd}_{A\times B}(\mathrm{id}_{A\times B}) =: \mathsf{snd}$$

Now the question: does this satisfy the universal property for products?

First, let's show that there exists an $h$ such that the diagram commutes:




Where can we find $h$? We can let $h = \mathsf{Pair}_C(f, g)$.

We need to show that the diagram commutes. The left triangle commutes by the following equational argument:

$$\begin{aligned}
\mathsf{fst} \circ h &= \mathsf{Fst}_{A\times B}(\mathrm{id}_{A\times B}) \circ \mathsf{Pair}_C(f,g) && \text{expanding definitions} \\
&= \mathsf{Fst}_C(\mathrm{id}_{A\times B} \circ \mathsf{Pair}_C(f,g)) && \text{naturality of Fst} \\
&= \mathsf{Fst}_C(\mathsf{Pair}_C(f,g)) && \text{identity law} \\
&= f && \beta
\end{aligned}$$

The right triangle commutes similarly.

Finally, we need to show that $\mathsf{Pair}_C(f, g)$ is the unique $h$ making the above diagram commute. For this we use the $\eta$ law. Suppose $h'$ satisfies $\mathsf{fst} \circ h = f$ and $\mathsf{snd} \circ h = g$. Then the following calculation

uses the $\eta$ law to establish that $h' = h$.

$$\begin{aligned}
h' &= \mathsf{Pair}_\Gamma(\mathsf{Fst}_\Gamma(h'), \mathsf{Snd}_\Gamma(h')) && \eta \\
&= \mathsf{Pair}_\Gamma(\mathsf{Fst}_{A\times B}(\mathsf{id}_{A\times B}) \circ h', \mathsf{Snd}_{A\times B}(\mathsf{id}_{A\times B}) \circ h') && \text{substitution} \\
&= \mathsf{Pair}_\Gamma(\mathsf{fst} \circ h', \mathsf{snd} \circ h') \\
&= \mathsf{Pair}_\Gamma(f, g) && \text{assumption} \\
&= \mathsf{Pair}_\Gamma(\mathsf{fst} \circ h, \mathsf{snd} \circ h) && \text{shown above} \\
&= \mathsf{Pair}_\Gamma(\mathsf{Fst}_{A\times B}(\mathsf{id}_{A\times B}) \circ h, \mathsf{Snd}_{A\times B}(\mathsf{id}_{A\times B}) \circ h) \\
&= \mathsf{Pair}_\Gamma(\mathsf{Fst}_\Gamma(h), \mathsf{Snd}_\Gamma(h)) && \text{substitution} \\
&= h && \eta
\end{aligned}$$

□

We can also go in the other direction: every product in the ordinary categorical sense can be turned into a PL-product.

**Construction 6.0.3.** Any product can be turned into a PL-product.

*Proof.* The previous construction provides a strong hint as to how to do this one. Given a product $(A \times B, \mathsf{fst}, \mathsf{snd})$, define

$$\begin{aligned}
\mathsf{Fst}_\Gamma(f) &= \mathsf{fst} \circ f \\
\mathsf{Snd}_\Gamma(f) &= \mathsf{snd} \circ f
\end{aligned}$$

for all morphisms $\Gamma \xrightarrow{f} A \times B$, and

$$\mathsf{Pair}_\Gamma(f, g) = \langle f, g \rangle$$

for all morphisms $\Gamma \xrightarrow{f} A$ and $\Gamma \xrightarrow{g} B$. The fact that Fst and Snd respect substitution boil down to the following equations:

$$\begin{aligned}
(\mathsf{fst} \circ f) \circ s &= \mathsf{fst} \circ (f \circ s) \\
(\mathsf{snd} \circ f) \circ s &= \mathsf{snd} \circ (f \circ s)
\end{aligned}$$

The fact that Pair respects substitution boils down to Proposition 3.4.1.

The $\beta$ law follows from the equations $\mathsf{fst} \circ \langle f, g \rangle = f$ and $\mathsf{snd} \circ \langle f, g \rangle = g$, and the $\eta$ law from the uniqueness property of $\langle f, g \rangle$. □

Finally, we show that the two constructions just described roundtrip to the identity.

**Proposition 6.0.4.** Constructions 6.0.2 and 6.0.3 are mutually inverse.

*Proof.* We show both roundtrips are the identity. First, suppose one has a categorical product of $A$ and $B$, hence a tuple $(A \times B, \mathsf{fst}, \mathsf{snd})$

satisfying the universal property of product. Applying Theorem 6.0.3 yields a PL-product $(A \times B, \mathsf{Fst}, \mathsf{Snd}, \mathsf{Pair})$ where

$$\begin{aligned}\mathsf{Fst}_\Gamma(f) &= \mathsf{fst} \circ f \\ \mathsf{Snd}_\Gamma(f) &= \mathsf{snd} \circ f \\ \mathsf{Pair}_\Gamma(f,g) &= \langle f, g \rangle\end{aligned}$$

Next, applying Theorem 6.0.2 to this PL-product yields a tuple $(A \times B, \mathsf{fst}', \mathsf{snd}')$ where

$$\begin{aligned}\mathsf{fst}' &= \mathsf{Fst}_{A\times B}(\mathsf{id}_{A\times B}) \\ \mathsf{snd}' &= \mathsf{Snd}_{A\times B}(\mathsf{id}_{A\times B})\end{aligned}$$

The goal is now to show that $\mathsf{fst} = \mathsf{fst}'$ and $\mathsf{snd} = \mathsf{snd}'$. This follows from a straightforward calculation. We show only the proof of $\mathsf{fst} = \mathsf{fst}'$; the case of $\mathsf{snd}$ is analogous.

$$\begin{aligned}\mathsf{fst}' &= \mathsf{Fst}_{A\times B}(\mathsf{id}_{A\times B}) \\ &= \mathsf{fst}_{A\times B} \circ \mathsf{id}_{A\times B} \\ &= \mathsf{fst}_{A\times B}\end{aligned}$$

Now for the other roundtrip. Suppose one has a PL-product $(A \times B, \mathsf{Fst}, \mathsf{Snd}, \mathsf{Pair})$. Applying Theorem 6.0.2 to this PL-product yields the tuple product $(A \times B, \mathsf{fst}, \mathsf{snd})$ where

$$\begin{aligned}\mathsf{fst} &= \mathsf{Fst}_{A\times B}(\mathsf{id}_{A\times B}) \\ \mathsf{snd} &= \mathsf{Snd}_{A\times B}(\mathsf{id}_{A\times B})\end{aligned}$$

Then, applying Theorem 6.0.3 to this product yields $(A \times B, \mathsf{Fst}', \mathsf{Snd}', \mathsf{Pair}')$ where

$$\begin{aligned}\mathsf{Fst}_\Gamma(f) &= \mathsf{fst} \circ f \\ \mathsf{Snd}_\Gamma(f) &= \mathsf{snd} \circ f \\ \mathsf{Pair}_\Gamma(f,g) &= \langle f, g \rangle\end{aligned}$$

Note that in the final equation, $\langle f, g \rangle$ is guaranteed to exist by the universal property of $(A \times B, \mathsf{fst}, \mathsf{snd})$ established in Theorem 6.0.2.

The goal is now to show that $\mathsf{Fst} = \mathsf{Fst}'$ and $\mathsf{Snd} = \mathsf{Snd}'$ and $\mathsf{Pair} = \mathsf{Pair}'$. By function extensionality, it's enough to show these functions are equal for any object $\Gamma$ and morphisms out of $\Gamma$. First, we have

$$\mathsf{Fst}'_\Gamma(f) = \mathsf{fst} \circ f = \mathsf{Fst}_{A\times B}(\mathsf{id}_{A\times B}) \circ f = \mathsf{Fst}_\Gamma(f).$$

The case of $\mathsf{Snd}$ is analogous. Then, we have

$$\mathsf{Pair}'_\Gamma(f,g) = \langle f, g \rangle \overset{(!)}{=} \mathsf{Pair}_\Gamma(f,g).$$

Note that the equation marked (!) follows from the fact that, in the proof of Theorem 6.0.2, the unique mediating map $h$ making the relevant product diagram involving $f$ and $g$ commute was built using Pair.[3] □

[3] It is also possible to establish (!) without rifling through the proof of Theorem 6.0.2, by appealing to the uniqueness property of the morphism $\langle f, g \rangle$.

# 7
# *Yoneda II: indexed set theory*

In the last chapter we saw an important example of functions indexed by objects in a category, which formed the components of the PL-pair. This idea of sets and functions indexed by objects in a category is pervasive in programming languages and in category theory, and in this chapter we will spend some time exploring these constructions in their own right.

Many of the notions that you are familiar with from set theory will have $\mathcal{C}$-indexed analogues:

| **Ordinary** | **$\mathcal{C}$-Indexed** | **Intuition** |
|---|---|---|
| Sets | $\mathcal{C}$-indexed sets | A copy of $\mathcal{C}$ inside of sets |
| Functions | $\mathcal{C}$-indexed functions | Polymorphic functions |
| Cartesian products | $\mathcal{C}$-indexed product | Cartesian product that respects $\mathcal{C}$-indexing |
| ... | | |

We will make use of these analogies extensively. But before we do, here is the formal definition of a $\mathcal{C}$-indexed set:

Use your typechecker! Some of these compositions $\circ$ are composition of morphisms, and some are composition of functions.

**Definition 17: $\mathcal{C}$-indexed set**

For a category $\mathcal{C}$, a $\mathcal{C}$-indexed set $F$ is:

- A family of sets $F(X)$ for each object $X$ in $\mathcal{C}$
- For each morphism $X \xrightarrow{f} Y$ in $\mathcal{C}$, a function $F(f) : F(Y) \to F(X)$ satisfying the following two *functoriality properties*:
  - preserves identity: $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$
  - preserves composition: $F(g \circ f) = F(f) \circ F(g)$ for all $X \xrightarrow{f} Y \xrightarrow{g} Z$.

A $\mathcal{C}$-indexed set is also known as a **presheaf**. Let's see some examples of $\mathcal{C}$-indexed sets:

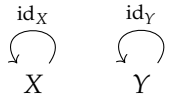
We will see soon that a $\mathcal{C}$-indexed set is an instance of a general phenomenon called a *functor*, which can be thought of as a structure-preserving map between categories.

- Consider a category with 1 object $\bullet$ and an identity morphism id:


(7.1)

Every $\mathcal{C}$-indexed set $F$ for this category is a set: for instance, we might have that $F(\bullet) = \{a, b\}$. By identity preservation, we must have that $F(\mathrm{id}) = \mathrm{id}_{\{a,b\}}$. Since there's only one morphism, composition preservation is trivial.

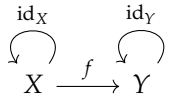
- Next, let's explore what $\mathcal{C}$-indexed sets look like on a more interesting indexing category, a category with two objects each with an identity morphism:




(7.2)

In this case, a $\mathcal{C}$-indexed set corresponds with a pair of sets. For example, $F(X) = \{a, b\}$ and $F(Y) = \{c, d\}$. Identity preservation means $F$ maps identities to identities, as before.

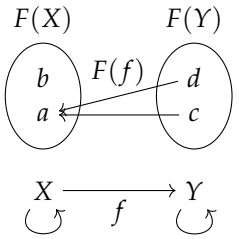
- Now let's see an example with a less trivial example of an indexing category with a non-identity morphism:



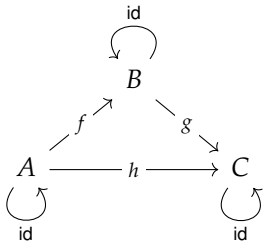

(7.3)

In this category, a $\mathcal{C}$-indexed set $F$ again maps each object to 2 sets, for example $F(X) = \{a, b\}$ and $F(Y) = \{c, d\}$. But now, $F$ must have an action on the morphism $f$: we can choose any function $F(f) : F(Y) \to F(X)$. Note that the direction of the function here is opposite the direction of the morphism in the indexing category. For example, we may choose $F(f) = x \mapsto a$. So, the $\mathcal{C}$-indexed sets on this category correspond to *functions between sets*.

It can be convenient to visualize $\mathcal{C}$-indexed sets $F$ as "balloon" diagrams: (1) above each object $X$ in $\mathcal{C}$ there is a visualization of the set $F(X)$, and (2) for each morphism $X \xrightarrow{f} Y$ in $\mathcal{C}$ there is an internal picture of the function $f : Y \to X$. For example, we can draw $F$ above as the following balloon diagram:



At this point it might be a good exercise to consider a few more cases of $\mathcal{C}$-indexed sets. In particular, try to draw a balloon diagram that shows an example of a $\mathcal{C}$-indexed set for the following 3-object category:




(7.4)

## 7.1 $\mathcal{C}$-indexed functions

Next, let's define the notion of a $\mathcal{C}$-indexed function, which should behave like a "polymorphic function":[1]

[1] We will see soon that a $\mathcal{C}$-indexed function is an instance of a general phenomenon called a *natural transformation* between functors $F$ and $G$.

**Definition 18: $\mathcal{C}$-indexed function**

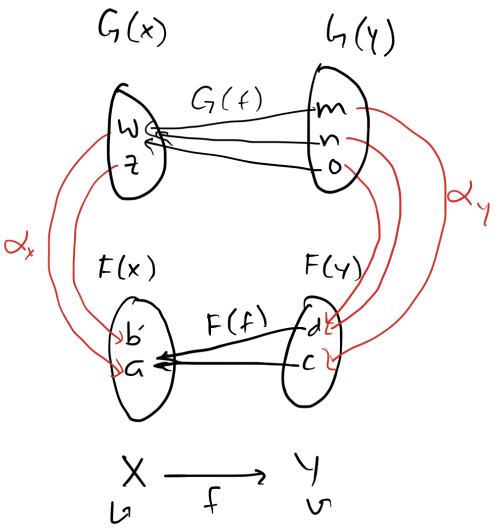
Given a category $\mathcal{C}$ and two $\mathcal{C}$-indexed sets $F$ and $G$, a $\mathcal{C}$-indexed function $\alpha : F \Rightarrow G$ is a family of functions $\alpha_X : F(X) \to G(X)$ for objects $X$ of $\mathcal{C}$ satisfying the following *naturality condition*: $\alpha_X \circ F(f) = G(f) \circ \alpha_Y$ for any $\mathcal{C}$ morphism $X \xrightarrow{f} Y$. This naturality condition can be visualized as a commutative square:

$$\begin{array}{ccc} F(Y) & \xrightarrow{\alpha_Y} & G(Y) \\ \downarrow{\scriptstyle F(f)} & & \downarrow{\scriptstyle G(f)} \\ F(X) & \xrightarrow{\alpha_X} & G(X) \end{array}$$

Once again let's work through some examples of $\mathcal{C}$-indexed functions to get a sense for how they behave.

Returning to the one-object category in (7.1), $\mathcal{C}$-indexed functions are simply functions between sets. Concretely, suppose $F(\bullet) = \{a, b\}$ and $G(\bullet) = \{c, d\}$. Then, there is a $\mathcal{C}$-indexed function $\alpha$ where $\alpha_\bullet : F(\bullet) \to G(\bullet)$ is a function. The naturality constraint is somewhat trivially satisfied, but is worth checking as an exercise. Note that there are 4 possible choices of $\alpha$ here, one for each possible function between the sets $F(\bullet)$ and $G(\bullet)$.

A much more interesting case is the 2-object category with a morphism between them in (7.3). We can visualize the data of a particular $\mathcal{C}$-indexed function $\alpha$ in this category by drawing a "stacked balloon diagram":




(7.5)

In this picture, we have $G(X) = \{w, z\}$, $G(Y) = \{m, n, o\}$, $F(X) = \{b, a\}$, and $F(Y) = \{c, d\}$. Then, a particular $\mathcal{C}$-indexed function $\alpha : G \Rightarrow F$ is a family of functions $\alpha_X : G(X) \to F(X)$ and $\alpha_Y : G(Y) \to F(Y)$; an example of one such choice of $\alpha$ is shown in red. To check that this is a valid choice of $\alpha$, we need to check that it satisfies the naturality condition, i.e. that $F(f) \circ \alpha_y = \alpha_X \circ G(F)$. To check this, we can play the "two finger game": put your finger on each particular element of $G(Y)$ and follow the two possible paths that it can take, and make sure your fingers end up in the same spot in $F(X)$. For instance, following the element $m$, we have $\alpha_Y(m) = c$, then $F(f)(c) = a$; in the other path, we have $G(f)(m) = w$, and $\alpha_X(w) = a$. So, these two paths agree on what to do with $m$, and so do all the other choices of starting element in $G(Y)$.

At this point, it's a good exercise to try to find all possible $\alpha : G \Rightarrow F$ for the $\mathcal{C}$-indexed sets $F$ and $G$ shown in (7.5).

## 7.2 *The category of $\mathcal{C}$-indexed sets (presheaf categories)*

At this point, you might be smelling something categorical: might it be the case that, for any category $\mathcal{C}$, one can form a category whose objects are all possible $\mathcal{C}$-indexed sets and whose morphisms are all possible $\mathcal{C}$-indexed functions? Indeed one can, but one requires the following to show this:

- Just as there is an identity function in set theory, there is an identity $\mathcal{C}$-indexed function in $\mathcal{C}$-indexed set theory. For each $\mathcal{C}$-indexed set $P$, there is a $\mathcal{C}$-indexed function $\mathsf{id}_P : P \Rightarrow P$ defined by $\mathsf{id}_{P,X} = \left( P(X) \xrightarrow{\mathsf{id}_{P(X)}} P(X) \right)$. The naturality condition amounts to

commutativity of the following diagram.

$$\begin{array}{ccc} F(Y) & \xrightarrow{\mathsf{id}_{F(Y)}} & F(Y) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle F(f)} \\ F(X) & \xrightarrow[\mathsf{id}_{F(X)}]{} & F(X) \end{array}$$

- $\mathcal{C}$-indexed functions can also be composed: the composition of $\alpha : Q \Rightarrow R$ and $\beta : P \Rightarrow Q$, written $\alpha \circ \beta : P \Rightarrow R$, is defined by $(\alpha \circ \beta)_X = \alpha_X \circ \beta_X$. The naturality condition amounts to the commutativity of the following diagram.

$$\begin{array}{ccccc} F(Y) & \xrightarrow{\beta_Y} & G(Y) & \xrightarrow{\alpha_Y} & H(Y) \\ \downarrow{\scriptstyle F(f)} & & \downarrow{\scriptstyle G(f)} & & \downarrow{\scriptstyle H(f)} \\ F(X) & \xrightarrow[\beta_X]{} & G(X) & \xrightarrow[\alpha_X]{} & H(X) \end{array}$$

From this we can conclude the following definition is well-formed:[2]

[2] This definition technically runs afoul of some cardinality limitations in our original definition of a category (Definition 1). Do not worry; the original definition of category can be adjusted to account for this fact without invalidating anything we have done so far.

**Definition 19: Category of $\mathcal{C}$-indexed sets**

Let $\mathcal{C}$ be a category. Then, the *category of $\mathcal{C}$-indexed sets* is the category whose objects are $\mathcal{C}$-indexed sets and whose morphisms are $\mathcal{C}$-indexed functions, where identities and composition of morphisms are defined as above.

This category is also called the **category of presheaves on** $\mathcal{C}$. In general, $\mathcal{C}$-indexed set theory looks kind of like set theory "with shape $\mathcal{C}$". The category of $\mathcal{C}$-indexed sets has fantastically rich structure: in fact, it is something called a *topos*, an idea that we will return to later. For now, the most exciting thing we can say about this category is that it is Cartesian-closed, meaning that it has products. Let's show this fact:

**Theorem 7.2.1.** Let $\mathcal{C}$ be a category. Then, the category of presheaves on $\mathcal{C}$ has products.

*Proof.* We prove this using the familiar internal argument involving the universal property of products. We begin by identifying a candidate $\mathcal{C}$-indexed set, and then we prove that it satisfies the universal property of products in the category of presheaves on $\mathcal{C}$:

**Definition 20: $\mathcal{C}$-indexed product**

Given a category $\mathcal{C}$ and two $\mathcal{C}$-indexed sets $F$ and $G$, the $\mathcal{C}$-indexed product of $F$ and $G$ is the $\mathcal{C}$-indexed set $F \times G$ defined

by:

- On objects $X \in \mathcal{C}$, $(F \times G)(X) = F(X) \times G(X)$
- On morphisms $X \xrightarrow{f} Y$, $(F \times G)(f)$ is the function $F(Y) \times G(Y) \to F(X) \times G(X)$ defined by $(F \times G)(f) = (x, y) \mapsto (F(f)(x), G(f)(y))$

Alongside $F \times G$ there are $\mathcal{C}$-indexed functions $\pi_1 : F \times G \Rightarrow F$ and $\pi_2 : F \times G \Rightarrow G$ defined:

$$\begin{aligned} \pi_{1,X} : (F \times G)(X) &\to F(X) \\ = (x, y) &\mapsto x \end{aligned}$$

and

$$\begin{aligned} \pi_{2,X} : (F \times G)(X) &\to G(X) \\ = (x, y) &\mapsto y \end{aligned}$$

First we need to show $F \times G$ is a $\mathcal{C}$-indexed set:

- *Preserves identity*:

$$\begin{aligned} (F \times G)(\mathrm{id}_X) &= (a, b) \mapsto (\mathrm{id}_{F(X)}(a), \mathrm{id}_{G(X)}(b)) \\ &= (a, b) \mapsto \mathrm{id}_{F(X) \times G(X)}(a, b) \\ &= \mathrm{id}_{F(X) \times G(X)}. \end{aligned}$$

- *Preserves composition*: For morphisms $X \xrightarrow{f} Y \xrightarrow{g} Z$:

$$\begin{aligned} (F \times G)(g \circ f) &= (a, b) \mapsto (F(g \circ f)(a), G(g \circ f)(b)) && \text{Def} \\ &= (a, b) \mapsto \big((F(f) \circ F(g))(a), (G(f) \circ G(g))(b)\big) && \text{Functoriality of } F \text{ and } G \\ &= (a, b) \mapsto \big[(F \times G)(f) \circ (F \times G)(g)\big](a, b) && \text{Def} \\ &= (F \times G)(f) \circ (F \times G)(g). \end{aligned}$$

We also need to show that $\pi_1$ and $\pi_2$ are $\mathcal{C}$-indexed functions. Let $X \xrightarrow{f} Y$ be a morphism in $\mathcal{C}$. Then, we must show the naturality diagram commutes:

$$\begin{array}{ccc} (F \times G)(Y) & \xrightarrow{\pi_{1,Y}} & F(Y) \\ \downarrow{\scriptstyle (F \times G)(f)} & & \downarrow{\scriptstyle F(f)} \\ (F \times G)(X) & \xrightarrow{\pi_{1,X}} & F(X) \end{array}$$

We can show this by simple calculation. Let $(a, b) \in (F \times G)(Y)$. Then, $\big[F(f) \circ \pi_{1,Y}\big](a, b) = F(f)(a)$. For the other path, we have

that $\big[(F \times G)(f)\big]$ applied to $(a, b)$ is the pair $(F(f)(a), F(f)(b))$, so $\pi_{1,X} \circ \big[(F \times G)(f)\big]$ applied to $(a, b)$ is the first component $F(f)(a)$. An identical argument holds for $\pi_{2,X}$.

Next, we need to show that $(F \times G, \pi_1, \pi_2)$ satisfies the universal property for products. Suppose there is a $\mathcal{C}$-indexed set $\Gamma$ with $\mathcal{C}$-indexed functions $f : \Gamma \Rightarrow F$ and $g : \Gamma \Rightarrow G$. Then, we must show there exists a unique $\mathcal{C}$-indexed function $\langle f, g \rangle : \Gamma \Rightarrow F \times G$ such that the relevant diagram commutes. Let's define $\langle f, g \rangle : \Gamma \Rightarrow F \times G$ as:

$$\begin{aligned} \langle f, g \rangle_X &: \Gamma(X) \to (F \times G)(X) \\ &= a \mapsto \Big( f_X(a), g_X(a) \Big) \end{aligned}$$

The naturality of this definition follows from the naturality of $f$ and $g$: for any morphism $Y \xrightarrow{s} X$ of $\mathcal{C}$, we have that

$$(F \times G)(s) \circ \langle f, g \rangle_X = \langle f, g \rangle_Y \circ \Gamma(s)$$

because, for any input $a \in \Gamma(X)$, it holds that

$$\begin{aligned} \text{LHS}(a) &= \big[(F \times G)(s)\big](f_X(a), g_X(a)) && \text{Def. of } \langle f, g \rangle_X \\ &= (F(s)(f_X(a)), G(s)(g_X(a))) && \text{Def. of } (F \times G)(s) \\ &= (f_Y(\Gamma(s)(a)), g_X(\Gamma(s)(a))) && \text{Naturality of } f \text{ and } g \\ &= \langle f, g \rangle_Y(\Gamma(s)(a)) && \text{Def. of } \langle f, g \rangle_Y \\ &= \text{RHS}(a). \end{aligned}$$

Next we have the following for any indexing object $X$, establishing that the relevant product diagram commutes.

$$\pi_{1,X} \circ \langle f, g \rangle_X = f_X, \qquad \pi_{2,X} \circ \langle f, g \rangle_X = g_X.$$

Finally, all that remains is to show uniqueness of $\langle f, g \rangle$. This follows immediately from function extensionality: if $\langle f, g \rangle$ is to satisfy $\pi_{1,X} \circ \langle f, g \rangle_X = f_X$ and $\pi_{2,X} \circ \langle f, g \rangle_X = g_X$ for all $X$, then it must be defined as above. □

# 8
# *Yoneda III: Representability*

The crucial idea of the Yoneda lemma is that a copy of $\mathcal{C}$ lives inside of its corresponding presheaf category. This lets ideas, definitions, and intuitions from the $\mathcal{C}$-indexed category – which, as we've already described, has fantastically rich structure – including products, exponentials, etc. – be reflected back into the indexing category $\mathcal{C}$.

## *8.1 Representables*

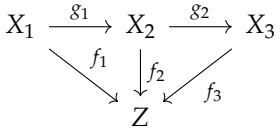
Our first step towards understanding this picture is to come up with a way of injecting a category $\mathcal{C}$ into its presheaf category. Here is an inscrutable definition that explains how to do this:

**Definition 21: The representable $\mathcal{C}$-indexed set at $X$**

The *representable $\mathcal{C}$-indexed set at $X$* is written $\yo X$ (pronounced "yo $X$") and defined by

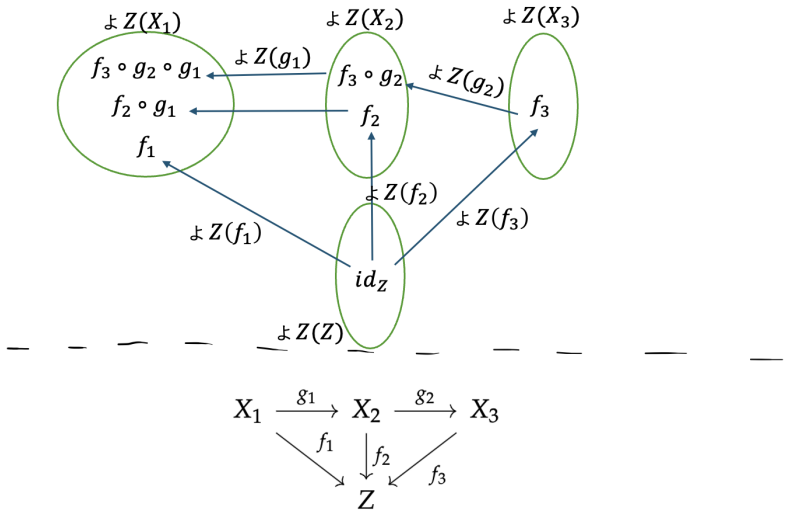
$$\begin{aligned}(\yo X)(A) &= \mathcal{C}(A, X)\\ (\yo X)(s : A' \to A) : (\yo A) &\to (\yo A')\\ &= (f : A \to X) \mapsto (f \circ s : A' \to X)\end{aligned}$$

This definition, on first glance, is very hard to unpack. So we will spend some time with it. The intuition you should have for $\yo X$ is that it is "the external view of the object $X$ in the category $\mathcal{C}$". Let's start by drawing a balloon diagram to picture one of these representables. Consider the following example category:




(8.1)

Let's draw a balloon diagram that visualizes $\yo Z$:




To build intuition, you should think of $\lambda Z$ as describing "the perspective every other object in $\mathcal{C}$ has on $Z$." Unpacking this more:

- Let's look at the action of $\lambda Z$ on $Z$ first. Unfolding the definition, we have:

$$(\lambda Z)(Z) = \mathcal{C}(Z, Z) = \{\mathrm{id}_Z\}$$

  So, somewhat vacuously, we might say: "the perspective $Z$ has on itself is the identity morphism". And, for $X_1$, we have:

$$(\lambda Z)(X_1) = \mathcal{C}(X_1, Z) = \{f_1, f_2 \circ g_1, f_3 \circ g_2 \circ g_1\}$$

  This is slightly more interesting: we would say, "$X_1$ has 3 perspectives on $Z$: (1) via the $f_1$ morphism, (2) by following $f_2 \circ g_1$; (3) by following $f_3 \circ g_2 \circ g_1$".

- Now let's look at the action on morphisms. These will correspond to "changes in perspective". Let's examine the action of $\lambda Z$ on the morphism $f_1$:

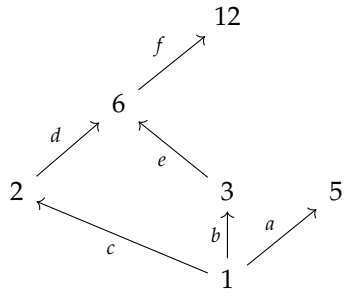
$$\begin{aligned}(\lambda Z)(f_1 : X_1 \to Z) : \lambda Z \to \lambda X_1 \\ = \mathrm{id}_Z \mapsto \mathrm{id}_z \circ f_1 = f_1\end{aligned}$$

  One can understand this as: "to move from "$X_1$'s perspective on $Z$ to $Z$'s perspective on $Z$, follow the $f_1$ morphism". Somewhat more interestingly, we can also shift perspectives between $X_1$ and $X_2$ by precomposing with the $g_1$ morphism:

$$\begin{aligned}(\lambda Z)(g_1 : X_1 \to X_2) : (\lambda Z)(X_2) \to (\lambda Z)(X_1) \\ = \begin{cases} f_2 \mapsto f_2 \circ g_1 \\ f_3 \circ g_2 \mapsto f_3 \circ g_2 \circ g_1 \end{cases}\end{aligned}$$

### 8.1.1 *Representables in the divisibility category*

Let's consider what representables look like in a familiar category to get a better sense of their properties. Consider the following subset of the natural numbers ordered by divisibility:




Let's consider the representables of each element in this category. First let's inspect $\text{ɟ}12$. We can compute its action on objects:

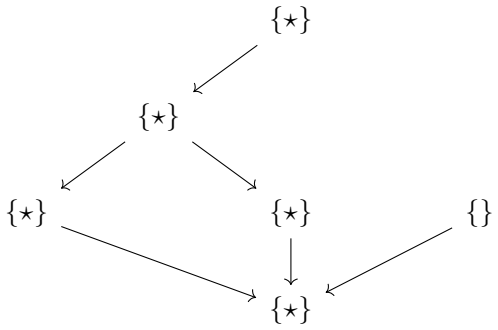
$$(\text{ɟ}12)(12) = \{\text{id}_{12}\} \quad (\text{ɟ}12)(6) = \{f\} \quad (\text{ɟ}12)(3) = \{f \circ e\}$$
$$(\text{ɟ}12)(2) = \{f \circ d\} \quad (\text{ɟ}12)(1) = \{f \circ d \circ c\} \quad (\text{ɟ}12)(5) = \{\}$$

Note that in this category there is at most 1 morphism between any 2 objects (i.e., we have that the morphisms $f \circ d \circ c = f \circ e \circ b$, so we only include 1 of them in $(\text{ɟ}12)(1)$). So, each representable $\text{ɟ}X$ is either a singleton (meaning $X \preceq 12$), or it is an empty set (meaning $X \not\preceq 12$). The action of $\text{ɟ}12$ on morphisms is not complicated; let's see what it is by unfolding definitions:

$$\begin{aligned}(\text{ɟ}12)(c) : (\text{ɟ}12)(2) &\to (\text{ɟ}12)(1) \\ &= (f \circ d) \mapsto (f \circ d) \circ c\end{aligned}$$

In this case we see that to shift perspective from 1 to 2, we should precompose by the $c$ morphism. From this, we can observe that there is a single morphism $\text{ɟ}Y \to \text{ɟ}X$ if and only if there is a morphism $X \to Y$ in $\mathcal{C}$.

Now, let's closely examine the structure of a representable $\text{ɟ}12$ and try to gain a more global picture of what data it contains. We can summarize $\text{ɟ}12$ by the following balloon, where we've replaced each object by its balloon:




We see that the collection of objects with a $\star$ in them are exactly those objects that are at or below 12 in the order. This collection of objects has a name in order theory: it is called the **principal ideal** of 12. Formally:

**Definition 22: Principal ideal**

Let $(X, \leq)$ be a preorder. Then, the **principal ideal** of an element $a \in X$, written $\downarrow a$, is the smallest collection of all elements less than or equal to $a$ in $X$, i.e. $\downarrow a = \{x \in X \mid x \leq a\}$.

So, we can summarize: the representable $\yoneda 12$ is the principal ideal $\downarrow 12$, i.e., it contains exactly as much information as is contained in the principal ideal of 12. In general, for any category formed by a preorder, it is the case that the representable of an element is a principal ideal in the preorder. This order-theoretic viewpoint sheds some light on the intuition that the representable of an element is "all the ways of viewing the element in the category". One can imagine "peering up" at an element $X$ from all the other objects: $\yoneda X$ is then all the vantage points below $X$ from which it can be seen.

## 8.2 *Representables in STLC*

So far we've been studying representables in thin categories, where there is a very pleasant order-theoretic understanding what they mean in terms of principal ideals. Let's see an example of representables in a category with more than 1 morphism between objects. Recall the syntax of STLC:

$$\begin{aligned} A, B &::= \mathsf{Unit} \mid A \times B \mid A \to B \\ M, N &::= \langle\rangle \mid \langle M, N\rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \mid \lambda\, x.\, M \mid M\, N \mid x \end{aligned} \tag{STLC}$$

Recall the category of simply-typed lambda calculus terms quotiented by equations:

**Definition 23: The category STLC**

The category of simply-typed lambda calculus terms (STLC) is the category where:

- Objects are types $A$;
- Morphisms $A \to B$ are equivalence classes of terms $[x : A \vdash e : B]$ where equality is given by the equational theory of the $\beta$ and $\eta$ laws;
- Composition is given by substitution:

$$[b : B \vdash N : C] \circ [a : A \vdash M : B] = [a : A \vdash N[M/b] : C]$$

- The identity morphism is the class of terms $[a : A \vdash a : A]$.

Let's see the representables in this category. For example, what do the representables $\text{ɣ}(\mathsf{Unit} \times \mathsf{Unit})$ look like? Considering a concrete example:

$$\begin{aligned}\text{ɣ}(\mathsf{Unit} \times \mathsf{Unit})(\mathsf{Unit}) &= \{\text{the ways of viewing terms of type } \mathsf{Unit} \times \mathsf{Unit} \text{ from terms of type } \mathsf{Unit}\} \\ &= \{[a : \mathsf{Unit} \vdash M : \mathsf{Unit} \times \mathsf{Unit}] \mid M \text{ an stlc term}\} \\ &= [a : \mathsf{Unit} \vdash \langle a, a \rangle : \mathsf{Unit} \times \mathsf{Unit}]\end{aligned}$$

In the case above, there is only 1 element of the set $\text{ɣ}(\mathsf{Unit} \times \mathsf{Unit})(\mathsf{Unit})$: this is a quirk of our choice of types. Intuitively, the representables $(\text{ɣ}B)(A)$ for two types $A$ and $B$ is the collection of all equivalence classes of terms $M$ of type $A \to B$. Then, naturally, the action on morphisms $(\text{ɣ}X)(f : A \to B) : \text{ɣ}B \to \text{ɣ}A$ ought to be the ways of shifting viewpoints from $A$ to $B$, i.e.:

$$\begin{aligned}(\text{ɣ}X)([a : A \vdash M : B]) &: (\text{ɣ}X)(B) \to (\text{ɣ}X)(A) \\ &= [b : B \vdash N : X] \mapsto [b : B \vdash N : X] \circ [a : A \vdash O : B] \\ &= [b : B \vdash N : X] \mapsto [a : A \vdash N[O/b] : X]\end{aligned}$$

Carefully check all the types in the above equations.

## 8.3 *Representability*

There can be many $\mathcal{C}$-indexed sets for a category, but the representables are very special due to their relationship with the indexing category. As usual in category theory, we are concerned not with particular objects but rather with equivalence classes of objects up to isomorphism, a notion we make formal here:

**Definition 24: Isomorphism of $\mathcal{C}$-indexed sets**

Two $\mathcal{C}$-indexed sets $P, Q$ are *isomorphic* if there are $\mathcal{C}$-indexed functions $\alpha : P \Rightarrow Q$ and $\beta : Q \Rightarrow P$ such that $\alpha \circ \beta = \mathsf{id}_Q$ and $\beta \circ \alpha = \mathsf{id}_P$.

With this definition in hand we can make precise the set of $\mathcal{C}$-indexed sets that are tied to the indexing category:

**Definition 25: Representability**

A $\mathcal{C}$-indexed set $P$ is *representable* if there is an isomorphism $\alpha : P \cong \yo X : \beta$ for some object $X$ of $\mathcal{C}$.

**Proposition 8.3.1.** The $\mathsf{FinSet}$-indexed set $\yo X \times \yo Y$ is representable.

*Proof.* for any finite set $A$, there is an isomorphism of sets

$$\alpha_A : \mathsf{FinSet}(A, X \times Y) \cong \mathsf{FinSet}(A, X) \times \mathsf{FinSet}(A, Y).$$

Concretely, $\alpha_A$ is the function that sends a $f \in \mathsf{FinSet}(A, X \times Y)$ to $(\pi_1 \circ f, \pi_2, \circ f) : \mathsf{FinSet}(A, X) \times \mathsf{FinSet}(A, Y)$. Its inverse is defined by $\beta_A(f, g) = \langle f, g \rangle$. Both $\alpha$ and $\beta$ can be checked satisfy the naturality condition. Thus $X \times Y$ represents $\yo X \times \yo Y$. □

Representability is the bridge between the internal view and the external view of a category: it lets us reflect properties presheaf category on $\mathcal{C}$ back into $\mathcal{C}$ itself. Concretely, in the next chapter, we will use the Yoneda lemma in full to show:

**Theorem 8.3.2.** Let $P, A, B$ be objects in a category $\mathcal{C}$. Then, $P$ is a product of $A$ and $B$ if and only if it represents the product in the presheaf category $\yo A \times \yo B$.

We will not prove this theorem here – it is a direct consequence of the Yoneda lemma, which we are now ready to appreciate – but one can understand this theorem as: an object $P$ is a product of $A$ and $B$ internally if and only if it behaves like a product externally.

# 9
# *Yoneda IV: universal constructions, elements, and properties*

Now we are ready to appreciate the Yoneda lemma:

**Theorem 9.0.1** (Baby Yoneda)**.** Let $\mathcal{C}$ be a category, $X$ be an object in $\mathcal{C}$, and $P$ be a $\mathcal{C}$-indexed set. Then, there is a bijective correspondence:[1]

$$\{\mathcal{C}\text{-indexed functions } \text{ɬ}X \Rightarrow P\} \cong P(X)$$

[1] This is called the "baby Yoneda lemma" because the full Yoneda lemma further stipulates that this isomorphism satisfies additional naturality conditions. The baby Yoneda lemma is still quite strong, however, and is much easier to prove.

On first glance this theorem is profoundly mysterious because it stipulates that there is a very strong relationship between two very different-seeming things. Let's unpack this slowly to really understand what is going on by instantiating the Yoneda lemma to a specific category and $\mathcal{C}$-indexed set. Let's consider a finite category with 3 objects:

$$X \xrightarrow{f} Y \xrightarrow{g} Z \tag{9.1}$$

Now, let's define a simple $\mathcal{C}$-indexed set $P$:

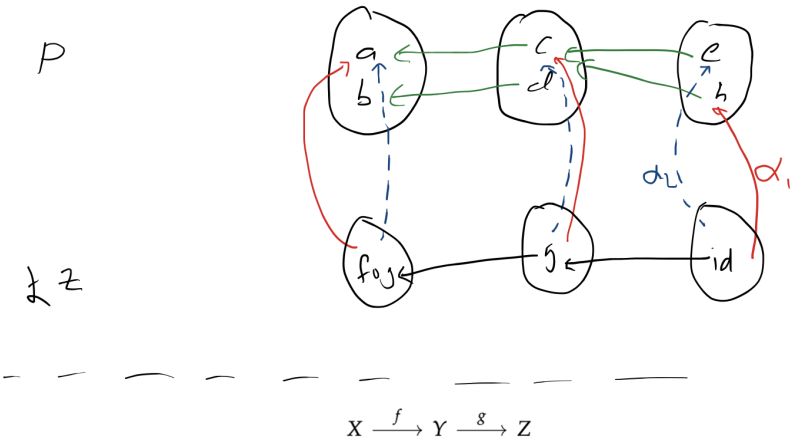
$$P(X) = \{a, b\} \quad P(Y) = \{c, d\} \quad P(Z) = \{e, h\}$$
$$P(f) = \begin{cases} c \mapsto a \\ d \mapsto b \end{cases} \qquad P(g) = \begin{cases} e \mapsto c \\ h \mapsto c \end{cases}$$

Continuing to unpack the data, we can define $\text{ɬ}Z$, which is also a $\mathcal{C}$-indexed set:

$$(\text{ɬ}Z)(Z) = \{\mathrm{id}_Z\} \quad (\text{ɬ}Z)(Y) = \{g\} \quad (\text{ɬ}Z)(X) = \{g \circ f\}$$
$$(\text{ɬ}Z)(f) = g \mapsto g \circ f \quad (\text{ɬ}Z)(g) = \mathrm{id}_Z \mapsto g$$

Next, let's enumerate the natural transformations $\text{ɬ}X \Rightarrow P$ If the Yoneda lemma is true, we don't have too much work to do: we know

there should be *only two of them*, since $P(X)$ has only two elements in it. Let's enumerate them on a balloon diagram:




Here we've drawn the two possible natural transformations $\alpha_1$ and $\alpha_2$ in solid red and dashed blue respectively. Now, let's understand why the Yoneda lemma holds by tracing the possible paths through this balloon diagram with your fingers. Recall the two finger rule: a $\mathcal{C}$-indexed function $\not\downarrow Z \Rightarrow P$ satisfies the naturality condition if, starting at any element in any balloon $\not\downarrow X$, your two fingers move in synchrony along all possible paths. So, let's start with $\not\downarrow Z = \{\mathrm{id}_Z\}$. Let's play the two finger game together to design the two possible natural transformations:

- Your left finger moves to $g$. Where can your right finger go? It can go either to $e$ or $h$. So, choose $h$ and so we are designing $\alpha_1$. *The rest of the moves are forced*: your right finger must now move to $c$, then to $a$. Your left finger is also forced: it must move to $g$ then to $f \circ g$. Now, all other decisions are forced, and we can fill in $\alpha_1$ so that all moves are in harmony.

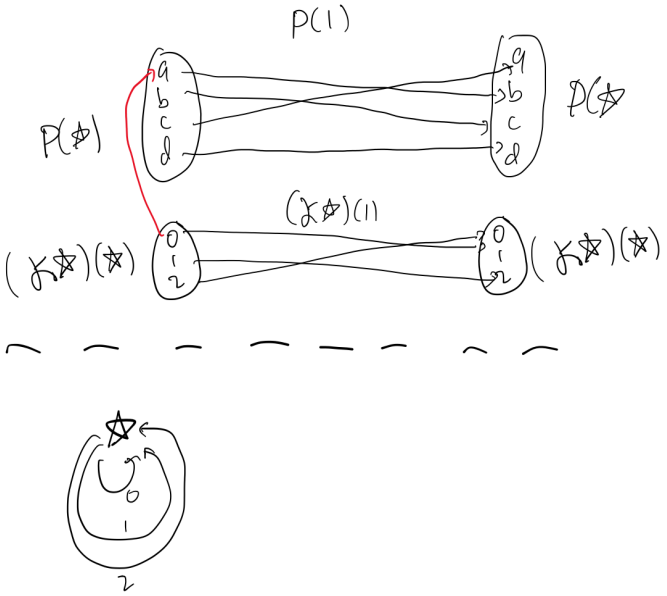
- Next, we can choose moving our right finger to $e$, and so we are designing $\alpha_2$. *Again, all moves are forced*: the right finger follows the functions in $P$, and the left finger follows the functions in $\not\downarrow Z$. Then, we are forced to choose each $\alpha$ to exactly line up exactly with how these fingers move: there is only one choice for $\alpha$ at each step.

Notice what is happening: because, in this example, there is a single element in $(\not\downarrow Z)(Z)$, then there must be *exactly 1* natural transformation for each possible element in $P(Z)$. Naturality is a very strong restraint to place on $\alpha$!

So, it is now hopefully quite clear that, if $\mathrm{id}_Z$ is the only element of $\not\downarrow Z$, the baby Yoneda lemma is true, since the entire action of a

natural transformation is determined by its action on the identity morphism of $Z$. But, what if there is more than one morphism in $(\text{ɟ}Z)(Z)$?

The crux is that the choice of $(\text{ɟ}Z)(\text{id}_Z)$ fully determines the action of $\text{ɟ}Z$ on all the other morphisms $Z \to Z$. Let's see this visually. Suppose we have as our indexing category the monoid corresponding to the natural numbers mod 3 under addition, the category $\mathbb{Z}_3$. This category has a single object $\star$ and 3 morphisms: the identity morphism 0, and morphisms 1 and 2, together with equations stipulating that $1 \circ 1 = 2, 1 \circ 2 = \text{id}, 2 \circ 2 = 1, 2 \circ 1 = \text{id}$. Let's visualize some of this data:




Here we've drawn an arbitrary $\mathcal{C}$-indexed set $P$ with its action on the 1 morphism, and we've chosen arbitrarily that the $\mathcal{C}$-indexed function $\alpha_1$ maps 0 to $a$. Note that we've drawn the balloon twice for the same object; this is for visualization purposes only.

Now, the key: *does this choice of $\alpha$ fully determine how $\alpha$ behaves on all other elements of* $(\text{ɟ}\star)(\star)$? Indeed it does! Observe: by naturality, we have that $P(1) \circ \alpha = \alpha \circ (\text{ɟ}\star)(1)$. This gives us a system of constraints that uniquely defines $\alpha$. Observe:

$$\begin{aligned}
&((P(1) \circ \alpha))(0) = b = (\alpha \circ (\text{ɟ}\star)(1))(0) \\
&\Rightarrow b = \alpha(((\text{ɟ}\star)(1))(0)) \\
&\Rightarrow b = \alpha(1).
\end{aligned}$$

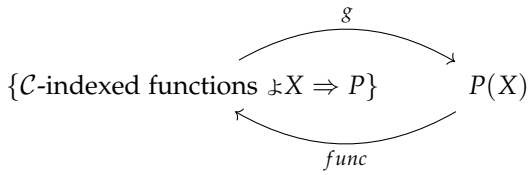
A similar chain of logic lets us derive $\alpha(2) = c$. Notice what is happening here: $P$ is a functor, so it must "behave like the monoid": i.e., it has the same equations and morphism compositions. Here, the choice of $\alpha(0)$ is *picking an element of $P$ that behaves like the identity*.

Now, how do we see that there are exactly 4 natural transformations $\alpha : \yoneda\star \Rightarrow P$? Let's look at $P$. Since $P$ is a functor, it has copies of the $\mathbb{Z}_3$ in its codomain. We can see two copies: the $\{a, b, c\}$ subset is an exact copy – the elements $a, b$, and $c$ correspond to distinct numbers, and $\alpha$ will pick an arbitrary 0 point – and the $\{d\}$ subset is a denerate copy with only a single number 0. So, there are 3 natural transformations that send 0 to $\{a, b, c\}$, each picking a different 0 point, and 1 natural transformation into $\{d\}$ that sends all elements of $(\yoneda\star)(\star)$ to $d$.

## 9.1 *Proving the baby Yoneda lemma*

Thanks to Bex Golovanov for the scribe notes for the next two subsections. up a bit.

Now we are ready to prove Theorem 9.0.1. Let $\mathcal{C}$ be a category, $P$ be a $\mathcal{C}$-indexed set, and $X$ be an object in $\mathcal{C}$. Then, we need to construct a bijection:




What does this bijective correspondence statement actually mean? We have some $func$ that, given a $p \in PX$, can construct a natural transformation $\alpha : \yoneda X \Rightarrow P$, and some $g$ that can extract $p$ from a given natural transformation.

1. Constructing natural transformation from $p \in PX$

   - Given a $p \in PX$, we defined $func(p)$ via a family of functions $func(p)_A$, where $func(p)$ is a natural transformation. We can think of the $func(p)_A$ as the components of $func(p)$, where for each object $A$ in $\mathcal{C}$, the component $func(p)_A$ maps from $(\yoneda X)(A)$ to $P(A)$.

     $$\begin{aligned} func(p)_A : (\yoneda X)(A) &\to PA \\ f &\mapsto (Pf)(p) \end{aligned}$$

     where $f : A \to \yoneda X$. The element $p$ by definition has type $PX$, and $P$ is contravariant so $Pf$ maps $PX$ to $PA$, giving us the desired codomain.

   - Naturality of $func(p)$. Let $g : B \to A$. I don't really want to

justify this but here is the square

$$\begin{array}{ccc} (\not\perp X)(A) & \xrightarrow{func(p)_A} & P(A) \\ {\scriptstyle (\not\perp X)(g)}\downarrow & & \downarrow{\scriptstyle P(g)} \\ (\not\perp X)(B) & \xrightarrow[func(p)_B]{} & P(B) \end{array}$$

2. Getting $p$ from natural transformation

   - How did we extract $p \in PX$ from some given natural transformation $\alpha$? We know $\alpha_X$ maps $(\not\perp X)(X) \to PX$, so applying $\alpha_X$ to $id_X$ gives us an element of $PX$, we can set $p := \alpha_X(id_X)$.
   - Now we need to show $\alpha_A = func(p)_A$. Since $\alpha_A$ acts on the elements of $(\not\perp X)(A)$, it suffices to show $\alpha_A(f) = func(p)_A = (Pf)(p)$ for every $f \in \mathcal{C}(A, X)$.
   - We have $(Pf)(p) = (Pf)(\alpha_X(id_X))$. We will use the naturality of $\alpha$. Specifically, $Pf \circ \alpha_X = \alpha_A \circ (\not\perp X)(f)$.

$$\begin{array}{ccc} (\not\perp X)(X) & \xrightarrow{\alpha_X} & P(A) \\ {\scriptstyle (\not\perp X)(f)}\downarrow & & \downarrow{\scriptstyle P(f)} \\ (\not\perp X)(A) & \xrightarrow[\alpha_A]{} & P(B) \end{array}$$

$$\begin{aligned} (Pf)(\alpha_X(id_X)) = (Pf \circ \alpha_A)(id_X) &= (\alpha_A \circ (\not\perp X)(f))(id_X) \\ &= \alpha_A((\not\perp X)(f)(id_X)) = \alpha_A(id_x \circ f). \\ &= \alpha_A(f) \end{aligned}$$

   So $(Pf)(p) = \alpha_A(f)$, and we are done!

**Remark 9.1.1.** Defining some natural transformation $\alpha := f(p)$ by defining how $f(p)$ acts on each object $A$ is like a "pointwise" definition of $\alpha$.

## 9.2 *The category of elements*

**Definition 26: Category of elements**

Let $P$ be a $\mathcal{C}$-indexed set. The *category of elements* of $P$, written $\int_{\mathcal{C}} P$, is a category where:

1. objects : pairs $(X, p \in PX)$
2. morphisms : from $(X, p \in PX)$ to $(Y, q \in PY)$ are morphisms $f : X \to Y$ of $\mathcal{C}$ such that $(Pf)(q) = p$. Alternatively, $q \cdot_P f =$

$p$. A morphism of this category of elements is a morphism from the original category with the *additional* structure of preserving the points of $PX$.

Definitions of composition and identity follow immediately from the definitions in $\mathcal{C}$, with the additional restriction of point preservation. It is relatively simple to show the desired categorical properties hold.

**Proposition 9.2.1.** Suppose $\mathcal{C}$-indexed set $P$ is representable by $X$. There exists a $u \in PX$ called the *P-universal element* which satisfies the following universal property : for any object $\Gamma$ of $\mathcal{C}$ and any $g \in P\Gamma$, there exists a unique $\hat{g} : \Gamma \to X$ such that $g = u \cdot_P \hat{g}$. Alternatively, $g = (P\hat{g})(u)$.

**Proposition 9.2.2.** A $\mathcal{C}$-indexed set $P$ is representable if $P \cong \yo X$ for some object $X$ of $\mathcal{C}$. Suppose $P$ is representable. Then there exists a *universal element* $u \in PX$ and $u$ is a terminal object in the *category of elements* of $P$.

**Example 9.2.3** (Category of elements of the terminal presheaf)**.** For any category $\mathcal{C}$, the presheaf category on $\mathcal{C}$ has a terminal object $\mathbf{1}$ defined:

$$\mathbf{1}(X) = \{\star\}$$
$$\mathbf{1}(X \xrightarrow{h} Y) = \star \mapsto \star$$

The category of elements $\int \mathbf{1}$ has objects $(X, p)$ where $p = \star$ and $X$ are objects of $\mathcal{C}$, and has morphisms $(X, p) \xrightarrow{h} (Y, q)$ where $F(h)(p) = q$; this point preservation is clearly trivially satisfied since $p = q = \star$. So, we can conclude that $\int \mathbf{1} = \mathcal{C}$, i.e., the category of elements of the terminal object in $\mathcal{C}$ is the original category $\mathcal{C}$.

We have from Theorem 9.2.2 the definition of the universal property that we want these mysterious universal elements to satisfy. We can think about this definition as stating that we can "factor" any $g \in P\Gamma$ (for any $\Gamma$) into $u$ and a unique $\hat{g}$.

What does it mean for $u \in P(X)$ to be terminal in the category of elements of $P$? That any object has a unique morphism mapping to $(X, u)$. Explicitly, for any $(A, p \in PA)$, there is a unique map $f : A \to X$ such that $(Pf)(u) = p$ (or $p = u \cdot_P f$). This looks pretty similar to the universal property given in Theorem 9.2.2 – we can "factor" any $p \in PA$ into $f$ and $u$.

How do we get Theorem 9.2.2 from the Yoneda Lemma? These are the key ideas:

- $X$ representing $P$ means we have an isomorphism between $\yo X$ and $P$. This isomorphism is a $\mathcal{C}-$indexed function, which we

have learned is usually called a natural transformation. So technically we have a bijective natural transformation (a natural isomorphism?)

- The Yoneda Lemma says that for every natural transformation $\text{ɟ}X \Rightarrow P$, we should have a corresponding $u \in PX$ from which we can construct said transformation. This means the natural isomorphism between $\text{ɟ}X$ and $P$ has a *corresponding* $u \in PX$ (suggestively labeled).
- Let $\alpha$ denote the natural isomorphism $\text{ɟ}X \Rightarrow P$. Because $\alpha$ is bijective, and $\alpha$ can be defined component-wise, each $\alpha_A$ for each object $A$ must also be bijective. So consider this bijection $\alpha_A : (\text{ɟ}X)(A) \Rightarrow PA$. From surjectivity, for any $p \in PA$, there exists some $\hat{p}$ in $(\text{ɟ}X)(A)$ such that $\alpha_A(\hat{p}) = p$. From injectivity, this $\hat{p}$ is unique.
- How do we connect this $\hat{p}$ to the corresponding $u \in PX$ we identified? (Look at how we constructed the natural transformations $\alpha_A$ in our proof of baby yoneda).
- From Yoneda Lemma we know we have a function $func(p)$ defined as a family of functions $func(p)_A$, where $func(p)_A$ maps $p$ to the component $\alpha_A$. Specifically, $\alpha_A$ is defined to take elements of $(\text{ɟ}X)(A)$, e.g. some $f : A \to X$, and map them to $(Pf)(p) = p \cdot_P f$, elements of $PA$. Then we know $\alpha_A(\hat{p}) = u \cdot_P \hat{p}$ and $\alpha_A(\hat{p}) = p$, meaning $p = u \cdot_P \hat{p}$ !

## 9.3 *Calculating a universal property*

Let's bring all these ideas together to find an object of a category that *internalizes* a $\mathcal{C}$-indexed set. This is an essential application of the Yoneda lemma: sometimes it is easier to describe objects by how they behave externally, and the Yoneda lemma lets us bring those external descriptions back to objects of the category. Let's use this idea to calculate the universal property of an *equalizer*.

**Definition 27: Equalizer**

An equalizer of $A \overset{f}{\underset{g}{\rightrightarrows}} B$ is a representation for the $\mathcal{C}$-indexed set with action on objects:

$$\mathsf{Eq}(f,g)(X) = \{X \xrightarrow{p} A \mid f \circ p = g \circ p\}$$

and action on morphisms works by precomposition:

$$\begin{aligned}\mathsf{Eq}(X \xrightarrow{h} Y) : \mathsf{Eq}(f,g)(Y) &\to \mathsf{Eq}(f,g)(X) \\ = (Y \xrightarrow{p} A \mid f \circ p = g \circ p) &\mapsto \\ (X \xrightarrow{h} Y \xrightarrow{p} A \mid f \circ p \circ h = g \circ p \circ h)&\end{aligned}$$

From this external definition, we can calculate the universal property for an internal definition of equalizer by computing a terminal object in the category of elements $\int \mathsf{Eq}(f,g)$. Let's calculate what it is. The objects are pairs $(X,p)$ where $X \xrightarrow{p} A$ and $f \circ p = g \circ p$ (i.e., $p \in \mathsf{Eq}(f,g)(X)$). Morphisms from $(X,p)$ to $(Y,q)$ is a $\mathcal{C}$-morphism $X \xrightarrow{h} Y$ such that $\mathsf{Eq}(f,g)(h)(q) = p$. Unfolding $\mathsf{Eq}(f,g)(h) = q \circ h$, we have that $q \circ h = p$.

Now we have a clearer picture of what objects in this category of elements looks like. They are exactly this configuration of objects:

$$E \xrightarrow{h} A \underset{g}{\overset{f}{\rightrightarrows}} B$$

Say we have another object in this category of elements:

$$Q \xrightarrow{h} A \underset{g}{\overset{f}{\rightrightarrows}} B$$

Then, morphisms in the category of elements are morphisms $Q \xrightarrow{f} P$:

$$\begin{array}{ccccc} R & \xrightarrow{h} & A & \underset{g}{\overset{f}{\rightrightarrows}} & B \\ {\scriptstyle f}\uparrow & \nearrow & & & \\ Q & & & & \end{array}$$

where the above diagram commutes. We call the terminal object in the category of elements $\int \mathsf{Eq}(f,g)$ the equalizer. From this we can finally calcuate the internal definition of an equalizer: it is the terminal object in this category of elements.

**Proposition 9.3.1.** Let $(E, E \xrightarrow{p} A)$ be an equalizer of $A \underset{g}{\overset{f}{\rightrightarrows}} B$. Then, $\curlywedge E \cong \mathsf{Eq}(f,g)$.

# 10
# *Functors and natural transformations*

We are ready to broaden our categorical perspective and language to begin comparing two categories with one another. We will see how some of the notions we've been encountering so far in our journey through the Yoneda lemma – in particular, the functoriality and naturality properties of $\mathcal{C}$-indexed sets and functions – are special cases of more general phenomena in category theory called functors and natural transformations.

## *10.1 Functors*

A functor is a structure-preserving map between categories:

**Definition 28: Functor**

Given two categories $\mathcal{C}$ and $\mathcal{D}$, a *functor $F$ from $\mathcal{C}$ to $\mathcal{D}$*, written $F : \mathcal{C} \to \mathcal{D}$, consists of

- An "action on objects", which is a function sending each object $X$ of $\mathcal{C}$ to an object $F(X)$ of $\mathcal{D}$
- An "action on morphisms", which is a function sending each morphism $X \xrightarrow{f} Y$ of $\mathcal{C}$ to a morphism $F(X) \xrightarrow{F(f)} F(Y)$ of $\mathcal{D}$

such that the following *functoriality conditions* are satisfied:

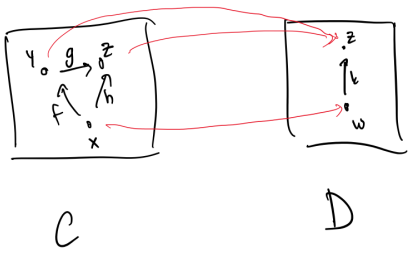
- Identity preservation: for all objects $X$ of $\mathcal{C}$ it holds that $F(\mathsf{id}_X) = \mathsf{id}_{F(X)}$
- Composition preservation: for all composable pairs of morphisms $X \xrightarrow{f} Y \xrightarrow{g} Z$ of $\mathcal{C}$ it holds that $F(g \circ f) = F(g) \circ F(f)$.

Intuitively, if there is a functor $F : \mathcal{C} \to \mathcal{D}$, then there is an "abstract copy of $\mathcal{C}$ inside $\mathcal{D}$".

### 10.1.1 *Examples of functors*

The simplest example of a functor is one between two finite categories. Consider the following example of a functor between two finite categories $\mathcal{C}$ and $\mathcal{D}$:

A good exercise: how many functors are there from $\mathcal{C}$ to $\mathcal{D}$?




The functor $F : \mathcal{C} \to \mathcal{D}$ is drawn in red; we've only shown its actions on objects here. Written out, we have:

$$F(Y) = Z, \quad F(X) = W, \quad F(Z) = Z$$

This action on objects defines the action on morphisms. This isn't always the case; there may be more than one morphism between two objects in $\mathcal{C}$ and $\mathcal{D}$, in which case, there may be a choice on which morphisms to map to. But here, it is simple, and we can conclude:

$$F(\mathrm{id}_Y) = \mathrm{id}_Z \quad F(\mathrm{id}_X) = \mathrm{id}_Z \quad F(\mathrm{id}_Z) = \mathrm{id}_Z$$
$$F(g) = \mathrm{id}_Z \quad F(f) = k \quad F(h) = k \quad F(g \circ f) = k$$

We need to check that the functoriality properties are satisfied. Clearly, identity preservation is satisfied. The only interesting case to check is the compositional property that $k = F(g \circ f) = F(g) \circ F(f) = \mathrm{id}_Z \circ k = k$.

As usual, it's useful to gain an intuition for what functors look like between categories that represent preorders. They will correspond to monotone functions:

**Definition 29: Monotone function**

Let $(X, \leq_X)$ and $(Y, \leq_Y)$ be preorders. Then, a function $f : X \to Y$ is called *monotone* if, for any $x_1, x_2 \in X$ such that $x_1 \leq_X x_2$, it is the case that $f(x_1) \leq_Y f(x_2)$.

Now, let's see how functors between preorders correspond to monotone functions. Let $\mathcal{C}$ and $\mathcal{D}$ be categories corresponding to preorders and let $F : \mathcal{C} \Rightarrow \mathcal{D}$ be a functor between these two categories. Then, functoriality of $F$ requires that, for any morphism $X \xrightarrow{f} Y$ in $\mathcal{C}$, there is a morphism $F(X) \xrightarrow{f} F(Y)$. This is exactly the monotonicity requirement: the morphism $f$, considered order-theoretically, denotes that $X \leq Y$; similarly, the morphism $F(f)$ denotes that $F(X) \leq F(Y)$.

## 10.2 *Natural transformations*

Natural transformations are structure-preserving maps between functors:

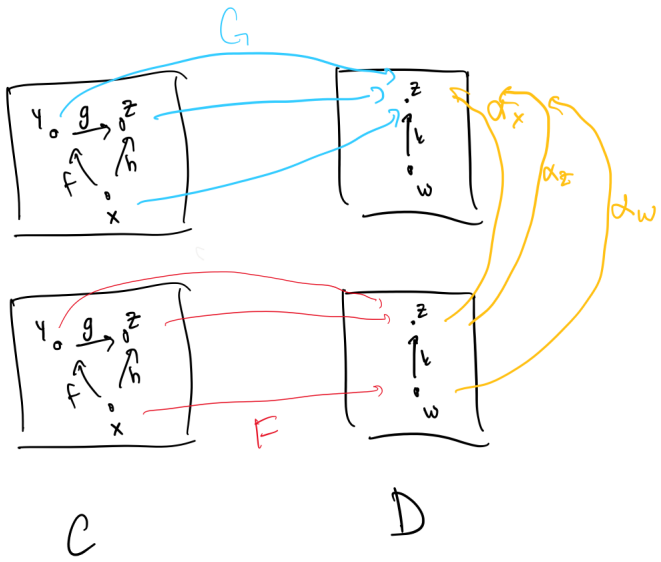
**Definition 30: Natural transformation**

Given two categories $\mathcal{C}$ and $\mathcal{D}$, and two functors $F, G : \mathcal{C} \to \mathcal{D}$, a *natural transformation* $\alpha$ *from* $F$ *to* $G$, written $\alpha : F \Rightarrow G$, consists of a function sending each object $X$ of $\mathcal{C}$ to a *morphism* $F(X) \xrightarrow{\alpha_X} G(X)$ of $\mathcal{D}$ such that the following *naturality condition* is satisfied: for each morphism $X \xrightarrow{f} Y$ of $\mathcal{C}$, the following square commutes:

$$\begin{array}{ccc} FX & \xrightarrow{\alpha_X} & GX \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle Gf} \\ FY & \xrightarrow[\alpha_Y]{} & GY \end{array}$$

In other words, $\alpha_Y \circ Ff = Gf \circ \alpha_X$ for all $X \xrightarrow{f} Y$.

Let's consider two functors $F$ and $G$ between finite categories and a natural transformation $\alpha : F \Rightarrow G$:




Breaking the data in this figure down, we have that $\alpha_X = \mathrm{id}_Z$, $\alpha_Z = \mathrm{id}_Z$, and $\alpha_W = k$. Now we can check the naturality square for every morphism:

- Checking for $f$: we need to show that $\alpha_Y \circ F(f) = G(f) \circ \alpha_X$. Substituting, this is $\mathrm{id}_Z \circ k = \mathrm{id}_Z \circ k$.

### *10.2.1 More examples of natural transformations*

Think back to $\mathcal{C}$-indexed functions. We drew these as squares involving two Balloon diagrams stacked together. It may help to think of the naturality condition as an abstraction of such a picture, where Balloons are replaced by objects of $\mathcal{D}$:

$$\begin{array}{ccc} GX & \xrightarrow{Gf} & GY \\ {\scriptstyle \alpha_X}\uparrow & & \uparrow{\scriptstyle \alpha_y} \\ FX & \xrightarrow[Ff]{} & FY \end{array}$$

$$X \xrightarrow{f} Y$$

**Definition 31: Identity natural transformation**

Let $F : \mathcal{C} \to \mathcal{D}$ be a functor between two categories $\mathcal{C}, \mathcal{D}$. The identity natural transformation $\mathsf{id}_F : F \Rightarrow F$ is defined by $\mathsf{id}_{F,X} = \mathsf{id}_{F(X)}$. Naturality is verified by the following "abstract balloon diagram":

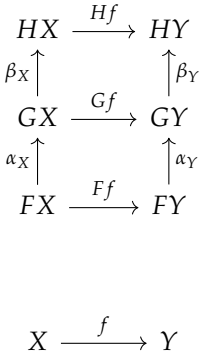
$$\begin{array}{ccc} FX & \xrightarrow{Ff} & FY \\ {\scriptstyle \mathsf{id}_{FX}}\uparrow & & \uparrow{\scriptstyle \mathsf{id}_{FY}} \\ FX & \xrightarrow{Ff} & FY \end{array}$$

$$X \xrightarrow{f} Y$$

**Definition 32: Composition of natural transformations**

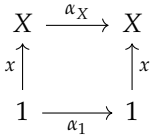
Given $\alpha : F \Rightarrow G$ and $\beta : G \Rightarrow H$, the composition $\beta \circ \alpha : F \Rightarrow H$ is defined by $(\beta \circ \alpha)_X = \beta_X \circ \alpha_X$. Naturality is verified by the

following "abstract balloon diagram":

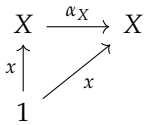



Examples

- Suppose you have two functors $F, G : X \to Y$ between preorders $X, Y$, aka two monotone functions. There is at most one natural transformation $\alpha : F \Rightarrow G$, which exists if and only if $F(x) \leq G(x)$ for all $x \in X$.
- Mind-bending example: there is an identity functor $\mathsf{id} : \mathsf{FinSet} \to \mathsf{FinSet}$. What does a natural transformation $\alpha : \mathsf{id} \Rightarrow \mathsf{id}$ look like? Think polymorphism: intuitively, $\alpha$ must have type "$\forall X.\mathsf{FinSet}(\mathsf{id}(X), \mathsf{id}(X))$", aka "$\forall X.X \to X$". This suggests that $\alpha$ is the identity, i.e., $\alpha_X = \mathsf{id}_X$ for all objects $X$ of $\mathsf{FinSet}$. Indeed, we can verify this follows from $\alpha$ being natural, as follows.
  - Let $X$ be an arbitrary object of $\mathsf{FinSet}$. We will show $\alpha_X$ is the identity function on $X$.
  - By function extensionality it's enough to show that for all $x \in X$ it holds that $\alpha_X(x) = x$.
  - Fix $x \in X$ arbitrary. We saw in a homework that $x$ corresponds to a $\mathsf{FinSet}$ morphism $\hat{x} : 1 \to X$, where 1 is the terminal object of $\mathsf{FinSet}$ (aka the one-point set). Now, by naturality of $\alpha$, the following square commutes:

    Because 1 is terminal, there can only be one morphism $1 \to 1$, namely the identity. Hence $\alpha_1 = \mathsf{id}_1$. Thus this square collapses to the following triangle:

But now this triangle expresses the fact that $\alpha_X(x) = x$, which is exactly what was to be shown.

## 10.3 *Functor categories*

**Definition 33: Functor category**

Let $\mathcal{C}$ and $\mathcal{D}$ be categories. Suppose that

1. The collection of functors from $\mathcal{C}$ to $\mathcal{D}$ forms a set
2. For any two functors $F, G : \mathcal{C} \to \mathcal{D}$, the collection of natural transformations from $F$ to $G$ forms a set

Then there is a *functor category*, written $[F; G]$, whose

- Objects are functors $\mathcal{C}$ to $\mathcal{D}$
- Morphisms from $F$ to $G$ are natural transformations $\alpha : F \Rightarrow G$

Identity morphism is the identity natural transformation, and composition is composition of natural transformations.

Examples

- As you can see, there are small set-theoretic nuisances with this definition. We will come back to this
- $\mathsf{FinSet}^{\mathsf{loop}}$ is a functor category! It's the functor category $[\mathcal{C}; \mathsf{FinSet}]$ where $\mathcal{C}$ is the category with one object $\star$, a special non-identity morphism $f$, and all composites one can make from this (aka $f^n$ for $n \in \mathbb{N}$). (In other words, $\mathcal{C}$ is the category you get from the monoid $(\mathbb{N}, +, 0)$ of natural numbers under addition.)
- $\mathsf{FinSet}^{\bullet \to \bullet}$, the category of finite functions, is a functor category! It's $[(\bullet \to \bullet); \mathsf{FinSet}]$ where $\bullet \to \bullet$ is the category with two objects and one non-identity arrow between them.
- As you can see, functor categories are kind of like "categories of diagrams": the objects of a functor category look like diagrams whose shape is given by the domain category. Keep this in mind; it'll come back later (limits and colimits)
- You may wonder: are $\mathcal{C}$-indexed sets a functor category? Hold that thought...

## 10.4 Full and faithful functors

We noted that a functor $F : \mathcal{C} \to \mathcal{D}$ ensures an abstract copy of $\mathcal{C}$ is inside $\mathcal{D}$. But, what if we want *more* of $\mathcal{C}$ to be visible inside $\mathcal{C}$? We can coax more structure out by enforcing that the functor does not collapse any structure.

Note that for each functor $F : \mathcal{C} \to \mathcal{D}$ and each pair of objects $X, Y$ in $\mathcal{C}$, you get a function $\mathcal{C}(X, Y) \to \mathcal{D}(FX, FY)$.

- A functor is *full* if this function is surjective: for any morphism $FX \xrightarrow{\hat{f}} FY$ in $\mathcal{D}$, there exists a morphism $X \xrightarrow{f} Y$ in $\mathcal{C}$ such that $\hat{f} = F(f)$.
- A functor is *faithful* if this function is injective: for any morphism for any morphisms $f, g : X \to Y$ in $\mathcal{C}$, if $Ff = Fg$ as morphisms $FX \to FY$ in $\mathcal{D}$, then $f = g$.

In particular, if a functor is *full and faithful*, then each function $\mathcal{C}(X, Y) \to \mathcal{D}(FX, FY)$ is a bijection, showing that $\mathcal{D}(FX, FY) \cong \mathcal{C}(X, Y)$ for all objects $X, Y$ of $\mathcal{C}$.

## 10.5 The category of sets

Naive definition, paralleling Definition 2: the category Set consists of

- Objects: the set of all sets
- Morphisms: tuples $(A, f, B)$ where $A, B$ are sets and $f : A \to B$ is a function

Obviously this definition is unworkable: there is no set of all sets. We have to finally confront the set-theoretic minutiae that we have swept under the rug so far. Russell famously showed that the collection of all sets does not form a set; it is "too big". We will adopt a way out which has become standard when doing highly categorical things: postulate the existence of a "very big set" that contains "enough sets to look like the set of all sets". Such a set is called a *Grothendieck universe*. We won't give the formal definition here; it's not essential to understand the details.[1] For our purposes, all that matters is that every Grothendieck universe $\mathcal{U}$ is a model of ZFC, hence "looks like the set of all sets". From now on we assume the following axiom.

[1] If you're curious, check out the nLab page for Grothendieck universe.

**Axiom 10.5.1.** There exists a Grothendieck universe $\mathcal{U}$.

The set $\mathcal{U}$ splits the world of sets into two halves:

- On one side there are the sets $X$ such that $X \in \mathcal{U}$. These sets are "small enough to fit into $\mathcal{U}$", so called *$\mathcal{U}$-small*.

- On the other side there are sets $X$ such that $X \notin \mathcal{U}$. These sets are "too large to fit", so called $\mathcal{U}$-*large*.

The key point is that, while we cannot define a category of *all* sets, we can define a category of $\mathcal{U}$-*small* ones.

**Definition 34: the category of ($\mathcal{U}$-small) sets**

Set is the category whose

- Objects are elements of $\mathcal{U}$, aka $\mathcal{U}$-small sets
- Morphisms from $X$ to $Y$ are functions $f : X \to Y$

Just as sets can be $\mathcal{U}$-small or $\mathcal{U}$-large, so too can categories. There are actually three tiers of "small"-ness for categories. First, a category can fit perfectly inside of the universe $\mathcal{U}$.

**Definition 35**

A category $\mathcal{C}$ is $\mathcal{U}$-*small* if both of its sets of objects and morphisms are.

More generally, a category could be too large to fit inside of $\mathcal{U}$, but it could be the case that for any two objects, the collection of *morphisms* between those two objects does fit inside of $\mathcal{U}$. Intuitively, this is like saying the category is $\mathcal{U}$-small if you zoom in on the morphisms betwen two given objects; hence this condition is called being "locally" small.

**Definition 36**

A category $\mathcal{C}$ is $\mathcal{U}$-*locally-small* if, for each pair of objects $X, Y$, the set $\mathcal{C}(X, Y)$ is $\mathcal{U}$-small.

Finally, a category is $\mathcal{U}$-large if it is neither small nor locally small.

### *10.5.1 Indexed set theory as a category*

Now we can show that $\mathcal{C}$-indexed sets form a category, so long as we take Set to mean the category of $\mathcal{U}$-small sets, and $\mathcal{C}$ is locally $\mathcal{U}$-small.

**Definition 37: category of $\mathcal{C}$-indexed sets**

Let $\mathcal{C}$ be a locally $\mathcal{U}$-small category. Then $\mathcal{C}$-indexed sets and functions between them form a functor category $[\mathcal{C}^{\mathsf{op}}; \mathsf{Set}]$.

**Proposition 10.5.2.** Let $\mathcal{C}$ be a locally $\mathcal{U}$-small category. The Yoneda embedding defines a functor $\yo : \mathcal{C} \to [\mathcal{C}^{\mathsf{op}}; \mathsf{Set}]$, and this functor is full and faithful.

*Proof.* Let $X$ and $Y$ be objects. To show that $\yo$ is full and faithful, we must show that $\mathcal{C}(X, Y) \cong [\mathcal{C}^{\mathsf{op}}; \mathsf{Set}](\yo X, \yo Y)$.

Let $X$ and $Y$ be objects. Then, by specializing the baby Yoneda lemma, $\{\alpha : \yo X \Rightarrow \yo Y\} \cong \yo(Y)(X)$. We have immediately that $\{\alpha : \yo X \Rightarrow \yo Y\} = [\mathcal{C}^{\mathsf{op}}; \mathsf{Set}](\yo X, \yo Y)$ by the definition of a functor category, and that $\yo(Y)(X) = \mathcal{C}(X, Y)$ by the definition of $\yo$. □

# 11
# Monoidal Categories

Thanks Jialu Bao for this lecture, and Shubh Agrawal for these scribe notes.

## 11.1 Categorifying Monoids

We have been working thus far with typed PLs, specified using typing rules. We used categories to give them semantics and diagrams to depict the categories. We will now discuss PLs with resources, and use *monoidal categories* to give them semantics, and use string diagrams to depict them.

First, let us recall the definition of monoid:

**Definition 38: monoid**

A triple $(M, \cdot, e)$ is a monoid if it satisfies:

1. **Associativity**: $\forall x, y, z \in M \,.\, (x \cdot y) \cdot z = x \cdot (y \cdot z)$
2. **Identity**: $\forall x \in M \,.\, e \cdot x = x = x \cdot e$

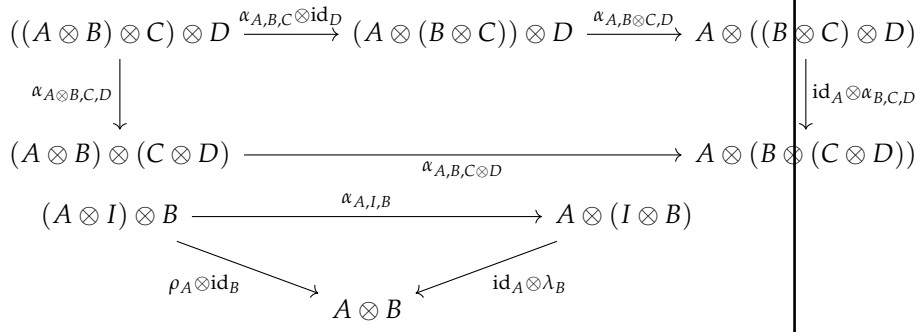
We have seen one way of categorifying monoids: construct a cateogry with one object and a morphism for each element of the monoid. The monoid operation then corresponds to sequential composition and the monoid axioms correspond to the category axioms.

What if we instead categorify monoids by making the *objects* form a monoid and where the monoid operation corresponds to some notion of parallel composition. Such categories are called *monoidal categories*. Monoidal categories are a category $\mathcal{C}$ together with an operation $\otimes$, called the tensor product, and a distinguished object $I$. The axioms are then encoded as a certain relationship between corresponding objects. Formally:

**Definition 39: monoidal category**

A category $\mathcal{C}$ is a monoidal category if:

1. there exists a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$
2. there exist natural isomorphisms:
   - $\alpha_{A,B,C} : A \otimes (B \otimes C) \Rightarrow (A \otimes B) \otimes C$
   - $\lambda_A : I \otimes A \Rightarrow A$
   - $\rho_A : A \Rightarrow A \otimes I$
3. such that the following diagrams commute:




The conditions that these diagrams commute are often simply referred to as "coherence conditions." The intuition is that the three natural isomorphisms need to interact with each other well in the sense that if we take any two paths between objects, then they are equivalent. Proving those two diagrams commute is sufficient to prove the *coherence theorem*, which makes the notion described above precise.

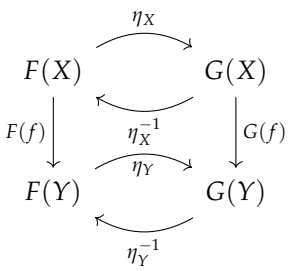
Let us define some of the other terms in the definition:

**Definition 40: bifunctor**

A functor from the product category $\mathcal{C} \times \mathcal{C}$ to $\mathcal{C}$.

**Definition 41: natural isomorphism**

natural isomorphism: a natural transformation $\eta : F \Rightarrow G$ is a natural isomorphism if for every object $X \in \mathcal{C}$, the component $\eta_X$ is an isomorphism. More concretely, we have the following diagram for every $f : X \to Y$ in $\mathcal{C}$:

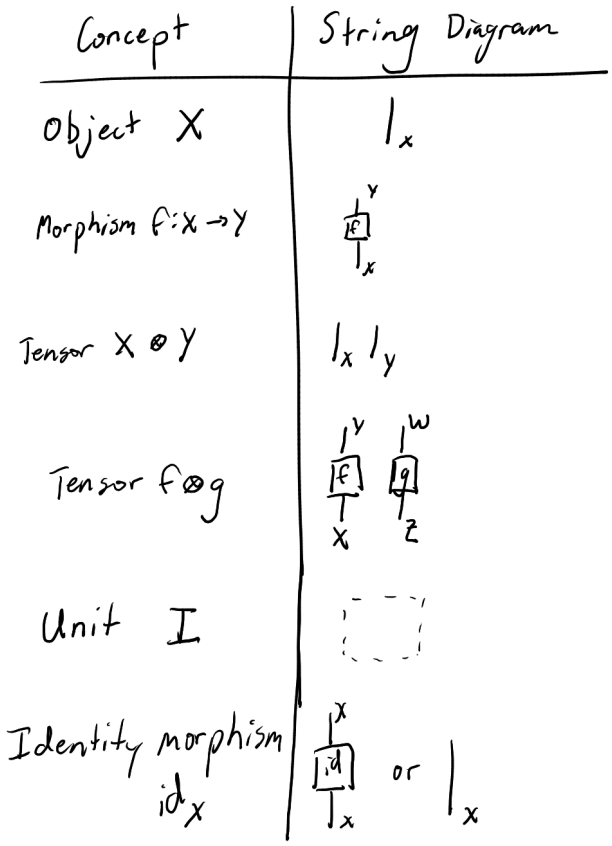



With these in hand, we can now define how to turn a monoid $(M, \cdot, e)$ into a monoidal category:

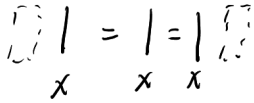
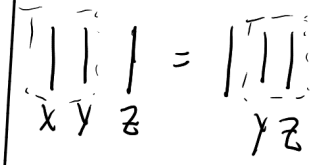
- The objects are the elements of $M$
- Morphisms are only the identity morphism on each object
- The tensor product $a \otimes b$ is the monoid operation $a \cdot b$
- The tensor product on morphisms $f \otimes g$ is trivial because both $f$ and $g$ are identity morphisms; thus, we have $\mathrm{id}_A \otimes \mathrm{id}_B = \mathrm{id}_{A \otimes B}$.
- We must show that the $\otimes$ defines a functor. It clearly respects identity by the definition above. showing it respects composition amounts to proving the following: $(f_1 \otimes f_2) \circ (g_1 \otimes g_2) = f_1 \circ g_1 \otimes f_2 \circ g_2$. This holds because all morphisms are the identity
- The objects we need to define natural transformations between are exactly equal by the monoid axioms, so we just define the transformations to be the identity on each component.

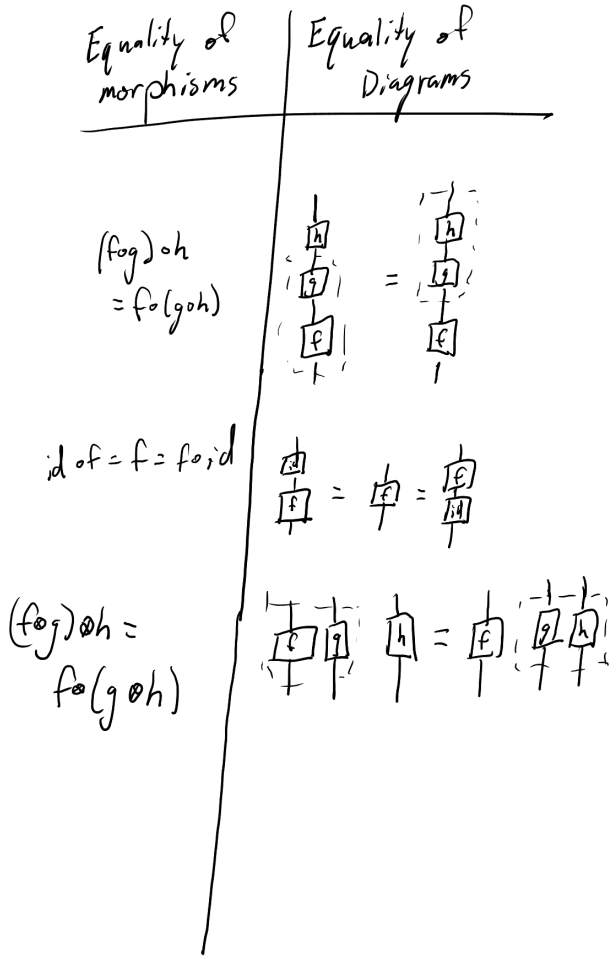
## *11.2 String Diagrams*

In string diagrams, we represent objects with wires and morphisms with boxes (including an identity). Sequential compositions correspond to wiring boxes together. Tensor product for objects corresponds to juxtaposing wires next to each other, and tensor products of wires is putting boxes next to each other. The monoid unit corresponds to a "phantom wire" which need not be drawn.




Many of the laws correspond to extremely intuitive facts about diagrams:

| Isomorphism of objects | Equality of diagrams |
| --- | --- |
| $I \otimes X \cong X \cong X \otimes I$ |  |
| $(X \otimes Y) \otimes Z \cong X \otimes (Y \otimes Z)$ |  |



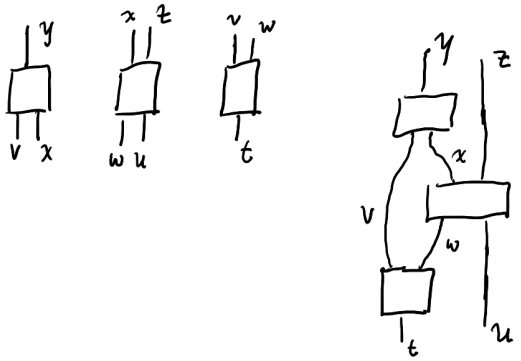

In any monoidal category, we have a notion of "generating objects and morphisms," which can't be decomposed into the tensor product of other objects; the category is then formed by taking all the possible tensor products.

## *11.3 Example: Real Numbers*

Consider the preorder of real numbers viewed as a category. We can define the tensor product on objects to be addition and the monoidal unit to be 0. Given two morphisms $n_1 \to n_2$, $m_1 \to m_2$, the tensor product is the morphism $n_1 + m_1 \to n_2 + m_2$, which we know exists by monotonicity of addition. Similar to the monoid example, the natural isomorphisms are the identity on each component because the corresponding objects are equal.

We can also use string diagrams to prove facts about this preorder. Here, we have diagrams representing proofs of $v + x \leq y$, $w + u \leq$

$x + z$, $t \leq v + w$. We can then wire the boxes together to get a proof of $t + u \leq y + z$:

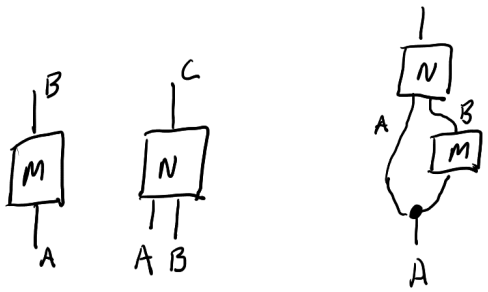



## *11.4 Example: Cartesian Categories*

If $\mathcal{C}$ has a product $\times$ and terminal object $T$, then this forms a monoidal category with $\otimes = \times$ and $I = T$.

The natural isomorphism for associativity was proven in Homework 1: $\alpha = \langle\langle \pi_1, \pi_1 \circ \pi_2\rangle, \pi_2 \circ \pi_2\rangle$. We also have $\lambda = \pi_1$ and $\rho = \langle \star, \mathrm{id}_A \rangle$.

In the typed PL perspective, we had rules for operating with products and the terminal object. We could then use these to construct more complicated terms. For example, we might ask given terms $x : A \vdash M : B$ and $\langle x, y\rangle : A \times B \vdash N : C$, how can we construct a term $T$ such that $x : A \vdash T : C$? We can do this with our term language. We can also do it in string diagrams:

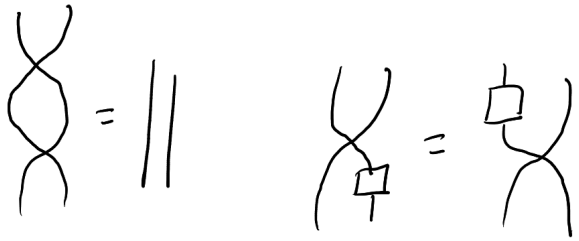



Doing this requires a *copy* operation, which allows us to duplicate $A$.

## 11.5 Additional Structure

### 11.5.1 Symmetry

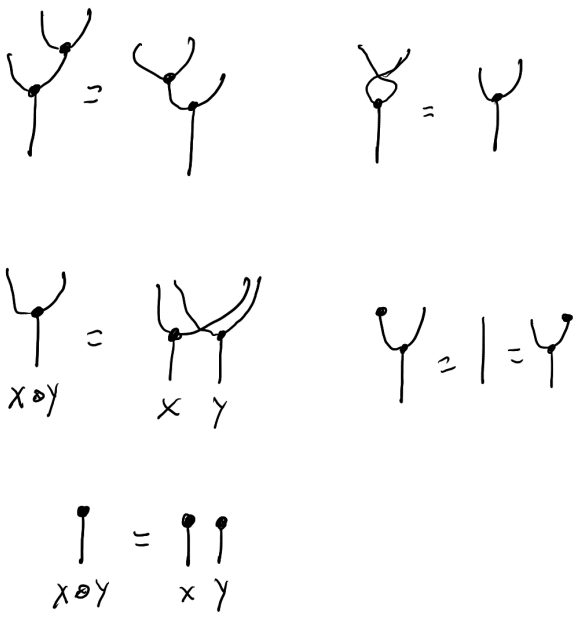
Notice that in both examples given, we have that the tensor product is commutative. The categories where this holds are called *symmetric monoidal categories*, which come equipped with an additional natural transformation $\gamma : A \otimes B \Rightarrow B \otimes A$. We get the following laws:



### 11.5.2 Copy and Discard

In order to encode all the same operations we expected in our STLC with pairs and unit, we needed a copy operation. The copy operation is a morphism $A \to A \otimes A$ for each $A$. In general, monoidal categories won't always have a copy operation. This corresponds to the intuition of monoidal categories as representing resources – if our resource can be copied, then our category would have a copier. If not, then it wouldn't.

Often, when we have a copy operation, we also have a discard operation $A \to I$. These operations will satisfy the following rules:


X⊗Y
X Y

# 12
# *Limits and colimits*

So far, we've been building a "categorical dictionary" of structure that categories might have that makes them useful models for programming languages. Categorical products let us model products in programming languages, exponential objects let us model higher-order functions. Limits and colimits take these ideas to their extreme: we will see that a category that has limits and colimits has a rich vocabulary of universal constructions of which products and exponentials are a special case. The utility of limits and colimits is in their generality, both in the kinds of constructions they give you and also how they are characterized. We will see that limits are *preserved* or *created* by the presence of certain functors, enabling us to characterize a large amount of categorical structure with relatively little work.

As an example of how limits generalize products, let

$$\mathsf{Point} = \{x : \mathbb{R}, y : \mathbb{R}, z : \mathbb{R}\}$$

This leads to the idea of indexed products.

**Definition 42: Indexed product**

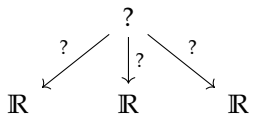
Let $(X_i)_{i \in I}$ be an $I$-indexed family of objects of a category $\mathcal{C}$. An *indexed product* of $(X_i)_{i \in I}$ consists of an object $P$ and a family of morphisms $(\pi_i : P \to X_i)_{i \in I}$ satisfying the following universal property: for any object $Y$ and any family of morphisms $(f_i : Y \to X_i)_{i \in I}$, there exists a unique morphism $\langle f_i \rangle_{i \in I} : Y \to P$ such that $\pi_i \circ \langle f_i \rangle_{i \in I} = f_i$ for all $i$ in $I$.

While records are typically finite, the definition of indexed product works just as well for infinite products. For example,

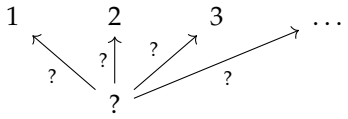
**Proposition 12.0.1.** 1 is the indexed product of $(k)_{k \in \mathbb{N}}$ in the category of natural numbers under divisibility.

More generally, indexed products in preorders are like infima.

Indexed products are in some sense the "largest" way to map into a tuple of objects. For example, the product for Point is the "largest" way of filling in the question marks in this diagram:




Similarly, 1 is the "largest" way to fill in the question marks in this diagram:



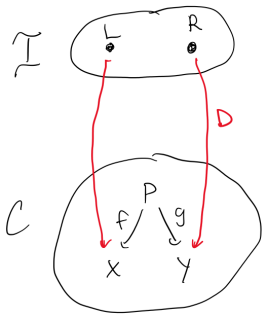

All of these diagrams are "discrete": they consist only of objects, with no morphisms connecting them. More generally, we can ask if it is possible to find the largest way of mapping into an arbitrary diagram consisting of both objects and morphisms. This is called a *limit*.

To be more precise about this, it is useful to give a formal definition of what a diagram is:

**Definition 43: Diagram**

For two categories $\mathcal{I}$ and $\mathcal{C}$, a *diagram of shape* $\mathcal{I}$ is a functor $F : \mathcal{I} \to \mathcal{C}$. The category $\mathcal{I}$ is called the *shape of the diagram*.

It is typical for the shape of a diagram to be a simple category with only a few objects and morphisms. For example, we can consider $\mathcal{I} = \bullet \quad \bullet$; then, a diagram of shape $\mathcal{I}$ in some category $\mathcal{C}$ would pick out a pair of two objects in $\mathcal{C}$. Here is a simple example of a diagram:




The essential idea of a limit is that it that it is the largest way of "mapping into a diagram". The way to formalize this is via the notion of a *cone*:

**Definition 44: Cone**

Let $D : \mathcal{I} \to \mathcal{C}$ be a diagram of shape $\mathcal{I}$ in a category $\mathcal{C}$. A *cone* over $D$ is a pair $(P, p)$ where $P$ is an object of $\mathcal{C}$ and $p$ is a family of morphisms $(p_i : P \to D(i))_{i \in I}$ such that $D(f) \circ p_i = p_j$ for all morphisms $f : i \to j$ of $\mathcal{I}$.

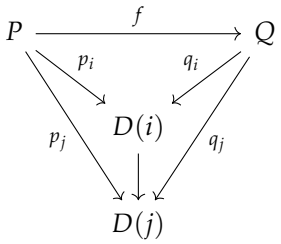
Let's understand this definition for the simple diagrams of shape $\mathcal{I} = \bullet \quad \bullet$ before we consider more more interesting shapes with morphisms in them. In the above example diagram, the tuple $(P, p)$ is a cone where $p_L = f$ and $p_R = g$.

Cones form a category:

**Definition 45: Morphism of cones**

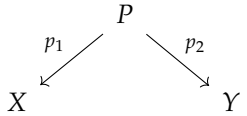
Let $D : \mathcal{I} \to \mathcal{C}$ be a diagram of shape $\mathcal{I}$ in a category $\mathcal{C}$. Let $(P, p)$ and $(Q, q)$ be two cones over $D$. A *morphism of cones from* $(P, p)$ *to* $(Q, q)$ is a morphism $f : P \to Q$ such that $p_i = q_i \circ f$ for all $i \in \mathcal{I}$.

A morphism of cones can be visualized as a kind of 3d prism:



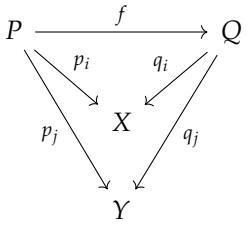

At the bottom of this prism, we have drawn a piece of the diagram $D$. The "bottom left face" of the prism is the cone $(P, p)$; the "bottom right face" is the cone $(Q, q)$. A morphism of cones from $(P, p)$ to $(Q, q)$ is a morphism $f : P \to Q$ as shown that makes all the triangles in the prism commute.

Looking at specific choices of diagram $D$ recovers the kinds of figures we were drawing. For example, suppose $D$ is the discrete diagram on two objects $X$ and $Y$. We get that a cone $(P, p)$ is a diagram of shape

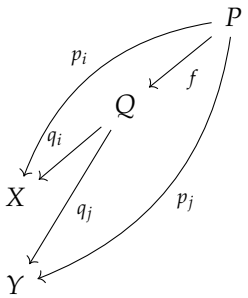



and a morphism from one such cone $(P, p)$ to another $(Q, q)$ is a

diagram of shape:




Now imagine taking the object $P$ and "dragging it up and to the right", until it's in the same plane as the triangle involving $Q, X, Y$:




Hopefully this diagram looks familiar: up to a rotation, it is the kind of diagram we were drawing when examining the universal property of product.

Together, cones over a diagram $D$ and morphisms between them form a category:

**Definition 46: category of cones**

Let $D : \mathcal{I} \to \mathcal{C}$ be a diagram in a category $\mathcal{C}$ of shape $\mathcal{I}$. The category of cones over $D$, written $\mathsf{Cone}(D)$, is the category whose objects are cones over $D$ and whose morphisms are morphisms of cones. (Because cone morphisms are simply morphisms of $\mathcal{C}$ satisfying a certain property, the definition of composition for cone morphisms and of the identity cone morphism are inherited from composition and identity in $\mathcal{C}$.)

The category of cones is designed so that it captures all the ways of mapping into a diagram. Recall our motivation that a limit ought to be the biggest way to map into a diagram. This hints at our final definition: the "biggest" object in a category is the terminal object,

**Definition 47: Limit**

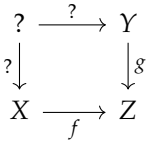
Let $D : \mathcal{I} \to \mathcal{C}$ be a diagram of shape $\mathcal{I}$ in a category $\mathcal{C}$. A *limit* of $D$ is a terminal object in $\mathsf{Cone}(D)$.

Now we can see how products are terminal in the categories of cones of shape $\mathcal{I} = \bullet \quad \bullet$. In this category, objects look like this:

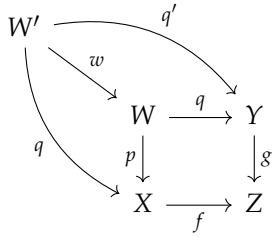
## 12.1 *Examples of limits*

We have seen two examples of limits already, in the homework.

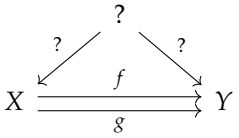
- Given two morphisms $X \xrightarrow{f} Z \xleftarrow{g} Y$, the *pullback* is the largest way of mapping into $f, g$, in other words the largest way of filling in the question marks in this diagram:



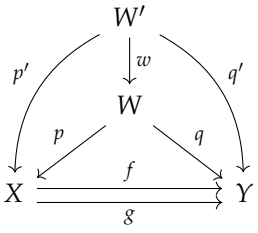

  Explicitly, a pullback is a 3-tuple $(W, p, q)$ where $p : W \to X$ and $q : W \to Y$ and $fp = gq$ satisfying the following universal property: for any other 3-tuple $(W', p', q')$ where $p' : W' \to X$ and $q' : W' \to Y$ and $fp' = gq'$, there exists a unique morphism $w : W' \to W$ such that $pw = p'$ and $qw = q'$.



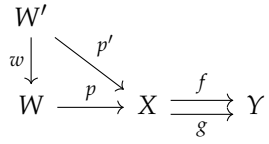

- Given two morphisms $X \underset{g}{\overset{f}{\rightrightarrows}} Y$, their *equalizer* is the largest way of mapping into $f, g$ in other words the largest way of filling in the question marks in this diagram:




  Explicitly, the equalizer is a 3-tuple $(W, p, q)$ where $p : W \to X$ and $q : W \to Y$ and $fp = q$ and $gp = q$ satisfying the following universal property: for any other 3-tuple $(W', p', q')$ where $p' : W' \to X$ and $q' : W' \to Y$ and $fp' = q'$ and $gp' = q'$ there exists a unique $w : W' \to W$ such that $pw = p'$ and $qw = q'$.



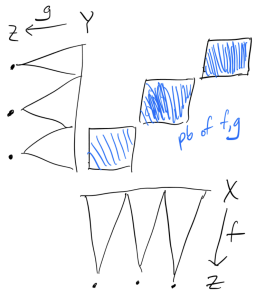
This definition can be simplified a bit: notice that in the 3-tuple $(W, p, q)$, the morphism $q$ is forced to be $fp$. So it is equivalent to say that an equalizer is a 2-tuple $(W, p)$ such that $fp = gp$, which satisfies the universal property that for any other $(W', p')$ with $fp' = gp'$, there exists a unique $w : W' \to W$ such that $wp = p'$ and $wq = q'$.




The equalizer looks like a refinement type:

$$\text{equalizer of } \; X \underset{g}{\overset{f}{\rightrightarrows}} Y \qquad \rightsquigarrow \qquad \{x : X \mid f(x) = g(x)\}$$

The pullback looks like a "fiber product":




There is an equivalent way of defining cones, that requires a bit more machinery.

**Definition 48**

Given an object $X$ of a category $\mathcal{C}$, and an arbitrary category $\mathcal{I}$, the *constant functor at $X$ from $\mathcal{I}$ to $\mathcal{C}$*, written $\Delta(X) : \mathcal{I} \to \mathcal{C}$, is the functor defined by $\Delta(X)(i) = X$ on objects $i$ of $\mathcal{I}$ and by $\Delta(X)(i \xrightarrow{f} j) = \mathsf{id}_X$ on morphisms.

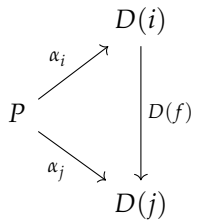
**Proposition 12.1.1.** A cone over $D : \mathcal{I} \to \mathcal{C}$ is equivalent to a pair $(P, \alpha)$ where $P$ is an object of $\mathcal{C}$ and $\alpha$ is a natural transformation $\Delta(P) \Rightarrow D$.

*Proof.* Unwinding definitions, the natural transformation $\alpha$ is a $\mathcal{I}$-indexed family of morphisms $(\alpha_i : \Delta(P)(i) \to D(i))_{i \in \mathcal{I}}$ satisfying the

following naturality square for all morphisms $i \xrightarrow{f} j$ of $\mathcal{I}$:

$$\begin{array}{ccc} \Delta(P)(i) & \xrightarrow{\alpha_i} & D(i) \\ {\scriptstyle \Delta(P)(f)}\downarrow & & \downarrow{\scriptstyle D(f)} \\ \Delta(P)(j) & \xrightarrow[\alpha_j]{} & D(j) \end{array}$$

But by definition of $\Delta(P)$, it holds that $\Delta(P)(i) = \Delta(P)(j) = P$ and $\Delta(P)(f) = \mathrm{id}_P$. Hence the left arrow in this commutative square is the identity, and the square can be collapsed into the following triangle which has to commute for all $i \xrightarrow{f} j$ in $\mathcal{I}$:




Hence the natural transformation $\alpha : \Delta(P) \Rightarrow D$ amounts to a family of morphisms $(\alpha_i : P \to D(i))_{i \in \mathcal{I}}$ satisfying the property that $D(f) \circ \alpha_i = \alpha_j$ for all $i \xrightarrow{f} j$ in $\mathcal{I}$. This is precisely the property of being a cone over $D$. □

Last time we saw by the Yoneda lemma that a representation for a $\mathcal{C}$-indexed set $P$ is equivalent to a terminal object in $\int P$, the category of elements of $P$. This fact provides an interesting equivalent definition of limit:

**Proposition 12.1.2.** Let $\mathcal{I}$ be a $\mathcal{U}$-small category. A limit of a diagram $D : \mathcal{I} \to \mathcal{C}$ is equivalent to a representation for the $\mathcal{C}$-indexed set $P$ defined by

$$P(X) = \text{the set of natural transformations } \Delta(X) \Rightarrow D.$$

*Proof.* The $\mathcal{C}$-indexed set $P$ is designed so that the category of cones over $D$ is equivalent to the category of elements of $P$. Hence, to have a terminal cone is to have a terminal object in this category of elements, which is equivalent to having a representation for $P$. □

### 12.1.1 *Some important facts*

**Definition 49: Limit preservation**

Let $\mathcal{C}$ and $\mathcal{D}$ be categories, let $F : \mathcal{C} \to \mathcal{D}$, and let $D : \mathcal{I} \to \mathcal{C}$ be a diagram in $\mathcal{C}$. Suppose $D$ has a limit $(P, p)$ in $\mathcal{C}$.

Then, the functor $F$ *preserves* this limit if the diagram $\tilde{D} = F \circ D$ in $\mathcal{D}$ has limit $(F(P), (F(p_i) : P \to D(i))_{i \in \mathcal{I}})$.

**Proposition 12.1.3.** The Yoneda embedding preserves limits.

We say a functor *reflects* limits if the converse of Definition 49 holds.

**Definition 50: Limit reflection**

Let $\mathcal{C}$ and $\mathcal{D}$ be categories, let $F : \mathcal{C} \to \mathcal{D}$, and let $D : \mathcal{I} \to \mathcal{C}$ be a diagram in $\mathcal{C}$. Let $(P, p)$ be a cone of shape $D$ in $\mathcal{C}$, and suppose the $\tilde{D} = F \circ D$ has limit $(F(P), (F(p_i) : P \to D(i))_{i \in \mathcal{I}})$ in $\mathcal{D}$. We say this limit is *reflected* into $\mathcal{C}$ if the cone $(P, p)$ is limiting in $\mathcal{C}$.

**Proposition 12.1.4.** Any full and faithful functor reflects limits.

## 12.2 *Every limit is the equalizer of a product*

We've seen two important examples of simple diagrams: the discrete diagram where there are no non-identity morphisms, and the equalizer diagram where there are exactly two morphisms between exactly two objects. These are both simple enough on their own, but what is interesting and powerful is that *all limits can be built out of these two simple diagrams*:

**Theorem 12.2.1.** Every limit is an equalizer of an indexed product.

Consequently, a category with indexed products and equalizers has all limits.

## 12.3 *Colimits*

The essence of colimits is essentially to flip the direction of all the arrows of limits. This seems simple at first, but surprisingly, colimits behave quite differently than limits. Let's quickly state the relevant definitions:

**Definition 51: Cocone**

Let $D : \mathcal{I} \to \mathcal{C}$ be a diagram of shape $\mathcal{I}$ in a category $\mathcal{C}$. A *cocone* over $D$ is a pair $(P, p)$ where $P$ is an object of $\mathcal{C}$ and $p$ is a family of morphisms $(p_i : D(i) \to P)_{i \in I}$ such that $p_i \circ D(f) = p_j$ for all morphisms $f : i \to j$ of $\mathcal{I}$.

**Definition 52: Morphism of cocones**

Let $D : \mathcal{I} \to \mathcal{C}$ be a diagram of shape $\mathcal{I}$ in a category $\mathcal{C}$. Let $(P, p)$ and $(Q, q)$ be two cocones over $D$. A *morphism of cocones from* $(P, p)$ *to* $(Q, q)$ is a morphism $f : P \to Q$ such that $f \circ p_i = q_i$ for all $i \in \mathcal{I}$.

Then, the category of cocones for a diagram $\mathcal{D}$ is written $\mathsf{CoCone}(\mathcal{D})$, analogous to the case for the category of cones $\mathsf{Cone}(\mathcal{D})$. Now we can state the definition of colimit:

**Definition 53: Colimit**

Let $\mathcal{D} : \mathcal{I} \to \mathcal{C}$ be a diagram of shape $\mathcal{I}$ in $\mathcal{C}$. A *colimit* of $\mathcal{D}$ is a cocone $(P, p)$ that is initial in $\mathsf{CoCone}(\mathcal{D})$.

Two examples in finite sets:

- coproduct, end up "tagged unions"
- coequalizer is unfortuantely much trickier. It ends up being a "generalized quotient"

# 13
# *Adjunctions*

- So far we've mostly focussed on characterizing properties of an individual category. Now let's broaden our focus to discuss in more detail relationships between categories.
- First, let's study the strongest possible relationship between two categories: does it mean for two categories to be equivalent? Following intuition from set isomorphism, we will want to design two functors between the categories that behave like inverses.
- For this definition we will need a way of talking equivalence of functors:

**Definition 54: Natural isomorphism of functors**

Let $\mathcal{C}$ and $\mathcal{D}$ be categories and $F, G : \mathcal{C} \to \mathcal{D}$ be functors. Then, $F$ and $G$ are *naturally isomorphic*, written $F \cong G$, if there is a natural transformation $\alpha : F \Rightarrow G$ whose components are isomorphisms.

- Now, to give an equivalence of categories, we must show that there exists a pair of functors $F$ and $G$ between them whose compositions are naturally isomorphic to the identity functors on each category:

**Definition 55: Equivalence of categories**

Let $\mathcal{C}$ and $\mathcal{D}$ be categories. Then, $\mathcal{C}$ is equivalent to $\mathcal{D}$, written $\mathcal{C} \cong \mathcal{D}$, if there are two functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{C} \to \mathcal{D}$ satisfying $F \circ G \cong \mathrm{id}_{\mathcal{C}}$ and $G \circ F \cong \mathrm{id}_{\mathcal{D}}$, where $\mathrm{id}_{\mathcal{C}}$ and $\mathrm{id}_{\mathcal{D}}$ are the identity functors on $\mathcal{C}$ and $\mathcal{D}$ respectively.

- There are some surprising categorical equivalences that are worth thinking about:

**Proposition 13.0.1.** The category of finite pointed sets $\mathsf{FinSet}^*$ is equivalent to the category of finite sets and partial functions $\mathsf{FinSet}_{\mathrm{par}}$.

The intuition behind this equivalence is interesting. First let's identify the functors that form the equivalence. Let $F : \mathsf{FinSet}_{\mathrm{par}} \to \mathsf{FinSet}^*$. The action on objects is straightforward: $F(X) = (X \uplus \{\star\}, \star)$ i.e., $F$ sends a set $X$ to a pointed set with $X$ extended by some element $\star$ with $\star$ as the point. The action on morphisms is interesting. Let $f : X \to Y$ be a partial function. Then, we define:

$$F(f) : (X \uplus \{\star\}, \star) \to (Y \uplus \{\star\}, \star)$$
$$= x \mapsto \begin{cases} \star & \text{if } f(x) \text{ undefined} \\ f(x) & \text{otherwise.} \end{cases}$$

Intuitively, the $\star$ stands in for the subset of $X$ on which $f$ is undefined. The inverse $G$ is also straightforward: its action on objects is to forget remove point, $G((X, \star)) = X \setminus \star$. Then, its action on morphisms map the point to undefined:

$$G(g) : X \setminus \star \to Y \setminus \star$$
$$= x \mapsto \begin{cases} \bot & \text{if } g(x) = \star \\ g(x) & \text{otherwise} \end{cases}$$

Now, show that there is a natural isomorphism $\alpha : F \circ G \Rightarrow \mathrm{id}_{\mathsf{FinSet}^*}$. This is not too hard and a good exercise. It requires showing that there is a $\mathsf{FinSet}^*$ isomorphism $(X, a) \cong ((X \setminus a) \uplus \star, \star)$.

- Categorical equivalences are very powerful when they arise but the condition is often too strong. A natural weakening of this notion will be an *adjunction*, which will be pairs of functors between categories are a form of "weak equivalence".

- The natural notion of weakening here will be as follows. Let $\mathcal{C}$ and $\mathcal{D}$ be categories and $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ be functors. Then, instead of requiring that round-trips of $F$ and $G$ be naturally isomorphic identity functors, an adjunction will require instead that there simply exist some pair of natural transformations:

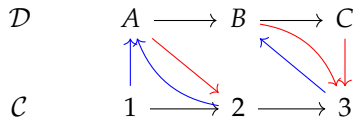
$$\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow G \circ F \qquad \epsilon : F \circ G \Rightarrow \mathrm{id}_{\mathcal{D}} \tag{13.1}$$

The natural transformation $\eta$ is called the **unit** of the adjunction and $\epsilon$ is called the **counit**, the functor $F$ is called the **left adjoint**, and the functor $G$ the **right adjoint**.

- There is one more coherence requirement we place on these natural transformations in order for them to form an adjunction. But, before we do that, let's examine this notion of notion of "weak equivalence" in a simple preorder category to understand what it means before we give the full definition of adjoint functors for general categories.

## 13.1 *Adjunctions for preorder categories: Galois connections*

- As usual it's very useful to study new category theory definitions in pre-order categories to gain order-theoretic intuition about them before generalizing to the general setting where there can be more than one morphism between any two objects.
- Let $(X, \leq_X)$, $(Y, \leq_Y)$ be preorders. Recall that a *monotone map* is a function $f : X \to Y$ that respects the preorder, i.e., if $x_1 \leq_X x_2$ then $f(x_1) \leq_Y f(x_2)$. Recall that functors between preorder categories are exactly monotone maps.
- Let's interpret the definition of adjoint in this setting. First, let's give two example functors between two preorder categories:




  Here we've drawn $F : \mathcal{C} \to \mathcal{D}$ in blue and $G : \mathcal{D} \to \mathcal{C}$ in red.
- Let's check if these functors satisfy the adjoint requirement as we've seen it so far. First, let's try to find a unit by finding a natural transformation $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow G \circ F$. Then, $\eta$ is a $\mathcal{C}$-indexed family of morphisms in $\mathcal{C}$:

$$\eta_1 = 1 \to 2 \quad \eta_2 = \mathrm{id}_2 \quad \eta_3 = \mathrm{id}_3$$

  We can also build the counit, which is a $\mathcal{D}$-indexed family of morphisms in $\mathcal{D}$:

$$\epsilon_A = \mathrm{id}_A \quad \epsilon_B = \mathrm{id}_B \quad \epsilon_C = B \to C$$

  Intuitively, these two natural transformations are characterizing what happens after round trips of following $F$ and $G$. The unit $\eta$ says that, after following a round trip of $F$ and then $G$, we must end up "in a higher place" than we started; dually, the counit $\epsilon$ is saying that after a round trip of $G$ then $F$ we must end up "in a lower place" than we started.

- Now we can reverse engineer an order-theoretic understanding for the unit and the counit of the adjunctions $F$ and $G$. This relationship has a name in order theory:

**Definition 56: Galois connection**

Let $(X, \leq_X)$ and $(Y, \leq_Y)$ be preorders and let $\alpha : X \to Y$ and $\gamma : Y \to X$ be monotone functions. Then, $\alpha$ and $\gamma$ form a *Galois connection* if, for all $x \in X$ and $y \in Y$:

$$x \leq \gamma(\alpha(x)) \qquad \text{and} \qquad \alpha(\gamma(y)) \leq y$$

- Pause for a moment to absorb how close a Galois connection is to an inverse. If $\gamma$ and $\alpha$ above were inverses, then indeed this would be an inverse.
- An equivalent definition to the above is that, for all $x \in X, y \in Y$, it holds that:

$$\alpha(x) \leq y \text{ if and only if } x \leq \gamma(y) \tag{13.2}$$

  Exercise: Why is this equivalence true?

- A programming languages example: Galois connections are important in defining abstract interpreters. The map $\alpha$ is called the *abstraction function* and $\gamma$ is called the *concretization function*, and $\alpha$ is left adjoint to $\gamma$.

### *13.1.1 Adjoints preserve structure*

- An important property of adjoints is that left adjoints preserve colimits and right adjoints preserve limits. This property is of course true for Galois connections, so let's see what it means in that setting first:

  **Theorem 13.1.1.** Let $(X, \leq)$ and $(Y, \leq)$ be complete partial orders (i.e., they both have all meets and joins) and let $\alpha : X \to Y$ and $\gamma : Y \to X$ be monotone functions where $\alpha$ is left adjoint to $\gamma$. Then, for any $x_1, x_2 \in X$ we have that:

$$\alpha(x_1 \sqcup x_1) = \alpha(x_1) \sqcup \alpha(x_2) \tag{13.3}$$

  and, for any $y_1, y_2 \in Y$ we have that:

$$\gamma(y_1 \sqcap y_2) = \gamma(y_1) \sqcap \gamma(y_2) \tag{13.4}$$

  *Proof.* Let's prove that the left adjoint preserves meets; the other one is a good exercise. By monotonicity, $\alpha(x_1) \sqcup \alpha(x_2) \leq \alpha(x_1 \sqcup x_2)$.

So, we need to show $\alpha(x_1 \sqcup x_2) \leq \alpha(x_1) \sqcup \alpha(x_2)$. We will use Eq. (13.2) to show this. Let $m = x_1 \sqcup x_2$ and $m^* = \alpha(x_1) \sqcup \alpha(x_2)$. Then,

$$\alpha(m) \leq m^* \iff m \leq \gamma(m^*)$$

Proceeding on the right hand side, we use the fact that $\gamma$ is monotone to build an intermediate inequality:

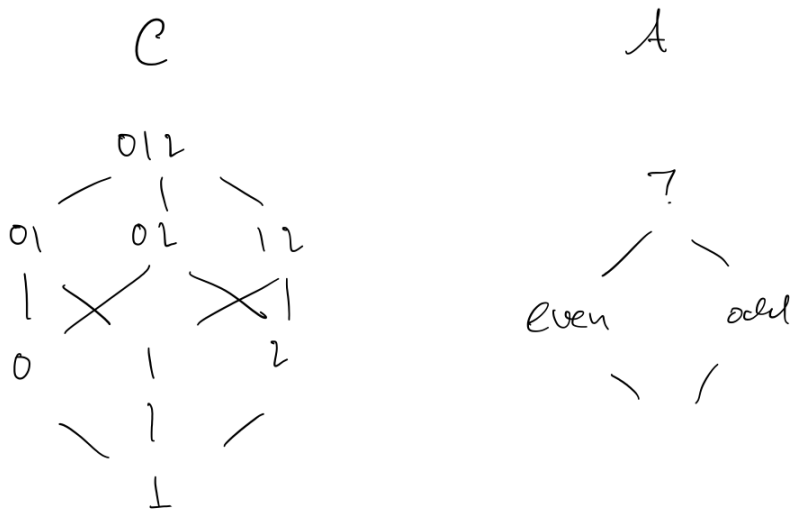
$$\gamma(\alpha(x_1)) \sqcup \gamma(\alpha(x_2)) \leq \gamma(\alpha(x_1) \sqcup \alpha(x_2))$$

So, if we can show $x_1 \sqcup x_2 \leq \gamma(\alpha(x_1)) \sqcup \gamma(\alpha(x_2))$, we are done. This follows directly from adjoint closure: $x \leq \gamma(\alpha(x))$ for all $x$. □

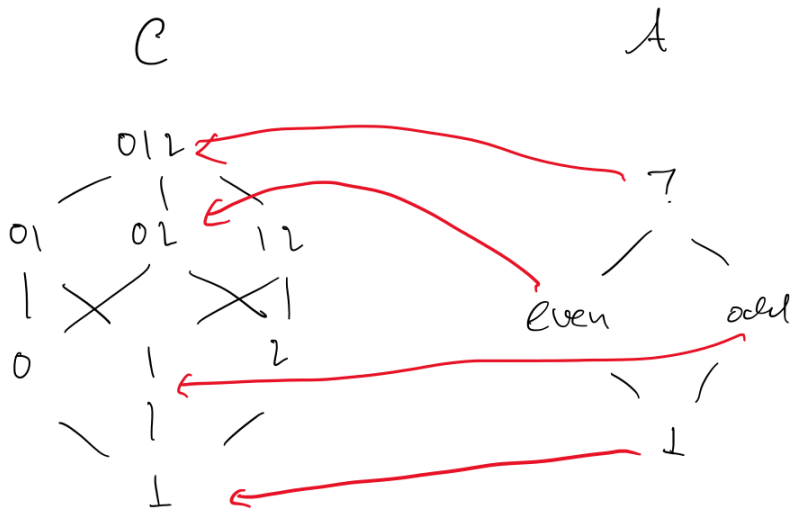
## 13.2 *Adjunctions for preorders, continued*

Notes from John's lecture

- Recall the Galois connection example:




- We defined a pair of functions $\alpha : \mathcal{C} \to \mathcal{A}$ and $\gamma : \mathcal{D} \to \mathcal{C}$, where $\alpha$ is left-adjoint to $\gamma$. We defined an example $\gamma$ as follows (shown in red):

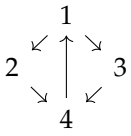



- To get to the general definition of preorder, we will study the properties of $\alpha$ and $\gamma$. We will want our general definition of adjoint to preserve *all* of this structure
- What is the property we want $\alpha$ to have, given this definition of $\gamma$? In some sense, it is *the most precise abstraction* would can make.
- For example, we can define a really bad $\alpha$ that is very imprecise. For instance, we could let $\alpha(\_) = ?$. But if we did this, then $\alpha$ would *not be an adjoint*.
- There is a collection of interesting properties we want $\alpha$ to have so that it is in some sense "most precise":
  1. $\alpha(c)$ is the smallest $a \in \mathcal{A}$ such that $c \leq \gamma(a)$.
  2. $\gamma(a)$ is the largest $c \in \mathcal{C}$ such that $\alpha(c) \leq a$.
  3. For all $a \in \mathcal{A}$ and $c \in \mathcal{C}$, $\alpha(c) \leq a$ if and only if $c \leq \gamma(a)$.
- These fun facts are more than just fun facts: they are in fact *equivalent definitions* of adjoints (at least for preorders).

  **Theorem 13.2.1.** Let $\mathcal{C} \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \mathcal{A}$ be a pair of functors. Then the following are equivalent:

  1. $\alpha \dashv \gamma$
  2. For any $c \in \mathcal{C}$, $\alpha(c)$ is the smallest $a \in \mathcal{A}$ such t hat $c \leq \gamma(a)$
  3. For any $a \in \mathcal{A}$, $\gamma(a)$ is the largest $c \in \mathcal{C}$ such that $\alpha(c) \leq a$.
  4. For all $a \in \mathcal{A}$ and $c \in \mathcal{C}$, $\alpha(c) \leq a$ if and only if $c \leq \gamma(a)$.

- This is going to be a big proof. First we'll prove that (1) implies (2) and (3). Then we'll prove that (2) and (3) both imply (4). Finally, we'll prove that (4) implies (1), giving the following strongly connected graph:




- (1) implies (3): Suppose $\alpha \dashv \gamma$. Then by definition $\alpha(\gamma(a)) \leq a$. So it only remains to show that $\gamma(a)$ is the largest $c$ satisfying $\alpha(c) \leq a$. Suppose $c \in \mathcal{C}$ such that $\alpha(c) \leq a$. Need to show that $c \leq \gamma(a)$. By $\gamma$ monotone,

$$\gamma(\alpha(c)) \leq \gamma(a)$$

  By definition of adjoint, $c \leq \gamma(\alpha(c))$, which completes the proof.

- (1) implies (2): analogous to the previous case.

- Proof that (2) implies (4).

  Suppose that for any $c \in \mathcal{C}$, $\alpha(c)$ is the smallest $a \in \mathcal{A}$ such that $c \leq \gamma(a)$. Want to show that for all $a \in \mathcal{A}$, $c \in \mathcal{C}$, $\alpha(c) \leq a \iff c \leq \gamma(a)$.

  ($\Rightarrow$): Have that $\alpha(c) \leq a$, need to show that $c \leq \gamma(a)$. By monotonicity, we have that $\gamma(\alpha(c)) \leq \gamma(a)$. By assumption, $c \leq \gamma(\alpha(c))$.[1]

  ($\Leftarrow$): Have $c \leq \gamma(a)$. Want $\alpha(c) \leq a$. Since $\alpha(c)$ is the smallest $a'$ such that $c \leq \gamma(a')$, and $a$ is one such $a'$, it must be that $\alpha(c) \leq a$.

[1] Note: cannot appeal to adjoint roundtrip property here. This is the assumption in equivalent condition (2).

- Proof that (4) implies (1). If for all $a \in \mathcal{A}, c \in \mathcal{C}$ it holds that $\alpha(c) \leq a$ iff $c \leq \gamma(a)$, then for all $a \in \mathcal{A}, c \in \mathcal{C}$ it holds that $\alpha(\gamma(a)) \leq a$ and $c \leq \gamma(\alpha(c))$.

  First, show $\alpha(\gamma(a)) \leq a$. By assumption this is true if and only if $\gamma(a) \leq \gamma(a)$, which vacuously holds. The other side is identical.

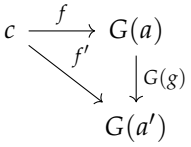
- Define some helpful notation: the sets $c \downarrow \gamma = \{a \mid c \leq \gamma(a)\}$ and $\alpha \downarrow a = \{c \mid \alpha(c) \leq a\}$. Then the previous theorem can be re-expressed as

$$\begin{aligned} & \alpha \dashv \gamma \\ \iff & (\forall c.\, \alpha(c) = \min(c \downarrow \gamma)) \\ \iff & (\forall a.\, \gamma(c) = \max(\alpha \downarrow a)) \\ \iff & (\forall a\, c.\, \alpha(c) \leq a \iff c \leq \gamma(a)). \end{aligned}$$

## 13.3 *Generalizing beyond preorders*

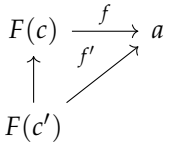
- We want a general definition of adjoints that generalizes the above four-way equivalence.
- To get there, we will first become more formal about this downarrow.
- Let $\mathcal{C}$ and $\mathcal{A}$ be categories. Then, for an object $c \in \mathcal{C}$, the category $c \downarrow G$ is a category whose objects are morphisms $c \to G(a)$ for each object $a \in \mathcal{A}$. Morphisms are commuting triangles:

  This is an instance of a *comma category*, and explains the notation we chose for the sets $c \downarrow \gamma$ and $\alpha \downarrow a$ in the previous section.




  where $c \xrightarrow{f} G(a)$ and $c \xrightarrow{f'} G(a')$ are two objects.
- Now that we have a category we have a notion of "smallest element" that generalizes categorically. This means: the categorically generalization of property (2) will be that for all objects $c \in \mathcal{C}$, the category $c \downarrow G$ has an initial object.

  This is a *definition of adjoint*.
- Generalizing (3), we need to build a category $F \downarrow a$. This category has objects $F(c) \xrightarrow{f} a$ for objects $c$ and morphisms $f$. Then, a morphism in $F \downarrow a$ is again a triangle:




- Let $\mathcal{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathcal{A}$

  Finally generalizing (1) is a bit tricky. Let $\eta : \mathrm{id} \Rightarrow G \circ F$ and $\varepsilon : F \circ G \Rightarrow \mathrm{id}$. Then, we have *coherence conditions*[2]
- The final generalization is (4). Categorically, this says for all $c \in \mathcal{C}$ and $a \in \mathcal{A}$, $\mathcal{A}(F(c), a) \cong \mathcal{C}(c, G(a))$ that is natural in $a$ and $c$.

  What does "natural in $a$ and $c$" mean? Well, we need two functors with compatible domains. Intuition: take all the things that the definitions says are natural and "poke a hole in them". Here are

[2] The idea of a coherence condition is essentially "equations you add to your definitions when you generalize from preorders to categories". Here, we've introduced extra ways to get between $\mathcal{C}$ and $\mathcal{A}$ via the units. This could add lots of extra morphisms to our category if we aren't careful. So, the coherence conditions restrict this proliferation of morphisms. It's useful to think about what happens without these coherence conditions; the easiest is to see that initiality is violated in the category $c \downarrow G$.

the holes: $\mathcal{A}(F(-_1), -_2) : \mathcal{C}^{\text{op}} \times \mathcal{A} \to \text{Set}$ and $\mathcal{C}(-_1, G(-_2)) : \mathcal{C}^{\text{op}} \times \mathcal{A} \to \text{Set}$. Then, the definition says there must be a natural isomorphism between these two functors.

- Packaging all these definitions up:

**Definition 57: Adjunction**

Let $\mathcal{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathcal{D}$ be a pair of functors. Then, the functor $F$ is *left adjoint* to $G$, written $F \dashv G$, if any of the following equivalent definitions hold:

- *Unit & co-unit*: There is a pair of natural transformations $\eta : \text{id}_{\mathcal{C}} \Rightarrow G \circ F$ and $\epsilon : F \circ G \Rightarrow \text{id}_{\mathcal{D}}$ that satisfy the *triangle identities*:

$$F \xRightarrow{F\eta} FGF \qquad G \xRightarrow{G\eta} GFG$$
$$\searrow^{1_F} \Downarrow \epsilon F \qquad \searrow^{1_G} \Downarrow \epsilon G$$
$$F \qquad\qquad G$$

- *Natural isomorphism of hom-sets:* For any objects $c \in \mathcal{C}$ and $d \in \mathcal{D}$ there is an isomorphism:

$$\mathcal{D}(Fc, d) \cong \mathcal{C}(c, Gd)$$

that is *natural in c and d*, i.e., there is a natural isomorphism between the two functors:

$$\mathcal{C}^{\text{op}} \times \mathcal{D} \underset{\mathcal{C}(-,G-)}{\overset{\mathcal{D}(F-,-)}{\rightrightarrows}} \text{Set}$$

- *Initiality condition*: For every object $c \in \mathcal{C}$, the category $c \downarrow G$ has an initial object.
- *Terminality condition*: For every object $d \in \mathcal{D}$, the category $F \downarrow d$ has a terminal object.

## 13.4 Examples

- A category $\mathcal{C}$ has terminal objects if and only if the there is a functor $F : \mathcal{C} \to 1$ (1 is the 1-object category) that has a right adjoint.
-

# 14
# *Elementary topoi*

This chapter is heavily based on [21] chapters 32 and 33. This chapter is work-in-progress.

- A *topos* is a category whose internal language is like set theory. If you like, you can think of it as a "`#lang` for math": it gives a reinterpretation internal to some category of all the mathematical symbols like $\wedge$, $\vee$, $\exists$, $\forall$, $\Rightarrow$, set-builder notation, etc.
- Many of the categories we've been discussing so far – presheaf categories, the category of $G$-sets, the category of finite transition systems – are examples of toposes.
- Today we'll discuss elementary toposes, which are toposes whose internal language is intuitionistic higher order logic. This means that the law of excluded middle and the axiom of choice may not hold in these topoi (i.e., logical statements like $\neg\neg\varphi$ may not entail $\varphi$)
- To get there, we will study the critical set-theoretic relationship of a *subset*, which is a special relationship between two objects that says one object a *part* of another. Once we have the categorification of subset in hand, we will use it to define other logical connectives.

## *14.1 Parts of objects*

- Recall the definition of a monomorphism, which is the categorification of an inclusion map (or injective function): from the homework:

**Definition 58: Monomorphism**

Let $\mathcal{C}$ be a category. A morphism $X \xrightarrow{f} Y$ is a *monomorphism* if for any morphisms $Z \underset{g_2}{\overset{g_1}{\rightrightarrows}} X$, if $f \circ g_1 = f \circ g_2$ then $g_1 =$

$g_2$. This property is called *left cancellation* of $f$. Monomorphisms are typically drawn with a hook arrow like $X \hookrightarrow Y$.

- In the homework we saw that, in the category of finite sets, a function is injective if and only if it is a monomorphism. Injective maps identify subsets, so monomorphisms will identify subobjects.
- Monomorphisms give us a natural notion of a categorification of subset, which will be called a *part of an object*:

**Definition 59: Part of an object**

Let $\mathcal{C}$ be a category and $X$ be an object in $\mathcal{C}$. Then, $Y$ is a *part* of $X$ if there is a monomorphism $Y \hookrightarrow X$.

## 14.2 *Subobject classifier*

- The subobject classifier is the categorification of a characteristic function.

**Definition 60: Characteristic function**

Let $S$ be a set and $A \subseteq S$. Then, the *characteristic function of* $A$, written $\chi_A : A \to \{\top, \bot\}$, is the function:

$$\chi_A = x \mapsto \begin{cases} \top & \text{if } x \in A \\ \bot & \text{otherwise.} \end{cases}$$

- Intuitively, characteristic functions ought to be in bijection with the subsets they are defined over. We can see this as follows. Suppose we have the following diagram in the category of finite sets:

$$\begin{array}{ccc} & & \{\star\} \\ & & \downarrow{\scriptstyle true} \\ S & \xrightarrow{\chi_A} & \{\top, \bot\} \end{array}$$

  Define $true : \star \mapsto \top$. Then, there is a set $B$ that is the pullback along the morphism $! : B \to \star$ and $\chi_A$, and this set $B$ must be isomorphic to $A$.

- This setup lends itself naturally to a categorical definition:

**Definition 61: Subobject classifier**

Let $\mathcal{C}$ be a category with terminal objects and $\Omega$ be an object in $\mathcal{C}$. Then, a morphism $true : \mathbf{1} \to \Omega$ is called a *subobject classifier of* $\mathcal{C}$ if for every monomorphism $j : U \hookrightarrow X$ there is a unique morphism $\chi_j : X \to \Omega$ called the *classifying morphism for the subobject represented by* $j$ such that the following diagram:

$$\begin{array}{ccc} U & \xrightarrow{!} & 1 \\ \downarrow{\scriptstyle j} & & \downarrow{\scriptstyle true} \\ X & \xrightarrow{\chi_j} & \Omega \end{array}$$

is a pullback along $\chi_j$ and !. In this situation, the object $\Omega$ is called the *object of truth values*.

- In the category of finite sets, the two-element set $\{\top, \bot\}$ is the object of truth values and the map $true : \{\star\} \to \top$ is the subobject classifier.

- The subobject classifier can be much more interesting in different categories! Let's see an example of one.

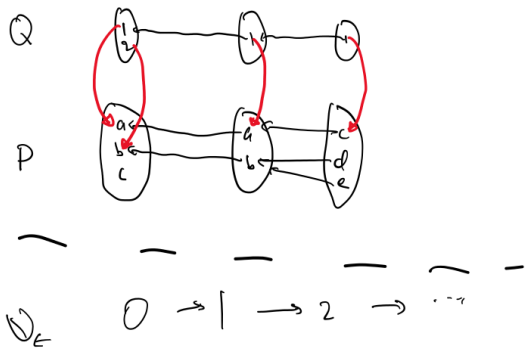
**Definition 62: Topos of trees**

Consider the following category called $\mathbb{N}_{\to}$:

$$0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow \ldots$$

The *topos of trees* is the category of presheaves on $\mathbb{N}_{\leftarrow}$ written $\mathsf{Psh}(\mathbb{N}_{\to})$.

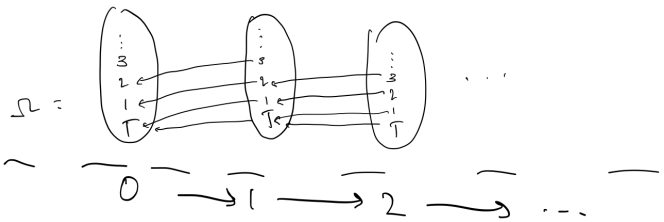
- We haven't defined what a topos is yet, so just treat the above as the definition for a special kind of presheaf category. But, let's see what a subobject classifier is in this category. First, let's look at what monomorphisms look like. Here is an example of a monomorphism:




The red arrow is a natural transformation $Q \Rightarrow P$ where each component of the natural transformation is an injective function. In a situation analogous to the one in finite sets, a natural transformation $\alpha : P \Rightarrow Q$ is a monomorphism in a presheaf category if and only if each component is an injective map.

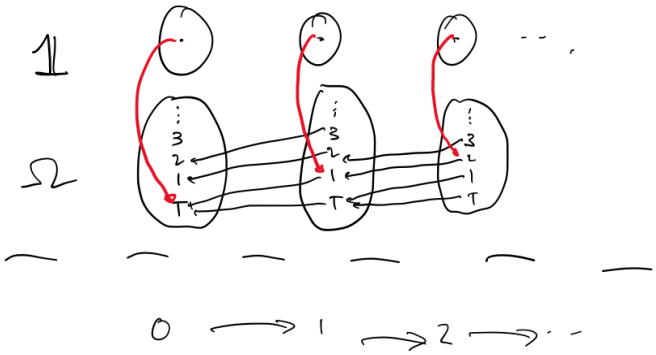
- Objects of this category can be thought of as "sets of facts that are true at particular times". They will correspond to *Kripke models*, and a monomorphism will correspond to a Kripke sub-world.

- Now we can think about the object of truth values. It is this:




Formally, $\Omega$ is a presheaf that maps each object to the set $\{\top, 1, 2, \cdots\}$, and each morphism to the predecessor function that maps 1 to $\top$ and $\top$ to $\top$. Intuitively, think of each element of each set as representing "the number of steps until something is true". With this intuition, a non-zero value represents that something is false at a particular time step.

- Given this object of truth values, we can design the subobject classifier as taking each time step and mapping it to the number of steps of "gas" that timestep has left:




- Let's use this object of truth values to find the classifying morphism for $\alpha$ above. We need to find a morphism $P \xrightarrow{\chi_\alpha} \Omega$
- The mere existence of subobject classifiers already forces your category to behave in set-like ways. An example: Every category with a subobject classifier is *balanced*, meaning that if there is an epimorphism and monorphism between two objects, they are isomorphic. This is just like in sets where the existence of an injection and surjection implies an isomorphism.

## 14.3 *Generalized elements*

Important picture: the subobject classifier gives us an interpretation for set-builder.

Think of $\varphi(x)$ as an open term with free variable of type $X$.

$$\begin{array}{ccc} \{x \in X \mid \varphi(x)\} & \longrightarrow & 1 \\ \downarrow & & \downarrow{\scriptstyle true} \\ X & \xrightarrow{\varphi(x)} & \Omega \end{array}$$

We can say *an element of $X$ satisfies $\varphi$* if we can find a morphism $x_0$ such that the following commutes:

$$\begin{array}{ccccc} & & \{x \in X \mid \varphi(x)\} & \longrightarrow & 1 \\ & \nearrow & \downarrow & & \downarrow{\scriptstyle true} \\ 1 & \xrightarrow{x_0} & X & \xrightarrow{\varphi(x)} & \Omega \end{array}$$

This is often too strong, e.g. Kripke semantics. Enter generalized elements:

We say $U \vDash \varphi$ (read "$U$ forces $\varphi$").

## 14.4 *Kripke-Joyal Semantics*

- The Kripke-Joyal semantics gives us a full picture of working internal to a topos

- These will look very familiar if you are familiar with separation logic papers

## *14.5 Elementary topos*

- The Kripke-Joyal semantics tells us what features we need our category to support, which will be an elementary topos:

**Definition 63: Elementary topos**

A category $\mathcal{C}$ is an elementary topos if and only if it has products, coproducts, exponentials, an initial object, a terminal object, an object of truth values, and satisfies that for every object $X$ the slice category $\mathcal{C}/X$ has products.

- Every presheaf category is an elementary topos (!!)

# 15
# Sheaf semantics

This chapter is heavily based on these notes from Alex Simpson: https://alexksimpson.github.io/Talks/TutorialOnSheafSemantics.pdf

Last time we mentioned that sheaves serve as a "hash-lang for math". More precisely, this means that sheaves provide a model of *higher-order intuitionistic logic*. In this logic it is possible to express a large amount of math; as a rough approximation, think of what can be written down as using Prop in Rocq (and without Inductives), or what can be expressed in Isabelle/HOL. While the standard model of this logic interprets propositions as classical truth values (either true or false), each category of sheaves provides a non-standard interpretation which often turns out to be convenient for modeling situations that arise in PL. This chapter provides a pared down presentation of the key idea behind how sheaves accomplish this. Rather than covering full higher-order intuitionistic logic, we will cover the intuitionistic propositional logic:

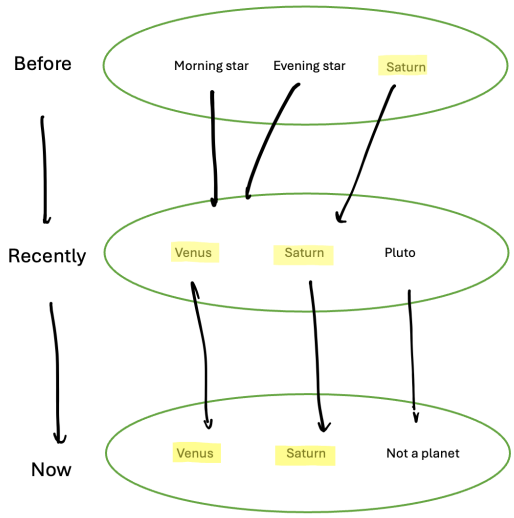
$$\varphi, \psi ::= \top \mid \bot \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid a$$

This logic consists of the usual intuitionistic logical connectives, plus some set of atomic propositions $a$.

Eventually we will see how to interpret this logic using the so-called *sheaf semantics*. We will build up to it by first talking about Kripke semantics, then presheaf semantics.

## 15.1 Kripke semantics

Kripke semantics are concerned with describing beliefs over time. For example, in the past, we believed that in the night sky there were 3 objects: a morning star and an evening star. Then, recently, we learned that in fact the morning star and evening star were both venus, and discovered new planets like Saturn and Pluto.




We can describe this situation by interpreting the connectives of intuitionistic propositional logic with a generalized set of truth values. The key idea of Kripke semantics is to introduce a set $W$ of *possible worlds*, whose elements represent states of knowledge. These states of knowledge are partially ordered according to what an *accessibility relation* $\leq$; the relation $w \leq w'$ (pronounced "world $w'$ is accessible from $w$") represents the situation where a state of knowledge $w$ can evolve into $w'$. For the planetary example above, one could set $W$ to the three-element set $\{\text{before}, \text{recently}, \text{now}\}$ ordered by before $\leq$ recently $\leq$ now.

Each proposition $\varphi$ is interpreted as a subset of $W$ giving the set of possible worlds in which $\varphi$ holds. Crucial to this interpretation is a *monotonicity* property that models the fact that knowledge is stable over time: if $\varphi$ holds in world $w$, then it also holds in all worlds accessible from $w$. The Kripke semantics of intuitionistic propositional logic is as follows.

$$
\begin{aligned}
[\![-]\!] &: \text{Formula} \to \wp(W) \\
[\![\top]\!] &= W \\
[\![\bot]\!] &= \varnothing \\
[\![\varphi \wedge \psi]\!] &= [\![\varphi]\!] \cap [\![\psi]\!] \\
[\![\varphi \vee \psi]\!] &= [\![\varphi]\!] \cup [\![\psi]\!] \\
[\![\varphi \Rightarrow \psi]\!] &= \{w \mid \text{for all } w' \geq w, \text{ if } w' \in [\![\varphi]\!] \text{ then } w' \in [\![\psi]\!]\}
\end{aligned}
$$

Of these definitions, the interpretation of $\Rightarrow$ is the most interesting. A naive interpretation that says $\varphi \Rightarrow \psi$ holds in world $w$ if and only if $w \in [\![\varphi]\!]$ implies $w \in [\![\psi]\!]$ is not valid here, because it fails to satisfy monotonicity.

These semantics are designed so that you have sound interpre-

tations of the proof rules of intuitionistic propositional logic: if $\varphi_1, \cdots, \varphi_n \vdash \psi$ then $[\![\varphi_1]\!] \cap \cdots \cap [\![\varphi_n]\!] \subseteq [\![\psi]\!]$, where $\vdash$ is intuitionistic syntactic entailment. In this sense the Kripke semantics provides a "domain-specific reinterpretation" of the usual rules of propositional logic, where each formula $\varphi$ denotes a monotone subset $[\![\varphi]\!]$ of the poset $W$.

As an example of this kind of reinterpretation, consider the poset $\mathbb{N}$ ordered by *reverse* inclusion, so that a natural number $n$ is accessible from $m$ if $n$ is *smaller* than $m$. This gives a semantics of intuitionistic propositional logic that is used in *step-indexing* [1].

[1] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 71–80. IEEE, 2009

## 15.2 *Presheaf semantics*

Presheaf semantics generalizes the poset $(W, \leq)$ from Kripke semantics to a category $\mathcal{C}$. Elements $w \in W$ are generalized to objects $c$ of $\mathcal{C}$ and the accessibility relation $w' \geq w$ is generalized to a morphism $f : c' \to c$. There are two things to note about this generalization. First, there can now be multiple such morphisms $f$ between any two given objects; intuitively, this is like a generalization of traditional Kripke semantics where there can be multiple different "ways" of going from one world to another. Second, there is a bit of reversal of direction going on: the accessibility relation $w' \geq w$ becomes the existence of a morphism $c' \to c$. This reversal of direction has to do with the fact that presheaves on a category $\mathcal{C}$ are contravariant; i.e., functors $\mathcal{C}^{\mathsf{op}} \to \mathsf{Set}$.

Rather than interpreting each formula as a monotone subset of $W$, the presheaf semantics interprets each formula as what is called a *sieve* on $\mathcal{C}$.

**Definition 64**

A *sieve* on a category $\mathcal{C}$ is a collection of morphisms $S$ in $\mathcal{C}$ that is closed under precomposition in the sense that if $c' \xrightarrow{f} c$ is in $S$ then so is $c'' \xrightarrow{g} c' \xrightarrow{f} c$ for any morphism $c'' \xrightarrow{g} c'$.

Intuitively, a sieve $S$ containing a morphism $c' \xrightarrow{f} c$ represents a proposition $\varphi$ where, if one is currently in world $c$, then $\varphi$ will become true if one transitions to world $c'$ via $f$. The requirement that $S$ be closed under precomposition provides a generalization of monotonicity.

The *presheaf semantics* is the corresponding generalization of

Kripke semantics where propositions are interpreted as sieves:

$$
\begin{aligned}
[\![-]\!] &: \text{Formula} \to \{\text{sieves on } \mathcal{C}\} \\
[\![\top]\!] &= \text{the set of all morphisms in } \mathcal{C} \\
[\![\bot]\!] &= \varnothing \\
[\![\varphi \wedge \psi]\!] &= [\![\varphi]\!] \cap [\![\psi]\!] \\
[\![\varphi \vee \psi]\!] &= [\![\varphi]\!] \cup [\![\psi]\!] \\
[\![\varphi \Rightarrow \psi]\!] &= \{c' \xrightarrow{f} c \mid \text{for all } c'' \xrightarrow{g} c', \text{ if } f \circ g \in [\![\varphi]\!] \text{ then } f \circ g \in [\![\psi]\!]\}
\end{aligned}
$$

Just like in the Kripke semantics, the presheaf semantics for Kripke logic will also validate intuitionistic logic entailments.

The following proposition verifies that the presheaf semantics indeed generalizes the posetal case.

**Proposition 15.2.1.** For any poset $(W, \leq)$, monotone subsets of $W$ are the same as sieves on $W^{\mathsf{op}}$, where $W$ is considered as a thin category, and the presheaf semantics agrees with the Kripke semantics.

In particular, the Kripke semantics for step-indexed logic corresponds to the presheaf semantics for the usual poset of natural numbers $(\mathbb{N}, \leq)$ made into a thin category. (Note that, unlike in the last section, this poset has the natural numbers ordered in the usual way. The appearance of $\mathsf{op}$ in the presheaf semantics automatically performs the reversal of the ordering relation that we did manually in the last section.)

A more interesting example is the presheaf semantics for name generation. We will take as our indexing category the category of finite injective maps $\mathsf{FinInj}^{\mathsf{op}}$. Objects are finite sets $\Gamma$ and homs are injection functions $\Gamma \to \Gamma'$. Intuitively, this represents how names can change over time: an injective function $\Gamma \to \Gamma'$ expands the set of available names from $\Gamma$ to $\Gamma'$, while possibly renaming the variables in $\Gamma$ along the way.

In the resulting presheaf semantics, the interpretation $[\![\varphi]\!]$ of a proposition $\varphi$ is the set of renamings $r : \Gamma \to \Gamma'$ such that $\varphi$ holds when $\Gamma'$ is "available", i.e., $\Gamma$ is rewritten into some new context $\Gamma'$ that validates $\varphi$.

Notice that this is *proof-relevant* in the sense that the kind of substitution performed matters: there can be many injective maps $\Gamma \to \Gamma'$. As usual the interesting case here is implication, which can be read off directly from the Kripke semantics above.

## *15.3 Sheaves*

At this point we've hit most of the common PL applications. Presheaf semantics get you very far: it is not that common to need to use

sheaves.

Whereas presheaf and Kripke semantics are well-suited to modeling *when* a proposition becomes true, sheaves are well-suited to modeling *where* a proposition becomes true. Sheaves interpret objects $c$ of the indexing category $\mathcal{C}$ not as states of knowledge, but rather as *spaces*; morphisms $f : c' \to c$ represent *subspace inclusions* that map a smaller space $c'$ into a larger space $c$. The sheaf semantics then models the spaces where a given proposition is true. The monotonicity requirements of Kripke and presheaf semantics continue to be relevant under this spatial reading: if a proposition holds in a space $c$, then one would intuitively still expect it to hold in a subspace $c'$ picked out by some inclusion $f : c' \to c$. But the spatial setting provided by sheaf semantics adds an additional twist not present in the more temporal reading provided by presheaf and Kripke semantics: it is possible for a family of "subspaces" $\{f_i : c'_i \to c\}_{i \in I}$ to jointly *cover* a space $c$.

The spatial interpretation offered by sheaf semantics affects the reading of disjunction. The Kripke/presheaf reading of disjunction says that $\varphi \vee \psi$ holds on a space $w$ if either $\varphi$ holds on all of $w$ or $\psi$ holds on all of $w$. But there is a third possibility: perhaps $\varphi$ holds on some piece $w_1$ of $w$, and $\psi$ holds on some piece $w_2$ of $w$, and together $w_1, w_2$ cover $w$.

A simple example is given by interpreting "space" as "subset of $\mathbb{R}$" and "inclusion" as subset inclusion. These spaces along with their inclusion relations forms a thin category $\wp(\mathbb{R})$. Families of morphisms with a common codomain $U$ in this setting are families $(U_i)_{i \in I}$ of subsets of $\mathbb{R}$ where $U_i \subseteq U$ for all $i \in I$. It is then natural to say that such a family *covers* $U$ if they union to $U$: $\bigcup_i U_i = U$. The sheaf semantics for this notion of cover interprets each proposition $\varphi$ as a collection $[\![\varphi]\!]$ of subsets of $\mathbb{R}$; a subset $U$ is in $[\![\varphi]\!]$ if "$\varphi$ is true everywhere in $U$". For example, one can define an atomic proposition $\varphi$ ="is less than one" whose interpretation $[\![\varphi]\!]$ is the collection of all those subsets $U$ of $\mathbb{R}$ contained in the $(-\infty, 1)$, and an atomic proposition $\psi$ ="is greater than zero" whose interpretation $[\![\psi]\!]$ is the collection of all those subsets contained in $(0, \infty)$. Now consider the disjunction $\varphi \vee \psi$. Intuitively, this proposition should hold *everywhere* in $\mathbb{R}$, as every real number is either less than one or greater than zero: we should expect $[\![\varphi \vee \psi]\!] = [\![\top]\!] =$ the set of all subsets of $\mathbb{R}$. But the presheaf semantics says that $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, which consists only of those subsets of $\mathbb{R}$ are either contained in $(-\infty, 1)$ or $(0, \infty)$. The sheaf semantics corrects this by taking into account the fact that $(0, \infty)$ and $(-\infty, 1)$ form a *cover* of $\mathbb{R}$: a disjunction $\varphi \vee \psi$ holds on a subset $U$ of $\mathbb{R}$ under the sheaf semantics if there is a covering $U = U_1 \cup U_2$ of $U$ by two subspaces $U_1, U_2$ such that $\varphi$ holds on

$U_1$ and $\psi$ holds on $U_2$.

More generally, the sheaf semantics is parameterized by a relation $T$ called a *coverage* that says when a given family of morphisms $\{f_i : c_i \to c\}_{i \in I}$ with common codomain $c$ counts as a cover of $c$.

**Definition 65**

A *coverage* on a category $\mathcal{C}$ is a relation $T$ between families of morphisms with common codomain $c$ and objects $c$ of $\mathcal{C}$. If $T$ relates a family $\{f_i : c_i \to c\}_{i \in I}$ to an object $c$ we say that $\{f_i : c_i \to c\}_{i \in I}$ is a *$T$-cover* of $c$. The relation $T$ must satisfy the following properties:

- Reflexivity: the singleton set $\{\mathsf{id}_c : c \to c\}$ is in $T$ for every object $c$
- Transitivity: if $\{f_i : c_i \to c\}_{i \in I}$ is in $T$ and for every $i \in I$ there is a family $\{f_{ij} : c_{ij} \to c_i\}_{j \in J_i}$ in $T$, then the family $\{f_i f_{ij} : c_{ij} \to c\}_{i \in I, j \in J_i}$ is in $T$
- Stability: if $\{f_i : c_i \to c\}_{i \in I}$ is in $T$ and $g : c' \to c$ is an arbitrary morphism into $c$, then there exists a family $\{f'_j : c'_j \to c'\}_{j \in J}$ such that every $g f'_j$ factors through some $f_i$; that is, for every $j \in J$ there exists an $i \in I$ such that $g f_j = f_i g'$ for some $g'$.

These coverage conditions are best visualized as closure conditions on families of morphisms drawn as trees. Reflexivity says that the trivial tree with just an identity morphism counts as a cover; transitivity says that a family of covering trees can be grafted onto another to form a new covering tree; stability says that a covering tree can be "pulled back" along a morphism $g : c' \to c$.

With the definition of coverage in hand, the sheaf semantics of intuitionistic propositional logic is as follows.

$$
\begin{aligned}
[\![-]\!] &: \text{Formula} \to \{\text{sieves on } \mathcal{C}\} \\
[\![\top]\!] &= \text{the set of all morphisms in } \mathcal{C} \\
[\![\bot]\!] &= \{f : c' \to c \mid \varnothing \text{ is a } T\text{-cover of } c\} \\
[\![\varphi \wedge \psi]\!] &= [\![\varphi]\!] \cap [\![\psi]\!] \\
[\![\varphi \vee \psi]\!] &= \{f : c' \to c \mid \text{there is a } T\text{-cover } \{f_i : c_i \to c'\}_{i \in I} \text{ of } c' \text{ such that } f f_i \in [\![\varphi]\!] \text{ or } f f_i \in [\![\psi]\!] \text{ for all } i\} \\
[\![\varphi \Rightarrow \psi]\!] &= \{c' \xrightarrow{f} c \mid \text{for all } c'' \xrightarrow{g} c', \text{ if } f \circ g \in [\![\varphi]\!] \text{ then } f \circ g \in [\![\psi]\!]\}
\end{aligned}
$$

Like the presheaf semantics, this semantics interprets each proposition as sieve. But the sheaf semantics satisfies an additional invariant. Each sieve produced by the sheaf semantics is what is known as *$T$-closed*; we omit the formal definition here, but intuitively this says

that if a proposition holds for every component of a covering family, then it holds for the space covered.

Examples of coverages:

- The notion of cover for subsets of $\mathbb{R}$ forms a coverage on $(\wp(\mathbb{R}), \subseteq)$ considered as a thin category.
- One can place a coverage on the category $\mathsf{Formula}$ of propositional formulae that says that a finite set $\varphi_1, \ldots, \varphi_n$ covers a formula $\varphi$ if $\varphi_1 \vee \cdots \vee \varphi_n \dashv\vdash \varphi$. Intuitively, this notion of coverage corresponds to the notion of cover one would get by intepreting formulas as sets of models.
- A more exotic notion of coverage, for the category $\mathsf{FinInj}^{\mathsf{op}}$, is given by declaring any singleton set to be a cover (that is, $\{r : \Gamma \to \Gamma'\}$ covers $\Gamma$ for any $\Gamma, \Gamma', r$.). This coverage corresponds to a category of sheaves known as the *Schanuel topos*, and produces a sheaf semantics that can be used to talk about name generation as modeled by *nominal sets* [2].

[2] Andrew M Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge University Press, 2013

### *15.3.1 Where are the categories?*

One might have noticed that categories of presheaves or sheaves have not played a role in any of the above. This is because we have restricted the discussion to propositional logic. Moving from propositional to first-order logic requires interpreting formulas-in-context $\Gamma \vdash \varphi$, which in turn requires interpreting contexts $\Gamma$ and upgrading the semantics of formulas to include substitutions $\gamma \in [\![\Gamma]\!]$. It will turn out that this upgrade requires interpreting each $[\![\Gamma]\!]$ as a presheaf (in the presheaf semantics) or sheaf (in the sheaf semantics).

We briefly sketch how this interpretation works. Recall the planetary picture from before. Turning this picture on its side, we obtain a balloon diagram lying over the category $W^{\mathsf{op}}$, where $W$ is the poset before $\leq$ recently $\leq$ now. This balloon diagram defines a presheaf $\mathsf{Sky}$ on $W^{\mathsf{op}}$, whose elements model our conception of objects in the night sky over time. This presheaf can serve as the interpretation of a basic sort $\mathtt{Sky}$ of a first-order intuitionistic logic. Logical predicates $x : \mathtt{Sky} \vdash \varphi(x)$ then denotes subpresheaves: for instance, the points in the diagram highlighted in yellow depict a subpresheaf corresponding to the predicate $\varphi$ that out those objects that are considered planets through to the present day.

# 16
# Monads

A monad is just a monoid in the category of endoburritos.

Monads are widely used for structuring effectful computation in functional programs. For example, we can have a small functional programming language with a monadic type-former $\mathsf{T}$:

$$\begin{aligned} A, B &::= \mathsf{Unit} \mid A \times B \mid A \to B \mid \mathsf{T}\ A \\ M, N &::= \langle\rangle \mid \langle M, N\rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid \lambda\, x.\, M \mid M\ N \mid x \\ &\quad \mid \mathsf{pure}\ M \mid x \leftarrow M; N \end{aligned} \tag{16.1}$$

The syntax $\mathsf{pure}\ M$ *lifts* a pure computation into a monadic one, and the syntax $x \leftarrow M; N$ is a *monadic bind* that enables computation inside the monad. The way that these syntactic forms are used is best understood via their typing judgments:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{pure}\ M : \mathsf{T}\ A} \qquad \frac{\Gamma \vdash M : \mathsf{T}\ A \qquad \Gamma, x : A \vdash N : \mathsf{T}\ B}{\Gamma \vdash x \leftarrow M; N : \mathsf{T}\ B}$$

If you've programmed in Haskell, you're quite familiar with examples of monads. A simple example is the `Maybe` monad, which lets you program with values that can be either `Nothing` or `Just` a value. This makes it a convenient pattern for implementing and sequencing partial functions, as the following Haskell code illustrates:

```
divide :: Double -> Double -> Maybe Double
divide a 0 = Nothing
divide a b = Just $ a / b

my_computation :: Double -> Maybe Double
my_computation x = do
  div1 <- divide 10 x;
  div2 <- divide 20 x;
  divide div1 div2
```

Let's consider now the denotational semantics of monads. Let's start with the type former $\mathsf{T}\ M$. We typically think of the interpretation of types as "a space of values that behave like a type", where

we use universal properties to characterize what it means to behave like a type. A monad is a new and exciting creature when viewed from this lens: it *takes in a type* $A$ (so it is higher-order), and it produces a *new type* $[\![\mathsf{T}\ A]\!]$ that has $A$ living inside of it. This feels very functorial. We can couple this intuition with the requirement that denotational semantics be defined compositionally, and we can arrive at the conclusion that $[\![\mathsf{T}]\!]$ *is a functor*:

$$[\![\mathsf{T}\ A]\!] = [\![\mathsf{T}]\!]\,([\![A]\!]) \tag{16.2}$$

Concretely, suppose we are interpreting our small higher-order language in some Cartesian-closed category $\mathcal{C}$. Then, $[\![\mathsf{T}]\!]$ is an *endofunctor* $[\![\mathsf{T}]\!] : \mathcal{C} \to \mathcal{C}$, a particular kind of functor from a category into itself. One way of understanding what $T$ is is that it maps elements to *generalized elements* that are imbued with the additional structure of the monad.

[1] has a very nice discussion of this motivation of monads as "generalized elements of a category".

As usual we don't know what kind of properties the functor $[\![\mathsf{T}\ A]\!]$ must have yet; to see those, we'll need to reverse-engineer them from how monads are used. Let's start by examining `pure`. By our usual translation of typing judgments into morphisms, we can conclude that $[\![\Gamma \vdash \mathsf{pure}\ M : \mathsf{T}\ A]\!]$ must have type $[\![\Gamma]\!] \to [\![\mathsf{T}]\!]\,([\![A]\!])$. We need to build this morphism. We know how to get close to it: $[\![M : A]\!]$ has type $[\![\Gamma]\!] \to [\![A]\!]$. But, we need an extra morphism to get from $A$ to $\mathsf{T}\ A$: this is called the *unit* of the monad at $[\![A]\!]$, and will be denoted $\eta_{[\![A]\!]}$:

$$[\![\Gamma \vdash \mathsf{pure}\ M : \mathsf{T}\ A]\!] = [\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![A]\!] \xrightarrow{\eta_{[\![A]\!]}} [\![\mathsf{T}]\!]\,([\![A]\!])$$

In Haskell, $\eta$ is called `pure` or `return`. Notably, in Haskell, `pure` is a polymorphic function of type `a -> m a` for some monad `m` and type parameter `a`. This is a polymorphic function, so it makes sense to think of $\eta$ as a *natural transformation* $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow T$.

```
class Functor m => Monad m where
  return :: alpha -> m alpha
  join :: m (m alpha) -> m alpha
```

Interpreting bind:

$$\left[\!\!\left[\frac{\Gamma \vdash M : \mathsf{T}\ A \qquad \Gamma, x : A \vdash N : \mathsf{T}\ B}{\Gamma \vdash x \leftarrow M; N : \mathsf{T}\ B}\right]\!\!\right] : [\![\Gamma]\!] \to [\![\mathsf{T}]\!]\,([\![B]\!])$$

$$= [\![\Gamma]\!] \xrightarrow{\langle \mathrm{id}, [\![M]\!] \rangle} [\![\Gamma]\!] \times [\![\mathsf{T}]\!]\,([\![A]\!])$$

$$\xrightarrow{\text{strength}} [\![T]\!]\,([\![\Gamma]\!] \times [\![A]\!])$$

$$\xrightarrow{[\![\mathsf{T}]\!]([\![N]\!])} [\![\mathsf{T}]\!]\left([\![\mathsf{T}]\!]\,([\![B]\!])\right)$$

$$\xrightarrow{\mu} [\![\mathsf{T}]\!]\,([\![B]\!])$$

## 16.1 What's the problem?

"A monad is just a monoid in the category of endofunctors, what's the problem?"

We've seen how a monad consists of (1) an endofunctor $T : \mathcal{C} \to \mathcal{C}$; (2) a *unit* natural transformation $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow T$ (3) a *multiplication (or join)* natural transformation $\mu : T^2 \Rightarrow T$. As usual, these special morphisms must satisfy certain laws in order to behave coherently. In particular, these two natural transformations need to satisfy natural *monoid laws*. Recall the definition of a monoid:

**Definition 66: Monoid**

A *monoid* is a triple $(X, \bullet, e)$ where $X$ is a set, $\bullet$ is a *multiplication operator* $\bullet : X \times X \to X$, and $e \in X$ is a distinguished *unit element* satisfying: (1) *associativity*: for any $a, b, c \in X$ it is the case that $a \bullet (b \bullet c) = (a \bullet b) \bullet c$ and (2) unitality: for any $a \in X$ it is the case that $a \bullet e = e \bullet a = a$.

A monad will satisfy categorical analogs of the monoid laws: the unit natural trasnformation $\eta$ must behave like a monoidal unit, and the multiplication natural transformation $\mu$ must behave like a monoidal product. Let's examine associativity first. This requirement can be captured by the following commutive square:

$$\begin{array}{ccc} T^3 & \xRightarrow{T\mu} & T^2 \\ \Downarrow{\scriptstyle \mu T} & & \Downarrow{\scriptstyle \mu} \\ T^2 & \xRightarrow{\mu} & T \end{array} \tag{16.3}$$

This square has some unfamiliar notation in it: the natural transformations $T\mu$ is given by composing the natural transformation $\mu$ with the functor $T$. Concretely, the natural transformation $T\mu$ has components $(T\mu)_A : T^3(A) \to T^2(A)$ given by composing $T$ with each component of $\mu$, i.e. $(T\mu)_A = T \circ \mu_A$. The definition for $\mu T$ is analogous.

The next requirement on the monad natural transformations is that $\eta$ behaves like a monoidal unit. This is captured by the following commuting diagram:

$$\begin{array}{ccccc} T & \xRightarrow{\eta T} & T^2 & \xLeftarrow{T\eta} & T \\ & \searrow^{\mathrm{id}_T} & \Downarrow{\scriptstyle \mu} & \swarrow_{\mathrm{id}_T} & \\ & & T & & \end{array} \tag{16.4}$$

There are two faces of this diagram each capturing one of the unitality requirements. The left square asserts that $\mu \circ \eta T = \mathrm{id}_T$, and the

right square asserts that $\mu \circ T\eta = \mathrm{id}_T$; these equations correspond naturally to the monoid unitality law. Together, these equations define a monad:

**Definition 67: Monad**

Let $\mathcal{C}$ be a category. A *monad* on $\mathcal{C}$ is a triple $(T, \eta, \mu)$ where $T : \mathcal{C} \to \mathcal{C}$ is an endofunctor, $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow T$ is the *unit* natural transformation, and $\mu : T^2 \Rightarrow T$ is a *multiplication (or join)* natural transformation, satisfying the associativity law in Eq. (16.3) and unit law in Eq. (16.4).

## 16.2 Examples of monads

### 16.2.1 Maybe monad

The *maybe monad* is often the first example of a monad one encounters in Haskell. Intuitively, the type Maybe $A$ represents values that behave like $A$ extended with the option of being "nothing".

- Let $F : \mathsf{FinSet} \to \mathsf{FinSet}$ be a functor:
  - Action on objects: $A \mapsto A \uplus \{\star\}$
  - Action on morphisms $f : A \to B$:

$$F(f) = a \mapsto \begin{cases} \mathsf{inl}\ f(x) & \text{if } a = \mathsf{inl}\ x \\ \mathsf{inr}\ \star & \text{if } a = \mathsf{inr}\ \star \end{cases}$$

- The unit $\eta : \mathrm{id}_{\mathsf{FinSet}} \Rightarrow F$ has components that straightforwardly inject $A$ into $F(A)$:

$$\begin{aligned} \eta_A &: \mathrm{id}_{\mathsf{FinSet}}(A) \to F(A) \\ &= x \mapsto \mathsf{inl}\ x \end{aligned}$$

- The multiplication $\mu : F^2 \to F$ merges the two nested distinguished elements $\star$ into a single one and flattens the nesting.

### 16.2.2 Powerset monad

The *powerset monad* is defined by

- The *powerset functor* $\wp : \mathsf{FinSet} \to \mathsf{FinSet}$ that sends each finite set $A$ to the set of all subsets $\{A_i \mid A_i \subseteq A\}$ is a monad.
- The unit $\eta : \mathrm{id}_{\mathsf{FinSet}} \Rightarrow \wp$ has components that map sets to the singleton set containing them:

$$\begin{aligned} \eta_A &: A \to \wp(A) \\ &= \{A\} \end{aligned}$$

- The multiplication $\mu : \wp^2 \Rightarrow \wp$ has components that "unfold the double powerset":

$$\begin{aligned}\mu_A : \wp(\wp(A)) &\Rightarrow \wp(A) \\ = \mathcal{S} &\mapsto \bigcup_{A \in \mathcal{S}} A\end{aligned}$$

## 16.3 Monads from adjunctions

A surprising fact: all adjunctions form a monad, and all monads come from an adjunction pair!

Let's see first how every adjunction forms a monad:

**Proposition 16.3.1.** Consider functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ where $F \dashv G$. Then, there is a unit to the adjunction $\eta : \mathrm{id}_C \Rightarrow G \circ F$ and co-unit $\nu : F \circ G \Rightarrow \mathrm{id}_D$. Then, there is a monad $(T, \eta, \mu)$:

- Endofunctor $T$ given by the roundtrip $T = G \circ F$
- Unit given by the unit of the adjunction
- Multiplication $\mu : G \circ F \circ G \circ F \Rightarrow G \circ F$ given by $\mu = G \circ \epsilon \circ F$.

### 16.3.1 Maybe monad from a free/forgetful adjunction

- Consider the categories FinSet and FinSet$^*$.
- There is a functor forget : FinSet$^*$ $\to$ FinSet called the *forgetful functor* that "forgets about the distinguished element".
- There is a functor free : FinSet $\to$ FinSet$^*$ that adds on a distinguished element
- There is an adjunction free $\dashv$ forgetful.
- The round trip free $\circ$ forgetful is exactly the monad $F$ we built earlier for the maybe monad.

### 16.3.2 List monad

- List monad: Free/forgetful adjunction between sets and monoids. Unit is singleton list construction; join is concatenation.

Example: the free $\dashv$ forgetful adjunction between pointed sets and ordinary sets is the `Maybe` monad.

# 17
# *Modeling effects with syntax*

Monads have found extensive application in computer science and langauge ideas for dealing with effects. The previous chapter showed how to interpret a monadic type, but it was not exactly clear how this captures the notion of an *effectful* computation. Here our aim is to make this clear by elucidating the connection between a monad and its *Klesli category*, which can be thought of as a syntactic representation of the effects being modeled. For any particular monad $T$ on a category $\mathcal{C}$, the monad will come about as an adjunction between $\mathcal{C}$ and its Klesili category.

We can start by characterizing effects using tiny languages that capture only the effectful fragments of computation. For example, we can represent the computations taking place under the finite probability monad Dist using the following tiny grammar:

$$e_{\mathsf{Dist}} ::= \mathsf{pure}(v) \mid \mathsf{if}\ \mathsf{flip}(p)\ e_1\ e_2$$

The syntax is extremely pared down: it does not have any features at all except for the ability to refer to values $v$, or branch on a random Boolean that is true with probability $p$. The set of all terms that can be formed out of this grammar is parameterized by the collection of values $v$: for each set of values $\mathcal{V}$ we denote the set of all terms from $e_{\mathsf{Dist}}$ using values $\mathcal{V}$ as $\mathsf{Exp}_{\mathsf{Dist}}(\mathcal{V})$.

As usual, given a language it is useful to characterize its equivalences. This is a design question: you, as the designer of the effect, must decide which effectful programs ought to behave the same.

Some equations are quite natural from this perspective:

$$\frac{}{\mathsf{if}\ \mathsf{flip}(p)\ e\ e \equiv e}\ (\textsc{Drop})$$

$$\frac{}{\mathsf{if}\ \mathsf{flip}(p)\ e_1\ e_2 \equiv \mathsf{if}\ \mathsf{flip}(1-p)\ e_2\ e_1}\ (\textsc{Swap})$$

$$\frac{}{\mathsf{if}\ \mathsf{flip}(1)\ e_1\ e_2 \equiv e_1}\ (\textsc{Then}) \qquad \frac{}{\mathsf{if}\ \mathsf{flip}(0)\ e_1\ e_2 \equiv e_2}\ (\textsc{Else})$$

The above equations capture natural reasoning principles. But, is this *all* the necessary equations that characterize? Here is an example of two tiny effectful programs that ought to be equivalent but aren't yet:

$$\mathsf{if}\ \mathsf{flip}(1/2)\ (\mathsf{if}\ \mathsf{flip}(1/3)\ (\mathsf{pure}\ \top)\ (\mathsf{pure}\ \bot))\ (\mathsf{pure}\ \bot) \stackrel{?}{\equiv} \mathsf{if}\ \mathsf{flip}(1/6)\ (\mathsf{pure}\ \top)\ (\mathsf{pure}\ \bot)$$

This motivates one more equation that captures all the equivalences for this effect:

$$\frac{p_1 + p_2 + p_3 = 1 \qquad p_{12} = p_1 + p_2 \qquad p_{23} = p_2 + p_3}{\mathsf{if}\ \mathsf{flip}(p_1 + p_2)\ (\mathsf{if}\ \mathsf{flip}(p_1/p_{12})\ e_1\ e_2)\ e_3 \equiv \mathsf{if}\ \mathsf{flip}(p_1)\ e_1\ (\mathsf{if}\ \mathsf{flip}(p_2/p_{23})\ e_2\ e_3)}\ (\textsc{Assoc})$$

Now we have a tiny language that captures *only* the effect of allocating and branching on random quantities. Using this tiny language, we can bolt on sampling as a monad in the category of sets Set. Recall that the set $\mathsf{Exp}_{\mathsf{Dist}}(\mathcal{V})$ of probabilistic computations with values in $\mathcal{V}$ is parameterized by a set $\mathcal{V}$. This can be used to define an endofunctor $D : \mathsf{Set} \to \mathsf{Set}$:

$$\begin{aligned} &D : \mathsf{Set} \to \mathsf{Set} \\ &D(\mathcal{V}) = \mathsf{Exp}_{\mathsf{Dist}}(\mathcal{V})/_{\equiv} \\ &D(f : \mathcal{V} \to \mathcal{V}') = e \mapsto e[f(v)/v] \end{aligned}$$

The action of this functor on morphisms sends a function $f$ to the function that uses $f$ to substitute each value $v$ in a given computation $e$ with $f(v)$.

This endofunctor $D$ forms a monad. The unit $\eta$ is defined by the coercion from values to computations:

$$\begin{aligned} &\eta_{\mathcal{V}} : \mathcal{V} \to D(\mathcal{V}) \\ &\eta_{\mathcal{V}}(v) = \mathsf{pure}(v) \end{aligned}$$

The multiplication is defined by *substitution*: given a computation $e$ that produces computations, the multiplication $\mu(e)$ is "flattening" of

the syntax tree of $e$:

$$\begin{aligned}
&\mu_{\mathcal{V}} : D(D(\mathcal{V})) \to D(\mathcal{V}) \\
&\mu_{\mathcal{V}}(\mathsf{pure}(v)) = v \\
&\mu_{\mathcal{V}}(\mathsf{if\ flip}(p)\ e_1\ e_2) = \mathsf{if\ flip}(p)\ \mu_{\mathcal{V}}(e_1)\ \mu_{\mathcal{V}}(e_2)
\end{aligned}$$

Note in the first case that the "value" $v$ is an element of $D(\mathcal{V})$. Thinking of computations as abstract syntax trees, this multiplication operation flattens a tree of trees into an ordinary tree. One can check that the monad laws are satisfied; they correspond to properties one might expect to hold of tree flattening.

The finite probability distribution monad $\mathcal{D} : \mathsf{Set} \to \mathsf{Set}$ sends sets $X$ to the set of finitely-supported functions $\mu : X \to [0,1]$ such that $\sum_{x \in X} \mu(x) = 1$.

As more convenient notation, we can describe a probability distribution on $X$ as a *formal series*: a distribution $\mu : X \to [0,1]$ can be written as an expression $c_1[x_1] + \cdots + c_n[x_n]$ where $c_i \in [0,1]$ are real-valued numbers satisfying $\sum_i c_i = 1$, and each $[x_i]$ is a *formal parameter* (i.e., a variable). There is a natural equivalence relation on formal series given by combining the coefficients of formal parameters:

$$c_1[x_1] + \cdots + c_i[x_i] + c_i'[x_i] + \cdots c_n[x_n] \equiv c_1[x_1] + \cdots + (c_i + c_i')[x_i] + \cdots c_n[x_n]$$

The additional equations of formal series are the natural equivalences given by the commutativity and associativity of addition and multiplication.

Now we can give the unit and multiplication natural transformations for $\mathcal{D}$:

- The unit $\eta : \mathrm{id}_{\mathsf{Set}} \Rightarrow \mathcal{D}$ has components $\eta_X : X \to \mathcal{D}(X)$ that place all the probability mass on a single point: $\eta_X(x) = 1[x]$.
- Multiplication $\mu : \mathcal{D}^2 \Rightarrow \mathcal{D}$ merges the probability distributions in the following way:

$$\begin{aligned}
&\mu_X : \mathcal{D}\mathcal{D}(X) \to \mathcal{D}(X) \\
&\quad = (c_1[c_{11}[x_1] + \cdots + c_{n1}x[n]] + \cdots + c_n[c_{1n}[x_1] + \cdots + c_{nn}x[n]]) \\
&\qquad \mapsto (c_1(c_{11}[x_1] + \cdots + c_{n1}x[n]) + \cdots + c_n(c_{1n}[x_1] + \cdots + c_{nn}x[n]))
\end{aligned}$$

**Proposition 17.0.1.** The monads $D$ and $\mathcal{D}$ are isomorphic; that is, there is a natural isomorphism $D \cong \mathcal{D}$ that commutes with the unit and multiplication operations of $D$ and $\mathcal{D}$ respectively.

Outline:

- From what just happened to "Kleisli category is syntactic description of a monad".
  - Taking stock of what happened with $D$ and $\mathcal{D}$: starting from a syntactic description of probability, we recovered finitely-supported PMFs.
  - Plotkin+Power's idea, inspired by Moggi's proposal to use monads: this is a recipe for modelling effects more generally: start from a syntactic description, add equations one expects should hold, and try to see what mathematical objects come out (c.f. algebraic effects)
  - Examples:
    * Failure: $e ::= v \mid \mathsf{fail}$. This recovers the Maybe monad.
    * Nondeterminism: $e ::= v \mid \mathsf{amb}(e_1, e_2)$. This recovers the finite nonempty powerset monad.
    * Logging: $e ::= v \mid \mathsf{log}(s); e$ (where $s$ is drawn from some set of strings). This recovers the writer monad.
    * Global state (one Boolean): $e ::= v \mid x \leftarrow \mathsf{get}_b; e(x) \mid \mathsf{put}_b; e$. This recovers the state monad, if you work hard to find the right equations [1].

[1] Gordon Plotkin and John Power. Notions of computation determine monads. In *International Conference on Foundations of Software Science and Computation Structures*, pages 342–356. Springer, 2002

- This approach of starting from syntax is essentially universal algebra. One nice consequence is that we can use results from universal algebra to study effects. We look at one example now: the idea of a Kleisli category of a monad.
  - Definition of Kleisli category. Morphisms $\mathcal{V} \to T\mathcal{W}$ are like substitutions, showing how to transform each value $\mathcal{V}$ into a term $T\mathcal{W}$.
  - Example: for Dist, Kleisli morphisms are transition matrices. In the continuous case, Kleisli morphisms are Markov kernels.
  - Example: for Maybe, Kleisli morphisms are partial functions.
- Given a monad, find its Kleisli category and corresponding adjunction.
- The Kleisli monad models call-by-value computation. Universal algebra shows how to model call-by-name: Eilenberg-Moore, from the perspective of "models of tiny language".

# 18
# Realizability

This chapter was written by Mingtong Lin. It is heavily based on [4]. The reader is only assumed to have acquaintance with previous chapters.

Realizability originated from Kleene [18] as a semantics for Heyting arithmetic. In this article, Kleene motivates the concept by calling logical statements "incomplete communication" and "partial judgment" from the perspective of constructivists and intuitionists. For instance, a proposition "$A$ implies $B$", when interpreted intuitionistically, is an incomplete communication of another statement which gives a general procedure that: when receiving the information that completes $A$, it finds the information that completes $B$. In other words, Kleene is seeking the computational content of logical statements. To achieve this, Kleene presents a way of encoding the items of information with partial recursive functions[1]. Then, we may say that certain recursive function *realizes* a statement, as a kind of intuitionistic truth notion of the statement.

[1] In Gödel numbers, which is what you will get if you search for "realizability" on Wikipedia.

In the context of PL research, the more relevant way of viewing realizability is motivated by a generalized question: "given a mathematical structure, what should its implementation look like?" [4] Such interpretation of realizability is of great interest in the area of compiler correctness [5, 29], since it naturally establishes a relation between implementations and specifications. Also, instead of partial recursive functions in Gödel numbers, we will use a generalization of the $\lambda$-calculus that allows applications to be partially defined for our realizers, namely the *partial combinatory algebra*, or PCA. But before defining PCAs, let's first introduce an even more generalized structure underlying them, namely the *partial applicative structure*, or PAS.

The discussion on PAS, PCA, and combinatory completeness is largely taken from [15].

**Definition 68: Partial applicative structure (PAS)**

A *partial applicative structure* $(A, \cdot)$ is a set $A$ equipped with a *partial* binary operation:

$$\cdot : A \times A \rightharpoonup A$$

For $f, a \in A$, if $(f, a) \in \mathsf{dom}(\cdot)$, we write $f \cdot a \downarrow$, otherwise $f \cdot a \uparrow$. Sometimes people may omit $\cdot$ and just write $fa$ for $f \cdot a$. Application associates to the left, just like that in $\lambda$-calculus, so $abc$ stands for $(ab)c$.

The definition of PAS reveals a key feature of combinatorial structures. Consider $f, a \in A$ and the application $f \cdot a$, the element $f$ acts as a function, while the element $a$ acts as an argument. In other words, the program (functions) and the data (arguments) live in the same realm and are represented in the same way.

Using our intuition from $\lambda$-calculus, a PAS defines an operation analogous to $\lambda$-applications, but what about $\lambda$-abstractions? To talk about them, we first need to incorporate variables.

**Definition 69: Terms (PAS)**

Let $V = \{x_0, x_1, \ldots\}$ be a countable set of fresh variables, the set of *terms* over $A$, written $T(A)$, is the least set satisfying:

- if $x \in V$, then $x \in T(A)$;
- if $a \in A$, then $a \in T(A)$;
- if $t_1, t_2 \in T(A)$, then $t_1 \cdot t_2 \in T(A)$.

We will use the familiar notations from the $\lambda$-calculus for free variables and substitutions.

**Notation 18.0.1.** $FV(t)$ denotes the set of free variables in the term $t \in T(A)$. A term $t$ is *closed* if $FV(t) = \emptyset$.

**Notation 18.0.2.** $t[a/x]$ stands for "substituting $a$ for $x$ in term $t$", exactly as in the $\lambda$-calculus.

Unfortunately, we are still not yet able to get something like $\lambda$-abstraction, but what exactly is going wrong?

**Example 18.0.3.** Let $A \coloneqq \{1, 2\}$ and $\cdot : A \times A \to A$ be natural number multiplication. Of course, we are able to write square of elements in $A$ in our PAS $(A, \cdot)$:

$$1 \cdot 1 \downarrow$$
$$2 \cdot 2 \uparrow$$

Now, suppose we want to abstract over the squaring operation, immediately we try to write a term $t \in T(A)$ such that

$$t = x \cdot x$$

Next, we need an element $f \in A$ in our PAS that behaves like $t$, so that we can actually apply it for the square of an arbitrary $a$:

$$f \cdot a = t[a/x] = a \cdot a$$

However, by inspection, none of the elements in $A$ suffices.

The above example reveals the discrepancy that, when given a term $t \in T(A)$, we might fail to find an element that behaves like it in $A$. In order to fill the gap, we shall first formalize what "behaves like" generally means for partial functions.

Unfortunately, the notation for Kleene equality collides with that for categorical equivalence. We will reserve $\simeq$ for Kleene equality exclusively throughout this chapter.

**Definition 70: Kleene equality**

*Kleene equality* establishes the quality between partial functions. Given an argument, either both functions are undefined, or both are defined and their values on that arguments agree. We write $f \simeq g$ for Kleene equality.

Then, we are able to state what it means to have an element $a \in A$ that behaves like a specific term $t$.

**Definition 71: Representability**

Let $t \in T(A)$ be a term with $FV(t) \subseteq \{x_0, \ldots, x_n\}$ and $f \in A$ an element of A, we say that $f$ *represents* $t$ if for all $a_0, \ldots, a_n \in A$:

- $f \cdot a_0 \cdot \ldots \cdot a_{n-1} \downarrow$
- $t[a_0/x_0, \ldots, a_n/x_n] \simeq f \cdot a_0 \cdot \ldots \cdot a_n$

In words, $f$ is a total function in the first $n$ arguments, and as an $n+1$-ary partial function, $f$ is equal to $t$.

When a term $t$ is representable (i.e., represented by one or more elements in $A$), we have the internalization of $t$ in $A$. In other words, if all of the terms $t \in T(A)$ are representable, we can always find at least one element that encodes our desired abstraction (e.g., the squaring operation in Theorem 18.0.3).

**Definition 72: Combinatory completeness**

A PAS $(A, \cdot)$ is *combinatorially complete* if every term $t \in T(A)$ can be represented by some $a \in A$.

**Remark 18.0.4.** From the perspective of recursion theory, the definition of PAS abstracts the concept of the *enumeration theorem* [19][2]: let $A$ be $\mathbb{N}$ (the Gödel numbers) and define $\cdot$ to be the universal function $\Phi$, we have $f \cdot a \coloneqq \Phi(f, a)$.

[2] Note that a PAS is not necessarily *universal*, therefore we are not referring to the *utm theorem*.

Conversely, combinatory completeness corresponds to the *smn theorem*. Thus, an equivalent definition of combinatory completeness is: for a term $t \in T(A)$ and a variable $x$, there is a term $t'$ such that $x \notin FV(t')$ and $t' \downarrow$, satisfying $t' \cdot a \simeq t[a/x]$ for all $a \in A$.

**Definition 73: Partial combinatory algebra (PCA)**

A PAS $(A, \cdot)$ is called a *partial combinatory algebra* if it is combinatory complete.

The usual way of defining PCAs is to say that a PCA is a PAS with $\mathtt{K}, \mathtt{S} \in A$. This simplification is due to Curry [10], known as the *bracket abstraction* algorithm. For more related details in the context of PCAs, see [25].

**Proposition 18.0.5.** A PAS $(A, \cdot)$ is combinatory complete if and only if there exists $\mathtt{K}, \mathtt{S} \in A$ such that for all $a, b, c \in A$:

- $\mathtt{K} \cdot a \cdot b = a$, and
- $\mathtt{S} \cdot a \cdot b \cdot c \simeq (a \cdot b) \cdot (b \cdot c)$ and $\mathtt{S} \cdot a \cdot b \downarrow$.

**Fact 18.0.6.** The first Kleene algebra $\mathcal{K}_1$, combinatory logic $\mathbb{CL}$, and untyped $\lambda$-calculus $\Lambda$ are all PCAs.

Availing us with the language of PCAs, we can now return to the discussion of realizability.

**Definition 74: Assembly**

An *assembly* $X$ over a PCA $(A, \cdot)$ is a pair $(|X|, \Vdash_X)$ where $|X|$ is the underlying set and $\Vdash_X$ is a relation between $A$ and $|X|$, such that for every $x \in |X|$, there exists an $r \in A$ and $r \Vdash_X x$. We read $r \Vdash_X x$ as "$r$ realizes $x$".

Tthe assembly requires every $x \in |X|$ to be realized by some $r \in A$, but there is no constraint about their correspondence. Therefore, an $x \in |X|$ may be realized by one or more realizers in $A$, and an $r \in A$ may also realize one or more elements in $|X|$.

John always has a comment that "when you ask Amal about realizability, she will likely tell you it's just *binary logical relations*." Indeed, an equivalent formulation is to view the realizability relation as the subset: $\Vdash_X \subseteq A_\tau \times |X|$. Note that we just sneaked in types for our PCA (we intend to not discuss typed-PCAs [23, 24] due to the lack of space). Then, we can write the realizability relation as a binary $\mathcal{V}$-relation:

$$(r, x) \in \mathcal{V}[\![\tau]\!]$$

**Definition 75: Tracker**

For two assemblies $(|X|, \Vdash_X)$ and $(|Y|, \Vdash_Y)$ over the same PCA $(A, \cdot)$, and a function $f : |X| \to |Y|$, a *tracker* for $f$ is an element $t \in A$ such that for every $a \Vdash_X x$, we have $t \cdot a \downarrow$ and $t \cdot a \Vdash_Y f(x)$. We say that $f$ is *realized* (or *tracked*) by $t$.

**Definition 76: Assembly map**

An *assembly map* $X \to Y$ between assemblies $X$ and $Y$ over a PCA

$\mathbb{A}$ is the pair $(f, t)$ where $f : |X| \to |Y|$ and $t \in A$ tracks $f$.

**Definition 77: Assembly category (Asm)**

Assemblies over a PCA $\mathbb{A} = (A, \cdot)$ form a category $\mathsf{Asm}(\mathbb{A})$ where:

- objects are assemblies $X = (|X|, \Vdash_X)$ over $\mathbb{A}$;
- morphisms are assembly maps;
- for two morphisms $(f, t)$ and $(g, t')$ where $f : |X| \to |Y|$ and $g : |Y| \to |Z|$, composition is defined as

$$(g, t') \circ (f, t) = (g \circ f, \mathtt{B} \cdot t' \cdot t)$$

  where $\mathtt{B} \in A := \mathtt{S} \cdot (\mathtt{K} \cdot \mathtt{S}) \cdot \mathtt{K}$;
- the identity morphism is given by the pair $(\mathrm{id}, \mathtt{I})$, where id is the identity function and $\mathtt{I} \in A := \mathtt{S} \cdot \mathtt{K} \cdot \mathtt{K}$.

The combinator B used in the above definition is not in the standard SKI basis of combinatory logic, but is a handy combinator for composition:

$$\mathtt{B} \cdot g \cdot f \cdot x \simeq g \cdot (f \cdot x)$$

**Example 18.0.7** (Unit assembly $\mathbb{1}$)**.** The *terminal* assembly $\mathbb{1}$ over $\mathbb{A}$ is $(\{\star\}, \Vdash_{\mathbb{1}})$, where for all $r \in A$, we have the trivial realizability relation $r \Vdash_{\mathbb{1}} \star$. The unit assembly $\mathbb{1}$ is the terminal object in $\mathsf{Asm}(\mathbb{A})$.

Previously, we said that a realizer $a \in A$ may realize more than one elements in $|X|$, which is not ideal for some cases. Therefore, we introduce a restricted version of an assembly in which no two elements share a common realizer.

**Definition 78: Modest set**

A *modest* assembly $X$, also called a *modest set*, is an assembly in which elements do not share realizers. That is, for any two $x, y \in |X|$, if an $r \in A$ realizes both $x$ and $y$, we have $x = y$.

**Fact 18.0.8.** Modest sets over a PCA $\mathbb{A}$ form a full subcategory $\mathsf{Mod}(\mathbb{A})$ of $\mathsf{Asm}(\mathbb{A})$.

**Proposition 18.0.9.** Both $\mathsf{Asm}(\mathbb{A})$ and $\mathsf{Mod}(\mathbb{A})$ are cartesian-closed. The proofs is left as an exercise for the readers ([4, Chapter 3.6] has some of the proofs).

**Definition 79: Partial equivalence relation (PER)**

Given a modest set $X$, we may induce a *partial equivalence relation* $\approx_X$ on $A$ which relates $r$ and $r'$ when they realize the same element $x \in |X|$:

$$r \approx_X r' \iff \exists x \in |X|.\, r \Vdash_X x \wedge r' \Vdash_X x$$

**Definition 80: Total realizer**

A realizer $r \in A$ is *total* if $r \approx_X r$. The set of total realizers is denoted by $\|X\| = \{r \in A \mid r \approx_X r\}$. Each $r \in \|X\|$ determines the equivalence class $[r]_{\approx_X} = \{a \in A \mid r \approx_X a\}$.

The name for total realizers is confusing–it is not the converse of "partial" in the computational sense. Replacing the other realizer $r'$ with another $r$ in the above definition reduces to:

$$r \approx_X r \iff \exists x \in |X|.\, r \Vdash_X x$$

which amounts to the fact that $r$ realizes something in $|X|$, and therefore is not some syntactic "junk" in $A$. A total realizer $r$ can definitely realize a partial function such as integer division. However, a realizer for $n/0$ is not total, since $n/0$ is absent in $|X|$.

**Example 18.0.10.** Let us consider the modest assembly $B = (\{\top, \bot\}, \Vdash_B)$ over some PCA $\mathbb{A}$, and the induced PER $\approx_B$. The set of total realizers $\|B\|$ contains those who realize something in $|B|$, namely for all $r \in |B|$, ether $r \Vdash_B \top$ or $r \Vdash_B \bot$. Then, we have the following two equivalence classes:

$$[r_\top]_{\approx_B} = \{r \in A \mid r_\top \approx_B r\} \qquad \text{where } r_\top \Vdash_B \top$$
$$[r_\bot]_{\approx_B} = \{r \in A \mid r_\bot \approx_B r\} \qquad \text{where } r_\bot \Vdash_B \bot$$

Note that since $B$ is modest, $[r_\top]_{\approx_B}$ and $[r_\bot]_{\approx_B}$ are disjoint.

**Definition 81: Extensional realizer**

Given two PERs $\approx_X$ and $\approx_Y$, an *extensional realizer* between them is a $p \in A$ such that, for all $r, r' \in A$, if $r \approx_X r'$ then $p\ r \downarrow$, $p\ r' \downarrow$, and $p\ r \approx_Y p\ r'$. Extensional realizers $p$ and $p'$ are *equivalent* when $r \approx_X r'$ implies $p\ r \approx_Y p'\ r'$.

**Remark 18.0.11.** An extensional realizer therefore "sends related inputs to related outputs", where the relation is induced by the modest set.

**Example 18.0.12.** Let us consider an extensional realizer between $B$ and itself. We have already discussed the set of total realizers and equivalence classes of $B$ above.

An example extensional realizer $p$ sends the input to the same equivalence class for the output: if the input is in $[r_\top]_{\approx_B}$, then the output is also in $[r_\bot]_{\approx_B}$, and vice versa. If our $\mathbb{A}$ is $\lambda$-calculus, we have $p\ r\ \downarrow$ and $p\ r'\ \downarrow$ for free since application of two $\lambda$ terms is always defined. One qualified $p$ is the identity $\lambda x.\, x$, and the extensionality condition is trivially true.

**Notation 18.0.13.** In stead of writing $\approx_X$ for the PER induced by the modest set $X$, it is common to denote it by $X$ directly. This convention is slightly ambiguous, but is somewhat reasonable since the correspondence is one-to-one.

**Remark 18.0.14.** Note that in the discussion of extensional realizers, we did not mention what in $|X|$ is realized by $p$. This is fundamentally different from what we did when defining trackers, where we said $f : |X| \to |Y|$ is realized by $t$. In fact, the underlying set $|X|$ only appeared at the very beginning for defining the PER itself throughout our discussion of PERs.

This marks a fundamental perspective shift. In the assembly view, we relate "implementations" (the realizers) to "specifications" (the mathematical objects in $|X|$). In the PER view, we relate implementations to implementations, by quotienting them through a PER.

**Definition 82: PER category** $\mathsf{Per}(\mathbb{A})$

PERs and equivalence classes of extensional realizers form a category $\mathsf{Per}(\mathbb{A})$ where:

- objects are PERs $X$ on $\mathbb{A}$;
- morphisms $X \to Y$ are equivalence classes of extensional realizers $[p] : X \to Y$;
- for two morphisms $[p] : X \to Y$ and $[q] : Y \to Z$, composition is defined as
$$[q] \circ [p] = [q \circ p]$$
where $[q \circ p]$ is defined as $[\mathtt{B} \cdot q \cdot p]$.
- the identity morphism is $\mathtt{I}$.

**Proposition 18.0.15.** The categories $\mathsf{Mod}(\mathbb{A})$ and $\mathsf{Per}(\mathbb{A})$ are equivalent.

For details of the equivalence proof, consult [4, Chapter 3.3.3]. There is also an Agda mechanization of this proof in [7].

**Definition 83: Uniform assembly**

Let $X$ be a set and $\mathbb{A}$ a PCA, the *uniform assembly* (or *constant assembly*)

$$\nabla_{\mathbb{A}} X = (X, \Vdash_{\nabla_{\mathbb{A}} X})$$

is the assembly in which the elements are "uniformly realized". That is, for all $r \in A$ and $x \in X$:

$$r \Vdash_{\nabla_{\mathbb{A}} X} x$$

**Notation 18.0.16.** When not ambiguous, we write $\nabla X$ for $\nabla_{\mathbb{A}} X$.

**Fact 18.0.17.** Uniform assemblies over a PCA $\mathbb{A}$ form a full subcategory of $\mathsf{Asm}(\mathbb{A})$.

**Proposition 18.0.18.** The unit assembly $\mathbb{1}$ is uniform.

In a uniform assembly $\nabla X$, every $x \in X$ is realized by every $r \in A$, meaning that the realizability condition is trivial and the realizers provide *no information* about the elements in $X$. Think about the trackers: normally, a tracker $t$ of $f$ needs to carefully maintain the condition that when applied to an argument that realizes a valid input $x$ of $f$, it should realize the result $f(x)$. But within uniform assemblies, the condition collapses to only require $t \cdot a \downarrow$, since it is guaranteed to realize $f(x)$. In this sense, $\nabla X$ is devoid of any computational content.

**Proposition 18.0.19.** Given an assembly $X$ and a uniform assembly $\nabla Y$ over a PCA $\mathbb{A}$, every function $|X| \to Y$ is an assembly map $X \to \nabla Y$.

*Proof.* For a function $f : |X| \to Y$, we pair it with a trivial tracker $\mathtt{I}$, where for all $a \in A$, $\mathtt{I} \cdot a$ is always defined. □

**Definition 84: Uniform assembly functor ($\nabla$)**

Let $\nabla : \mathsf{Set} \to \mathsf{Asm}(\mathbb{A})$ be the functor defined as follows:

- on objects, it sends a set $X$ to the uniform assembly $\nabla X$;
- on $\mathsf{Set}$ morphism $f : X \to Y$, $\nabla f = f$.

**Proposition 18.0.20.** The functor $\nabla$ is fully faithful.

**Proposition 18.0.21.** An assembly $X$ is uniform if and only if it is isomorphic to some $\nabla Y$. Therefore, the functor $\nabla_{\mathbb{A}}$ is essentially surjective onto the full subcategory of uniform assemblies over a PCA $\mathbb{A}$.

**Corollary 18.0.22.** The category of uniform assemblies over a PCA $\mathbb{A}$ is equivalent to Set.

**Definition 85: Forgetful functor ($\Gamma$)**

Let $\Gamma : \mathsf{Asm}(\mathbb{A}) \to \mathsf{Set}$ be the forgetful functor defined as follows:

- on objects, $\Gamma X = |X|$;
- on assembly map $(f, t) : X \to Y$, $\Gamma(f, t) = f$.

**Proposition 18.0.23.** The the forgetful functor $\Gamma$ is left adjoint to the uniform assembly functor $\nabla$.

*Proof.* Recall the natural transformations $\eta : \mathrm{id}_{\mathsf{Asm}(\mathbb{A})} \Rightarrow \nabla\Gamma$ and $\varepsilon : \Gamma\nabla \Rightarrow \mathrm{id}_{\mathsf{Set}}$ are the unit and counit of the adjunction.

- For any $X \in \mathsf{Asm}(\mathbb{A})$, the component of the unit $\eta_X : X \to \nabla\Gamma X$ is defined by the assembly map $(\mathrm{id}_{|X|}, \mathtt{I})$, where $\mathtt{I}$ is a trivial tracker.
- For any $S \in \mathsf{Set}$, the component of the counit $\varepsilon_S : \Gamma\nabla S \to S$ is given by the identity function $\mathrm{id}_S$.
- Show $\varepsilon\Gamma \circ \Gamma\eta = \mathrm{id}_\Gamma$: for an assembly $X \in \mathsf{Asm}(\mathbb{A})$,

$$(\varepsilon\Gamma \circ \Gamma\eta)_X = \varepsilon_{\Gamma X} \circ \Gamma(\eta_X) = \mathrm{id}_{|X|} \circ \mathrm{id}_{|X|} = \mathrm{id}_{|X|}$$

- Show $\nabla\varepsilon \circ \eta\nabla = \mathrm{id}_\nabla$: for a set $S \in \mathsf{Set}$,

$$(\nabla\varepsilon \circ \eta\nabla)_S = \nabla(\varepsilon_S) \circ \eta_{\nabla S} = (\mathrm{id}_S, \mathtt{I}) \circ (\mathrm{id}_S, \mathtt{I}) = (\mathrm{id}_S, \mathtt{I})$$

□

# 19
# *Categorical Noninterference*

This chapter was written by Bex Golovanov.

Noninterference is a property in formal security to establish the safety of private information with respect to public information. Essentially, a program/language/protocol satisfies noninterference if publicly observable operations do not depend on private information; behavior using private data has no distinguishable effect on public data.

This property is very strong, and almost always too strong. One might be motivated to research different models for reasoning about noninterference, with the hope of obtaining a weakened (yet formal) version. As was mentioned at the beginning of this semester, the answer to understanding a problem is rarely to categorify it. But why not give it a try?

In this chapter, we explore Sterling and Harper's approach to noninterference via a sheaf semantics in their paper titled *Sheaf Semantics of Termination-Insensitive Noninterference*[1].

[1] We do not cover anything pertaining to termination, sensitivity or lack thereof. Our exploration is also localized to pages 6 and 7 of the paper, as well as one observation on page 12 which we prove at the end of this chapter.

The category theory in this chapter is primarily sheaf-theoretic; the content is almost entirely self-contained. Sources and supplementary reading can be found in these course notes, §I and §II of *Sheaves in Geometry and Logic* by MacLane and Moerdijk, *Category Theory in Context* by Emily Riehl, and of course ncatlab.org.

## *19.1 Information Flow Basics*

This section introduces some terminology and basic ideas around formal reasoning for information flow. We start with the notion of a *security lattice*. The simplest form is the set $\mathcal{L} = \{L, H\}$ with $L \leq H$, where $L$ represents low-security (or public) data, and $H$ represents high-security (or secret) data.

One way to think about this lattice with respect to programming is by "labelling" variables with some $\ell \in \mathcal{L}$, e.g. $x_L$ represents that $x$ is a low-security variable. This labelling can be implemented with *security*

*types*. For example, if we want to talk about $x_L$ of type Int with this framework, we would say $x$ has type $Int_L$, where $Int_L$ represents public ints.

When working with information flow, we ideally want use of public (observable) data to not reveal anything about secret data. The strongest statement of this is requiring that public data never depends on secret data, usually referred to as noninterference.

**Definition 86: $\ell-$equivalence**

Let $L : \text{Var} \to \mathcal{L}$ be a *labelling* from variables $x$ to elements $\ell$ of the security lattice $\mathcal{L}$.

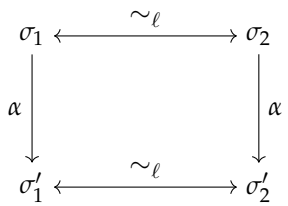
A state $\sigma : \text{Var} \to \text{Val}$ maps variables to values. Two states $\sigma, \sigma'$ are *$\ell$-equivalent*, denoted $\sigma \sim_\ell \sigma'$, when for all $\ell' \leq \ell$, if $L(x) = \ell'$, then $\sigma(x) = \sigma'(x)$. This requirement is equivalent to $L(x) \leq \ell$ for all $x$, which we can interpret as $x$ being "public with respect to $\ell$".

**Definition 87: Noninterference**

A program $\alpha$ satisfies *noninterference* if, for any states $\sigma_1, \sigma_2$ such that $\sigma_1 \sim_L \sigma_2$, and $\sigma_1 \xrightarrow{\alpha} \sigma_1'$, $\sigma_2 \xrightarrow{\alpha} \sigma_2'$, we have $\sigma_1' \sim_L \sigma_2'$. If initial states agree on all public values, the respective post-states should agree on all public values.

Let **Rel** be the category whose objects are sets and whose morphisms $A \sim_R B$ are relations $\sim_R \subseteq A \times B$. We can think of $\ell$-equivalence as a morphism in **Rel**, where $\sigma \sim_\ell \sigma'$ means $(\sigma, \sigma') \in \sim_\ell \subseteq \Sigma \times \Sigma'$ for $\Sigma, \Sigma'$ sets of states. Since functions $f : A \to B$ can be represented as graphs $f \subseteq A \times B$, where $(a, b) \in f$ if $f(a) = b$, we can consider functions to be relations as well, so $\alpha : \Sigma \to \Sigma'$ is also a morphism in **Rel**.

Then noninterference can be understood as the requirement that the following diagram commutes:




This property is also referred to as $\sigma_1$ and $\sigma_2$ being *indistinguishable* with respect to public values.

This indistinguishability property is equivalent to requiring any map from some type $\tau_H$ to type $\tau_L$ to be constant. One characterization of this definition is the following:

**Proposition 19.1.1.** For any closed function $\cdot \vdash f : \tau_H \to \tau_L$, there exists a closed $\cdot \vdash v : \tau_L$ such that $f \simeq \lambda\_.v$, meaning $f$ is observationally (extensionally) equivalent to a constant $v$ function.

## 19.2 *Categorical Background Potpourri*

In this section, we explain the medley of categorical concepts needed to understand Sterling and Harper's construction.

### 19.2.1 *Lattices and Logic*

**Definition 88: Lattices**

A lattice $\mathcal{L}$ is a partially ordered set which has all *meets* and *joins*.

- Two elements $l, k \in \mathcal{L}$ have a *meet* if there exists an $m \in \mathcal{L}$ such that $m$ is the greatest lower bound of $l$ and $k$. *Joins* are symmetrically least upper bounds.
- Having all meets and joins means for any pair of elements in the lattice, you can find a greatest lower bound as well as a least upper bound. This property can also be phrased as having all binary products and all binary coproducts.

A lattice with **0** and **1** (unique initial and terminal objects) has all finite limits and all finite colimits.

Intuitionistic propositional logic can be thought of as a *Heyting algebra* $H$, which is a lattice with **0** and **1**, and has for each pair $x, y$ an exponential object $y^x$, usually written as $x \Rightarrow y$. Disjunction and conjunction are joins and meets, respectively, with the initial object **0** and terminal object **1** as their respective identities. That is,

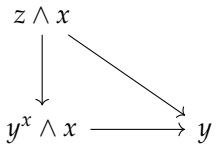
$$x \vee \mathbf{0} = x, \quad x \wedge \mathbf{1} = x$$

A *distributive lattice* is a lattice in which

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$

holds for all $x, y, z \in \mathcal{L}$. This identity also implies the dual, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$.

The exponential $x \Rightarrow y$ corresponds to implication. Writing out our usual diagram in the above language,




we can characterize $y^x$ or $x \Rightarrow y$ by the following:

$$z \wedge x \leq y \quad \iff \quad z \leq x \Rightarrow y$$

We can interpret $x \Rightarrow y$ as a least upper bound for $z$ such that $z \wedge x \leq y$. Negation in a Heyting algebra can be defined in the "implies absurdity" manner:

$$\neg x := x \Rightarrow \mathbf{0}$$

With our definition of $\Rightarrow$, this implies

$$y \leq \neg x \quad \iff \quad y \wedge x = \mathbf{0},$$

where we have $=$ instead of $\leq$ because $\mathbf{0}$ is initial.

An alternative (cursed) answer to the question "what is a topology?" : a topology is *just* something you define sheaves over, and sheaves are *just* something you define a cohomology on, so the question you should really be asking is, what is a cohomology?

**Definition 89: Topology**

A *topology* on a set $X$ can be defined as a collection $\mathcal{T}$ of subsets of X, called **open sets** and satisfying the following axioms:

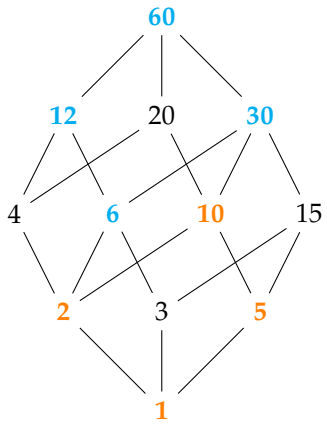
1. The empty set and $X$ itself belong to $\mathcal{T}$.
2. Any arbitrary union of members of $\mathcal{T}$ belongs to $\mathcal{T}$.
3. The intersection of any finite number of members of $\mathcal{T}$ belong to $\mathcal{T}$.

For a topological space $X$, the set of all open sets of $X$, notated $\mathcal{O}(X)$, forms a Heyting algebra. This implies we can model the intuitionistic propositional calculus with any topological space $X$, by looking at the associated set $\mathcal{O}(X)$.

**Definition 90: Special Subsets of Lattices**

- An *upper set* U on a poset $(\mathcal{P}, \leq)$ is upwards closed: if $x \in U$ and $x \leq z$, then $z \in U$.
- Symmetrically, a *lower set* U on $\mathcal{P}$ is downwards closed: if $x \in U$ and $z \leq x$, then $z \in U$.
- A *filter* $\mathcal{F}$ on $\mathcal{P}$ is an upper set with the additional condition of every finite subset of $\mathcal{F}$ having a lower bound in $\mathcal{F}$ (filters are closed under finite meets)

**Example 19.2.1.** Recalling that the natural numbers $\mathbb{N}$ form a poset ordered under divisibility, here is a Hasse diagram of the factors of 60, where the blue numbers are the elements of the upper set generated by 6 (and in this case the upper set is also a filter), and the orange are elements of the lower set generated by 10.




For an example of an upper set which is *not* a filter, consider the lattice of divisors of 12 ordered by division. The upper set defined by $U = \{4, 6, 12\}$ is not a filter, since the meet (in this case gcd) of 4 and 6 is 2, which is not contained in $U$. By contrast, $\{6, 12\}$ satisfies filter requirements

**Definition 91: Alexandroff Topology**

The *Alexandroff Topology* is a natural topology on the underlying set of a pre-ordered set, where the open sets are the upper sets.

(Foreshadowing): If we order filters $\mathcal{F}$ on a poset $\mathcal{P}$ by inclusion, we get a poset of the set of filters. This would imply there exists an Alexandroff topology to be defined here, with upper sets of the filters.

Let's check that this definition satisfies the axioms for open sets.

**Proposition 19.2.2.** Upper sets of a poset $\mathcal{P}$ satisfy the following axioms:

1. The empty set and $\mathcal{P}$ itself are upper sets.
2. Upper sets are closed under arbitrary unions.
3. Upper sets are closed under finite intersections.

*Proof.* .

1. $\emptyset$ as an upper set is vacuously true; there are no elements to check. For $\mathcal{P}$, suppose $x \in \mathcal{P}$ and $x \leq y$. Well $y \in \mathcal{P}$ because $\mathcal{P}$ is the entire set.

2. Let $\{U_i\}_{i \in I}$ be a collection of upper sets. We wish to show $\bigcup_{i \in I} U_i$ is an upper set. Suppose $x \in \bigcup_{i \in I} U_i$ and $x \leq y$. Then $x \in U_j$ for some $j \in I$. $U_j$ is an upper set and $x \leq y$, so $y \in U_j$. Then $y \in \bigcup_{i \in I} U_i$ so arbitrary union of upper sets is also an upper set.

3. Let $U, V$ be upper sets. We want to show $U \cap V$ is an upper set. Suppose $x \in U \cap V$ and $x \leq y$. Then $x \in U$ and $x \in V$. Since $U$ and $V$ are both upper sets and $x \leq y$, we have $y \in U$ and $y \in V$. Then $y \in U \cap V$! So upper sets are also closed under finite intersection.

□

### *19.2.2 Sheaves*

In the sheaf semantics chapter we defined a *coverage*, which was defined to be a relation $T$ between families of morphisms with common codomain $c$ and objects $c$ of $\mathcal{C}$. If a family $\{f_i : c_i \to c\}$ and object $c$ are in $T$, then $\{f_i : c_i \to c\}$ is a $T$-cover of $c$. Conditions for $T$ can be found in the aforementioned chapter. A general intuition for these families of morphisms is to think of them as a way to partition an object $c$ by images of $f_i$.

**Definition 92: What is a sheaf, actually?**

Let $\mathcal{C}$ be a category with a coverage $T$. For any object $A \in \mathcal{C}$, we have a T-cover $\{a_i : A_i \to A\}_{i \in I}$ indexed by some $I$. A presheaf $F$ on $\mathcal{C}$ is a sheaf if it satisfies two additional conditions for *any* T-cover:

1. *locality*: For any $f, g \in F(A)$, if $f \mid_{A_i} = g \mid_{A_i}$ for all $i \in I$, then $f = g$. The notation $f \mid_{A_i}$ is equivalent to applying $F(p_i)(f)$ where $p_i$ is the morphism $A_i \to A$.

   If two elements agree locally (on each part of the cover), they agree everywhere. We can also think of this as being able to partition any $f \in F(A)$ into the "sum of its parts", where the parts correspond to an arbitrary cover of $A$.

2. *gluing*: Let $f \in F(A)$. For any $i, j \in I$, we have $F(\pi_i)(f \mid_{A_i}) = F(\pi_j)(f \mid_{A_j})$ in $F(A_i \times_A A_j)$, where $\pi_i$ is the projection $A_i \times A_j \to A_i$, $\pi_j$ is the projection for $A_j$ and $A_i \times_A A_j$ is the pullback of $A$ along $f \mid_{A_i}$ and $f \mid_{A_j}$.

   If $\mathcal{C}$ is a preorder with meets, this simplifies to for any $x \in A_i \cap A_j$, we have $f \mid_{A_i} x = f \mid_{A_j} x$.

   - Consider the case where $F(A)$ maps to sets of functions. In this case, an equivalent definition to the above which may have more of a "gluing" connotation says that, for any $f_i : A_i \to A$ and $f_j : A_j \to A$ such that $f_i(a) = f_j(a)$ for $a \in A_i \cap A_j$, there exists a unique $f$ such that $f \mid_i = f_i$ and $f \mid_j = f_j$. That is, when we have two maps agreeing at their intersection, we can "glue" them together to get another map.

**Proposition 19.2.3.** Let $\mathcal{C}$ be a category with a coverage and $F : \mathcal{C}^{op} \to Set$. For any $A \in \mathcal{C}$, for any cover $\{a_i : A_i \to A\}$ of $A$, $F$ is a sheaf iff $F$ satisfies the equalizer diagram

$$F(A) \xrightarrow{e} \prod_{i \in I} F(A_i) \underset{q}{\overset{p}{\rightrightarrows}} \prod_{i,j \in I} F(A_i \cap A_j),$$

where $p(f_i) = f_i \mid_{A_i \cap A_j}$ and $q(f_i) = f_j \mid_{A_i \cap A_j}$.

Brief sketch on why the above is true:

**Lemma 19.2.4.** $e$ (or any equalizing map) is a monomorphism. Conversely, the morphism associated to the coequalizer is an epimorphism. These are not too hard to prove.

Can see Solution 19.4.4

- Since $F$ maps to **Set**, $e$ is also injective. We can think of $e$ as the map that partitions some $f \in F(A)$, i.e. $e(f) = \{f \mid_{A_i}\}_{i \in I}$. Then

agreeing locally, $f \mid_{A_i} = g \mid_{A_i}$, is equivalent to saying $e(f) = e(g)$. Injectivity of $e$ gives us $f = g$.

- The equalizer-ness of $F$ means $F(A)$ only consists of those $f$ where $p \circ e(f) = q \circ e(f)$. We are guaranteed $\left(f \mid_{A_i}\right) \mid_{A_i \cap A_j} = \left(f \mid_{A_j}\right) \mid_{A_i \cap A_j}$, which is equivalent to saying $f \mid_{A_i} x = f \mid_{A_j} x$ for all $x \in A_i \cap A_j$.

**Definition 93: Topological Vocabulary**

- Let $U$ be an open subset of some $S$ with topology $\mathcal{T}$. The family of open sets $C = \{U_i\}_{i \in i}$ indexed by some $I$ is an *open cover* of $U$ iff $\bigcup_{i \in I} U_i = U$.
- A family of open subsets $\{B_i\}_{i \in I}$ of $S$ form a *basis* for $\mathcal{T}$ iff for any $U \subset S$, there exists some $J \subset I$ such that $\{B_i\}_{i \in J}$ is an open cover of $U$. That is, any element of $\mathcal{T}$ (open set) can be represented as a union of some subset of the basis elements.

For a space $S$, the topological definition of a covering corresponds to a coverage on $\mathcal{O}(S)$, where the families of morphisms are inclusion maps. Recall $\mathcal{O}(S)$ denotes the set of open sets of $S$, which form a poset (in fact, a heyting algebra) under inclusion.

Taking $C$ as an open cover of $U$ as above, we can write $C$ as $\{U_i \hookrightarrow U\}_{i \in I}$.

**Definition 94: Subterminal Sheaf**

Fix a category $\mathcal{C}$

- Given a functor $F$, the functor $G$ is a *subfunctor* of $F$ if $G(C) \subseteq F(C) \ \forall \ C \in \mathcal{C}$. If $F$ is a sheaf on $\mathcal{C}$, then $G$ is a *subsheaf* if and only if for every open $U$ of $\mathcal{C}$ and every element $f \in FU$, and every open covering $U = \bigcup U_i$, one has $f \in GU$ if and only if $f \mid_{U_i} \in GU_i$ for all $i$.
- The terminal sheaf is the terminal object $\mathbf{1}$ of $Sh(\mathcal{C})$. A *subterminal sheaf* $F$ is a subsheaf of $\mathbf{1}$. $F$ is a subfunctor, meaning $FC \subseteq \mathbf{1}C = 1$ for all $C \in \mathcal{C}$. This means $FC$ can only be either 1 or $\emptyset$.

**Proposition 19.2.5.** A *subobject* of a sheaf $F$ in the category $Sh(\mathcal{C})$ is isomorphic to a subsheaf of $F$.

*Proof.* See §II.3 in *Sheaves in Geometry and Logic* by MacLane and Moerdijk. □

**Theorem 19.2.6.** For any space $X$, there is an isomorphism

$$\mathcal{O}(X) \cong Sub_{Sh(X)}(\mathbf{1})$$

of partially ordered sets.

*Proof.* $\Rightarrow$

Given any open set $W$ of $X$, define a functor $S_W$ on open sets $U$ by $S_W(U) = 1$ if $U \subseteq W$ and $\emptyset$ otherwise. Recall $\mathbf{1}U = 1$ for all open $U$. Let $\{V_i\}$ be an arbitrary covering of $U$. If $S_W(V_i) = 1$ for all $i$, then $\bigcup V_i \subseteq W$, meaning $U \subseteq W$, so $S_W(U) = 1$. In the other direction, if $S_W(U) = 1$, all $V_i \subseteq U$ so $S_W(V_i) = 1$ for all $i$. So, we can see $S_W$ is a subsheaf of $\mathbf{1}$.

$\Leftarrow$

Let $S$ be a subsheaf of $\mathbf{1}$. Each $S(U)$ must then be either 1 or $\emptyset$. If $S(U) = 1$ and $V \subseteq U$, and $S$ requires a morphism $S(U) \to S(V)$ meaning $1 \to S(V)$, it follows that $S(V)$ is nonempty and thus $S(V) = 1$. By the equalizer condition, if $\{V_i\}$ is an open cover of $U$ and $SU_i = 1$ for all $i$, then $SU = 1$.

Define $W = \bigcup\{U \in \mathcal{O}(X) | SU = 1\}$. Then by equalizer condition we have $SU = 1$ if and only if $U \subseteq W$. Then we have $S = S_W$, so we have the bijection $W \iff S_W$. □

### *19.2.3 Special Sheaf Functors*

The crux of the Sterling and Harper's information flow model is based on the following monad definition:

"For any subspace $Q \subset P$, a sheaf $A \in Sh(P)$ can be restricted to $Q$, as in $A\ |_Q \in Sh(Q)$, and then extended again to $P$. This composite defines an *idempotent* monad on $Sh(P)$ that we can interpret as removing any data from $P$ that cannot be seen from $Q$."

A monad $T$ is idempotent when $T^2 = T$ (i.e. $\mu$ =iso).

This construction is a specialization of a much more general property of sheaves, which we need to do some work to define.

**Definition 95**

Let $X, Y$ be topological spaces. A function $f : X \to Y$ is *continuous* if, for every open $V \subseteq Y$, the preimage $f^{-1}(V)$ is open in $X$.

**Exercise 1**

All monotone maps in the Alexandroff topology are continuous.

See Solution 19.4.1

**Definition 96: Direct Image Sheaf Functor**

Given a continuous $f$, each sheaf $F$ on $X$ yields a sheaf $f_*F$ on $Y$ defined for $V$ open in $Y$, by $(f_*F)V = F(f^{-1}V)$. Essentially, $f_*F$ is the composite functor

$$\mathcal{O}(Y)^{op} \xrightarrow{f^{-1}} \mathcal{O}(X)^{op} \xrightarrow{F} \mathbf{Set}$$

This sheaf is called the *direct image* of $F$ under $f$. The map $f_* : Sh(X) \to Sh(Y)$ is a functor of sheaf categories.

In the measure theory/qbs lecture, we briefly discussed the pushforward of a measure along a measurable map: $\mu_B(b) = \mu_A(f^{-1}(b))$ where $f^{-1}$ is the preimage. We can similarly think of $\mu_B$ as the composite functor $\Sigma_B \xrightarrow{f^{-1}} \Sigma_A \xrightarrow{\mu_A} [0,1]$, where we might want to denote $\mu_B$ by $f_*\mu_A$, and say $f_*$ is a functor of probability measures.

A continuous $f : X \to Y$ also induces a functor $f^* : Sh(Y) \to Sh(X)$, where each sheaf $G$ on $Y$ yields a sheaf $f^*G$ on $X$. We typically can't define $f^*$ (the Inverse Image Functor) as simply as we were able to define $f_*$ because we don't know if $f(U)$ is open (continuity only gives us information about preimages).

First we have the general and scary definition (though not the most general because we assume sheaves over topological spaces):

**Definition 97: Inverse Image Sheaf Functor**

Let $f : X \to Y$ be a continuous map of topological spaces. For any sheaf $G$ on $Y$, we define the *inverse image* sheaf $f^*G$ on $X$ to be the *sheafification* of the presheaf $U \mapsto \operatorname{colim}_{V \supseteq f(U)} G(U)$, where $U$ is any open set in $X$, and the limit is taken over all open sets $V$ of $Y$ containing $f(U)$.

Luckily for us, the only continuous maps we care about in this chapter are inclusion maps. Here is some sketchy / informal reasoning as to why $i^*F(U) = F(U)$ when $X \subseteq Y$ is open for open $U \subseteq X$.

- When $X \subseteq Y$ is open any open $U \subseteq X$ is open in $Y$ as well.
- $F(U)$ would be the colimit of $F(V)$ where $V \supseteq i(U) = U$, because $U$ is clearly the greatest lower bound (limit) on sets containing $U$, so dually $F(U)$ would be the least upper bound (colimit) on $F(V)$ with $V \supseteq U$, since $F$ is contravariant.
- Then $i^*F(U) = F(U)$, and of course $F$ is a sheaf so $i^*F(U)$ automatically satisfies sheaf conditions for open $U$.

If $X$ is not open, we have a more complicated situation, where open $U \subseteq X$ is not guaranteed to be open in $Y$.

If we are working in a topology $\mathcal{T}$ where for every (not necessarily open) subset $V$ of a space $Y$ has a smallest open $U \subseteq Y$ such that $V \subseteq U$, then $i^*F(V) = F(U)$. Let's notate this smallest open $U$ for

$V$ by $o(V)$. To have $i^*F$ satisfy sheaf conditions, we need covers to "behave well", that is, for $\bigcup_j V_j = V$, we need $\bigcup_j o(V_j) = o(V)$. In general this is not guaranteed.

However, in this chapter we will only be looking at the Alexandroff topology. In this topology the open sets are upper sets, so for any set $V$, we have $o(V) = \uparrow V$, where $\uparrow V$ represents the upwards closure of $V$, basically turning $V$ into an upper set. The upwards closure $\uparrow$ respects/distributes across unions, that is $\bigcup_j \uparrow V_j = \uparrow \bigcup_j V_j$. So we do not need to worry about sheafifying anything!

**Theorem 19.2.7.** If $f : X \to Y$ is a continuous map, then the functor $f^*$, sending each sheaf $G$ on $Y$ to its inverse image (preimage) on $X$, is left adjoint to the direct image functor $f_*$ :

$$Sh(X) \underset{f^*}{\overset{f_*}{\rightleftarrows}} Sh(Y) \ , \quad f^* \dashv f_*$$

*Proof.* See §II.9 in *Sheaves in Geometry and Logic* by MacLane and Moerdijk. □

Theorem 19.2.7 is included to note that $f_* \circ f^*$ is a monad on $Sh(Y)$. Particularly, when $X \subset Y$ and $f$ is the inclusion map $i$, we have a monad given by $i_* \circ i^*$. It is precisely this composition which defines Sterling et al.'s idempotent monad.

The preimage of the inclusion map $i : Q \hookrightarrow P$ is defined $i^{-1}(V) = V \cap Q$, where $V$ is an open subset of $P$. With this, let's compute this monad for any inclusion. Fix a sheaf $F$ and $V$ open in $P$.

$$\begin{aligned} i_*(i^*F)(V) &= (i^*F)(i^{-1}V) \\ &= (i^*F)(V \cap Q) \end{aligned}$$

Recall that in our special case, the functor $i^*$ operates by $i^*F(U) = F(\uparrow U)$ for open $U$ in subspace $Q$, where $\uparrow U$ is the upward closure of $U$ in $P$. When $Q$ is open in $P$, $i^*F(U) = F(U)$.

For any sheaf $F$, $F(\emptyset) = 1$. We can see this from the equalizer definition, where $F$ in this case is the equalizer of the empty diagram, which is precisely the terminal object.

If $V \cap Q \neq \emptyset$, then $i_*(i^*F)(V) = F(\uparrow (V \cap Q))$. If $V \cap Q = \emptyset$, then $i_*(i^*F)(V) = 1$.

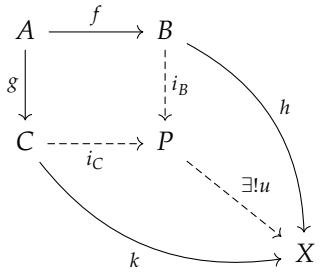
### *19.2.4 Pushouts*

Let $X$ be a topological space and $Y$ an open set in $X$. One of the monads that Sterling et al. define is the composition of $i_* \circ i^*$ defined by $i : X \setminus Y \hookrightarrow X$. The authors mention that this monad $i_* \circ i^*$ can be identified with a pushout construction

$$Y \sqcup_{Y \times (-)} (-) : Sh(X) \to Sh(X),$$

where $Y$ here is treated as the subterminal sheaf where $Y(U) = 1$ if $U \subseteq Y$, and $\emptyset$ otherwise. This construction is key for later non-interference proofs, so we provide some intuition for how pushouts work/behave.

Pushouts are dual to pullbacks, but it is not very intuitive to get a sense of what that means exactly. We can try to work it out by staring at the diagram in **Set**:

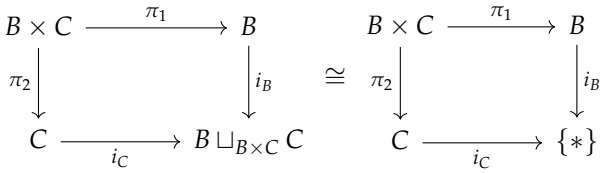



Where $P$ is the smallest object with morphisms $i_C, i_B$, such that $i_C \circ g = i_B \circ f$, i.e. $i_C(g(a)) = i_B(f(a))$ for all $a \in A$. Then we can see that this restriction is only for the images of $f$ and $g$; we don't have any equality condition on $b \notin im(f), c \notin im(g)$. These maps $i_B, i_C$ are canonically *quotient maps* modulo the relation by $f$ and $g$, meaning they map $b \in B, c \in C$ to equivalence classes $[b], [c] \in P$, with this extra equivalence condition.

So, we can interpret the commutativity of this square to mean that for $b, c$ where there exists an $a$ such that $f(a) = b, g(a) = c$, we require $i_B(b) = [b] = [c] = i_C(c)$.

In **Set**, $P$ is a "quotient of the disjoint union of $A$ and $B$", which can also be written $A \sqcup B/\sim$, where $\sim$ is the equivalence relation generated by the restriction detailed above, i.e. some $\sim$ satisfying $b \sim c$ when $\exists\ a \in A$ such that $f(a) = b, g(a) = c$. The pushout essentially collapses the images of $f$ and $g$ to equivalence classes in $P$. We provide a couple of examples in **Set**.

**Example 19.2.8** (pushout of product)**.** Let $B, C$ be nonempty sets.




By definition of product, for every $b \in B$ and for every $c \in C$, there exists an $a \in A$, namely $(b, c)$, such that $\pi_1((b, c)) = b$ and $\pi_2((b, c)) = c$. Then every element of $B$ is in the same equivalence class as every element of $C$, and by transitivity of equality, every element of $B$ is in the same equivalence class, as well as every element

of $C$. Then $P$ only has one equivalence class, so $P$ is a singleton set and thus isomorphic to the terminal object.

**Example 19.2.9** (pushout of the initial object, o)**.** Recall that if a category has a terminal object and pullbacks, then it has products. This is the dual for "initial and pushout imply coproducts".

$$\begin{array}{ccc} 0 & \xrightarrow{!} & B \\ {\scriptstyle !}\downarrow & & \downarrow{\scriptstyle i_B} \\ C & \xrightarrow[i_C]{} & B \sqcup_0 C \end{array} \quad \cong \quad \begin{array}{ccc} 0 & \xrightarrow{!} & B \\ {\scriptstyle !}\downarrow & & \downarrow{\scriptstyle i_B} \\ C & \xrightarrow[i_C]{} & B \sqcup C \end{array}$$

The empty set is the initial object in **Set**. There are no $b, c$ such that $f(a) = b, g(a) = c$ because there are no $a \in A \cong \emptyset$. Then each $b$ and $c$ are distinct, resulting in the disjoint union of $B$ and $C$.

Going back to our product example, suppose $B$ is empty and $C$ is nonempty. The product $B \times C$ is isomorphic to 0, so we would get $B \sqcup C = \emptyset \sqcup C = C$. If $B$ and $C$ are empty, then naturally the pushout would be empty as well.

Let's look at how pushouts behave with presheaves. Let $F, G \in [C^{op}; Set]$.

**Example 19.2.10** (pushout of $F \times G$)**.** This example is interesting because it does not automatically collapse to $\{*\}$. Functors are defined by how they act on objects, so for some $A \in \mathcal{C}$ we have

$$\begin{array}{ccc} (F \times G)(A) & \xrightarrow{\pi_{1A}} & F(A) \\ {\scriptstyle \pi_{2A}}\downarrow & & \vdots{\scriptstyle i_B} \\ G(A) & \xrightarrow[i_C]{} & F(A) \sqcup_{(F\times G)(A)} G(A) \end{array}$$

Where the projection maps are now natural transformations, also defined component-wise. Recall $(F \times G)(A) \cong F(A) \times G(A)$. We end up with 3 (really 4) cases for the pushout:

1. F(A) and G(A) are nonempty. Then we have the typical product - pushout, so in this case we do get 1 or $\{*\}$.
2. F(A) is nonempty and G(A) is empty. Then $F(A) \times G(A) = \emptyset$. Pushout from the initial object gives us $F(A) \sqcup G(A) = F(A) \sqcup \emptyset = F(A)$. We have a symmetric case for F(A) empty and G(A) nonempty.
3. F(A) and G(A) are empty. This yields $\emptyset \sqcup \emptyset = \emptyset$

Sheaves are a little more complicated. If we compute a pushout of a product of sheaves $S, P$ as above, we can define the result point-wise as a presheaf, but we are not guaranteed that this presheaf satisfies the sheaf conditions.

The general solution to this is to *sheafify* the computed presheaf pushout, which is currently out of the scope of this chapter.

## 19.3 *Sheaf Semantics for Noninterference (putting it all together)*

We are now ready to talk about some of the central ideas of the paper. This section will culminate in a proof of a form of noninterference where sheaves are interpreted as types.

Fix a poset $\mathcal{P}$ of security levels closed under finite meets.

**Definition 98**

- An *abstract behavior* $x$ is a filter on the poset $\mathcal{P}$. We can interpret this as saying $x$ denotes the security levels where a behavior is permitted.
  - For example, a filter generated by M would include H and $\top$, capturing that medium-security information/behavior can be used at higher-security levels.
- A *security policy* U is a lower set in $\mathcal{P}$. The notation $U \vdash x$ is pronounced "U permits $x$", and means $U \cap x \neq \emptyset$.
  - Consider the poset $\{L \leq M \leq H\}$. Supposing we have a security policy generated by M: this includes L and M, and can be understood as denoting the security levels *above* which some information/behavior is permitted. Alternatively, the set where this behavior is not allowed.

**Definition 99**

The authors define $P$ to be the topological space where points are abstract behaviors, and the open sets are of the form $\{x \mid U \vdash x\}$.

**Exercise 2**

Prove these sets are open with the openness axioms.

See Solution 19.4.2

Each security level $\ell \in \mathcal{P}$ represents a security policy $\langle \ell \rangle$ of the form $\{a \in \mathcal{P} \mid a \leq \ell\}$. The corresponding open subspace of $P$, $(\{x \mid \langle \ell \rangle \vdash x\})$ is denoted $P_{\langle \ell \rangle}$. The complement, $P \setminus P_{\langle \ell \rangle}$ is denoted $P_{\bullet \langle \ell \rangle}$.

**Proposition 19.3.1.** The predicate $\langle \ell \rangle \cap x \neq \emptyset$ is equivalent to $\ell \in x$.

To see the above, suppose we have some nonempty intersection for $\langle \ell \rangle$ and $x$. Then there exists some $k \in \langle \ell \rangle$ such that $k \in x$. Since $k \in \langle \ell \rangle$, this means $k \leq \ell$. Since $x$ is a filter and hence upwards closed, since $\ell \geq k$ and $k \in x$, we have $\ell \in x$. The other direction is immediate since $\ell \in \langle \ell \rangle$ so $\ell \in x$ clearly implies a nonempty intersection.

**Proposition 19.3.2.** Let $\langle \ell_1 \rangle$ and $\langle \ell_2 \rangle$ be security policies such that $\ell_1 < \ell_2$. Then $P_{\langle \ell_1 \rangle} \subset P_{\langle \ell_2 \rangle}$.

*Proof.* Let $x$ be an abstract behavior in $P_{\langle \ell_1 \rangle}$, i.e. a filter on $\mathcal{P}$ such that $\langle \ell_1 \rangle \cap x \neq \emptyset$. Then there is some $y$ in this intersection, meaning $y \in x$ and $y \leq \ell_1$. Since $\ell_1 < \ell_2$ and posets have transitivity, $y < \ell_2$, which means $y \in \langle \ell_2 \rangle$. Then $y \in x \cap \langle \ell_2 \rangle$, so $x \in P_{\langle \ell_2 \rangle}$. □

### *19.3.1 Sheaf Interpretation*

In this subsection, we parse Sterling et al.'s statement: *"our intention is to interpret each type of a dependency core calculus as a sheaf on the space P of abstract behaviors. to see why this interpretation is plausible as a basis for secure information flow, we note that a sheaf on P is the same thing as a presheaf on the poset $\mathcal{P}$"*.

- In this context, the only relevance of dependency core calculus is that it is a typed programming language where types can be annotated with security levels from some poset.
- This statement assumes a presheaf interpretation over $\mathcal{P}$ is understood to be a plausible basis for modeling information flow, so let's justify this quickly:
  - Consider the presheaf category over $\mathcal{P}$. This has a natural representation for information flow : for each security level $\ell$, we have a set $F(\ell)$ representing what data is visible under security level $\ell$. Since presheaves are contravariant functors, for each $k \leq l$ in $\mathcal{P}$, we get a morphism $F(\ell) \to F(k)$, demonstrating the restriction of information to that at a lower security level.
- Now we have this note : a sheaf on $P$ is "the same thing" as a presheaf on the poset $\mathcal{P}$. This sounds like a bijection.

**Theorem 19.3.3.** There is a bijection between the presheaf category $Psh(\mathcal{P})$ and the sheaf category $Sh(P)$.

Recall the topological space at hand, $P$, has open sets of the form $\{x \mid U \vdash x\}$, where $U$ is a lower set, $x$ is a filter, and $U \vdash x$ means

$U \cap x \neq \emptyset$. Note that the open sets are defined by the lower sets of $\mathcal{P}$, so we will denote them by $O_U$, where $U$ is the lower set of choice. Now we have a series of lemmas.

**Lemma 19.3.4.** The family $\{P_{\langle \ell \rangle}\}_{\ell \in \mathcal{P}}$ is a basis for the topology on $P$.

*Proof.* We know each $P_{\langle \ell \rangle} = \{x \mid \langle \ell \rangle \vdash x\}$ are open by definition.

Let $U$ be an upper set of $\mathcal{P}$ and $O_U$ the corresponding open set of $P$, so $O_U = \{x \mid x \cap U \neq \emptyset\} = \{x \mid \exists\, \ell \in U : \ell \in x\}$. We can tranform this into

$$\bigcup_{\ell \in U} \{x | \ell \in x\} = \bigcup_{\ell \in U} \{x \mid \langle \ell \rangle \vdash x\} = O_U,$$

where the second expression comes from Theorem 19.3.1. So, we have a family of nonempty open sets of the form $P_{\langle \ell \rangle}$ indexed by some subset of $\mathcal{P}$ (as some of the sets in the above union will be naturally empty) such that their union is equal to $O_U$ in $P$. □

**Lemma 19.3.5.** Let $\ell, k \in \mathcal{P}$. We claim $P_{\langle \ell \rangle} \cap P_{\langle k \rangle} = P_{\langle \ell \wedge k \rangle}$

*Proof.* Note we have $\ell \wedge k$ for any $\ell, k \in \mathcal{P}$ because $\mathcal{P}$ is closed under finite meets. Suppose we have some $x \in P_{\langle l \wedge k \rangle}$, meaning $\ell \wedge k \in x$. Since $x$ is upwards closed and $\ell \wedge k$ is the meet of $\ell$ and $k$, it follows immediately that $\ell \in x$ and $k \in x$, meaning $x \in P_{\langle \ell \rangle} \cap P_{\langle k \rangle}$.

For the other direction, we begin with $\ell \in x$ and $k \in x$. Since $x$ is a filter, any finite subset of $x$ has a lower bound in $x$. Considering $\{\ell, k\}$ as our subset, suppose we have some lower bound $q \in x$. By definition of meets, $q \leq \ell \wedge k$, so $\ell \wedge k \in x$, meaning $x \in P_{\langle \ell \wedge k \rangle}$. This reasoning also tells us that filters on a poset are closed under finite meets. □

There is an alternate (perhaps more elegant) theorem in MacLane and Moerdijk demonstrating the same property as Theorem 19.3.6: Let $\mathcal{B}$ be a basis for a topology on a space $X$. The restriction functor $Sh(X) \to Sh(\mathcal{B})$ is an equivalence of categories. They leave the proof as an exercise for the reader.

**Lemma 19.3.6.** Any sheaf on $P$ is determined by values on the basis $\{P_{\langle \ell \rangle}\}_{\ell \in \mathcal{P}}$.

*Proof.* Fix some sheaf $F$ on $P$ and let $O_U$ be open in $P$. There exists some subset $\mathcal{Q} \subset \mathcal{P}$ such that $O_U = \bigcup_{\ell \in \mathcal{Q}} P_{\langle \ell \rangle}$, so $\{P_{\langle \ell \rangle}\}_{\ell \in \mathcal{Q}}$ is a covering of $U$. Since $F$ is a sheaf, it must satisfy

$$F(O_U) \xrightarrow{e} \prod_{\ell \in \mathcal{Q}} F(P_{\langle \ell \rangle}) \overset{\xrightarrow{p}}{\underset{\xrightarrow{q}}{}} \prod_{\ell, k \in \mathcal{Q}} F(P_{\langle \ell \rangle} \cap P_{\langle k \rangle}),$$

which is equal to

$$F(O_U) \xrightarrow{e} \prod_{\ell \in \mathcal{Q}} F(P_{\langle \ell \rangle}) \overset{\xrightarrow{p}}{\underset{\xrightarrow{q}}{}} \prod_{\ell, k \in \mathcal{Q}} F(P_{\langle \ell \wedge k \rangle}).$$

Then to compute behavior of $F$ on $O_U$ such that the sheaf conditions are satisfied, we only need to know how $F$ acts on the basis, since we have rewritten the rest of the diagram in terms of the basis elements.

□

Okay, now we are ready to prove Theorem 19.3.3!

*Proof.* .

1. $Psh(\mathcal{P}) \to Sh(P)$: Let $E$ be a presheaf on $\mathcal{P}$. We will use $E$ to define a sheaf $F$ on $P$. We have the natural behavior for the basis elements: $F(P_{\langle \ell \rangle}) := E(\ell)$.

   As we showed above, all the other open sets follow immediately when we set $F(O_U)$ as the equalizer of the diagram with the covering of choice being the basis elements. Note that this extends to arbitrary coverings, as each open set can be written itself as a union of basis elements, so no matter what we can always reduce a covering to its basis elements.

   F maintains all the presheaf requirements as well: suppose $P_{\langle \ell \rangle} \subset P_{\langle k \rangle}$. We've shown this implies $\ell \leq k$. Then we have a restriction morphism $E(k) \to E(\ell)$, and by definition this yields a restriction morphism $F(P_{\langle k \rangle}) \to F(P_{\langle \ell \rangle})$. Defining $F(O_U)$ to be the equalizer of the given diagram gives us the desired sheaf conditions, as sketched in Theorem 19.2.3.

2. $Sh(P) \to Psh(\mathcal{P})$: Set $E(\ell)$ to $F(P_\ell)$ for all $\ell \in \mathcal{P}$.

□

We could also be done after this definition, since we have a unique definition of $F$ on the basis, and the fancier theorem says $F(\mathcal{B})$ is equivalent to $F(P)$, where $\mathcal{B}$ is a basis for $P$.

### 19.3.2 *Transparency and Sealing Monads*

These transparency and sealing monads are precisely instances of $i_* \circ i^*$, as defined in Section 19.2.3.

**Definition 100**

1. The *transparency monad* $\langle \ell \rangle \Rightarrow A$ replaces $A$ with whatever part of it can be viewed under policy $\langle \ell \rangle$. If we define $i : P_{\langle \ell \rangle} \hookrightarrow P$, the composition $i_* \circ i^*$ defines the unit $\eta_{\Rightarrow} : Sh(P) \to Sh(P)$ where $A \mapsto (\langle \ell \rangle \Rightarrow A)$. When the unit is an isomorphism at $A$, we say that $A$ is $\langle \ell \rangle$-*transparent*.
2. The *sealing monad* $\langle \ell \rangle \bullet A$ removes from $A$ whatever part of it can be viewed under policy $\langle \ell \rangle$. If we define $i : P \setminus P_{\langle \ell \rangle} \hookrightarrow P$, the composition $i_* \circ i^*$ defines the unit $\eta_{\bullet} : Sh(P) \to Sh(P)$ where $A \mapsto (\langle \ell \rangle \bullet A)$.
   - The sealing monad can be constructed as the pushout $P_{\langle \ell \rangle} \sqcup_{P_{\langle \ell \rangle} \times A} A$. When the unit is an isomorphism at $A$, we say $A$ is $\langle \ell \rangle$-*sealed*.

**Proposition 19.3.7.** Sterling et al. claim the transparency monad "is" the function space $A^{P_{\langle \ell \rangle}}$, i.e. $A^{P_{\langle \ell \rangle}} \cong \langle \ell \rangle \Rightarrow A$. We will sketch a proof for the general case of open $Y$ in $P$.

You may notice we are treating the open space $P_{\langle \ell \rangle}$ (and general $Y$) as a sheaf. Recall Theorem 19.2.6, which proves Sterling et al.'s statement that an open set of $P$ is "the same" as a subterminal sheaf.

*Proof.* (sketch)

Let $j : Y \hookrightarrow P$ and $A$ be a sheaf on $P$. We want $j_* j^* A \cong A^Y$. One definition of the exponential object on sheaves over $\mathcal{O}(X)$ is

$$A^Y(U) \cong \mathrm{Hom}(Y \mid_U, A \mid_U)$$

where $Y \mid_U, A \mid_U$ are the sheaves restricted to subsets of $U$. Since $Y \mid_U (V) = 1$ if $V \subseteq U \cap Y$ and $\emptyset$ otherwise, any morphisms (natural transformations) $Y \mid_U \to A \mid_U$ are characterized by $V \subseteq U \cap Y$.

Let $\alpha$ be a natural transformation and $\alpha_V$ the component associated to $V \subseteq U \cap Y$, i.e. $\alpha_V : Y \mid_U (V) \to A \mid_U (V) = A(V)$. This simplifies to $\alpha_V : 1 \to A(V)$. There is a natural isomorphisms between $\mathrm{Hom}(1, A)$ and $A$, for any object $A$, so $\alpha_V$ is naturally isomorphic to $A(V)$. The subsets $V$ form a cover of $U \cap Y$, so we should be able to glue the nontrivial components of $\alpha$ (i.e., $\alpha \mid_{U \cap Y}$) to get $A(U \cap Y)$. Then we have

$$\begin{aligned} \mathrm{Hom}(Y \mid_U, A \mid_U) &\cong \mathrm{Hom}(Y \mid_{U \cap Y}, A \mid_{U \cap Y}) \\ &\cong \{A(V) \mid V \subseteq U \cap Y\} \\ &\cong A(U \cap Y) \end{aligned}$$

In Section 19.2.3, we demonstrated $j_* j^* A(U) = A(U \cap Y)$ for $j : Y \hookrightarrow P$, so we have $j_* j^* A \cong A^Y$. □

Given space $P$ and arbitrary $\langle \ell \rangle \in P$, the subspace defining the transparency monad $\langle \ell \rangle \Rightarrow$ is $P_{\langle \ell \rangle}$, and the subspace defining the sealing monad $\langle \ell \rangle \bullet$ is $P_{\bullet \langle \ell \rangle}$.

Closed sets are defined to be complements of open sets. It is rare that the complement of an open set will also be open (called clopen sets). Note that $P_{\bullet \langle \ell \rangle}$ is not guaranteed to be open – this is where the caveats of Definition 97 with respect to the Alexandroff topology come in. That is, for every open $U \in P_{\bullet \langle \ell \rangle}$, we have $i^* F(U) = F(\uparrow U)$ where $\uparrow U$ is the upward closure of $U$ in P.

**Example 19.3.8.** Let's fix $\mathcal{P} := \{L \subset M \subset H \subset \top\}$.

There are only four filters on $\mathcal{P}$, so we can write all of them out:

$$p_0 = \{L, M, H, \top\}$$
$$p_1 = \{M, H, \top\}$$
$$p_2 = \{H, \top\}$$
$$p_3 = \{\top\}$$

From here, we can also enumerate the principal open sets on $P$:

$$P_{\langle L \rangle} = \{p_0\}$$
$$P_{\langle M \rangle} = \{p_0, p_1\}$$
$$P_{\langle H \rangle} = \{p_0, p_1\, p_2\}$$
$$P_{\langle \top \rangle} = \{p_0, p_1\, p_2, p_3\}$$

To get a better idea of how these monads work, we will work through explicitly what $\langle M \rangle \Rightarrow A$ and $\langle M \rangle \bullet A$ look like.

1. $\langle M \rangle \Rightarrow A$ is the monad defined by composition of the inverse and direct images of $i : P_{\langle M \rangle} \hookrightarrow P$

$$(i_* \circ i^*) A(P_{\langle L \rangle}) = A(P_{\langle L \rangle} \cap P_{\langle M \rangle}) = A(P_{\langle L \rangle})$$
$$(i_* \circ i^*) A(P_{\langle M \rangle}) = A(P_{\langle M \rangle} \cap P_{\langle M \rangle}) = A(P_{\langle M \rangle})$$
$$(i_* \circ i^*) A(P_{\langle H \rangle}) = A(P_{\langle H \rangle} \cap P_{\langle M \rangle}) = A(P_{\langle M \rangle})$$
$$(i_* \circ i^*) A(P_{\langle \top \rangle}) = A(P_{\langle \top \rangle} \cap P_{\langle M \rangle}) = A(P_{\langle M \rangle})$$

2. $\langle M \rangle \bullet A$ is the monad defined by composition of the inverse and direct images of $i : P_{\bullet \langle M \rangle} \hookrightarrow P$

$$(i_* \circ i^*) A(P_{\langle L \rangle}) = A(\uparrow (P_{\langle L \rangle} \cap P_{\bullet \langle \ell \rangle})) = A(\emptyset) = 1$$
$$(i_* \circ i^*) A(P_{\langle M \rangle}) = A(\uparrow (P_{\langle M \rangle} \cap P_{\bullet \langle \ell \rangle})) = A(\emptyset) = 1$$
$$(i_* \circ i^*) A(P_{\langle H \rangle}) = A(\uparrow (P_{\langle H \rangle} \cap P_{\bullet \langle \ell \rangle})) = A(\uparrow (\{p_2\})) = A(P_{\langle H \rangle})$$
$$(i_* \circ i^*) A(P_{\langle \top \rangle}) = A(\uparrow (P_{\langle \top \rangle} \cap P_{\bullet \langle \ell \rangle})) = A(\uparrow (\{p_2, p_3\})) = A(P_{\langle \top \rangle})$$

This aligns with intuition that sealing at level $\langle \ell \rangle$ preserves all data "higher" than $\ell$, but collapes/makes all data at or "lower" than $\ell$ indistinguishable.

**Exercise 3**

The authors mention that the sealing monad also has a pushout construction. Check that evaluating $P_{\langle M \rangle} \sqcup_{P_{\langle M \rangle} \times A} A$ at the principal sets ($P_{\langle \ell \rangle}$ for $\ell \in \mathcal{P}$) agrees with the composition of the direct and inverse images of the inclusion map.

(Hint : use Theorem 19.2.10; don't worry about sheafification)

See Solution 19.4.3

**Proposition 19.3.9.** The $\langle \ell \rangle$-transparent part of an $\langle \ell \rangle$-sealed sheaf is trivial, i.e. $(\langle \ell \rangle \Rightarrow (\langle \ell \rangle \bullet A)) \cong \{*\}$.

*Proof.* Let's expand the definition of $\langle \ell \rangle \Rightarrow$ in the above expression:

$$i_*(i^*(\langle \ell \rangle \bullet A))(V) = (i^*(\langle \ell \rangle \bullet A))(i^{-1}V) = (i^*(\langle \ell \rangle \bullet A))(V \cap P_{\langle \ell \rangle}) = (\langle \ell \rangle \bullet A)(V \cap P_{\langle \ell \rangle})$$

We know $V \cap P_{\langle \ell \rangle} \subset P_{\langle \ell \rangle}$, so from our computation above we can see $(\langle \ell \rangle \bullet A)(V \cap P_{\langle \ell \rangle}) = \{*\}$ for any $V \in P$. □

### *19.3.3 Noninterference*

We've made it!
The noninterference property will be represented by the constant definition. That is, if we have a morphism from higher security to lower security, this morphism satisfies noninterference if it is constant.

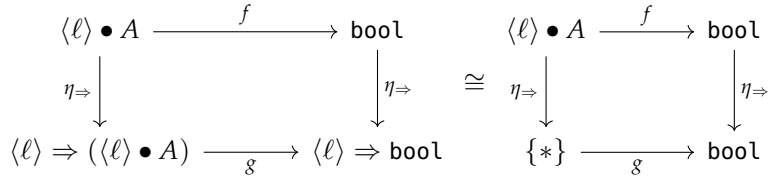
Recall that we aim to view these sheaves as types: we can think of $A(U)$ where $U \subset P$ as the data of type $A$ which has some associated security restrictions/permissions. If we say a morphism $A \to B$ is constant, this means that the behavior of $f$ does not reveal any information about data of type $A$.

**Theorem 19.3.10.** Any map $\langle \ell \rangle \bullet A \to \texttt{bool}$ is constant.

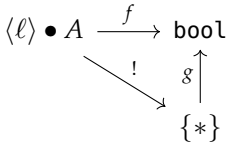
*Proof.* The boolean sheaf `bool` assigns to each open the set of boolean values $\{\texttt{true}, \texttt{false}\}$.

Let's see what happens when we apply $\langle \ell \rangle \Rightarrow \texttt{bool}$. This evaluates to $\texttt{bool}(P_{\langle \ell \rangle} V) = \{\texttt{true}, \texttt{false}\}$ for any open $V \in P$. By function extensionality, $\langle \ell \rangle \Rightarrow \texttt{bool} = \texttt{bool}$ for all $\ell$, so `bool` is $\langle \ell \rangle$-transparent for all $\ell$, meaning $\langle \ell \rangle \Rightarrow \texttt{bool} \cong \texttt{bool}$.

The unit $\eta_{\Rightarrow} : Id \to (\langle \ell \rangle \Rightarrow \_)$ is a natural transformation, so the following diagram commutes:




The diagram on the left comes from Theorem 19.3.9 and the fact that `bool` is $\langle \ell \rangle$-transparent for all $\ell$. This simplifies to




Since $f$ factors through the terminal object, $f$ must be constant. □

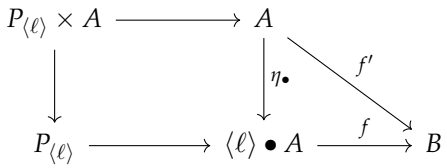
Since `bool` is a constant sheaf, it is essentially public. So, the above theorem is stating the mapping to bool from any type of $\langle \ell \rangle$ sealed data will not reveal anything about this "more secure" data.

**Theorem 19.3.11.** Morphisms $\langle \ell \rangle \bullet A \to B$ are in bijective correspondence with morphisms $A \to B$ that restricts to a constant function under $\langle \ell \rangle$.

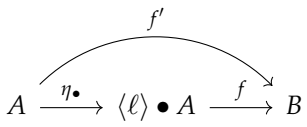
*Proof.* We take "restricts under $\langle \ell \rangle$" to mean "under application of the $\langle \ell \rangle$ transparency monad". Recall $\eta_\bullet$ refers to the unit for sealing.

1. Begin with $f : \langle \ell \rangle \bullet A \to B$.

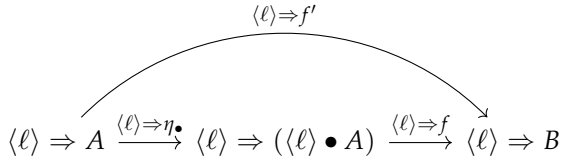
   We can use the pushout definition of the sealing monad to get the following diagram:



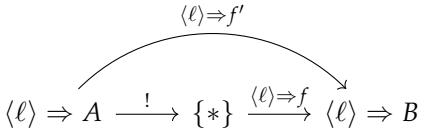

   where $f'$ is defined to be $f \circ \eta_\bullet$. Based on our understanding of restriction under $\langle \ell \rangle$, we want to see how $\langle \ell \rangle \Rightarrow f'$, mapping $\langle \ell \rangle \Rightarrow A \to \langle \ell \rangle \Rightarrow B$ behaves, so really we only care about




   We now use the functoriality of $\langle \ell \rangle \Rightarrow$ to have that the following diagram commutes,



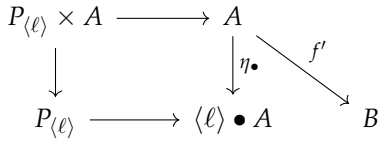

which simplifies to




So, since $\langle \ell \rangle \Rightarrow f'$ factors through the terminal object, it must be constant, meaning the $f'$ defined by $f \circ \eta_\bullet$ restricts to a constant function under $\langle \ell \rangle$.

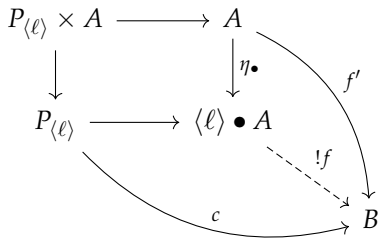
2. Begin with an $f' : A \to B$ such that $\langle \ell \rangle \Rightarrow f'$ is constant.

   We again want to observe the pushout diagram for $\langle \ell \rangle \bullet A$:




   Note we do not have a defined arrow $\langle \ell \rangle \bullet A$ to $B$. By the universal property of the pushout, if we have morphisms $A \to B$ and $P_{\langle \ell \rangle} \to B$, then there exists a unique morphism $\langle \ell \rangle \bullet A \to B$ such that everything is nice etcetera. To obtain a morphism $P_{\langle \ell \rangle} \to B$, we use the constant under $\langle \ell \rangle$ assumption for $f'$.

   $\langle \ell \rangle \Rightarrow f'$ constant means for any $x \in \langle \ell \rangle \Rightarrow A$, we have $(\langle \ell \rangle \Rightarrow f')(x) = c$ for some constant $c \in \langle \ell \rangle \Rightarrow B$. Theorem 19.3.7 states that the transparency monad is isomorphic to the function space $A^{P_{\langle \ell \rangle}}$ (with open sets being treated as subterminal sheaves), so this constant $c$ is isomorphic to a map $P_{\langle \ell \rangle} \to B$, so we can complete our diagram to get




   For each $f' : A \to B$ restricting to a constant function under $\langle \ell \rangle$, there exists a unique $f : \langle \ell \rangle \bullet A \to B$.

□

The noninterference intuition for the above theorem is that, if we have a map which relies on sealed (secret,private,etc) data, then when we try to take a "less secure" view, the behavior of this map is indistinguishable.

## *19.4 Solutions*

**Solution 19.4.1.** Solution to Exercise 1:

Let $U$ be open in $Y$. Then $U$ is an upper set, meaning if $a \in U$ and $a \leq b$, then $b \in U$. Suppose we have $x \in f^{-1}(U)$ and $y \in X$ such that $x \leq y$. We wish to show $y \in f^{-1}(U)$.

Since $x \in f^{-1}(U)$, we know $f(x) \in U$. By monotonicity, we know $f(x) \leq f(y)$, meaning $f(y) \in U$. Then $y \in f^{-1}(U)$, so $f^{-1}(U)$ is an upper set and thus open.

**Solution 19.4.2.** Solution to Exercise 2:

1. The empty set, along with being an upper set, is also a lower set (for the same vacuous reason). Then we can write the set $\{x \mid x \cap \emptyset \neq \emptyset\} = \emptyset$, so the empty set fits this characterization. Now we look at $\{x \mid x \cap P \neq \emptyset\}$, but this in fact contains every $x$ because each $x$ is nonempty, so this set is equal to $P$.

2. Arbitrary unions: we want to write $\bigcup_{i \in I} \{x \mid U_i \vdash x\}$ as $\{x \mid ? \vdash x\}$.What does it mean for some $x'$ to be in this union? That there is some $U_j$ such that $x' \cap U_j \neq \emptyset$, which means there exists some $y$ such that $y \in x'$ and $y \in U_j$. So we can reformulate this as saying $x$ is in the arbitrary union if there exists some $j \in I$ and $y \in x$ such that $y \in U_j$. The existence of some $j$ can be rephrased as requiring there exist some $y \in x$ such that $y \in \bigcup_{i \in I} U_i$. Then this is equivalent to the existence of some $y \in x \cap \bigcup_{i \in I} U_i$, so we can rewrite the arbitrary union as

$$\{x \mid x \cap \bigcup_{i \in I} U_i\}.$$

   The arbitrary union of lower sets is also a lower set — the proof is very similar to that for upper sets, you can do it if you are not convinced. So, we have that these sets are closed under arbitrary unions.

3. Finite intersections: Similarly, we can rewrite $O_V \cap O_U$ as $\{x \mid (U \cap V) \vdash x\}$, and lower sets are also closed under finite intersections.

**Solution 19.4.3.** Solution to Exercise 3:

We will compute $\langle M \rangle \bullet A$ as the pushout of $P_{\langle M \rangle} \times A$. We will use Theorem 19.2.10, which was for presheaves. Because this space $Sh(P)$ is particularly nice, we don't have to do any fancy sheafifying after computing the pointwise values. Recall $P_{\langle M \rangle}$ is being treated as a subterminal sheaf, so since $P_{\langle H \rangle} \not\subset P_{\langle M \rangle}$ for example, $P_{\langle M \rangle}$ applied to $P_{\langle H \rangle}$ will be $\emptyset$. On the other hand, both $P_{\langle M \rangle}$ and $P_{\langle L \rangle}$ subset $P_{\langle M \rangle}$, so the application returns 1.
These are the resulting values of applying the monad:

$$(\langle M \rangle \bullet A)(P_{\langle L \rangle}) = 1$$
$$(\langle M \rangle \bullet A)(P_{\langle M \rangle}) = 1$$
$$(\langle M \rangle \bullet A)(P_{\langle H \rangle}) = A(P_{\langle H \rangle})$$
$$(\langle M \rangle \bullet A)(P_{\langle \top \rangle}) = A(P_{\langle \top \rangle})$$

**Solution 19.4.4.** Solution to Lemma 19.2.4 (sketch)

Let $e$ be an equalizer map for $A \overset{p}{\underset{q}{\rightrightarrows}} B$. We want to show $e$ is a monomorphism, meaning for any $f : C \to A, g : C \to A$, if we have $e \circ f = e \circ g$, this implies $f = g$. So, suppose $e \circ f = e \circ g$.

By equalizerness, we know $p \circ e = q \circ e$, so we can set $p \circ e \circ f = q \circ e \circ f$. Then $e \circ f$ is an equalizer of $p$ and $q$. By the universal property, there exists a unique $u$ such that $e \circ u = e \circ f$, and uniqueness of $u$ gives us $u = f = g$, so we have $f = g$.

The coequalizer $\implies$ epimorphism reasoning is very similar.

# 20
# *Type Refinements as Indexed Inductive Types*

This chapter was written by Joseph Rotella.

In the course of writing code in a simply typed language, one may find oneself wishing to constrain the values of a type based on some "feature" of its values. For instance, when writing functions over lists or trees, it may be desirable to enforce constraints on the length or height of the input, or to ensure that the values it stores conform to some precondition. A classic example is the list `zip` operation: the author of this function may wish to enforce that its arguments have the same length.

*Refinement types* enrich simple type systems to allow programmers to express such constraints. For example, the type of lists of length `n` can be written

```
{xs :  List A | length(xs) = n}
```

In dependently typed languages, such refinement types can be naïvely translated as dependent pairs, attaching to lists proofs that they satisfy the length constraint:

$$\sum_{xs \,:\, \texttt{List}\ \alpha} \texttt{length}(xs) = n$$

However, dependent types permit a more elegant encoding: such constraints can be directly "baked into" the definition of an inductive type via type indices, ensuring that all values inhabiting the type are correct by construction. Our example above, for instance, yields the type of length-indexed vectors:[1]

```
inductive Vector (A : Type) : Nat -> Type
  | vnil  : Vector A 0
  | vcons : {n : Nat} -> A -> Vector A n -> Vector A (succ n)
```

[1] Throughout this chapter, we will write code snippets in a syntax similar to that of Lean 4, though we assume all constructors live in the global namespace. Those familiar with Agda can read `inductive` as `data` and `Type` as `Set`.

The type `Vector` naturally internalizes the `length` refinement from the simply typed setting. In this chapter, we will use a categorical construction to generalize this transformation to arbitrary inductive types and a broad class of refinements. In other words, we answer the following question: how can a refinement of a type in a simply

typed language be translated into an indexed inductive type in a dependently typed one?

Although this question serves as our motivation, the actual aim of this chapter is threefold:

- To give an account of a categorical semantics of inductive and indexed inductive types;
- To illustrate an application of these tools in the form of the translation discussed above; and
- To provide motivating intuition for the study of *fibrations*, which we regrettably do not have space to treat herein.

The content of this chapter is largely based on the work of Atkey, Johann, and Ghani [2]. The content of the first section on inductive types also draws on Wadler's "Recursive Types for Free!" [36] and scribed lecture notes from New's presentation of similar material [28].

## 20.1 Categorifying Inductive Types

We will begin by adding a new programming-language feature to our categorical repertoire: definitions of inductive datatypes.

### 20.1.1 Wherefore More Constructions?

Let us first consider why the facilities afforded to us by our existing semantic development are inadequate to express inductive type definitions in full generality. This may not be immediately obvious: for instance, as we have previously seen, the datatype of booleans

```
inductive Bool : Type where
  | true  : Bool
  | false : Bool
```

has elements isomorphic to those of `Unit` $\oplus$ `Unit`, which we interpret categorically as $1 \oplus 1$. We can even model finite parameterized datatypes: the option datatype

```
inductive Option (A : Type) : Type where
  | some : A -> Option A
  | none : Option A
```

has elements isomorphic to those of `A` $\oplus$ `Unit`, which we interpret as $[\![\texttt{A}]\!] \oplus 1$ (assuming that we are able to interpret `A`). More generally, to interpret the elements of a given inductive type, we can treat each constructor argument as a term in a product, then sum together the representations of each constructor. Where lies the problem?

You may notice that our wording above—interpreting "elements of the type" induced by the definition, rather than the "definition"

itself—is somewhat roundabout: this is intentional. An inductive type is characterized not merely by its ground inhabitants, but also by the *maps into* the type (i.e., its constructor forms). An inductive type also comes equipped with a *recursor* (also known as an *iterator* or *fold*) that admits recursive definitions over the type. The account above, which describes only the object denoted by the type itself, does not directly characterize these facets of an inductive definition.

Moreover—and more problematically—we also currently lack the facilities to provide a general categorical account of infinite inductive types, such as the type of natural numbers

```
inductive Nat : Type where
  | zero : Nat
  | succ : Nat -> Nat
```

or the type of lists

```
inductive List (A : Type) : Type where
  | nil  : List A
  | cons : A -> List A -> List A
```

since the tools we have previously used to give the semantics of lambda calculi—(finite) (co)products, exponential objects, and initial and terminal objects—do not admit the construction of sums and products of unbounded arity. To give a complete account of general inductive definitions, we will need to develop some new (and repurpose some old) tools.

We will motivate our translation of inductive datatypes using our standard approach to translating PL concepts into category theory: translating inference rules. Specifically, we will translate the inference rules for booleans and natural numbers into categorical constructions, then observe some patterns that will allow us to generalize these translations to arbitrary inductive types.

### *20.1.2 Categorifying Booleans*

To begin, we return to the example of the type `Bool`. Its constructors are characterized by the following typing rules:

$$\frac{}{\vdash \texttt{true} : \texttt{Bool}}\ \textsc{T-true} \qquad \frac{}{\vdash \texttt{false} : \texttt{Bool}}\ \textsc{T-false}$$

Note that we write these rules with an empty context, rather than an arbitrary $\Gamma$. Recalling that the empty context is represented by the terminal object 1, these rules correspond to the existence of two *global elements* $\texttt{true}, \texttt{false} : 1 \to [\![\texttt{Bool}]\!]$.

**Definition 101: Global element**

In a category with terminal objects, a *global element* of an object $A$ is a morphism $1 \to A$. It is the categorical interpretation of an "element" of an object $A$.

Here, our use of global elements corresponds to the fact that `true` and `false` are distinguished members of the "collection" denoted by `Bool`. The rules above translate into the following diagram:

$$1 \xrightarrow{\texttt{true}} [\![\texttt{Bool}]\!] \xleftarrow{\texttt{false}} 1$$

However, we want to be able to map *out* of our inductive types as well. To do so, we must define the recursor for the Boolean type, canonically known as `if`:[2]

$$\frac{\vdash t : \tau \qquad \vdash e : \tau}{b : \texttt{Bool} \vdash \texttt{if}(t,e)(b) : \tau}\ \text{T-IF}$$

[2] We will adopt the convention of writing recursors with their *major premise* (i.e., the value being "recursed on") as the final argument. This is opposite the usual notation in functional programming but will better agree with the categorical formulation.

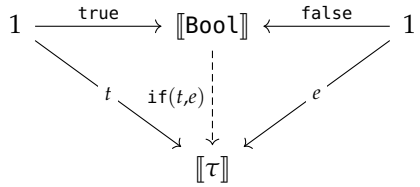
The recursor is operationally characterized by $\beta$ and $\eta$ laws. The $\beta$ laws say that `if` should "case correctly": it should evaluate to its "then" branch on a `true` input, and to its "else" branch on a `false` one. The $\eta$ law, meanwhile, says that "things that behave like `if`s are judgmentally equal to `if`s;" one can think of this as a restricted form of function extensionality:

$$\frac{\vdash t : \tau \qquad \vdash e : \tau}{\vdash \texttt{if}(t,e)(\texttt{true}) \equiv t : \tau}\ \beta_1^{\texttt{if}} \qquad \frac{\vdash t : \tau \qquad \vdash e : \tau}{\vdash \texttt{if}(t,e)(\texttt{false}) \equiv e : \tau}\ \beta_2^{\texttt{if}}$$

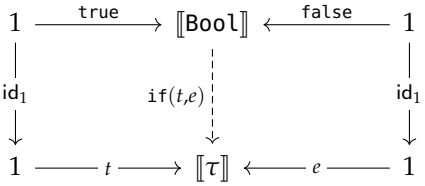
$$\frac{x : \texttt{Bool} \vdash d : \tau \quad \vdash d[\texttt{true}/x] \equiv t : \tau \quad \vdash d[\texttt{false}/x] \equiv e : \tau}{x : \texttt{Bool} \vdash d \equiv \texttt{if}(t,e)(x) : \tau}\ \eta^{\texttt{if}}$$

As in our previous translations of equational laws, the $\beta$ laws correspond to equalities of morphisms, while the $\eta$ law is a uniqueness criterion. In particular, the $\beta$ laws require that if we have morphisms $t, e : 1 \to [\![\tau]\!]$, then $\texttt{if}(t,e) \circ \texttt{true} = t$ and $\texttt{if}(t,e) \circ \texttt{false} = e$. Meanwhile, the $\eta$ law says that if we have some expression $d$ such that it can be substituted for $\texttt{if}(t,e)$ in the preceding two equalities[3]—i.e., $d \circ \texttt{true} \equiv t$ and $d \circ \texttt{false} \equiv e$—then it is equal to the recursor: in other words, the recursor denotes the unique morphism satisfying these equalities. Putting these together, we obtain the following pair of commutative triangles:

[3] Recall that we interpret substitution as composition of morphisms.




For reasons that will become clear shortly, it is helpful to expand these triangles into squares:




(20.1)

**Exercise 20.1**

Verify that $1 \oplus 1$ is a suitable choice for $[\![\texttt{Bool}]\!]$. What are the denotations of `true`, `false`, and $\texttt{if}(t, e)$?

### *20.1.3 Categorifying Natural Numbers*

We now perform a similar translation for natural numbers. The type `Nat` has two constructors, `zero` and `succ`, and is equipped with a recursor $\texttt{rec}(e_z, x.e_s)(n)$. Notice that, unlike `if`, the recursor for `Nat` binds a variable in the successor case: this will be the recursive result of evaluating the recursor on the substructure of $n$. The typing rules are given below:

$$\frac{}{\vdash \texttt{zero} : \texttt{Nat}}\ \text{T-ZERO} \qquad \frac{}{n : \texttt{Nat} \vdash \texttt{succ}(n) : \texttt{Nat}}\ \text{T-SUCC}$$

$$\frac{\vdash e_z : \tau \qquad x : \texttt{Nat} \vdash e_s : \tau}{n : \texttt{Nat} \vdash \texttt{rec}_{\texttt{Nat}}(e_z, x.e_s)(n) : \tau}\ \text{T-NATREC}$$

The $\beta$ rules for the recursor express its behavior as a fold over the structure of a natural number: applying the recursor to zero evaluates to the first arm, while applying it to a successor binds $x$ to the recursive result of folding over its predecessor, then evaluates $e_s$:

$$\frac{\vdash e_z : \tau \qquad x : \texttt{Nat} \vdash e_s : \tau}{\vdash \texttt{rec}_{\texttt{Nat}}(e_z, x.e_s)(\texttt{zero}) \equiv e_z : \tau}\ \beta_z^{\texttt{rec}_{\texttt{Nat}}}$$

$$\frac{\vdash e_z : \tau \qquad x : \texttt{Nat} \vdash e_s : \tau \qquad \vdash n : \texttt{Nat}}{\vdash \texttt{rec}_{\texttt{Nat}}(e_z, x.e_s)(\texttt{succ}(n)) \equiv e_s[\texttt{rec}_{\texttt{Nat}}(e_z, x.e_s)(n)/x] : \tau}\ \beta_s^{\texttt{rec}_{\texttt{Nat}}}$$

Finally, the $\eta$ rule says that any expression $e$ that satisfies the above equations is equal to the corresponding recursor application:

$$\frac{\begin{array}{ll} y : \mathtt{Nat} \vdash e : \tau & \vdash e[\mathtt{zero}/y] \equiv e_z : \tau \\ n : \mathtt{Nat} \vdash e[\mathtt{succ}(n)/y] \equiv e_s[e/x] : \tau & \end{array}}{y : \mathtt{Nat} \vdash e \equiv \mathtt{rec}_{\mathtt{Nat}}(e_z, x.e_s)(y) : \tau}\ \eta^{\mathtt{rec}_{\mathtt{Nat}}}$$

As with Booleans, categorifying the rules for natural numbers yields an object representing the type `Nat`, a morphism for each constructor, and, for each pair of "combining-function" morphisms running "parallel to" the constructors, a unique morphism denoted by the recursor.[4] In the case of natural numbers, however, the resulting construction has a special name: the *natural numbers object*.

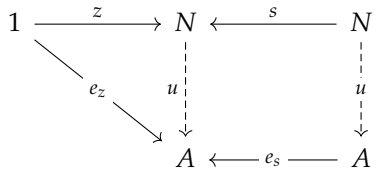
[4] Note that we obtain the uniqueness condition from the $\eta$ law by recalling that substitution is interpreted as composition. Like the corresponding condition for `Bool`, this uniqueness condition says that any morphism $e$ satisfying the $\beta$-equalities $e \circ \mathtt{zero} = e_z$ and $e \circ \mathtt{succ} = e_s$ is equal to the corresponding recursor morphism.

**Definition 102: Natural numbers object**

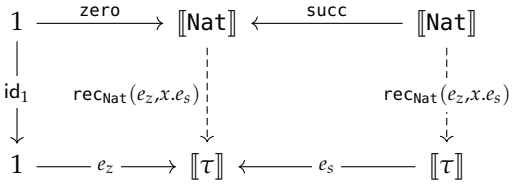
A *natural numbers object* in a Cartesian closed category $\mathcal{C}$ consists of

- An object $N$ of $\mathcal{C}$, and
- Morphisms $z : 1 \to N$ and $s : N \to N$,

such that, for every object $A$ of $\mathcal{C}$ and pair of morphisms $f_z : 1 \to A$ and $f_s : A \to A$, there exists a unique morphism $u$ such that the following diagram commutes:




Here, $u$ is $\mathtt{rec}_{\mathtt{Nat}}(e_z, x.e_s)$. As before, it will be helpful to instead view this diagram as a pair of squares:




(20.2)

**Exercise 20.2**

Give a categorical interpretation of the type of *extended natural numbers*, i.e., the type of natural numbers extended with an extra

element representing infinity, defined as follows:

```
inductive ExtNat : Type where
  | zero : ExtNat
  | succ : ExtNat -> ExtNat
  | inf  : ExtNat
```

*Hint:* The resulting diagram will look a bit different from the two preceding examples.

### 20.1.4 A General Categorification of Inductive Types

Let us now take stock of the preceding examples and generalize what we have seen. In each of our translations above, we did the following:

- First, interpret the constructors of an inductive type as morphisms from the interpretation of the constructor's domain to the type's denotation. That is, a constructor $\mathtt{c} : \mathtt{D} \to \mathtt{T}$ of $\mathtt{T}$ becomes a morphism $[\![\mathtt{D}]\!] \xrightarrow{\mathtt{c}} [\![\mathtt{T}]\!]$.

- Then, interpret the recursor as a unique morphism $[\![\mathtt{T}]\!] \to A$ for every collection of morphisms into $A$ "parallel to the constructors," i.e., morphisms $t_i$ that map into $A$ from the domain of the $i$th constructor where $[\![\mathtt{T}]\!]$ has been replaced by $A$. For instance, if $\mathtt{T}$ has two constructors

  $$\mathtt{c}_1 : \mathtt{Bool} \to \mathtt{T} \quad \text{and} \quad \mathtt{c}_2 : \mathtt{T} \times \mathtt{Nat} \to \mathtt{T},$$

  its recursor is interpreted as a unique morphism $u_{f_1,f_2} : [\![\mathtt{T}]\!] \to A$ for all objects $A$ and morphisms

  $$f_1 : [\![\mathtt{Bool}]\!] \to A \quad \text{and} \quad f_2 : A \times [\![\mathtt{Nat}]\!] \to A.$$

  so that the following squares commute:[5]

[5] Given morphisms $f : X \to Z$ and $g : Y \to W$, we write $f \times g$ to denote the morphism $\langle f \circ \pi_1, g \circ \pi_2 \rangle : X \times Y \to Z \times W$.

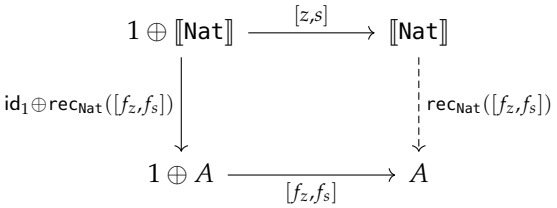
$$\begin{array}{ccccc}
[\![\mathtt{Bool}]\!] & \xrightarrow{c_1} & [\![\mathtt{T}]\!] & \xleftarrow{c_2} & [\![\mathtt{T}]\!] \times [\![\mathtt{Nat}]\!] \\
\downarrow{\scriptstyle \mathsf{id}_{[\![\mathtt{Bool}]\!]}} & & \downarrow{\scriptstyle u_{f_1,f_2}} & & \downarrow{\scriptstyle u_{f_1,f_2} \times \mathsf{id}_{[\![\mathtt{Nat}]\!]}} \\
[\![\mathtt{Bool}]\!] & \xrightarrow{f_1} & A & \xleftarrow{f_2} & A \times [\![\mathtt{Bool}]\!]
\end{array}$$

This procedure is a useful heuristic but is not yet a precise categorical account of an inductive type. For instance, the "replacement" performed in the second step is merely a syntactic operation. We will now make this categorical translation more precise.

We start by consolidating our commutative squares. Observe that any inductive type with multiple constructors can be equivalently expressed as a having a single constructor whose domain is the sum of the domains of the original constructors. That is, the constructors

$c_1 : A \to T$, $c_2 : B \to T$, and $c_3 : C \to T$ can be equivalently represented by a single constructor $c : A \oplus B \oplus C \to T$. The recursor is then parameterized by a single morphism running "parallel" to the constructor that maps into $A$ from an analogous coproduct over $A$.

Applying this rewriting to the `Nat` datatype depicted in Diagram (20.2), we obtain




and a similar transformation occurs for Diagram (20.1).[6]

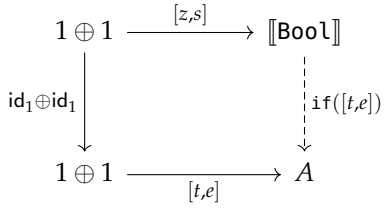
The top-left corner should look familiar: it is the body of the recursive type equation that defines the natural numbers. That is, using a least-fixed-point operator $\mu$, we would write

$$\texttt{Nat} \triangleq \mu t.\, 1 \oplus t,$$

so that `Nat` is the (smallest) type satisfying the type isomorphism

$$\texttt{Nat} \cong 1 \oplus \texttt{Nat}.$$

Similarly, Diagram (20.1) is "consolidated" into a single square with a constructor with domain $1 \oplus 1$:




With this consolidation, every inductive type now translates to a single object (the top-left corner of the square), a single morphism (the constructor), and a universal property that translates the recursor. The "shape" of the objects on the left-hand side of the commutative square are determined by the "shape" of the inductive type. More precisely, using the observation in the preceding paragraph, that "shape" is determined precisely by the expression $\tau$ in the fixed-point equation $\mu t.\, \tau$ for the inductive type.

Adopting a categorical lens, we might observe that an expression $t.\, \tau$ defines a mapping from a type $t$ to some type $\tau$ parameterized thereby. In our categorical interpretation, types form a category; thus, such a mapping from types to types ought to correspond to a functor. More specifically, since we are mapping from a category (of types)

[6] Above and hereafter, we use the notation $[f, g]$ to denote the unique morphism obtained from the universal property of coproducts. Thinking of morphisms as functions, $[f, g]$ does a case split on its input, mapping elements of the left-hand summand using $f$ and those of the right-hand summand using $g$. By analogy with $f \times g$, we write $f \oplus g$ for $[\mathsf{inl} \circ f, \mathsf{inr} \circ g]$.

into itself, this is an *endofunctor*. The endofunctor associated with the type of natural numbers is defined by

$$F_{\mathtt{Nat}}(X) = 1 \oplus X$$
$$F_{\mathtt{Nat}}(f : X \to Y) = \mathsf{id}_1 \oplus f : 1 \oplus X \to 1 \oplus Y$$

Of course, the mapping $t.\ \tau$ is not itself a type; rather, we apply the least-fixed-point operator $\mu$ to obtain an inductive type. We should expect a corresponding maneuver in the world of categories: we interpret an inductive type as the *least fixed point* $\mu F$ of an endofunctor $F$ on our semantic domain: i.e., the "least" object $\mu F$ satisfying $F(\mu F) = \mu F$.

This naturally invites the following question: in the absence of an *a priori* ordering on objects, how do we make precise the fact that $\mu F$ should be "least" among the fixed points of $F$? Appealing to our categorical intuition, we should expect this to be some kind of initiality or colimiting property (recall that, using our order-theoretic lens, we think of initial objects as "minimal"). And, in fact, as we will now demonstrate, we have already seen such a property in our Boolean and natural-number examples!

The key is to recall that an inductive type is defined not only by an object but also by a morphism representing its constructor that tells us how to "map into" the type. Thus, instead of merely considering the "smallest" object among the fixed points of $F$, we should instead consider the "smallest" *pair* of an $F$-fixed point *and* its corresponding constructor morphism. Such a pair is an example of an *algebra* over the endofunctor $F$, also known as an *$F$-algbera*.

**Definition 103: $F$-Algebra**

Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. An *$F$-algebra*, or *algebra over $F$*, is a pair $(A, \alpha)$ where $A$ is an object in $\mathcal{C}$ and $\alpha : FA \to A$ is a morphism.

$A$ is called the *carrier* of the $F$-algebra, and $\alpha$ is the *structure map*. Note that the structure map $\alpha$ alone is sometimes referred to as an $F$-algebra, leaving $A$ implicit.

We are looking for the "smallest" $F$-algebra $(A, \alpha)$ such that its carrier $A$ is a fixed point. Let us first, however, consider simply the "smallest" $F$-algebra, without any further restrictions. We do this because it turns out that there is a natural interpretation of "smallest" in terms of initiality. In particular, $F$-algebras form a category $\mathsf{Alg}_F$, so the "smallest" $F$-algebra is simply the initial object in this category.

**Definition 104: Category of $F$-algebras**

Fix an endofunctor $F : \mathcal{C} \to \mathcal{C}$. The *category of $F$-algebras* (or *category of algebras over $F$*) is defined by the following:

- Objects are $F$-algebras $(A, \alpha : FA \to A)$.
- Morphisms from $(A, \alpha)$ to $(B, \beta)$ are morphisms $f : A \to B$ in $\mathcal{C}$ such that the following square commutes:

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ A & \xrightarrow[f]{} & B \end{array}$$

- Identity and composition are lifted from $\mathcal{C}$.

**Definition 105: Initial $F$-algebra**

Fix an endofunctor $F : \mathcal{C} \to \mathcal{C}$. The *initial $F$-algebra* (or *initial algebra over $F$*), if it exists, is the initial object in the category of $F$-algebras.

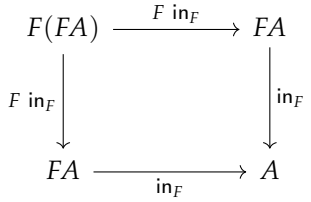
We write $\mathsf{in}_F$ for the (structure map of) the initial $F$-algebra. Given another $F$-algebra $(A, \alpha)$, we write $(\!|\alpha|\!)_F$ (or just $(\!|\alpha|\!)$ if $F$ is clear from context) for the unique morphism from the initial object to $(A, \alpha)$.

We have thus found a suitable notion of a "smallest" $F$-algebra. But what about the smallest $F$-algebra whose carrier is a fixed point of $F$? In fact, these are one and the same (up to isomorphism): it will always be the case that the carrier of the initial $F$-algebra is (isomorphic to) a fixed point of $F$. In other words, if $(A, \alpha)$ is the initial $F$-algebra, then $FA \cong A$. This is the content of *Lambek's lemma*.

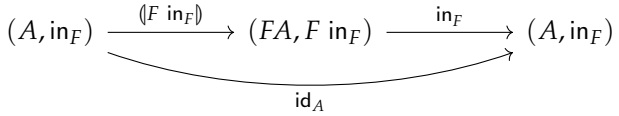
**Proposition 20.1.1** (Lambek's lemma)**.** Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. If $(A, \mathsf{in}_F : FA \to A)$ is the initial $F$-algebra, then $\mathsf{in}_F$ is an isomorphism.

*Proof.* Let $(A, \mathsf{in}_F : FA \to A)$ be the initial $F$-algebra. We will show that $\mathsf{in}_F$ is an isomorphism by showing that its inverse is $(\!|F\ \mathsf{in}_F|\!) : A \to FA$, the unique $F$-algebra morphism $(A, \mathsf{in}_F) \to (FA, F\ \mathsf{in}_F)$ induced by initiality.

First, observe that $\mathsf{in}_F$ is an $F$-algebra morphism $(FA, F\ \mathsf{in}_F) \to (A, \mathsf{in}_F)$, since the following diagram trivially commutes:




Thus, we have the following diagram in $\mathsf{Alg}_F$:



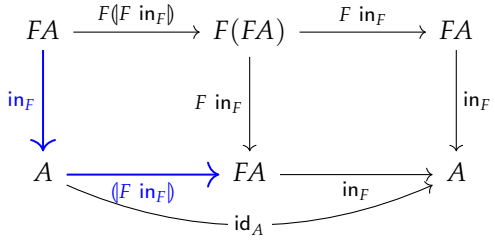

(20.3)

Since $(A, \mathsf{in}_F)$ is initial, there is only one $F$-algebra morphism from $(A, \mathsf{in}_F)$ to itself, so this diagram must commute, i.e.,

$$\mathsf{in}_F \circ (\!|F\ \mathsf{in}_F|\!) = \mathsf{id}_A. \tag{20.4}$$

Accordingly, $(\!|F\ \mathsf{in}_F|\!)$ is a right inverse for $\mathsf{in}_F$.

To show that it is also a left inverse, we first expand the $F$-algebra morphisms in Diagram (20.3) into commutative squares:




In order to show that $(\!|F\ \mathsf{in}_F|\!)$ is a left inverse of $\mathsf{in}_F$, we must show that the highlighted lower-left path through the left square is equal to $\mathsf{id}_A$.

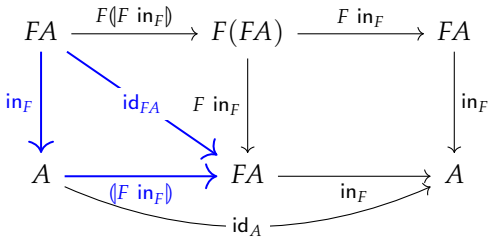
Observe, however, that the upper-right path through the same square is $F\ \mathsf{in}_F \circ F(\!|F\ \mathsf{in}_F|\!)$, which we can rewrite as follows by functoriality:

$$\begin{aligned} F\ \mathsf{in}_F \circ F(\!|F\ \mathsf{in}_F|\!) &= F\ (\mathsf{in}_F \circ (\!|F\ \mathsf{in}_F|\!)) \\ &= F\ \mathsf{id}_A && \text{(Equation 20.4)} \\ &= \mathsf{id}_{FA}. \end{aligned}$$

Adding this fact to our diagram, we can read off the desired equation

$$(\!|F\ \mathsf{in}_F|\!) \circ \mathsf{in}_F = \mathsf{id}_{FA}$$

by tracing the lower-left path:

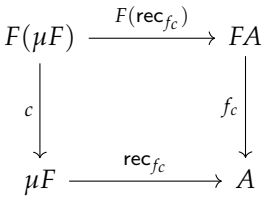



Thus, $\mathsf{in}_F : FA \to A$ is indeed an isomorphism. □

With Lambek's lemma in hand, we can now give a precise categorification of inductive types: the inductive type represented by $\mu t.\, \tau$ is interpreted as the initial algebra of the endofunctor $F(t) = [\![\tau]\!]$, where we extend our previous translation of types to map the free type variable $t$ to the functor variable of the same name.[7] The algebra's carrier $\mu F$ is the object representing the type, while its structure map $\mathsf{in}_F$ represents the type's constructor.

[7] The action of $F$ on morphisms must also be defined in the "obvious" way (which can be derived from viewing products and coproducts of interpreted types in $\mathcal{C}$ as categorical constructions in the functor category $[\mathcal{C}; \mathcal{C}]$). We elide the details, but one can inspect the definition of $F_{\mathtt{Nat}}$ at the beginning of this subsection for an example.

It remains only to precisify our former account of recursors. Recall our previous observation that recursors map out of an inductive type $\mu F$ into some other type $A$ using a "combining morphism" that runs "parallel" to the constructor morphism—i.e., one that maps into $A$ from an object that "replaces" instances of $\mu F$ with instances of $A$. Using our newfound language of functors, we can phrase this much more concisely: a recursor maps from $\mu F$ to some type $A$ using a "combining morphism" $f_c : FA \to A$. Or, as a diagram:




In fact, we already saw this diagram (Diagrams (20.1) and (20.2) are variations of it in the particular cases of natural numbers and Booleans). But having now developed the vocabulary of $F$-algebras, this should look familiar for another reason: recursors are $F$-algebra morphisms! More precisely, given an inductive type $\mu F$, the "unapplied" recursor function is represented by $⦇-⦈_F$; given a particular "combining morphism" $f_c : FA \to A$—which is precisely an $F$-algebra over $A$—the resulting recursor application is $⦇f_c⦈ : \mu F \to A$.

As an example, we translate the parity function on natural numbers, which returns `true` if its input is even and `false` if it is odd, as an application of the natural-number recursor. First, we identify the appropriate "combining function" for this fold. Since we are ulti-

mately producing a Boolean, this function should translate to a morphism of type $F_{\mathtt{Nat}}\,[\![\mathtt{Bool}]\!] \to [\![\mathtt{Bool}]\!]$. We first define its programmatic version below: [8]

$$\begin{aligned} \mathsf{f}_{\mathsf{even}} &: 1 \oplus \mathtt{Bool} \to \mathtt{Bool} \\ \mathsf{f}_{\mathsf{even}}\ (\mathsf{inl}\ ()) &= \mathtt{true} \\ \mathsf{f}_{\mathsf{even}}\ (\mathsf{inr}\ \mathsf{b}) &= \mathtt{not}(\mathsf{b}) \end{aligned}$$

This becomes the $F_{\mathtt{Nat}}$-algebra

$$\alpha_{\mathsf{even}} = [\mathtt{true}, \mathtt{not}]$$

which finally reveals our translation of the parity function to be $(\!|\alpha_{\mathsf{even}}|\!) : [\![\mathtt{Nat}]\!] \to [\![\mathtt{Bool}]\!]$.

Note that `not` is, in turn, defined by an algebra over $F_{\mathtt{Bool}}$, the constant-$(1 \oplus 1)$ functor. In particular, $[\![\mathtt{not}]\!] = (\!|\alpha_{\mathtt{not}}|\!)$ where $\alpha_{\mathtt{not}} : 1 \oplus 1 \to 1 \oplus 1$ is $[\mathsf{inr}, \mathsf{inl}]$.

To put together the developments in this section, let's consider one more example of a categorified inductive datatype, now expressed in terms of initial algebras. Recall the inductive datatype of lists:

```
inductive List (A : Type) : Type where
  | nil : List A
  | cons : A -> List A -> List A
```

Notice that this type is parameterized over a fixed type `A`; we will write $A$ for that type's categorical denotation. We can read the endofunctor of which `List A` is a least fixed point off of its constructors:[9]

$$F_{\mathtt{List_A}}(X) = 1 \oplus A \times X.$$

Lists are thus represented by the initial algebra $(\mu F_{\mathtt{List_A}}, \mathsf{in}_{F_{\mathtt{List_A}}})$. Returning to our example from earlier, we can define the list length function as the fold induced by the "combining function"

$$\begin{aligned} \mathsf{f}_{\mathsf{length}} &: \mathtt{Unit} \oplus \mathtt{A} \times \mathtt{Nat} \to \mathtt{Nat} \\ \mathsf{f}_{\mathsf{length}}\ (\mathsf{inl}\ ()) &= 0 \\ \mathsf{f}_{\mathsf{length}}\ (\mathsf{inr}\ (a, \ell)) &= \mathtt{succ}(\ell) \end{aligned}$$

which we translate into the $F_{\mathtt{List_A}}$-algebra

$$\alpha_{\mathsf{length}} = [z, s \circ \pi_1]$$

to obtain the fold $(\!|\alpha_{\mathsf{length}}|\!) : \mu F_{\mathtt{List_A}} \to [\![\mathtt{Nat}]\!]$.

**Exercise 20.3**

Below, we define a strange type of so-called "self-trees:" each node can have as many children as the parameterizing type has elements

[8] It may be helpful to compare this definition to the following equivalent, structurally recursive definition in a more familiar programmatic style:

```
def even : Nat -> Bool
  | 0 => true
  | n + 1 => not (even n)
```

Similarly, `not` would be defined as:

```
def not : Bool -> Bool
  | true => false
  | false => true
```

[9] We again elide the action on morphisms.

(i.e., a `SelfTree A` node can have, as it were, "`A` many children").

```
inductive SelfTree (A : Type) : Type where
  | node : A -> (A -> SelfTree A) -> SelfTree A
  | leaf : SelfTree A
```

1. Specify the functor $F_{\texttt{SelfTree}}$ that characterizes this type.
2. We define the "self-height" of a self-tree to be the number of nodes encountered by successively visiting the $a$th child of a node containing $a$. (The self-height of a leaf is 0.) Categorify this operation using a suitable $F_{\texttt{SelfTree}}$-algebra.

**Exercise 20.4**

Show that the morphism `app` induced by the universal property of exponentials is the structure map for an algebra whose carrier is the initial object. Use this to give a non-empty definition of an inductive type (i.e., an `inductive` declaration listing at least one putative constructor) that is isomorphic to the empty type.

*Hint:* Recall that in a CCC, $0^A \cong 0 \cong 0 \times A$.

### *20.1.5 On the Existence of Initial Algebras*

Up to this point, we have simply assumed that, given an endofunctor $F$, we can always find an initial $F$-algebra. This is not, however, always the case. And we should expect as much: after all, not all recursive type equations admit a least fixed point—this is why we insist that no non-positive occurrences of $t$ appear in $\tau$ when defining an inductive type $\mu t.\ \tau$. The failure to obtain a least fixed point in a recursive type equation corresponds to an analogous lack of an initial algebra for the corresponding functor. For instance, $\mu t.\ (t \to 2) \to 2$ (i.e., $F(X) = 2^{2^X}$) does not admit a least fixed point in every Cartesian closed category: for one, Lambek's lemma says that this would induce an isomorphism $\mu F \cong 2^{2^{\mu F}}$, which by Cantor's theorem never holds in the CCC Set. In fact, we can more generally prove the nonexistence of certain initial algebras in arbitrary categories, not just Set, but this would take us too far afield. For the purposes of this chapter, it suffices to know that all the usual strictly positive inductive types of interest are characterized by endofunctors that admit initial algebras.

**Exercise 20.5**

There is another way to think about interpreting inductive types.

While we gave an interpretation by translating the various features of inductive types' structure into categorical terms, we could have alternatively followed our previous method and directly translated the generalized inference rules for isorecursively formulated inductive types (cf. Harper, *Practical Foundations for Programming Languages (2nd ed.)*, Ch. 15).

Give a translation of these rules (shown below), and verify that it yields the same construction at which we just arrived.

$$\frac{\vdash t.\tau\ \mathtt{pos}}{e : \tau[\mu t.\tau/t] \vdash \mathtt{fold}(e) : \mu t.\tau}\ \textsc{T-fold}$$

$$\frac{x : \tau[\rho/t] \vdash e_1 : \rho}{e_2 : \mu t.\tau \vdash \mathtt{rec}(x.e_1)(e_2) : \rho}\ \textsc{T-rec}$$

$$\frac{x : \tau[\rho/t] \vdash e_1 : \rho}{e_2 : \tau[\mu t.\tau/t] \vdash \mathtt{rec}(x.e_1)(\mathtt{fold}(e_2)) \equiv e_1[\mathtt{map}(e_2)(y.\mathtt{rec}(x.e_1)(y))/x] : \rho}\ \beta^{\mathtt{rec}}$$

$$\frac{y : \tau[\rho/t] \vdash e : \rho \qquad e_2 : \tau[\mu t.\tau/t] \vdash e[\mathtt{fold}(e_2)/y] \equiv e_1[\mathtt{map}(e_2)(y.\mathtt{rec}(x.e_1)(y))/x]}{y : \mu t.\tau \vdash e \equiv \mathtt{rec}(x.e_1)(y) : \rho}\ \eta^{\mathtt{rec}}$$

Note that these rules introduce two new auxiliary definitions:

- The judgment $\vdash t.\tau\ \mathtt{pos}$ asserts that $t$ has only *positive* occurrences in $\tau$; i.e., $t$ never appears to the left of an arrow. Note that all of the rules implicitly carry this hypothesis, though we have only explicitly stated it for the first.
- The auxiliary function $\mathtt{map}(e)(e')$ "unpacks" the structure of $e$ based on its type and applies $e'$ to each "element" thereof. Its typing rule is as follows (where $t$ may appear free in $\tau$):

$$\frac{\vdash e : \tau[\rho/t] \qquad x : \rho \vdash e' : \rho'}{\vdash \mathtt{map}(e)(x.e') : \tau[\rho'/t]}\ \textsc{T-map}$$

Here are two of its equational rules (it has one for every type

former):

$$\frac{\vdash e : (\tau_1 \times \tau_2)[\rho/t] \qquad x : \rho \vdash e' : \rho'}{\begin{array}{l}\vdash \texttt{map}(e)(e') \equiv \langle \texttt{map}(\texttt{fst}\ e)(x.e'), \texttt{map}(\texttt{snd}\ e)(x.e')\rangle \\ \qquad : \tau_1[\rho'/t] \times \tau_2[\rho'/t]\end{array}}\ \beta^{\texttt{map}}_{\times}$$

$$\frac{\vdash e : (\tau_1 \oplus \tau_2)[\rho/t] \qquad x : \rho \vdash e' : \rho'}{\begin{array}{l}\vdash \texttt{map}(e)(e') \equiv \texttt{case}\ e\ \texttt{of} \\ \qquad \mid \texttt{inl}\ x_1 \mapsto \texttt{inl}\ (\texttt{map}(x_1)(x.e')) \\ \qquad \mid \texttt{inr}\ x_2 \mapsto \texttt{inr}\ (\texttt{map}(x_2)(x.e')) \\ \qquad : \tau_1[\rho'/t] \oplus \tau_2[\rho'/t]\end{array}}\ \beta^{\texttt{map}}_{\oplus}$$

Your translation should give an account of what these definitions mean categorically.

Here are some hints for this exercise:

- If parameterized types are translated as functors, to what does type-variable substitution correspond?
- $\texttt{map}(e)$ will not be translated as a morphism.

## 20.2 *The Families Fibration: Indexing Inductive Types*

Having given a categorical translation of simple inductive types, we now turn our attention to *indexed* inductive types. Recall that indexed inductive types enrich inductive types by allowing types to be parameterized by data values at the type level: for instance, in addition to the simple type `List A` of arbitrary-length lists with elements of type `A`, we can also define the indexed type `Vector A n` of `A`-containing lists of length `n`.

While we gave a generalized presentation of inductive types in terms of initial algebras of endofunctors over arbitrary categories, hereafter we will specifically consider a categorical semantics in terms of sets. While all of the ideas presented can be generalized to other categories, focusing on Set will simplify some of the required categorical machinery. (Note that our preceding interpretation of inductive types can indeed be carried out in Set, since it is Cartesian closed and contains finite coproducts—i.e., it is *bicartesian closed*.)

The category Set alone, however, lacks the structure to give an elegant semantics to indexed inductive types. Instead, we will turn to a new category built atop Set that is equipped with a notion of "indexing" that will afford a straightforward interpretation of our inductive types' indices.[10]

[10] Be careful when reading the following definition and elsewhere in this chapter: the notation $\prod_{a \in A} S(a)$ below refers to a *set-theoretic* indexed product, not a $\Pi$-type. The symbols $\prod$, $\sum$, and $\to$ are regrettably overloaded (we will see yet another denotation of $\sum$ later on), so the meanings of these notations are highly context-sensitive.

**Definition 106: Families fibration**

The *category of families (of sets)* $\mathsf{Fam}(\mathsf{Set})$ is defined as follows:

- Objects are pairs $(A, P : A \to \mathrm{Ob}(\mathsf{Set}))$ of an indexing set $A$ and a *family* of sets $P$ specifying a set $P(a)$ for each index $a \in A$.
- Morphisms from $(A, P)$ to $(B, Q)$ are pairs of functions $(f, f^\sim)$ such that $f : A \to B$ and $f^\sim : \prod_{a \in A} P(a) \to Q(f(a))$.

The forgetful functor $U : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ defined by

$$U(A, P) = A$$
$$U(f, f^\sim) = f$$

is called the *families fibration*.

Note that, while we will not treat them in full generality, *(Grothendieck) fibrations* are objects of particular interest in category theory and characterize a certain kind of structure (which we will attempt to intuitively motivate at several points hereafter). Throughout this chapter, you will encounter definitions phrased as features "of the families fibration:" these should be read as referring to the entire structure of $\mathsf{Fam}(\mathsf{Set})$, $\mathsf{Set}$, and this mapping between them, rather than particular characteristics of a single forgetful functor.

Let's now use the category of families to interpret indexed inductive types. We'll start with the type `BitVec n`, which is like the type `Vector A n` we have previously discussed, except that it contains only Booleans:[11]

[11] `Vector A n` is defined similarly. We choose `BitVec` for this example to avoid any potential confusion between the parameter `A` and the index `n` of `Vector`.

```
inductive BitVec : Nat -> Type
  | bvnil  : BitVec 0
  | bvcons : {n : Nat} -> Bool -> BitVec n -> BitVec (succ n)
```

Drawing inspiration from our preceding development of simple inductive types, we might expect to interpret this type as an endofunctor on $\mathsf{Fam}(\mathsf{Set})$. That endofunctor should produce a family that looks like a representation of the Boolean list type (i.e, the sum of an "empty" object and a Boolean–vector pair object), except that we can only inject into the left-hand side of the sum at index zero and into the right-hand side at successor indices. Concretely, such a functor looks as follows:[12]

[12] We will continue our convention of eliding morphism actions. As before, these mappings are "automatic" from our choice of action on objects.

$$F_{\texttt{BitVec}}(\mathbb{N}, X) = (\mathbb{N}, (n \mapsto \begin{pmatrix} \{\star\} & \text{if } n = 0 \\ \varnothing & \text{else} \end{pmatrix} \oplus \begin{pmatrix} \varnothing & \text{if } n = 0 \\ \{n_1\} \times [\![\texttt{Bool}]\!] \times X(n_1) & \text{if } n = \texttt{succ}(n_1) \end{pmatrix}))$$ (20.5)

But here we encounter a problem: the endofunctor we have just defined only acts on $\mathbb{N}$-indexed families. Therefore, it is *not* an endofunctor on $\mathsf{Fam}(\mathsf{Set})$, whose objects can have arbitrary indexing sets. Yet this $\mathbb{N}$-indexing really is essential: our functor cannot perform a case split on zero and successors if our indexing set does not have such elements! Indeed, this ability to analyze the structure of a type index is precisely one of the key attributes that makes indexed inductive types more powerful than mere parametric polymorphism.

Therefore, to represent an indexed inductive type, we take the initial algebra of an endofunctor not on $\mathsf{Fam}(\mathsf{Set})$ itself, but on a restriction of $\mathsf{Fam}(\mathsf{Set})$ where we consider only families indexed by the indexing set of the indexed type of interest. This smaller category is called the *fiber* of the families fibration over $A$.

**Definition 107: Fiber of the families fibration over $A$**

The *fiber* of the families fibration over $A$, written $\mathsf{Fam}(\mathsf{Set})_A$, is a subcategory of $\mathsf{Fam}(\mathsf{Set})$ in which all objects have indexing set $A$. Precisely:

- Objects are families $(A, P : A \to \mathrm{Ob}(\mathsf{Set}))$
- Morphisms are pairs $(\mathrm{id}_A, f^\sim : \prod_{a \in A} P(a) \to Q(a))$

That is, $\mathsf{Fam}(\mathsf{Set})_A$ is the preimage of $\mathsf{id}_A$ under $U$.

Thus, although it will at times be useful to consider the category $\mathsf{Fam}(\mathsf{Set})$ in its entirety, we will represent $A$-indexed inductive types by initial algebras of endofunctors on just the fiber of the families fibration over $A$. As before, the morphisms induced by initiality correspond to folds using the target algebras' structure maps.

**Exercise 20.6**

Identify the endofunctor on $\mathsf{Fam}(\mathsf{Set})_{\mathbb{N}}$ that characterizes `Fin n`, the type of "natural numbers less than $n$" (or, isomorphically, $\mathbb{Z}/n\mathbb{Z}$):

```
inductive Fin : Nat -> Type where
```

```
  | finZero : {n : Nat} -> Fin (succ n)
  | finSucc : {n : Nat} -> Fin n -> Fin (succ n)
```

*Hint:* Notice that, for a given index, there maybe multiple (or no) applicable constructors.

Notice that we can also interpret refinements of simple types in the category of families. If $T = \mu F$ is an inductive type, then given an $A$-valued combining function $f$ for $T$ (i.e., an $F$-algebra $f$ with carrier $A$) and a value $a : A$, the refinement of $T$ where $f$ yields value $a$, i.e.,

$$\texttt{\{t : T | rec}_{\texttt{A}}\texttt{(f)(t) = a\}}$$

can be translated as the family

$$(A, a \mapsto \{x \in \mu F \mid (\!|f|\!)_F(x) = a\}).$$

This is the "naïve" subset interpretation of refinements that we discussed previously. Our objective will be to turn this family into one that is the denotation of an indexed inductive type.

## 20.3 Constructing a Refinement Translation

Our basic recipe for the translation of $\texttt{\{t : T | rec}_{\texttt{A}}\texttt{(f)(t) = a\}}$ into an indexed inductive type will be as follows:

1. Lift $F$, an endofunctor on $\mathsf{Set}$, to an endofunctor $\hat{F}_A$ on $\mathsf{Fam}(\mathsf{Set})_A$.
2. Restrict the family produced by our lifting to those elements which are mapped to the appropriate value by $\texttt{rec}_{\texttt{A}}\texttt{(f)}$.

This recipe will require a number of auxiliary constructions, on which we now embark.

### 20.3.1 Another View of Families

For some of the constructions we give in this chapter, it will be helpful to have in mind an alternative perspective on $\mathsf{Fam}(\mathsf{Set})$. This perspective is inspired by the Curry–Howard correspondence between propositions and types.

Recall that the Curry–Howard correspondence suggests another way to interpret indexed inductive types: as *predicates* on the indexing type. For instance, we can define what it means for a natural number to be even by declaring a type of *proofs* of evenness:

```
inductive Even : Nat -> Type where
  | evZero : Even 0
  | evSucc : {n : Nat} -> Even n -> Even (succ (succ n))
```

A natural number n is even if the type Even n is inhabited. Correpsondingly, the interpretation of this inductive type as a family of sets $(\mathbb{N}, P_{\mathsf{Even}})$ can be thought of as defining a *predicate* $P_{\mathsf{Even}}$ on natural numbers. With this interpretation, we think of a family $(A, P)$ as containing proofs $P(a)$ that the represented predicate holds of elements $a \in A$. In particular, if $P(a)$ is inhabited, than the predicate holds of $a$; if it is empty, the predicate does not.

Adopting the logical perspective on families suggested by the Curry–Howard correspondence, it is natural to consider the family that represents the constantly-true predicate on a given set, known as the set's *truth predicate*.

**Definition 108: Truth predicate and truth functor**

Given a set $A$, the *truth predicate* for $A$ is the family $(A, a \mapsto 1)$, where 1 is the terminal object in Set (i.e., $\{\star\}$). The truth predicate is terminal in $\mathsf{Fam}(\mathsf{Set})_A$.

The functor $\top : \mathsf{Set} \to \mathsf{Fam}(\mathsf{Set})$ that sends sets to their truth objects (and maps morphisms by pairing them with $\_ \mapsto \mathsf{id}_1$) is called the *truth functor* for the families fibration.

### 20.3.2 *Comprehension and Reindexing*

Returning to our "recipe" from earlier, our first order of business is, given an endofunctor $F : \mathsf{Set} \to \mathsf{Set}$ with an initial algebra, to define an endofunctor $\hat{F}_A : \mathsf{Fam}(\mathsf{Set})_A \to \mathsf{Fam}(\mathsf{Set})_A$ whose initial algebra has a carrier representing an $A$-indexed type that "looks like" the inductive type characterized by $F$.

To define a translation between Set and Fam(Set), we need to be able to move between these two categories. We have already seen one functor in each direction: the truth predicate $\top : \mathsf{Set} \to \mathsf{Fam}(\mathsf{Set})$ maps sets to the trivial family thereover, while the families fibration $U : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ discards the family associated with an indexing set. We will now introduce one additional functor in the backward direction, which exploits the untypedness of Set to "collapse" a family in Fam(Set) into a single object in Set.

**Definition 109: Comprehension functor**

The *comprehension* of a family $(A, P)$, written $\{(A, P)\}$, is the set $\sum_{a \in A} P(a)$. We correspondingly define the *comprehension functor* $\{-\} : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$, which sends a family to its comprehension and a pair of morphisms to a morphism on pairs.

The comprehension functor $\{-\}$ and forgetful functor $U$ are re-

lated by the natural transformation $\pi : \{-\} \Rightarrow U$ whose morphisms project (hence the notation $\pi$) the index from an element of a ("comprehended") indexed family:

$$\pi_{(A,P)}(a, p) = a$$

We will also require means of mapping between $\mathsf{Fam}(\mathsf{Set})_A$ and $\mathsf{Fam}(\mathsf{Set})_B$ for distinct indexing sets $A$ and $B$. Intuitively, this is because we interpret our indexed inductive types as endofunctors only over a specific fiber, so we will need to move between fibers in order to, for instance, fold over an indexed inductive type into a distinct output type. This will be accomplished using another functor, known as the *op-reindexing functor*.

**Definition 110: Indexed coproduct**

Let $f : A \to B$ be a function, and let $(A, P)$ be a family of sets. The *indexed coproduct of* $(A, P)$ *along* $f$, written $\Sigma_f(A, P)$, is defined as

$$\left(B, b \mapsto \sum_{a \in A} (b = f(a)) \times P(a)\right)$$

where $(b = f(a))$ is a set that is inhabited if the equality holds and uninhabited if not.

This definition induces the *op-reindexing functor* $\sum_f : \mathsf{Fam}(\mathsf{Set})_A \to \mathsf{Fam}(\mathsf{Set})_B$, whose action on morphisms lifts functions to act on the relevant component of tuples:

$$\Sigma_f(\mathrm{id}_A, g^\sim) = (\mathrm{id}_B, b \mapsto (a, h, p) \mapsto (a, h, g^\sim(a)(p)))$$

In other words, $\Sigma_f(A, P)$ maps a given $b$ in the image of $f$ to the members of the family $P$ at each element of $b$'s preimage. Indeed, an isomorphic (and perhaps more intuitive) characterization of $\Sigma_f(A, P)$ is given by

$$\left(B, b \mapsto \sum_{a \in f^{-1}[\{b\}]} P(a)\right). \tag{20.6}$$

We summarize the developments of this section in the diagram below: the category of families sits "above" the category of sets, with each fiber situated atop its corresponding index set. As discussed previously, we have functors running in "orthogonal" directions, with $U$, $\top$, and $\{-\}$ mapping between $\mathsf{Fam}(\mathsf{Set})$ and $\mathsf{Set}$ while $\Sigma_f$ reindexes between fibers. We will have more to say about this diagram later.

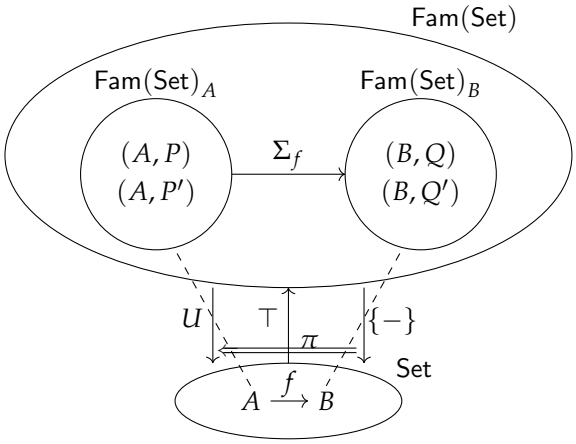



Figure 20.1: A summary of functors to be used in our translation.

**Exercise 20.7**

Fix a family $(A, P)$. Let $f : A \to B$ be a function. Consider the following diagram, where $\psi : (A, P) \to \Sigma_f(A, P)$ acts in the straightforward way:

$$\begin{array}{ccc} \{(A,P)\} & \xrightarrow{\{\psi\}} & \{\Sigma_f(A,P)\} \\ \downarrow{\scriptstyle \pi_{(A,P)}} & & \downarrow{\scriptstyle \pi_{\Sigma_f(A,P)}} \\ A & \xrightarrow[f]{} & B \end{array}$$

1. Show that this diagram commutes.
2. Show that $\{\psi\}$ is an isomorphism. (This shows that the families fibration has *very strong coproducts*.)

### *20.3.3 Liftings*

We now have all the tools required to define a lifting of an endofunctor $F$ on Set to an endofunctor $\hat{F}$ on Fam(Set), which we may then restrict to Fam(Set)$_A$. (It makes sense that we can lift "all the way" to Fam(Set), rather than just a particular fiber Fam(Set)$_A$, since the type whose endofunctor we are lifting does not depend on a particular index.) Before we attempt to define such a lifting, however, we should first set out the desiderata we expect such a construction to satisfy.

As a basic criterion, we ought to expect that the indexing set produced by our "lifted" endofunctor $\hat{F}$ is the set obtained by simply applying $F$ to the original indexing set. Indeed, we can now drop

the scare quotes: this is precisely the definition of the lifting of an endofunctor to $\mathsf{Fam}(\mathsf{Set})$.

**Definition 111: Lifting**

A *lifting* of an endofunctor $F : \mathsf{Set} \to \mathsf{Set}$ is an endofunctor $\hat{F} : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Fam}(\mathsf{Set})$ such that $F \circ U = U \circ \hat{F}$.

However, this criterion says nothing about the behavior of the *family* produced by $\hat{F}$ itself. Adopting the logical view again, we can formulate one such criterion: $\hat{F}$ should "respect truth." That is, if we think of $F(A)$ (given some $A$) as implicitly defining a predicate (namely, one which holds of all its elements), then $\hat{F}$ should yield the same predicate—i.e., it should yield a predicate that holds of the same values—when "applied to" $A$. Of course, this intuitive phrasing does not type-check: $F(A)$ is not a predicate (it is a set), and $\hat{F}$ cannot be applied to $A$ (since $A$ is not a family). But we can fill in some straightforward coercions to make this definition—which characterizes a so-called *truth-preserving* lifting—precise: to treat the objects in the codomain of $F$ as predicates, we post-compose the truth functor; to allow $\hat{F}$ to act on sets, we pre-compose the truth functor; and the notion that these functors always yield the "same predicate" as each other is captured by natural isomorphism.

**Definition 112: Truth-preserving lifting**

Let $F : \mathsf{Set} \to \mathsf{Set}$ be an endofunctor. A lifting of $F$ is *truth-preserving* if there exists a natural isomorphism $\top \circ F \cong \hat{F} \circ \top$.

**Exercise 20.8**

Verify that

$$\hat{F}(A, P) = (FA, \_ \mapsto 1)$$
$$\hat{F}(f, f^{\sim}) = (Ff, \_ \mapsto \mathrm{id}_1)$$

defines a trivial truth-preserving lifting of $F : \mathsf{Set} \to \mathsf{Set}$.

The truth-preserving lifting above, however, is degenerate in the sense that it entirely forgets the structure of the original family. We therefore instead define the following truth-preserving lifting:[13]

$$\hat{F}(A, P) = \Sigma_{F\pi_{(A,P)}} \top (F\{(A, P)\})$$

Unlike in the exercise, we incorporate the "structure of $P$ under $F$" by applying $F$ to the *comprehension* of the original family. We then

[13] One can find a description of the lifting's action on morphisms, along with a proof that it is truth-preserving, in [13].

re-separate the indexing set $FA$ and associated family "under $F$" by taking an indexed coproduct along $F\pi_{(A,P)}$. Thus, the resulting family can be seen as containing those elements which "wrap the structure of $P$ in $F$" (or, from a logical perspective, the resulting predicate holds only of those elements of $FA$ whose "contents" in $A$ were validated by $P$). Unfolding definitions makes these intuitions more apparent:

$$\begin{aligned}
\hat{F}(A,P) &= \Sigma_{F\pi_{(A,P)}} \top (F\{(A,P)\}) \\
&= \Sigma_{F\pi_{(A,P)}} (F\{(A,P)\}, _ \mapsto 1) \\
&= \left( FA, x \mapsto \sum_{y \in F\{(A,P)\}} (F\pi_{(A,P)}(y) = x) \times 1 \right) \\
&\cong (FA, x \mapsto \{y \in F\{(A,P)\} \mid (F\pi_{(A,P)})(y) = x\})
\end{aligned}$$

As a concrete example, consider the truth-preserving lifting of $F_{\mathsf{Nat}}$:

$$\begin{aligned}
\hat{F}_{\mathsf{Nat}}(A,P) &= \Sigma_{F_{\mathsf{Nat}}} \top (\hat{F}_{\mathsf{Nat}}\{(A,P)\}) \\
&= \Sigma_{F_{\mathsf{Nat}}\pi_{(A,P)}} (1 \oplus \{(A,P)\}, _ \mapsto 1) \\
&\cong \left( 1 \oplus A,\ b \mapsto \sum_{a \in F_{\mathsf{Nat}}\pi^{-1}_{(A,P)}[\{b\}]} 1 \right) \\
&\cong \left( 1 \oplus A, \begin{cases} \mathsf{inl}\ () \mapsto \mathsf{inl}\ () \\ \mathsf{inr}\ a \mapsto \{\mathsf{inr}\ ((a,p),\star) \mid p \in P(a)\} \end{cases} \right) \\
&\cong \left( 1 \oplus A, b \mapsto \begin{pmatrix} 1 & \text{if } b = \mathsf{inl}\ () \\ \varnothing & \text{if } b = \mathsf{inr}\ a \end{pmatrix} \oplus \begin{pmatrix} \varnothing & \text{if } b = \mathsf{inl}\ () \\ \{a\} \times P(a) & \text{if } b = \mathsf{inr}\ a \end{pmatrix} \right)
\end{aligned}$$

Finally, since our objective is ultimately to obtain an endofunctor on $\mathsf{Fam}(\mathsf{Set})_A$—not $\mathsf{Fam}(\mathsf{Set})$—we define the functor $\hat{F}_A$ to be the restriction of $\hat{F}$ to $\mathsf{Fam}(\mathsf{Set})_A$. The attentive reader may notice that that this is not an endofunctor: $\hat{F}_A$ maps from the fiber $\mathsf{Fam}(\mathsf{Set})_A$ to the fiber $\mathsf{Fam}(\mathsf{Set})_{FA}$. However, it is the first half of a construction that will yield the endofunctor we desire; we will fill in the remaining piece that takes us back to $\mathsf{Fam}(\mathsf{Set})_A$ in the following section.

### *20.3.4 Translating Refinements*

Recalling our original objective—to translate type refinements into indexed inductive types—we can see that truth-preserving liftings get us halfway to our goal: they transform the endofunctors that

characterize simple inductive types into analogous endofunctors on families. But our aim is not to produce an indexed inductive type that corresponds to the unrestricted simple one, but rather some refinement thereof. This refinement translation is the last piece we must develop.

In fact, we already have the tool we need to do this: op-reindexing. Recall that indexed coproducts (or, more generally, the op-reindexing functor) let us take a family and reindex it, such that the resulting family contains only those elements which are mapped by the reindexing function to a given value in its codomain. This is precisely what we do when we refine a datatype: given `P : T → A`, the type `{t : T | P(t) = a}` is an `A`-indexed type that, at a given index `a`, consists of the elements of `T` on which `P` produces value `a`. In other words, op-reindexing by $f$ corresponds precisely to refinement by $f$.

Therefore, the construction to which we have building all this time can finally be written as follows:

**Definition 113: Refinement translation**

Let $F : \mathsf{Set} \to \mathsf{Set}$ be an endofunctor characterizing the inductive type $\mu F$. Let $(A, \alpha : FA \to A)$ be an $F$-algebra. We define the translation of the refinement type characterized by the family

$$(A, a \mapsto \{x \in \mu F \mid (\!|\alpha|\!)_F(x) = a\})$$

as the indexed inductive type $\mu F^\alpha$ characterized by the endofunctor $F^\alpha : \mathsf{Fam}(\mathsf{Set})_A \to \mathsf{Fam}(\mathsf{Set})_A$ such that

$$F^\alpha = \Sigma_\alpha \circ \hat{F}_A.$$

Once again, it is illuminating to expand this functor's action on objects:

$$\begin{aligned}
F^\alpha(A, P) &= (\Sigma_A \circ \hat{F}_A)(A, P) \\
&\cong \Sigma_\alpha(FA, x \mapsto \{y \in F\{(A, P)\} \mid (F\pi_{(A,P)})(y) = x\}) \\
&\qquad\qquad \text{(As before)} \\
&= \left(A, a \mapsto \sum_{x \in FA} (a = \alpha(x)) \times \{y \in F\{(A, P)\} \mid (F\pi_{(A,P)})(y) = x\}\right) \\
&\cong (A, a \mapsto \{y \in F\{(A, P)\} \mid \alpha((F\pi_{(A,P)})(y)) = a\}) \qquad (20.7)
\end{aligned}$$

Upon cursory inspection, at least, the general shape of this family should seem reasonable: it is $A$-indexed, with elements drawn from a set of "$F$-shaped objects" restricted to those "on which" (after a certain projection) $\alpha$ attains value $a$. To grasp the significance of this

final expression more completely, it may be helpful to return to it after observing the example below.

We illustrate this construction in practice with our example from the beginning: deriving the type of length-indexed vectors from the refinement of lists by their length. To simplify, we will consider just lists of Booleans, and attempt to derive the type `BitVec` from before.

First, letting $2 \cong 1 \oplus 1$ be the object denoted by the Boolean type, we recall the definition of $F_{\texttt{BitList}} : \mathsf{Set} \to \mathsf{Set}$ and the algebra $(\mathbb{N}, \texttt{length})$ thereover:

$$\begin{aligned} F_{\texttt{BitList}}(X) &= 1 \oplus 2 \times X \\ F_{\texttt{BitList}}(f) &= \mathrm{id}_1 \oplus (\mathrm{id}_2 \times f) \end{aligned}$$

$$\begin{aligned} \texttt{length}\ (\mathsf{inl}\ ()) &= 0 \\ \texttt{length}\ (\mathsf{inr}\ (b, \ell)) &= \texttt{succ}(\ell) \end{aligned}$$

We now compute the corresponding indexed inductive type. Note that we use the isomorphic expansion of $F^\alpha$ from Equation (20.7) to avoid re-unfolding definitions:

$$\begin{aligned} & F^{\texttt{length}}_{\texttt{BitList}}(\mathbb{N}, P) \\ &\cong (\mathbb{N}, n \mapsto \{y \in F_{\texttt{BitList}}\{(\mathbb{N}, P)\} \mid \texttt{length}((F_{\texttt{BitList}}\pi_{(\mathbb{N},P)})(y)) = n\}) \\ &= (\mathbb{N}, n \mapsto \{y \in 1 \oplus 2 \times (\mathbb{N}, P) \mid (y = \mathsf{inl}\ () \text{ and } 0 = n) \text{ or} \\ &\qquad\qquad (y = \mathsf{inr}\ (b, (n_1, p)) \text{ and } \texttt{succ}(n_1) = n)\}) \\ &= \left(\mathbb{N}, \begin{cases} 0 \mapsto \{\mathsf{inl}\ ()\} \\ \texttt{succ}(n_1) \mapsto \{\mathsf{inr}\ (b, (n_1, p)) \mid b \in 2, p \in P(n_1)\} \end{cases}\right) \\ &= (\mathbb{N}, (n \mapsto \begin{pmatrix} 1 & \text{if } n = 0 \\ \varnothing & \text{else} \end{pmatrix} \oplus \\ &\qquad\qquad \begin{pmatrix} \varnothing & \text{if } n = 0 \\ 2 \times (\{n_1\} \times P(n_1)) & \text{if } n = \texttt{succ}(n_1) \end{pmatrix})) \end{aligned}$$

Up to isomorphism (specifically, with a minor rearrangement and reassociation in the right-hand summand), this is precisely the functor $F_{\texttt{BitVec}}$ derived in Equation (20.5)!

**Exercise 20.9**

Consider the following inductive type of trees with colored nodes (we refrain from calling these "red-black trees" in prose because they need not maintain the relevant invariants):

```
inductive RBList (A : Type) where
  | leaf  : RBTree
  | red   : A -> RBTree -> RBTree -> RBTree
```

```
4     | black : A -> RBTree -> RBTree -> RBTree
```

1. Define the endofunctor $F_{\texttt{RBTree}} : \mathsf{Set} \to \mathsf{Set}$ such that $[\![\texttt{RBTree}]\!] \cong \mu F_{\texttt{RBTree}}$.
2. Define an $F_{\texttt{RBTree}}$-algebra `bh` that computes the *black height* of an `RBTree`, i.e., the maximum number of black nodes on a path from the root to a leaf. You may assume the existence of a function $\texttt{max} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.
3. Using the above, compute $F^{\texttt{bh}}_{\texttt{RBTree}} : \mathsf{Fam}(\mathsf{Set})_{\mathbb{N}} \to \mathsf{Fam}(\mathsf{Set})_{\mathbb{N}}$, and read off of this expression an indexed inductive type of black height-indexed colored trees. Verify that your definition matches how you would intuitively define this type from scratch.

## *20.4 Correctness*

These illustrative examples are compelling, but we have yet to prove that our foregoing construction always characterizes the indexed inductive type of interest. To begin with, we do not even know that the functor $F^\alpha$ admits an initial algebra in the first place, nor—if it does—whether the carrier of this algebra is in fact isomorphic to our naïve subset formulation of the refinement of $\mu F$ by $\alpha$. In this section, we will sketch the proof of these facts, referring the reader to the paper by Atkey et al. [2] for further details.

### *20.4.1 Preliminaries*

We begin with a few auxiliary lemmas and definitions we will require for our proofs. First, recall our observation from the previous section that the functors $\top : \mathsf{Set} \to \mathsf{Fam}(\mathsf{Set})$ and $U : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ map in oppposite directions. In fact, we can say more: these form an adjoint pair.

**Lemma 20.4.1.** The functors $U \dashv \top$ form an adjoint pair.

*Proof.* We must show $\mathsf{Set}(U(A,P),B) \cong \mathsf{Fam}(\mathsf{Set})((A,P),\top B)$, i.e.,

$$\mathsf{Set}(A,B) \cong \mathsf{Fam}(\mathsf{Set})((A,P),(B,\_ \mapsto 1)).$$

Elements of the right-hand set are pairs of morphisms $(f : A \to B, f^\sim : \prod_{a \in A} P(a) \to 1)$. But by terminality, there is only ever one choice of $f^\sim$, so the right-hand side is isomorphic to the set of functions $f : A \to B$, i.e., $\mathsf{Set}(A,B)$. Naturality is straightforward. □

It turns out that $\top$ is also *left* adjoint, in particular to the comprehension functor $\{-\}$, but we will not make use of this fact.

As before, we now turn from functors between $\mathsf{Fam}(\mathsf{Set})$ and $\mathsf{Set}$ (i.e., between families and their index sets) to those between fibers $\mathsf{Fam}(\mathsf{Set})_A$ and $\mathsf{Fam}(\mathsf{Set})_B$ (i.e., between families with distinct index sets). While our construction in the previous section required only the op-reindexing functor $\Sigma_f$, the following alternative means of reindexing will be useful for our proof.

**Definition 114: Reindexing functor**

Let $f : A \to B$ be a function (of sets). The *reindexing functor* $f^* : \mathsf{Fam}(\mathsf{Set})_B \to \mathsf{Fam}(\mathsf{Set})_A$ is defined by

$$f^*(B, Q) = (A, Q \circ f)$$
$$f^*(\mathrm{id}_B, g^\sim) = (\mathrm{id}_A, g^\sim \circ f)$$

Notice that this functor, in contrast with $\Sigma_f$, is contravariant: the domain of $f^*$ is the fiber over the codomain of $f$, and vice versa. (Hence the "op" prefix in "op-reindexing.") Thus, given a function $f : A \to B$, we can produce functors running in both directions between $\mathsf{Fam}(\mathsf{Set})_B$ and $\mathsf{Fam}(\mathsf{Set})_A$. Moreover, these functors are adjoint.

**Lemma 20.4.2.** Given a function $f : A \to B$, the functors $\Sigma_f \dashv f^*$ form an adjoint pair.

*Proof.* We show

$$\mathsf{Fam}(\mathsf{Set})_B(\Sigma_f(A, P), (B, Q)) \cong \mathsf{Fam}(\mathsf{Set})_A((A, P), f^*(B, Q)).$$

Recall that morphisms in these fibers are fully characterized by their second component (i.e., their mapping on families), so, unfolding definitions (and using the isomorphic representation of the indexed coproduct from Equation (20.6)), it suffices to show

$$\left\{ g : \prod_{b \in B} \sum_{a \in f^{-1}[\{b\}]} P(a) \to Q(b) \right\} \cong \left\{ h : \prod_{a \in A} P(a) \to Q(f(a)) \right\}$$

satisfying naturality. It is straightforward to check that $\iota$ (mapping from left to right) and its putative inverse defined below indeed form such an isomorphism between these sets:

$$\iota(g) = a \mapsto p \mapsto g(f(a))(a, p)$$
$$\iota^{-1}(h) = b \mapsto (a, p) \mapsto h(a)(p)$$

□

We will also make use of a lemma that characterizes the morphisms into a reindexed family: they are isomorphic to those morphisms into the original family that mapped indexing sets according to $f$.

**Lemma 20.4.3.** Let $(A, P)$ and $(B, Q)$ be objects in $\mathsf{Fam}(\mathsf{Set})$. Let $f : A \to B$ be a function. Then there is an isomorphism between the set of $\mathsf{Fam}(\mathsf{Set})$-morphisms $h : (A, P) \to (B, Q)$ such that $Uh = f$ and $\mathsf{Fam}(\mathsf{Set})_A$ morphisms $(A, P) \to f^*(B, Q)$:

$$\{h \in \mathsf{Fam}(\mathsf{Set})((A, P), (B, Q)) \mid Uh = f\} \cong$$
$$\mathsf{Fam}(\mathsf{Set})_A((A, P), f^*(B, Q))$$

**Exercise 20.10**

Prove Lemma 20.4.3.

We can now update Figure 20.1 to summarize these additional results:

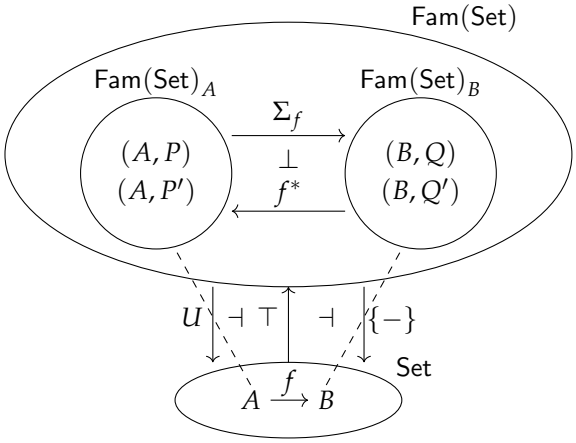



Figure 20.2: A revision of Figure 20.1, now with adjoint relationships and the reindexing functor ($\pi$ has been removed for clarity).

### *20.4.2 A Proof of Correctness*

We want to show that, given an endofunctor $F$ on $\mathsf{Set}$ with an initial algebra, the endofunctor $F^\alpha$ on $\mathsf{Fam}(\mathsf{Set})_A$ has one as well: that is, we must find an initial object in $\mathsf{Alg}_{F^\alpha}$. Since we already have an initial object in $\mathsf{Alg}_F$, it will be useful to relate $\mathsf{Alg}_{F^\alpha}$ to this category. Mirroring our construction of $F^\alpha$ itself, we will do this in two stages: first, we will consider algebras over $\hat{F} : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Fam}(\mathsf{Set})$, which we relate to algebras over $F$; then, we will limit our attention to an "$\alpha$-based restriction" of $\hat{F}$-algebras.

Algebras over $\hat{F}$ are elements of the category $\mathsf{Alg}_{\hat{F}}$. Since $\mathsf{Fam}(\mathsf{Set})$ "sits atop" $\mathsf{Set}$ (in the sense that a family $P$ is "atop" its indexing set $UP$), the category $\mathsf{Alg}_{\hat{F}}$ likewise "sits atop" $\mathsf{Alg}_F$, a fact witnessed by the following *algebras fibration*.

**Definition 115: Algebras fibration**

The *algebras fibration* $U^{\mathsf{Alg}}$ induced by the families fibration $U$ is a functor $U^{\mathsf{Alg}} : \mathsf{Alg}_{\hat{F}} \to \mathsf{Alg}_F$ defined by

$$U^{\mathsf{Alg}}(P, k : \hat{F}P \to P) = (UP, Uk)$$
$$U^{\mathsf{Alg}}(f : (P,k) \to (Q,k')) = Uf : (UP, Uk) \to (UQ, Uk')$$

It is worth pausing to type-check this definition, in which we have taken some notational shortcuts for concision. Recall that $U : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ maps families to their indexing sets. Since $\hat{F}$ is an endofunctor on *families*, $P$ comprises a family indexed by some set; $k$ is a morphism of families, which is a pair of functions; and $f$ is an algebra morphism (as defined in Definition 104) over an endofunctor on families and thus also a morphism of families (satisfying a certain commutativity condition). Thus, $U$ has an action on each of these.

The algebras over $F$ inherit an additional useful property from those over $F$: in particular, if $F$ has an initial algebra, than so does $\hat{F}$:

**Theorem 20.4.4.** Suppose $F$ has an initial algebra. Then $\hat{F}$ has an initial algebra too, and its carrier is $\top(\mu F)$.

*Proof.* The key insight for this proof is that $F$-algebras with carrier $\{(A, P)\}$ are interderivable with $\hat{F}$-algebras with carrier $(A, P)$. A proof of this fact is provided by Ghani et al. [13]; we will not give one here. Using this fact, the result follows almost immediately: the carrier $\mu F$ of the initial $F$-algebra is isomorphic to $\sum_{t \in \mu F} \{(\_ \mapsto \star)\}$ as a set, since the summand is a singleton. This set is precisely $\{\top(\mu F)\}$, so that $\top(\mu F)$ is the carrier of the initial $\hat{F}$-algebra by the aforementioned interderivability fact. □

However, since $F^\alpha$ is a restriction of $\hat{F}$ by values of an algebra $(A, \alpha)$, we want to look at a correspondingly restricted category of algebras, not $\mathsf{Alg}_{\hat{F}}$ in its entirety. We thus restrict the category $\mathsf{Alg}_{\hat{F}}$ to an appropriate *fiber* over our refining algebra. This definition, given below, is directly analogous to the definition of fibers over $\mathsf{Fam}(\mathsf{Set})$; however, we phrase it slightly differently to suggest the generality of such a construction. (One may find it worthwhile to "pattern-match" the tuple-valued variables below and unfold the definitions of $U^{\mathsf{Alg}}$ and $U$ to see this correspondence more directly.) Note that we are also now adopting the convention of abbreviating $(A, \alpha)$ as simply $\alpha$.

**Definition 116: Fiber of the algebras fibration over $\alpha$**

The *fiber* of the algebras fibration over $\alpha$, or equivalently the *category of $\hat{F}$-algebras over $\alpha$*, denoted by $(\mathsf{Alg}_{\hat{F}})_\alpha$, is defined as follows:

- Objects are $\hat{F}$-algebras $(P, k)$ such that $U^{\mathsf{Alg}}(P, k) = \alpha$.
- Morphisms are $\hat{F}$-algebra morphisms $f : (P, k) \to (Q, k')$ such that $Uf = \mathsf{id}_\alpha$.

Once again, it is worth briefly type-checking this definition. For instance, the criterion on objects is well defined: $k = (k_1, k_2)$ is a morphism $\hat{F}P \to P$, which is an abbreviation of $\hat{F}(A, P) \to (A, P)$ with the indexing set explicit. By the definition of $\mathsf{Fam}(\mathsf{Set})$, $k_1$ maps between the carriers; since $\hat{F}(A, P)$ has carrier $FA$ by definition, we thus have $k_1 : FA \to A$. And since $U^{\mathsf{Alg}}(P, k) = U^{\mathsf{Alg}}((A, P), (k_1, k_2)) = (A, k_1)$, this is of the same type as $\alpha$, i.e., as $(A, \alpha : FA \to A)$.

Moreover, it turns out that $(\mathsf{Alg}_{\hat{F}})_\alpha$ is the same category as $\mathsf{Alg}_{F^\alpha}$, just described as a "decomposition" in terms of $\hat{F}$ and $\alpha$.

**Lemma 20.4.5.** Let $(A, \alpha)$ be an $F$-algebra. Then there is an isomorphism of categories

$$(\mathsf{Alg}_{\hat{F}})_\alpha \cong \mathsf{Alg}_{F^\alpha}$$

*Proof.* Observe that an object of $(\mathsf{Alg}_{\hat{F}})_\alpha$ is a pair of a family and a $\mathsf{Fam}(\mathsf{Set})$-morphism—i.e., a tuple $((A, P), k : \hat{F}(A, P) \to (A, P))$—such that $Uk = \alpha$. By Lemma 20.4.3, we can characterize such a $\mathsf{Fam}(\mathsf{Set})$-morphism as a $\mathsf{Fam}(\mathsf{Set})_{FA}$-morphism via the reindexing functor, so the object is equivalently expressed as $((A, P), k' : \hat{F}(A, P) \to \alpha^*(A, P))$. In turn, we observe that morphisms $k' : \hat{F}(A, P) \to \alpha^*(A, P)$ are themselves in isomorphism with $\mathsf{Fam}(\mathsf{Set})_A$-morphisms $k'' : \Sigma_\alpha \hat{F}(A, P) \to (A, P)$ by Lemma 20.4.2. Recalling the definition of $F^\alpha$, we can rewrite this as $k'' : F^\alpha(A, P) \to (A, P)$. Thus, we can again equivalently rewrite our object as $((A, P), k'' : F^\alpha(A, P) \to (A, P))$, which is precisely an object in $\mathsf{Alg}_{F^\alpha}$. Transporting a morphism along the corresponding sequence of isomorphic transformations yields the required action on morphisms. □

In order to work with this new formulation of the category, we will reproduce for $(\mathsf{Alg}_{\hat{F}})_\alpha$ several results analogous to those we showed for the families fibration. The following lemmas are analogues of Lemmas 20.4.2 and 20.4.3; their proofs are technical but not exceedingly difficult, so we elide them.

**Lemma 20.4.6.** Let $f$ be an $F$-algebra morphism $(A, \alpha) \to (B, \beta)$. Then there are functors $\Sigma_f^{\mathsf{Alg}} : (\mathsf{Alg}_{\hat{F}})_\alpha \to (\mathsf{Alg}_{\hat{F}})_\beta$ and $f^{*\mathsf{Alg}} : (\mathsf{Alg}_{\hat{F}})_\beta \to (\mathsf{Alg}_{\hat{F}})_\alpha$ such that $\Sigma_f^{\mathsf{Alg}} \dashv f^{*\mathsf{Alg}}$ and for any $\hat{F}$-algebras $((A, P), k)$ and $((B, Q), k')$,

- The $\hat{F}$-algebra $\Sigma_f^{\mathsf{Alg}}((A,P),k)$ has carrier $\Sigma_f(A,P)$, and
- The $\hat{F}$-algebra $f^{*\mathsf{Alg}}((B,Q),k')$ has carrier $f^*(B,Q)$.

**Lemma 20.4.7.** Let $(A,\alpha)$ and $(B,\beta)$ be $\hat{F}$-algebras. Fix an object $((A,P),k)$ in $(\mathsf{Alg}_{\hat{F}})_\alpha$ and $((B,Q),k')$ in $(\mathsf{Alg}_{\hat{F}})_\beta$. Let $f : (A,\alpha) \to (B,\beta)$ be an $\hat{F}$-algebra morphism. Then there is an isomorphism between the set of $\mathsf{Alg}_{\hat{F}}$-morphisms $h : k \to k'$ such that $U^{\mathsf{Alg}}h = f$ and the set of morphisms $k \to f^{*\mathsf{Alg}}(k')$:

$$\{h \in \mathsf{Alg}_{\hat{F}}(k,k') \mid U^{\mathsf{Alg}}h = f\} \cong \mathsf{Alg}_{\hat{F}}(k, f^{*\mathsf{Alg}}(k')).$$

We can now draw a diagram analogous to Figure 20.2 for the algebras fibration. Note that we temporarily drop our abbreviated notation for families and algebras to better analogize this diagram to the one for the families fibration.

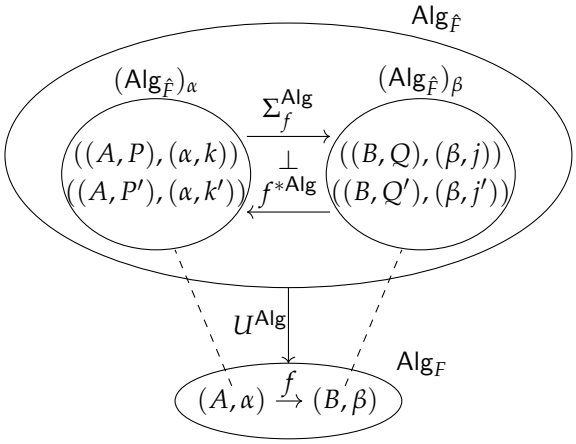



Figure 20.3: A summary of constructions for the algebras fibration.

Equipped with these intermediate results, we can now finally prove what we set out to show: that $F^\alpha$ admits an initial algebra, and that its carrier is isomorphic to the "naïve" refinement interpretation. We make one brief observation: we can rewrite our previous "naïve" interpretation of a refinement by $(\!|\alpha|\!) : \mu F \to A$ using the truth and op-reindexing functors as follows:

$$(A, a \mapsto \{x \in \mu F \mid (\!|\alpha|\!)(x) = a\}) \cong \Sigma_{(\!|\alpha|\!)} \top(\mu F).$$

Using this formulation, we now state our main theorem.

**Theorem 20.4.8.** The endofunctor $F^\alpha$ has an initial algebra, and its carrier is $\Sigma_{(\!|\alpha|\!)} \top(\mu F)$.

*Proof.* We will first construct a candidate algebra in $(\mathsf{Alg}_{\hat{F}})_\alpha$, then show that it is initial. Note that, by Lemma 20.4.5, this is equivalent to identifying an $F^\alpha$-algebra and showing its initiality in $\mathsf{Alg}_{F^\alpha}$.

From Theorem 20.4.4, we obtain an inital algebra $(\top(\mu F), \mathsf{in}_{\hat{F}})$ in $\mathsf{Alg}_{\hat{F}}$. We will use this to construct an initial object in the fiber over $\alpha$. Mirroring the construction of $F^\alpha$, we apply the op-reindexing functor $\Sigma^{\mathsf{Alg}}_{(\!|\alpha|\!)}$ for the algebras fibration to obtain

$$\left(\Sigma^{\mathsf{Alg}}_{(\!|\alpha|\!)} \top(\mu F),\ \Sigma^{\mathsf{Alg}}_{(\!|\alpha|\!)} \mathsf{in}_{\hat{F}} : \hat{F}(\top(\mu F)) \to \top(\mu F)\right).$$

We will aim to show that this object is our desired initial object in $(\mathsf{Alg}_{\hat{F}})_\alpha$.

First, observe that by Lemma 20.4.6, we have

$$\Sigma^{\mathsf{Alg}}_{(\!|\alpha|\!)} \top(\mu F) \cong \Sigma_{(\!|\alpha|\!)} \top(\mu F),$$

i.e., the carrier of this algebra is our naïve refinement encoding. So it remains only to show that the algebra is indeed initial.

We show initiality directly from the definition: we show that there is exactly one morphism between $\Sigma^{\mathsf{Alg}}_{(\!|\alpha|\!)} \mathsf{in}_{\hat{F}}$ and an arbitrary $\hat{F}$-algebra $k : \hat{F}(A, P) \to (A, P)$ (hereafter we will denote each algebra under discussion by its structure map alone). We proceed by considering a series of isomorphisms of hom-sets:

$$\begin{aligned}
&(\mathsf{Alg}_{\hat{F}})_\alpha \left(\Sigma^{\mathsf{Alg}}_{(\!|\alpha|\!)} \mathsf{in}_{\hat{F}},\ k\right) \\
&\cong (\mathsf{Alg}_{\hat{F}})_{\mathsf{in}_F} \left(\mathsf{in}_{\hat{F}},\ (\!|\alpha|\!)^*(k)\right) && \text{(Adjointness portion of Lemma 20.4.6)} \\
&\cong \{h \in \mathsf{Alg}_{\hat{F}}(in_{\hat{F}}, k) \mid U^{\mathsf{Alg}} h = (\!|\alpha|\!)\} && \text{(Lemma 20.4.7)}
\end{aligned}$$

Notice that this final set is a subset of morphisms out of an initial object, so it can have at most one element (namely, $(\!|k|\!)$). If this is to be a singleton set, then it must indeed contain $(\!|k|\!)$, so in particular it must be that

$$U^{\mathsf{Alg}}(\!|k|\!) = (\!|\alpha|\!).$$

Recall that $U^{\mathsf{Alg}}$ is left adjoint, so it preserves colimits: the above equation (of morphisms from initial objects) thus holds.

Accordingly, the algebra $\Sigma^{\mathsf{Alg}}_{(\!|\alpha|\!)} \mathsf{in}_{\hat{F}}$ is indeed initial in $(\mathsf{Alg}_{\hat{F}})_\alpha$, completing the proof. □

Before concluding this section, it is worth noting that the construction given here can be extended in various ways, such as to support refinements defined in terms of functions that are not directly expressed as folds. We direct the reader to Atkey et al. [2] for details.

Let us finally take stock of what we have done. We began by interpreting simple inductive types as initial algebras $(\mu F, \mathsf{in}_{\mu F})$ of endofunctors $F$. On this interpretation, maps out of the datatype defined in terms of folds are the $F$-algebra homomorphisms $(\!|\alpha|\!)$ guaranteed by initiality. We then gave a categorical semantics of indexed

inductive types in terms of initial algebras of endofunctors on *families*, in which we gave a "naïve" subset interpretation of the refinement $\{x : \mu F \mid (\!|\alpha|\!)(x) = a\}$. To identify the indexed inductive type corresponding to this semantic interpretation, we constructed the endofunctor on families $F^{\alpha}$ using a *lifting* of the functor $F$ characterizing the unrefined simple inductive type. Lastly, we showed that this construction yielded an initial algebra with the carrier we identified in our naïve translation.

## 20.5 A (Literal) Invitation to Fibered Category Theory

Throughout this chapter, we have followed Atkey et al. [2] in carefully presenting the material so as to admit a generalization to the abstract language of *fibrations*, and have gradually adjusted our presentation over the course of the chapter toward a more abstract framing in terms of fibrational constructions. Regrettably, we do not have the space here to present the full generalization—a construction that can be carried out in any *full Cartesian Lawvere category with very strong coproducts*—or to discuss fibrations in their own right.

Instead, we direct the reader to the following sources. The original development of this material by Atkey et al. [2] gestures further toward its generalization, in addition to providing several further constructions that translate broader classes of refinements into indexed inductive types. Ghani et al.'s paper [13] includes the original, abstract derivation of the truth-preserving lifting $\hat{F}$ we employ in this paper. Some relevant background can also be found in Jacobs' paper on comprehension categories [16], while the same author's textbook *Categorical Logic and Type Theory* [17] provides an exposition of fibered category theory and several applications. A selection of material from among these sources should provide sufficient background to interpret this chapter in a general fibrational setting, thereby characterizing the precise categorical structure required to unify refinement and indexed types.

## *Solutions to Exercises*

**Solution 20.1:**

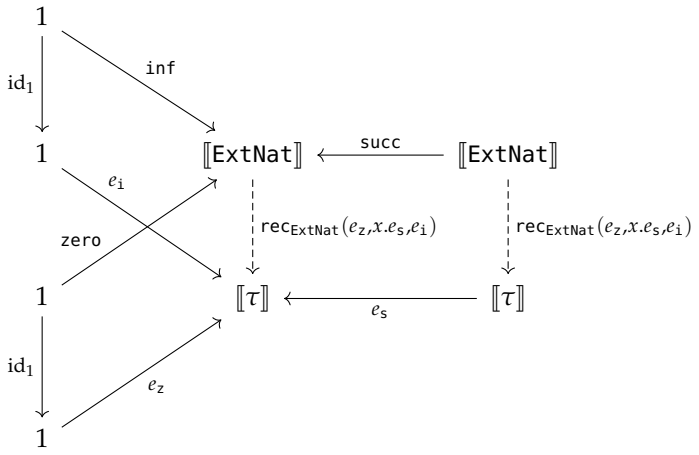
Pick $[\![\texttt{true}]\!] = \mathsf{inl}$ and $[\![\texttt{false}]\!] = \mathsf{inr}$, respectively the left and right injections into the coproduct, and take $[\![\texttt{if}(t,e)]\!] = \langle t, e\rangle$. Commutativity and recursor uniqueness are immediate from the universal property for coproducts.

**Solution 20.2:**

Since the recursor for `ExtNat` is not explicitly given, we first derive its typing rule (its equational laws are analogous to those for the regular natural-number recursor):

$$\frac{\vdash e_{\mathtt{z}} : \tau \qquad x : \tau \vdash e_{\mathtt{s}} : \tau \qquad \vdash e_{\mathtt{i}} : \tau}{\vdash \mathtt{rec}(e_{\mathtt{z}}, x.e_{\mathtt{s}}, e_{\mathtt{i}}) : \tau}$$

As the hint alludes to, this example differs from the preceding ones because there are three constructors rather than two, yielding a third square in our diagram; it is a "revolving door" rather than a pair of squares. It looks as follows:




**Solution 20.3:**

1. The actions of $F_{\mathtt{SelfTree}}$ on objects and morphisms are given below:

$$F_{\mathtt{SelfTree}}(X) = 1 \oplus A \times X^A$$
$$F_{\mathtt{SelfTree}}(e) = \mathsf{id}_1 \oplus (\mathsf{id}_A \times \lambda(e \circ \mathsf{app}))$$

2. The relevant algebra is

$$\alpha_{\mathtt{selfHeight}} : F_{\mathtt{SelfTree}}\,[\![\mathtt{Nat}]\!] \to [\![\mathtt{Nat}]\!] = [0, \mathsf{succ} \circ \mathsf{app} \circ \mathsf{flip}].$$

We denote by $\mathsf{flip} : A \times X^A \to X^A \times A$ the isomorphism that witnesses the commutativity of products.

**Solution 20.4:**

Fix objects $A, B$ and consider their exponential. Since $\mathtt{app} : B^A \times A \to B$, we are led to define the endofunctor $F(X) = X^A \times A$, whose action on morphisms is given by $F(e) = \lambda(e \circ \mathtt{app}) \times \mathsf{id}_A$. (Notice that, despite the type of `app` mentioning two objects, there is only one possible choice of endofunctor: if we tried to define the functor in $A$, it would be contravariant.)

We claim that the identity morphism for the initial object 0 is the initial $F$-algebra. To see why, first observe that $F(0) = 0^A \times A \cong 0 \times A \cong 0$, so the choice of initial algebra $\mathsf{id}_0 : F(0) \to 0$ (transported across the relevant isomorphisms) type-checks. Moreover, its initiality is immediate from the fact that 0 is initial.

By Lambek's lemma, we thus have that $\mu F \cong 0$, so the inductive type that this functor characterizes is empty. We can read off the corresponding `inductive` definition:

```
inductive Empty' (A : Type) : Type where
  | mk : (A -> Empty' A) -> Empty' A -> Empty' A
```

**Solution 20.5:**

We translate each rule in turn.

- T-FOLD: As previously discussed, the positivity hypothesis of this rule is analogous to the requirement that the endofunctor $F$ denoted by $t.\tau$ has an initial algebra. The substitution $\tau[\mu t.\tau/t]$ corresponds to plugging in $[\![\tau]\!]$ for the functor variable $t$ in $F(t)$: it is the functor action on objects. Thus, the conclusion of the rule induces the following morphism:

$$[\![F(\mu F)]\!] \xrightarrow{\mathtt{fold}} [\![\mu F]\!]$$

- T-REC: Now that we know how to interpret type-variable substitution, this translation proceeds mechanically: the hypothesis says we have a morphism $e_1 : [\![\rho]\!] \to F\,[\![\rho]\!]$, and the conclusion uses this to produce a morphism $\mu F \to [\![\rho]\!]$:

$$\begin{array}{ccc} & & [\![\mu F]\!] \\ & & \Big| \\ & & \mathsf{rec}(x.e_1) \\ & & \downarrow \\ F\,[\![\rho]\!] & \xrightarrow{e_1} & [\![\rho]\!] \end{array}$$

- $\beta^{\mathsf{rec}}$: Now we must contend with the function $\mathtt{map}(e)$. Observe that $e : (t.\tau)[\rho/t]$ from rule T-MAP, so we know that the type of $e$ denotes a functor application $F[\![\rho]\!]$. The key insight—which is most easily observed by inspecting the sample $\beta$ rules for this function—is that the function $\mathtt{map}(e)$ denotes precisely the action of $F$ on *morphisms*, i.e., terms. Thus, we translate $\mathtt{map}(e)$ as the morphism action $F(-)$.

  We can now interpret the $\beta^{\mathsf{rec}}$ rule. The left-hand side corresponds to the morphism $\mathtt{rec}(x.e_1) \circ \mathtt{fold}$. The right-hand side becomes (removing the explicit point $y$ for clarity, and recalling that substitution becomes morphism composition) $e_1 \circ [\![\mathtt{map}(e_2)(\mathtt{rec}(x.e_1))]\!]$, which unfolds to $e_1 \circ F(\mathtt{rec}(x.e_1))$. Since this rule is equational, we have that these two morphisms are equal, i.e.,

  $$\mathtt{rec}(x.e_1) \circ \mathtt{fold} = e_1 \circ F(\mathtt{rec}(x.e_1))$$

  This just says that if we stitch together the diagrams from the preceding two rules, the resulting square commutes:

  $$\begin{array}{ccc} F(\mu F) & \xrightarrow{\mathtt{fold}} & [\![\mu F]\!] \\ \downarrow{\scriptstyle \mathtt{F(rec}(x.e_1))} & & \downarrow{\scriptstyle \mathtt{rec}(x.e_1)} \\ F[\![\rho]\!] & \xrightarrow{e_1} & [\![\rho]\!] \end{array}$$

- $\eta^{\mathsf{rec}}$: The hypothesis of this rule translates to

  $$e \circ \mathtt{fold} = e_1 \circ F(\mathtt{rec}(x.e_1))$$

  and its conclusion to

  $$e = \mathtt{rec}(x.e_1).$$

  In other words, any morphism $e$ satisfying the equality from the $\beta$ rule must be equal to the recursor; that is, the recursor is unique. We reflect this in our diagram from above:

  $$\begin{array}{ccc} F(\mu F) & \xrightarrow{\mathtt{fold}} & [\![\mu F]\!] \\ \downarrow{\scriptstyle \mathtt{F(rec}(x.e_1))} & & \vdots{\scriptstyle \mathtt{rec}(x.e_1)} \\ F[\![\rho]\!] & \xrightarrow{e_1} & [\![\rho]\!] \end{array}$$

Observe that (renaming $\mathtt{fold}$ to $c$, $e_1$ to $f_c$, and $\rho$ to $A$) we have now arrived at the same diagram we derived in our previous translation!

**Solution 20.6:**

$$F_{\mathtt{Fin}}(\mathbb{N}, P) = \left(\mathbb{N}, \begin{cases} 0 \mapsto \varnothing \\ \mathsf{succ}(n) \mapsto 1 \oplus P(n) \end{cases}\right)$$

**Solution 20.7:**

First, note that the "straightforward" morphism $\psi$ is concretely

$$\psi(a, p) = (f, a \mapsto p \mapsto (a, p)).$$

(Note that we are using the isomorphic formulation of indexed coproducts given in Equation (20.6) to minimize bureaucracy.)

1. Unfolding definitions, we find that

   $$(f \circ \pi_{(A,P)})(a, p) = f(a)$$

   and

   $$(\pi_{\Sigma_f(A,P)} \circ \{\psi\})(a, p) = \pi_{\Sigma_f(A,P)}(f(a), a, p) = f(a)$$

   so that

   $$f \circ \pi_{(A,P)} = \pi_{\Sigma_f(A,P)} \circ \{\psi\}$$

   as required.

2. Define the putative inverse of $\{\psi\}$ to be $\iota : \{\Sigma_f(A, P)\} \to \{(A, P)\}$ given by

   $$\iota(b, (a, p)) = (a, p)$$

   Then observe that

   $$\{\psi\}(\iota(b, (a, p))) = \{\psi\}(a, p) = (f(a), (a, p)) = (b, (a, p))$$

   where the last equality follows by the definition of $\Sigma_f(A, P)$, and moreover

   $$\iota(\{\psi\}(a, p)) = \iota(f(a), (a, p)) = (a, p)$$

   so that $\{\psi\}$ is indeed an isomorphism with inverse $\iota$.

**Solution 20.8:**

This is clearly a lifting, since

$$F(U(A, P)) = FA = U(FA, \_ \mapsto 1) = U(\hat{F}(A, P))$$

and

$$F(U(f, f^{\sim})) = Ff = U(f, \_ \mapsto \mathsf{id}_1) = U(\hat{F}(f, f^{\sim})).$$

Moreover, that it is truth preserving is witnessed by the natural isomorphism

$$\eta_{(A,P)} = \mathsf{id}_{(A,P)}$$

since, noting that

$$\top(F(A)) = (FA, \_ \mapsto 1) = \hat{F}(\top(A))$$

and

$$\top(F(f)) = (f, \_ \mapsto \mathsf{id}_1) = \hat{F}(\top(f))$$

the following square trivially commutes:

$$\begin{array}{ccc} (\top \circ F)(A) & \xrightarrow{\eta_{(A,\_\mapsto 1)}} & (\hat{F} \circ \top)(A) \\ \Big\downarrow{\scriptstyle (\top \circ F)(f)} & & \Big\downarrow{\scriptstyle (\hat{F} \circ \top)(f)} \\ (\top \circ F)(B) & \xrightarrow{\eta_{(B,\_\mapsto 1)}} & (\hat{F} \circ \top)(B) \end{array}$$

**Solution 20.9:**

1. The endofunctor $F_{\mathsf{RBTree}}$ has the following actions on objects and morphisms:

$$\begin{aligned} F_{\mathsf{RBTree}}(X) &= 1 \oplus ((A \times (X \times X)) \oplus (A \times (X \times X))) \\ F_{\mathsf{RBTree}}(e) &= \mathsf{id}_1 \oplus ((\mathsf{id}_A \times (e \times e)) \oplus (\mathsf{id}_A \times (e \times e))) \end{aligned}$$

2. The desired $F_{\mathsf{RBTree}}$-algebra is

$$\alpha_{\mathsf{bh}} = [0, [\mathsf{succ} \circ \mathsf{max} \circ \pi_2, \mathsf{max} \circ \pi_2]]$$

3. We compute as follows:

$$\begin{aligned} & F^{\mathsf{bh}}_{\mathsf{RBTree}}(\mathbb{N}, P) \\ & \cong (\mathbb{N}, n \mapsto \{y \in F_{\mathsf{RBTree}}(\mathbb{N}, P) \mid \mathsf{bh}((F_{\mathsf{RBTree}}\pi_{(\mathbb{N},P)})(y)) = n\}) \\ & = (\mathbb{N}, n \mapsto \{y \in 1 \oplus ((A \times ((\mathbb{N}, P) \times (\mathbb{N}, P))) \oplus (A \times ((\mathbb{N}, P) \times (\mathbb{N}, P)))) \mid \\ & \qquad\qquad (y = \mathsf{inl}\ () \text{ and } 0 = n) \text{ or} \\ & \qquad\qquad (y = \mathsf{inr}\ (\mathsf{inl}\ (a, ((m_\ell, p_\ell), (m_r, p_r)))) \text{ and } n = \mathsf{succ}(\mathsf{max}(n_\ell, n_r))) \\ & \qquad\qquad (y = \mathsf{inr}\ (\mathsf{inr}\ (a, ((m_\ell, p_\ell), (m_r, p_r)))) \text{ and } n = \mathsf{max}(n_\ell, n_r))\}) \\ & = \left(\mathbb{N}, \left(n \mapsto \begin{pmatrix} 1 & \text{if } n = 0 \\ \varnothing & \text{else} \end{pmatrix} \oplus \right.\right. \\ & \qquad\qquad \left(\begin{pmatrix} A \times ((\{m_\ell\} \times P(m_\ell)) \times (\{m_r\} \times P(m_r))) & \text{if } n = \mathsf{succ}(\mathsf{max}(m_\ell, m_r)) \\ \varnothing & \text{else} \end{pmatrix} \oplus \right. \\ & \qquad\qquad \left.\left.\left.\begin{pmatrix} A \times ((\{m_\ell\} \times P(m_\ell)) \times (\{m_r\} \times P(m_r))) & \text{if } n = \mathsf{max}(m_\ell, m_r) \\ \varnothing & \text{else} \end{pmatrix}\right)\right)\right) \end{aligned}$$

One can verify that this is isomorphic to the translation of the following inductive type:

```
inductive BHRBTree : Nat -> Type where
  | leaf : BHRBTree 0
  | red  : {m n : Nat} -> A -> BHRBTree m -> BHRBTree n -> BHRBTree (succ
      (max m n))
  | red  : {m n : Nat} -> A -> BHRBTree m -> BHRBTree n -> BHRBTree (max
     m n)
```

**Solution 20.10:**

Recall that $\mathsf{Fam}(\mathsf{Set})$-morphisms from $(A, P)$ to $(B, Q)$ are pairs

$$\left(h : A \to B, h^{\sim} : \prod_{a \in A} P(a) \to Q(h(a))\right).$$

The left-hand set in the lemma statement is the restriction of such pairs to those whose first element is $f$, so they are of the form

$$\left(f, h^{\sim} : \prod_{a \in A} P(a) \to Q(f(a))\right).$$

By the definitions of $\mathsf{Fam}(\mathsf{Set})_A$ and $f^*$, morphisms in the right-hand set are pairs

$$\left(\mathsf{id}_A, f^{\sim} : \prod_{a \in A} P(a) \to Q(f(a))\right).$$

Since the first component of every pair in either set is fixed, both sets are isomorphic to the set of functions of type

$$\prod_{a \in A} P(a) \to Q(f(a))$$

and thus to each other.

# Algebras and automata

Michael Zhang

## 1 Algebras

In algebra, there are many interesting structures to study, such as groups, rings, vector spaces, etc. If we look at the definitions for some common algebraic structures:

- A group is a tuple $(G, \cdot, (-)^{-1})$ satisfying some axioms.
- A ring is a tuple $(R, +, \cdot)$ satisfying some axioms.
- A boolean algebra is a tuple $(B, \wedge, \vee, \neg)$ satisfying some axioms.

They all seem to look kind of familiar – a carrier, usually a set, followed by some operations on that carrier, satisfying some axioms. It may be useful to look for an abstraction over all of these.

In order to make it easier to think about abstractions for a universal algebra, we'll start by focusing our work in the category of sets $\mathbf{Set}$. So our carrier will be some object in $\mathbf{Set}$.

Now, consider an endofunctor $F : \mathbf{Set} \to \mathbf{Set}$. This will have two parts: an action on objects, which is simply a set function, and an action on morphisms. If we take $X$ to be the input carrier set, then $F(X)$ is essentially the representation of *inputs* to operations you can take with $X$. Then, we have a morphism $\alpha : F(X) \to X$, known as the *structure* map, which defines how the operations take place.

For example:

- the identity element of a group is not dependent on anything, so we say $F(X) = 1$, and so the morphism $\alpha : 1 \to X$ simply selects the identity element with no further input.
- the inverse operation of a group is a set-function $X \to X$. This translates to our framework as $F(X) = X$ and so $\alpha : X \to X$ in this case.
- the group operation is a binary set-function $(\cdot) : X \times X \to X$. We have $F(X) = X \times X$, so the morphism $\alpha : X \times X \to X$ has the correct type.

But each of these are still independent pieces of a group definition. We must combine them somehow. A single morphism $\alpha : F(X) \to X$ must represent all actions you can possibly take on a group. One answer would be to take $F(X) = 1 \uplus X \uplus X \times X$, where $\uplus$ is the disjoint set-union operation. This is usually written instead as $F(X) = 1 + X + X \times X$, as we will soon generalize disjoint unions in $\mathbf{Set}$ to coproducts in an arbitrary category.

So our combined morphism $\alpha : F(X) \to X$ essentially gives us a choice of which operation we would like to take, as well as how each is implemented. It should be clear at this point that each functor $F$ can possibly have multiple $\alpha : F(X) \to X$. This corresponds to the notion that there are multiple groups, multiple rings, multiple Boolean algebras, etc.

Of course, since $F$ is a functor, it must have an action on morphisms as well. In the case of $F(X) = A \times X + B$, a functor application to a morphism $F\left(X \xrightarrow{f} Y\right)$ would have type $A \times X + B \xrightarrow{F(f)} A \times Y + B$. It may be useful to think of $F(f)$ as being a coproduct of morphisms $A \times X \underset{1}{\xrightarrow{f}} A \times Y$ and $B \underset{2}{\xrightarrow{f}} B$. This is often true for the cases we will be studying, as many of the functors for the common algebras do exhibit this property. However, it may not be true in general, and the categorical

framework for algebra allows for functors that mix between the different components. Still, we will use the notation $F(f) = f_1 + f_2$ in the remainder of the notes.

**Definition 1.1 ($F$-algebra)**

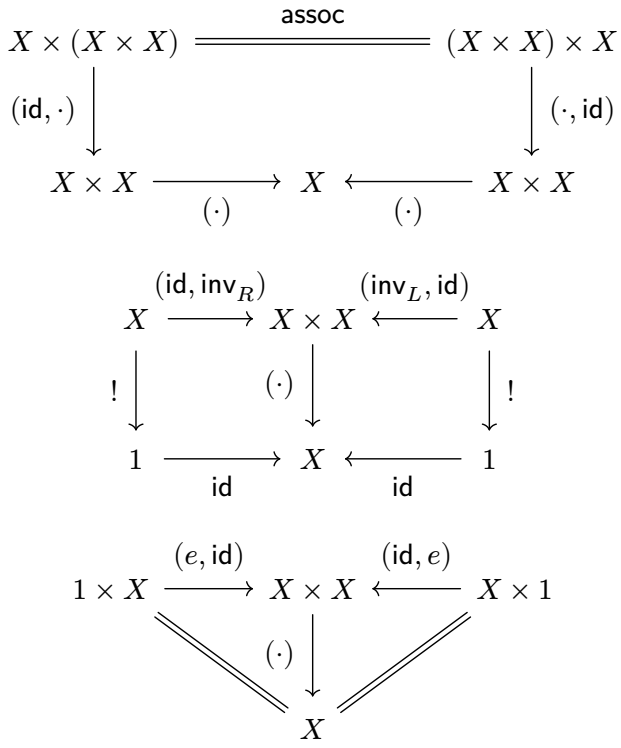
Let $\mathcal{C}$ be an ambient category. An $F$-algebra is a pair $(X, \alpha : F(X) \to X)$, where $X \in \mathsf{Ob}(\mathcal{C})$, $F$ is an endofunctor $\mathcal{C} \to \mathcal{C}$, and $\alpha : F(X) \to X$.

This doesn't cover all of the group properties yet. We are still missing associativity and the inverse laws. Unfortunately, the functor $F$ primarily deals with the types of operations in an algebra, also known as the "signature", so the extra laws must be separately required.[1] For example, we will define groups as follows:

A group is an $F$-algebra defined by the signature

$$F(X) = 1 + X + X \times X \tag{1}$$

where the components are named $[e, \mathsf{inv}, (\cdot)]$, such that the following diagrams commute:




It turns out there are some interesting $F$-algebras when we consider the category of $F$-algebras themselves. But before we get there, we must first define what it means to have a morphism of $F$-algebras. As is standard in category theory, an $F$-algebra is simply an underlying carrier plus some extra data. We will thus define morphisms as a function between the carrier along with proof of coherence between the data:

[1] There is a different construction which generates algebras from a monad, that can encapsulate having these extra axioms, but we won't really explore that here.

**Definition 1.2 ($F$-algebra morphism)**

Let $(X, \alpha_X)$ and $(Y, \alpha_Y)$ be two $F$-algebras in a category $\mathcal{C}$. An $F$-algebra morphism is then a morphism $f : X \to Y$ such that the following diagram commutes:

$$\begin{array}{ccc} F(X) & \xrightarrow{\alpha_X} & X \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle f} \\ F(Y) & \xrightarrow[\alpha_Y]{} & Y \end{array}$$

**Exercise 1.1.** *Verify that $F$-algebras form a category.*

Let us now interpret this commutative square in the context of the $F$-algebras generated by each of our group operations above.

- In the case of identity, the component of $F(X)$ we are focusing on is $F(X) = 1$. So both $F(X)$ and $F(Y)$ evaluate to 1, so $F(X) \to F(Y)$ can only be the unique identity morphism in 1. Thus, the left side collapses and we have a triangle:

$$\begin{array}{ccc} 1 & \xrightarrow{\alpha_X} & X \\ & {\scriptstyle \alpha_Y}\searrow & \downarrow{\scriptstyle f} \\ & & Y \end{array}$$

  Translating this to group theory, this means our choice of $\alpha_Y$, the function that picks out the group identity of $Y$, must agree with the result of mapping $f$ to the group identity of $X$. In other words, group homomorphisms preserve identities.
- In the case of inverse, $F(X) = X$, so we have the following diagram:

$$\begin{array}{ccc} X & \xrightarrow{\mathsf{inv}_X} & X \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f} \\ Y & \xrightarrow[\mathsf{inv}_Y]{} & Y \end{array}$$

  This translates to $\mathsf{inv}_Y \circ f = f \circ \mathsf{inv}_X$, which written algebraically says that for any $x \in X$, it must hold that $f(x)^{-1} = f(x^{-1})$. In other words, group homomorphisms preserve inverses.
- In the case of the binary operation $(\cdot)$, we have $F(X) = X \times X$, so we have the following diagram (recall the definition of the action of $F$ on morphisms):

$$\begin{array}{ccc} X \times X & \xrightarrow{(\cdot)_X} & X \\ {\scriptstyle (f,f)}\downarrow & & \downarrow{\scriptstyle f} \\ Y \times Y & \xrightarrow[(\cdot)_Y]{} & Y \end{array}$$

Translating this to algebra, we have $f(x \cdot_X y) = f(x) \cdot_Y f(y)$, which is the main group homomorphism property.

As you can see, the group homomorphism properties fell purely out of just instantiating our $F$-algebra morphism law with our specific group properties.

## 1.1 Initial $F$-algebras and induction

We said before that $F(X)$ builds up the input data to be used by $\alpha$. But that's a rather abstract description. To see an example of how this is used, consider a functor $F(X) = 1 + X$. We can define an $F$-algebra by providing an $\alpha : F(X) \to X$. Expanding the type gives us $\alpha : (1 + X) \to X$, which we can define by cases:

- $\alpha_1 : 1 \to X$
- $\alpha_X : X \to X$

Suppose we are working with an $(X, \alpha)$ that is initial in the category of $F$-algebras for $F(X) = 1 + X$. The defining property of an initial object is that there exists a morphism out of this $F$-algebra to any other $F$-algebra, such that the square commutes. This morphism is commonly known as the **catamorphism**.

For example, consider any other $F$-algebra $\beta : (1 + Y) \to Y$, with corresponding $[\beta_1, \beta_Y]$. Then, there exists a *unique* $f : X \to Y$ such that this square commutes:

$$\begin{array}{ccc} 1 + X & \xrightarrow{\alpha} & X \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle f} \\ 1 + Y & \xrightarrow[\beta]{} & Y \end{array}$$

where $F(f)$ is defined on cases:

- for the left case of the sum, there is a morphism $1_X \to 1_Y$
- for the right case of the sum, there is a morphism $X \to Y$

The square forces $f$ to respect the $F$-algebra structure.

There is a sense where the initial $(F, \alpha)$ shown above represents the natural numbers $\mathbb{N}$, and that the unique morphism $f$ is the natural number recursor. Recall that the natural number recursor looks something like this:

$$\mathsf{natrec} : (y_0 : Y) \to (y_s : Y \to Y) \to (\mathbb{N} \to Y) \tag{2}$$

If we change $y_0$ to be of type $1_Y \to Y$ (this is equivalent, it's just a function that ignores the input), it should be clear that the inputs $y_0$ and $y_s$ represents the two cases of $\beta$. So really, we can reframe this as: given $\beta : 1 + Y \to Y$, which is the definition of the $F$-algebra you are trying to produce, you get the unique morphism $\mathbb{N} \to Y$ back, since it's forced by the square.

**Exercise 1.1.1.** *Prove that* natrec *is the unique morphism that makes the above square commute.*

We defined natrec for naturals here, but the idea that the unique catamorphism is the recursor holds for arbitrary $F$-algebras.

**Definition 1.1.1 (Recursor)**

Given some functor $F$, and an initial $F$-algebra $(X, \alpha)$, the recursor is the unique $F$-algebra morphism to any other $F$-algebra $(Y, \beta)$.

This generalizes to other algebra-shaped structures as well. In fact, $F$-algebras generally build inductive data types. For example, lists are $F(X) = 1 + A \times X$ for $A$-typed lists. The initial object in this category is then the pair $(X, \alpha)$ defined by:

- $X$ is the set of all finite lists of type $A$.
- $\alpha : 1 + A \times X \to X$ is then defined by cases on the input:
  - In the case of $\mathsf{inl}(\star)$, just output the empty list.
  - In the case of $\mathsf{inr}(a, x)$, output the list whose head is $a$ and tail is $x$.

**Exercise 1.1.2.** *Define some other common algebraic structures using functors.*

So without initiality, what we have is only a formula for building some generic objects with this shape. But there is nothing really forcing the objects to take a coproduct shape, or to behave well. In fact, the initiality enforces the self-similarity of the structure map $\alpha$, which we will see in the next section.

## 1.2 Lambek's Theorem

There seems to be some parallel between the structure of $F(X)$ and $X$ itself. For example, the structure map $\alpha : F(X) \to X$ seemed to be perfectly split into the same cases as $X$ itself. It turns out that this structure map itself is an isomorphism.

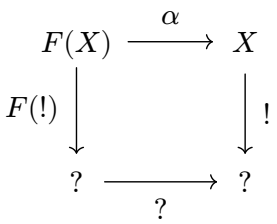
**Theorem 1.2.1 (Lambek's theorem)**

Given some functor $F$ and some $F$-algebra $(X, \alpha)$. If $(X, \alpha)$ is initial, then $\alpha$ is an isomorphism.

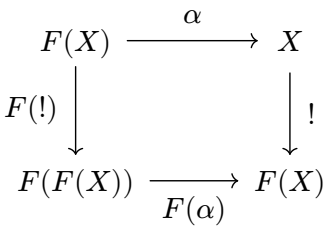
*Proof.* We begin with our structure map

$$F(X) \xrightarrow{\alpha} X$$

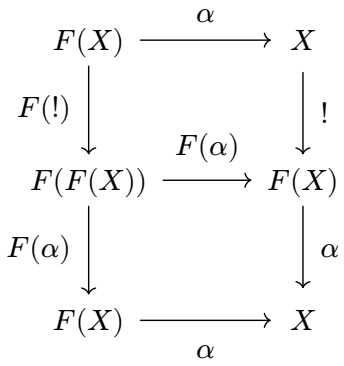
which is initial in the category of $F$-algebras. Let us "use" this initiality by coming up with another $F$-algebra to map this to. We will use ! to denote the catamorphism – the unique morphism out of the initial $F$-algebra.



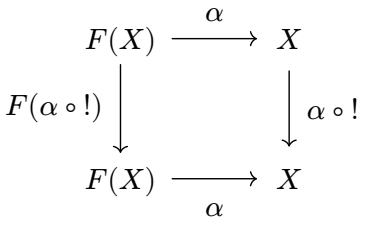

We will choose the particular $F$-algebra defined by applying $F$ again:




If we tack on a tautological square to the bottom, we get:




We can then "squish" the square using composition, and also using the functoriality of $F$ to simplify $F(\alpha) \circ F(!)$ into $F(\alpha \circ !)$:




This is exactly the $F$-algebra morphism diagram showing that $\alpha \circ !$ is a morphism from the $F$-algebra $(X, \alpha)$ to itself. Since the morphism out of an initial object is unique, we know that it must coincide with the identity morphism, so $\alpha \circ ! = \mathsf{id}_X$.

This is the first half of our proof. It remains to be shown that $! \circ \alpha = \mathsf{id}_{F(X)}$. Using the result that $\alpha \circ ! = \mathsf{id}_X$, apply $F$ to both sides, getting $F(\alpha \circ !) = F(\mathsf{id}_X)$, which by functoriality yields $F(\alpha) \circ F(!) = \mathsf{id}_{F(X)}$. But looking at this square from above:

$$\begin{array}{ccc} F(X) & \xrightarrow{\alpha} & X \\ {\scriptstyle F(!)}\downarrow & & \downarrow{\scriptstyle !} \\ F(F(X)) & \xrightarrow[F(\alpha)]{} & F(X) \end{array}$$

We know that $F(\alpha) \circ F(!) = ! \circ \alpha$. Therefore, $! \circ \alpha = \mathsf{id}_{F(X)}$, and $\alpha$ is an isomorphism with inverse $!$. □

Since $\alpha$ is essentially defining the "constructors" of the inductive type defined by $F(X)$, one consequence of Lambek's theorem is that all constructors are invertible. For example, we can derive an inverse $\mathsf{suc}$ function that takes any $\mathsf{suc}(n)$ and produces $n$.

**Exercise 1.2.1.** *Prove that there is no non-empty initial algebra for the functor $F(X) = A \times X$ in* **Set**, *for any arbitrary object $A$. What does this mean in English?*

*Applying Lambek's theorem.* Consider the list example again, $F(X) = 1 + A \times X$. Let us derive the initial $F$-algebra using Lambek's theorem. First, Lambek's theorem tells us that for initial algebras, $X \cong F(X)$ necessarily. Thus, $X \cong 1 + A \times X$. Since this is a fixpoint, we can start unfolding $X$ and see what happens:

$$\begin{aligned} X &\cong 1 + A \times X \\ &\cong 1 + A \times (1 + A \times X) \\ &\cong 1 + A \times (1 + A \times (1 + A \times X)) \\ &\cong \ldots \\ &\cong 1 + (A \times 1) + (A \times A \times 1) + (A \times A \times A \times 1) + \ldots \\ &\cong 1 + A + A^2 + A^3 + \ldots \end{aligned} \tag{3}$$

This is a union of $n$-products of $A$ for all $n$, which is exactly the set of all $n$-length sequences in $A$.

# 2 Coalgebra

As we know, we can get dual concepts in category theory for free by reversing the arrows. By dualizing algebras, we get coalgebras. Let's start with the definition and then try to extract some intuitions from it.

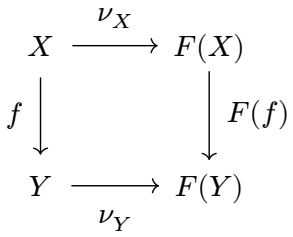
**Definition 2.1 ($F$-coalgebra)**

Let $\mathcal{C}$ be an ambient category. An $F$-coalgebra is a pair $(X, \nu)$ where $X \in \mathsf{Ob}(\mathcal{C})$, $F$ is an endofunctor $\mathcal{C} \to \mathcal{C}$ and $\nu : X \to F(X)$.

And as usual, we must also define morphisms:

**Definition 2.2 ($F$-coalgebra morphism)**

Let $(X, \nu_X)$ and $(Y, \nu_Y)$ be two $F$-coalgebras in a category $\mathcal{C}$. An $F$-coalgebra morphism is then a morphism $f : X \to Y$ such that the following diagram commutes:




Similar to how in algebras, we had the intuition of building up some kind of "input data" to our $\alpha$ morphism which then represented the constructors for our $X$, we would like to think of coalgebras as functions out of $X$, giving us some *one-step* observation data that may involve $X$.

Let's work through an example that might solidify this intuition. Once again, we are working temporarily in the category of sets, so let $X$ be a set. Consider the endofunctor $F(X) = 1 + A \times X$. In algebras, this corresponds to the signature for list-y structures.

We will see now what the dual to list-y things are. To complete the definition of an $F$-coalgebra, we need a structure map of type $\nu : X \to F(X)$, which when unfolded, is $\nu : X \to 1 + A \times X$. Whereas in algebras, the structure map served a purpose of consuming the structure into a single $X$, here we have $\nu$ expanding the $X$ into its observable data.

Our intuition tells us that $X$ is supposed to be list-y. So let's say our $\nu$ does case analysis to see if our list is cons or nil. In the case of cons, we return $A \times X$, which is the head and tail separately. In the case of nil, we return 1. So whereas our $F$-algebra from before was building *up* a list, now our $F$-coalgebra is essentially mapping *out* of a list.

Compared to the algebraic version, this one sort of axiomatizes observations out of the list rather than building up the exact list structure. We assume the list exists as a mystery object, and are able to decompose it into an element and "the rest of the stream," which continues being the mystery object.

We also have a version of Lambek's theorem for coalgebra:

**Theorem 2.3 (Lambek's theorem for coalgebra)**

Given some functor $F$ and some $F$-coalgebra $(X, \nu)$. If $(X, \nu)$ is terminal, then $\nu$ is an isomorphism.

**Exercise 2.1.** *Prove Theorem 2.3.*

Of course, with coalgebraic lists, we could also imagine a situation where we have a definition that continuously returns an element of the $A \times X$ side, and never returns the 1. In this case, the list would never end. We cannot formulate this type using algebraically defined data (Exercise 1.2.1), but with coalgebraically defined codata, this is completely in bounds. In fact, we may even specify that the stream *cannot* ever end. In this case, we only need to use signature functor $F(X) = A \times X$, opting not to include the extra 1. This is useful for reasoning about applications such as servers which are designed to run and continue responding to input forever. In the next section we will discuss the terminal $F$-coalgebra and coinduction.

## 2.1 Coinduction and bisimulation

Recall that a terminal object has morphisms into it from every other object. These morphisms are typically called **anamorphisms**. To see an example, let's work in **Set** again, with the stream functor, where $F(X) = A \times X$, for some type $A$. This represents streams of elements with type $A$.

We can use Theorem 2.3 to derive what the terminal coalgebra should look like. Starting with $X \cong A \times X$, we can unfold several times to get $X \cong A \times A \times A \times \ldots$. This is an unending product of $A$ by itself, which is the same as the function space $\mathbb{N} \to A$. We can also write this as $A^{\mathbb{N}}$, using exponential notation for functions.

Next is to define the structure map, $\nu_X : X \to A \times X$. Of course, we don't really have a choice in this matter either – it must follow from the terminality conditions. An obvious choice would be to just pick the head and tail and return that as a pair.

**Definition 2.1.1**

The structure map for the terminal $F$-coalgebra defined by $F(X) = A \times X$ is

$$\begin{aligned} \nu_X &: X \to A \times X \\ \nu_X((a_0, a_1, a_2, \ldots)) &= a_0, (a_1, a_2, \ldots) \end{aligned} \tag{4}$$

But does this choice work? We must check that all the rules are satisfied.

**Lemma 2.1.2**

For any $F$-coalgebra $(Y, \nu_Y)$, there exists a morphism $f : (Y, \nu_Y) \to (X, \nu_X)$.

*Proof.* First, we must show that for any $(Y, \nu_Y)$, that an $f : Y \to X$ even exists. We will do so by generating a sequence of $a$'s from $\nu_Y$. First, begin with $y \in Y$, and run $\nu_Y(y)$ to get $y_1 : A \times Y$. We'll take the first element $\mathsf{head}(y_1) : A$ as the first observation, and keep the rest which is $\mathsf{tail}(y_1) : Y$. Then, we can repeat the process, running $\nu_Y(y_1)$ to get $y_2 : A \times Y$. By continuing to repeat this process, we'll just end up with an infinite series of $a$'s, which we can write as $a_0, a_1, a_2, \ldots$. We'll refer to this as the *behavior* of $Y$. Notice that we didn't need to depend on what the structure of $Y$ was.

Now we need to verify that our choice of $f$ is well-behaved, with regards to all of the conditions we defined earlier. Let us revisit the commuting square from Definition 2.2:

$$\begin{array}{ccc} Y & \xrightarrow{\nu_Y} & A \times Y \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle F(f)} \\ A^{\mathbb{N}} & \xrightarrow[\nu_X]{} & A \times A^{\mathbb{N}} \end{array}$$

Translating this to an equation, we get:

$$\nu_X \circ f = F(f) \circ \nu_Y \tag{5}$$

First, note that it is necessary that $F(f) = \mathsf{id} \times f$. This is because of the functoriality of $F$, similar to how our $F$s earlier had to be a coproduct. So really, we have:

$$\nu_X \circ f = (\mathsf{id} \times f) \circ \nu_Y \tag{6}$$

This requires us to show that $\nu_Y$ indeed produces the head element of the list that we are manipulating in $\nu_X$. More literally, if we have $y \in Y$, then $f(\mathsf{tail}(\nu_Y(y)))$ must produce the same thing as just taking the tail end of $f(y)$. This is just true by definition. To spell it out with elements, we have:

$$\begin{aligned} \nu_X(f(y)) &= (\mathsf{id} \times f)(\nu_Y(y)) \\ \nu_X((a_0, a_1, a_2, \ldots)) &= \\ a_0, (a_1, a_2, \ldots) &= \\ &= (\mathsf{id} \times f)(\nu_Y(y)) \\ &= (\mathsf{id} \times f)(a_0, y_1) \\ &= a_0, f(y_1) \end{aligned} \tag{7}$$

As noted above, $y_1$ is just the next sequence after $y$, so it will produce $a_1, a_2, \ldots$ by $f$. So this commuting square checks out. □

**Lemma 2.1.3**

The morphism $f$ defined in Lemma 2.1.2 is unique.

For this, we'll need to use a different kind of reasoning than before. We will have to use **coinduction**, which is dual to induction and is perfect for working with codata such as streams.

*Proof.* Suppose we have $g : Y \to A^{\mathbb{N}}$ that respects the square. We would like to show that $f = g$. Extensionally, this means $f$ and $g$ produce the same output on all inputs, as functions. Now, $g$ is an unknown function, but it has the shape $Y \to A^{\mathbb{N}}$, so we know it produces an infinite stream of $a$'s. Furthemore, we know that:

$$\begin{array}{ccc} Y & \xrightarrow{\nu_Y} & A \times Y \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \mathsf{id} \times g} \\ A^{\mathbb{N}} & \xrightarrow[\nu_X]{} & A \times A^{\mathbb{N}} \end{array}$$

which says that $\nu_X \circ g = (\mathsf{id} \times g) \circ \nu_Y$. This means at least the first element of the sequence produced by $g$ must be picked out by $\nu_Y$. Otherwise, $\mathsf{id}(\mathsf{head}(\nu_Y(y)))$ would not produce the same result as $\mathsf{head}(\nu_X(g(y)))$. Thus, the first element of the sequence must be $\mathsf{head}(\nu_Y(y))$.

Then, regarding the tail end of the sequence, we would like to argue coinductively that it must follow the same shape. Specifically, let $\nu_Y(y) = (a_0, y')$. Following the top-right path, we can see that the tail part of the result is $g(y')$. Since $g$ is unknown, we can't really say what this is yet, but we can look at the left-bottom path, which tells us that the tail part of the result must be $\mathsf{tail}(g(y))$, since as a

reminder, our $\nu_X$ specifically splits a sequence $A^{\mathbb{N}}$ into its $\mathsf{head}$ and $\mathsf{tail}$ parts. This tells us that $g(y') = \mathsf{tail}(g(y))$, which is exactly what we expect.

So putting this together with $f$, we know that both $\mathsf{head}(f(y))$ and $\mathsf{head}(g(y))$ result in $\mathsf{fst}(\nu_Y(y))$. So $f(y)$ and $g(y)$ agree on that part. For the tail segment, we know that $\mathsf{tail}(g(y)) = g(y')$. It turns out this is also true of $f$, due to how we defined it: $\mathsf{tail}(f(y)) = f(y')$. Thus, to determine if $\mathsf{tail}(f(y)) = \mathsf{tail}(g(y))$, it suffices to show that $f(y') = g(y')$. But since $y'$ is exactly self-similar to $y$, by coinduction, we know that $f(y') = g(y')$. □

In the above proof, we used function extensionality to show that $f$ and $g$ are equal. But for showing two coalgebra are "the same," it is often more convenient to use a weaker notion of equivalence. A classic example is of automata, which we will see more about in a later section. It is completely sensical to have two automata that capture the same language (yielding the same transitions), yet have different non-reconcilable states. For this, we will introduce bisimulation.

**Definition 2.1.4 (Bisimulation of streams)**

Let $(X, \nu_X)$ and $(Y, \nu_Y)$ be $F$-coalgebras, where $F(X) = A \times X$. Let $R$ be a relation between $X$ and $Y$ – specifically, $R \subset X \times Y$. $R$ is a **bisimulation** iff for any $(x, y) \in R$:

$$\mathsf{head}(x) = \mathsf{head}(y) \wedge (\mathsf{tail}(x), \mathsf{tail}(y)) \in R \tag{8}$$

Then, it is said that $X$ and $Y$ are **bisimilar**. This sort of looks like a more generalized version of the reasoning we did as a part of the uniqueness proof, but with equality replaced with the relation $R$. But the reasoning is still stream-specific, and relatively ad-hoc. We'd like to not have to make arbitrary bisimulation conditions for every type we'd like to analyze.

It turns out if we just take everything that has specifically to do with streams and abstract it, we can arrive at a more general notion of bisimulation.

**Definition 2.1.5 (Aczel-Mendler bisimulation)**

Let $\mathcal{C}$ be a category with products, and $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. Let $(X, \nu_X)$ and $(Y, \nu_Y)$ be $F$-coalgebras. Let $R$ be a subobject of the product object $X \times Y$ with projection morphisms $\pi_X$ and $\pi_Y$ such that there exists a morphism $\nu_R : R \to F(R)$ such that the following diagram commutes:

$$\begin{array}{ccccc} & & \overset{\pi_1}{\longleftarrow} & & \overset{\pi_2}{\longrightarrow} \\ X & & R & & Y \\ {\scriptstyle \nu_X}\downarrow & & \downarrow{\scriptstyle \nu_R} & & \downarrow{\scriptstyle \nu_Y} \\ F(X) & \underset{F(\pi_1)}{\longleftarrow} & F(R) & \underset{F(\pi_2)}{\longrightarrow} & F(Y) \end{array}$$

Let's quickly make sense of this in the context of our previous Definition 2.1.4 again. We have some relation $R$ which is a subobject of the product. This corresponds exactly with $R \subset X \times Y$. The rough interpretation of this is that at the current state, $X$ and $Y$ are related through $R$.

Then, we have the object $F(R)$, which is a lift of the product through $F$. So for streams, since $F(X) = A \times X$, we have $F(R) \subset (A \times X) \times (A \times Y)$. Notice how this part is completely determined by the specific $F$'s interaction with the relation. The $\nu_R$ lifting operation simply calls `head` and `tail` component-wise on both the left and right sides. Then, the $F(\pi_1)$ projection operator would grab the components separately and turn them into a product, and $F(\pi_2)$ does the corresponding thing for $F(Y)$.

The commuting squares, like earlier, essentially force us to take this component-wise motion.

**Exercise 2.1.1.** *Does the category of $F$-algebras have an initial object for any $F$? Does the category of $F$-coalgebras have a terminal object for any $F$?*

# 3 Automata

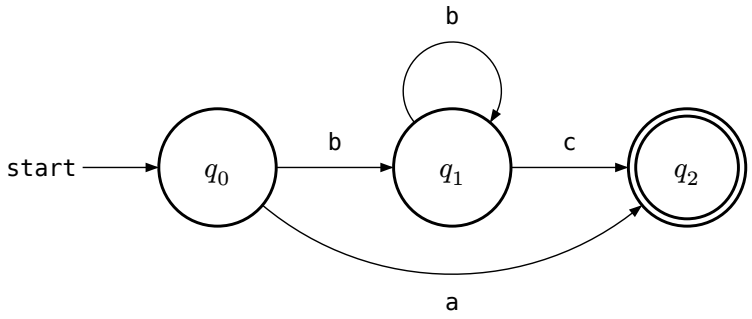
We have been working a lot with streams so far, so from here on we will be introducing some richer coalgebras that have applications in other fields. Automata are a classic example of coalgebras in practice. At a high level, an automaton is a state machine that has states and transitions between them. It has a notion of "accept" states which allow the machine to respond with a success response. This is often used in recognizing languages, such as in parsing for compiler development, but also for network protocols and hardware state machines.

We'll look at how automata can be abstracted using algebras and coalgebras, and then work through a generalization of Brzozowski's minimization algorithm using our definition, in order to see how to use the duality of algebras and coalgebras in order to achieve minimization. Let's start with a formal definition of automata.

**Definition 3.1 (Deterministic automaton)**

For some alphabet $\Sigma$, a **deterministic automaton (DA)** is a 4-tuple $(Q, \delta, i, F)$:
- $Q$, a set of states
- $\delta : Q \times \Sigma \to Q$, a transition map
- $i \in Q$, an initial state
- $F \subseteq Q$, final states

Visually, we can represent an automaton like this:




This visualization represents the formal automaton defined as $(\{q_0, q_1, q_2\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \delta, q_0, \{q_2\})$ where $\delta$ would map:
- $(\mathtt{a}, q_0) \mapsto q_2$

- $(\texttt{b}, q_0) \mapsto q_1$
- $(\texttt{b}, q_1) \mapsto q_1$
- $(\texttt{c}, q_1) \mapsto q_2$
- all other combinations of states would resolve to an implicit "error" state (which would also be in $Q$), which only has transitions back to itself. It does not appear in $F$, the set of final states. For all intents and purposes in these notes, we can safely just ignore the fact that it exists, although we should formally make a mental note of the erroring behavior, since $\delta$ is not a partial function.

The double-circle means that $q_2$ is a final state, in other words, that $q_2 \in F$. This automaton will recognize a language consisting of either the exact string `a` or any non-zero number of `b`s followed by a `c`. This is typically written using the terse regular expression syntax as `a|(b+)c`.

Formally, a language $\mathcal{L}$ is just a series of strings consisting of symbols in the alphabet $\Sigma$. An automaton **recognizes** a language if for every string $s$ in the language, running $s$ through the automaton would result in a final state.

There is also a variant which allows non-deterministic transitions – for example, it would be allowed to have multiple arrows from the same state with the same transition. This machine is known as a *non-deterministic* automaton.

**Definition 3.2 (Non-deterministic automaton)**

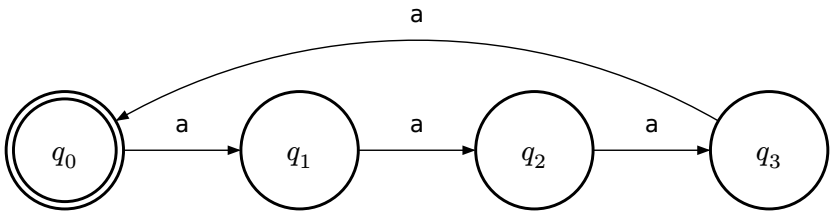
For some alphabet $\Sigma$, a **non-deterministic automaton** is a 4-tuple $(Q, \delta, i, F)$:

- $Q$, a set of states
- $\boxed{\delta : Q \times \Sigma \to \mathcal{P}(Q), \text{ a transition map}}$
- $i \in Q$, an initial state
- $F \subseteq Q$, final states

The $\delta$ is the only difference from the DA definition. Instead of only transitioning to one state, it could possibly transition to any number of states. A non-deterministic automata would accept a string from a language if taking any path using the symbols of the string would result in acceptance.

Traditionally, non-deterministic automata would be converted deterministic using an algorithm such as the powerset construction, in which you would consider the set of all states that could possibly result following a particular transition. There are also some methods of evaluating non-deterministic automata directly without first doing the conversion, as the conversion may produce an explosion ($O(2^n)$) of states in the resulting DA.

For this section, we will consider a particular problem concerning automata: there is a sense in which an NFA might not be minimal. For example, consider this automaton:



You could easily imagine collapsing all of these down to a single node. Due to this, there are several minimization algorithms. A well-known algorithm, due to Brzozowski, involves reversing the NFA, determinizing it using the powerset construction, and then reversing it again. In this next section, we will investigate a category-theoretic approach by (Bonchi et al. 2014) to show why this works, specifically for the case of deterministic automata.

## 3.1 The category of automata

In order to categorify automata, we must first identify a category. We want the objects to be automata, so we must define some morphisms.

**Definition 3.1.1 (Automaton morphism)**

For some alphabet $\Sigma$, a morphism between two automata $(Q_1, \delta_1, i_1, F_1)$ and $(Q_2, \delta_2, i_2, F_2)$ is defined as a function $f : Q_1 \to Q_2$ such that the following hold:

- $f$ commutes with the transition map, or in other words, for any $s \in \Sigma$, this square commutes:

$$\begin{array}{ccc} Q_1 & \xrightarrow{f} & Q_2 \\ {\scriptstyle \delta_1(-,s)}\downarrow & & \downarrow{\scriptstyle \delta_2(-,s)} \\ Q_1 & \xrightarrow[f]{} & Q_2 \end{array}$$

- $f$ preserves the initial state, or in other words $f(i_1) = i_2$
- $f$ preserves the final states, or in other words $f(q) \in F_2$ for all $q \in F_1$

Thus, we can form a category of DAs:

**Definition 3.1.2 (Category of deterministic automata, $\mathsf{DA}_{\mathcal{L}}$)**

For some alphabet $\Sigma$ and language over that alphabet $\mathcal{L}$, the category $\mathsf{DA}_{\mathcal{L}}$ is defined as a category where:

- objects are DAs that recognize the language $\mathcal{L}$
- morphisms are defined as in Definition 3.1.1

It's important that the category itself is parameterized by a language $\mathcal{L}$, since morphisms need to preserve the language. If this was not the case, then we could not have initial or final objects at all – it would essentially require that all automata at all recognize the same language, which is not the case. Such a category would still exist, and in fact $\mathsf{DA}_{\mathcal{L}}$ is a subcategory of that category, but it is not useful to us.

**Remark**

We can characterize a DA categorically as *both* and algebra and as a coalgebra. As an algebra, it uses the signature functor

$$F(Q) = 1 + \Sigma \times Q \tag{9}$$

In order to represent an automata as an algebra, we would need a structure map of type $\alpha : 1 + \Sigma \times Q \to Q$. The $1 \to Q$ part represents the initial state, since it takes no input, and $\Sigma \times Q \to Q$ is the type of a transition map – it takes a transition symbol, a state, and outputs the next state.

As a coalgebra, it can be characterized with the signature functor:

$$F(Q) = 2 \times Q^{\Sigma} \tag{10}$$

For the coalgebra case, we need a structure map $\nu : Q \to 2 \times Q^{\Sigma}$. For any input state $q \in Q$, the $Q^{\Sigma}$ part of the output represents all possible states that the input state could transition to, and 2 refers to a tag on each state that is assigned the value 1 if that state is a final state, or the value 0 if it isn't.

## 3.2 Minimality of automata

Next, let us consider the initial and terminal objects of $\mathsf{DA}_{\mathcal{L}}$. We'd like to determine some automaton that recognizes $\mathcal{L}$ and has a morphism to every other automaton. A pretty trivial "maximal" object that lets us do this is to add every single possible word as a state.

Formally, let's define this automaton:

**Definition 3.2.1 (Initial object of $\mathsf{DA}_{\mathcal{L}}$)**

Define the initial automaton $\mathsf{Init}$ as:

- $Q_{\mathsf{Init}}$ is defined as $\Sigma^*$, the set of all strings over the alphabet $\Sigma$
- $\delta_{\mathsf{Init}} : Q \times \Sigma \to Q$ simply concatenates the currently running state's string with the transition symbol
- $i_{\mathsf{Init}}$ is the state corresponding to the empty string
- $F_{\mathsf{Init}}$ are the states corresponding to all strings in $\mathcal{L}$

Of course, we must show that it is in fact initial.

**Theorem 3.2.2**

The automaton described above is initial in $\mathsf{DA}_{\mathcal{L}}$.

*Proof.* Consider some other automaton $(Q', \delta', i', F')$. We must define a morphism from $\mathsf{Init}$ to this automaton.

First, define $f : Q_{\mathsf{Init}} \to Q'$. For every state $q \in Q_{\mathsf{Init}}$, there is a corresponding string $s \in \Sigma^*$. We will define $f$ to map $s$ to a state in $Q'$ by running $Q'$ – in other words, starting with $i'$ and transforming it using $\delta'$ on each symbol of $s$, and then returning the final state that it landed in.

We must show that $f$ commutes with the transition map, or in other words, for any symbol $t \in \Sigma$:

$$f \circ \delta_{\mathsf{Init}}(-, t) = \delta'(-, t) \circ f \tag{11}$$

Let $q \in Q_{\mathsf{Init}}$ be an arbitrary state in $\mathsf{Init}$, and $s \in \Sigma^*$ be the corresponding string. Then, we are trying to show that $f(\delta_{\mathsf{Init}}(q, t)) = \delta'(f(q), t)$. Since $\delta_{\mathsf{Init}}$ is defined by concatenation, we have $f(\delta_{\mathsf{Init}}(q, t)) = f(s + t)$ on the left side. Since $f$ simply runs $\delta'$ iteratively, this is actually true *by the definition* of $f$.

It is trivial to show that $f$ also preserves the initial and final states.

In order to show $f$ is the unique such morphism, assume $g : Q_{\mathsf{Init}} \to Q'$. We need to show for any state $q$, that $f(q) = g(q)$. Since the states map one-to-one with all strings $\Sigma^*$, perform induction on strings. For the empty string, the corresponding state is the initial state, which is preserved by the requirement of the automaton morphism. Thus, $f(i_{\mathsf{Init}}) = g(i_{\mathsf{Init}}) = i'$.

For the inductive case, assume $f(q) = g(q)$. Then, for any symbol $t \in \Sigma$, identify the state corresponding to concatenating $s + t$ (once again letting $s$ be the string corresponding to $q$). We must show $f(s + t) = g(s + t)$. By the commutative square, we can rewrite this property to $\delta'(f(q), t) = \delta'(g(q), t)$. Since $f(q) = g(q)$ by our inductive hypothesis, the two sides are identical. □

*Alternate proof.* Note that automata behave algebraically. Use Theorem 1.2.1 to deduce that the structure map $1 + \Sigma \times Q \to Q$ must be an isomorphism, so:

$$Q \cong 1 + \Sigma \times Q \tag{12}$$

Expanding this, we get: $Q \cong 1 + \Sigma \times (1 + \Sigma \times (1 + \Sigma \times ...))$, which simplifies to $1 + \Sigma + \Sigma^2 + \Sigma^3 ....$ This is exactly $\Sigma^*$ – all strings of all lengths, sourced from the alphabet.[2] □

The rough design of this morphism is that for some arbitrary automaton $Q$, we're running every possible string in $\Sigma^*$ to see which states are able to be reached via $Q$, and capturing that in a single function $f$.

We're interested specifically in automata where every state is reachable. This means there's no redundancies in the states – if you had some redundant state that isn't final, then only running $\delta$ until completion would completely miss that state, excluding it from the image of the morphism out of the initial automaton. We capture this with the following definition:

**Definition 3.2.3 (Reachable automaton)**

An automaton $Q$ is **reachable** if the unique morphism out of the initial automaton $\mathsf{Init} \to Q$ is an epimorphism.

In sets, epimorphisms (set functions) capture the notion of surjectivity, which means every element of the codomain has a pre-image. Since automata are essentially sets with more structure, a morphism being epic captures the same notion, which in automata corresponds to every state being reached by a morphism. If this property is true of the morphism out of the initial automaton (which remember, runs every possible string through the automaton), we know every state is being used, which captures our desired property of reachability.

There is a similar construction on the coalgebraic side. We can imagine a terminal automaton, which has morphisms going into it from any other automaton. For some arbitrary automaton $Q$, we'd imagine that the result of a morphism should be some *observation* of the states in $Q$. So the terminal automaton should be a collective observation about all possible automata in $\mathsf{DA}_{\mathcal{L}}$.

**Definition 3.2.4 (Terminal object in $\mathsf{DA}_{\mathcal{L}}$)**

The terminal automaton $\mathsf{Term}$ is defined as:

[2] In fact, this approach can be used to derive the initial automaton. I didn't realize this until after I wrote the previous proof... However, I left the original proof in, because it actually defines the function needed for the morphism.

- $Q_{\mathsf{Term}}$ is the set of all possible subsets of $\Sigma^*$. This represents all languages that could possibly be recognized for this alphabet.
- $\delta_{\mathsf{Term}} : Q \times \Sigma \to Q$ maps every $(q, s) \in Q \times \Sigma$ to the state representing the remainder of the language after consuming the prefix $s$. This is also known as the *Brzozowski derivative*.
- $i_{\mathsf{Term}}$ is the language $\mathcal{L}$.
- $F_{\mathsf{Term}}$ is the set of all languages that contain the empty string $\epsilon$.

Staring at this definition, it may kind of make sense why this could capture all observations, but let's make this concrete by actually proving terminality.

**Theorem 3.2.5**

The automaton described above is final in $\mathsf{DA}_{\mathcal{L}}$.

*Proof.* This proof largely follows in the same vein as the initial object proof. As such, we will skip some parts of it for brevity. Just trust me 😉.

Consider some other automaton $(Q', \delta', i', F')$. We'll now define $f : Q' \to Q_{\mathsf{Term}}$. For any state $q \in Q'$, we will consider the automaton that has the same states and transitions as $Q'$, but with initial state $q$. The language that this automaton recognizes will be a set of a strings, which is a subset of $2^{\Sigma^*}$. This language is the output of $f$.

The exercise of verifying that $f$ preserves the initial and final states as well as the transition map is left to the reader. □

**Exercise 3.2.1.** *Verify that $f$ satisfies the requirements for being a $\mathsf{DA}_{\mathcal{L}}$-morphism.*

*Alternative proof.* Same as before, notice that $\mathsf{DA}_{\mathcal{L}}$ is coalgebraic and use Theorem 2.3. The structure map $\nu : Q \to 2 \times Q^{\Sigma}$ must be an isomorphism, so:

$$Q \cong 2 \times Q^{\Sigma} \tag{13}$$

Expanding this, we get

$$\begin{aligned} Q &\cong 2 \times \left(2 \times (2 \times (\ldots)^{\Sigma})^{\Sigma}\right)^{\Sigma} \\ &= 2 \times 2^{\Sigma} \times (2 \times (\ldots)^{\Sigma})^{\Sigma^2} \\ &= 2 \times 2^{\Sigma} \times 2^{\Sigma^2} \times (\ldots)^{\Sigma^3} \\ &= 2 \times 2^{\Sigma} \times 2^{\Sigma^2} \times 2^{\Sigma^3} \times \ldots \\ &= 2^{1+\Sigma+\Sigma^2+\Sigma^3+\ldots} \\ &= 2^{\Sigma^*} \end{aligned} \tag{14}$$

Thus, $2^{\Sigma^*}$ is the state space for the terminal automaton. □

The rough design for this morphism is that we're mapping the automaton to what language it recognizes, inside of $\mathsf{Term}$. In that sense, this morphism would capture the behavior of all the strings that could possibly be recognized by the automaton.

This captures another important property, which is that every state is distinct. Without this property, you could have two states which recognize the same suffix language. That wouldn't be very minimal. For minimality, we care specifically that there are no duplicate states that can just be merged. This can be captured by the definition of observability, which is dual to Definition 3.2.3:

**Definition 3.2.6 (Observable automaton)**

An automaton $Q$ is **observable** if the unique morphism into the terminal automaton $Q \to \mathsf{Term}$ is a monomorphism.

Again, to make a comparison to sets, a set function is monic if it is injective, or that distinct inputs map to distinct outputs. Thus, a morphism in $\mathsf{DA}_{\mathcal{L}}$ is monic if it maps distinct input states to distinct output states. If our output state is the terminal automaton, then two states mapping into the same output state means that they recognized the same language, which would mean they were duplicated.

Putting these together, we can finally define what a minimal automaton is:

**Definition 3.2.7 (Minimal automaton)**

An automaton $Q$ is **minimal** if it is both reachable and observable.

The reachability property enforces that no states can be deleted, and the observability property enforces that no states can be merged, at least without changing the language being recognized. These are the only two ways to reduce an automaton without changing the language. This claim could be justified by the fact that the definition of minimization used above coincides with the minimization definition for a coalgebra defined through epi-mono factorization (Bezhanishvili et al. 2012).

On the computation side of things, reachability is pretty trivial to compute algorithmically – just traverse the state machine and grab all the states visited. On the other hand, observability requires quotienting by equivalence classes of a hard-to-compute property of basically *every* state. In practice, algorithms such as Hopcroft's algorithm optimize this by incrementally comparing states and quotienting as you go.

## 3.3 Reversing automata

The crux of Brzozowski's minimization algorithm involves reversing the automaton. The intuition here is, if reachability is more straightforward, we should aim to do it. Then, we can find some dualization property that allows us to somehow transfer reachability of an automaton into observability of the reverse language, and then use reachability *again* to get both desired properties on the same language.

The actual reversal process amounts to reversing the arrows of the transition map, as well as exchanging the initial and final states. Note that normally, when you do this to a deterministic automaton, it becomes a *non-deterministic* one. This is because there could be multiple final states, but only one initial state. When you reverse it, you have a number of initial states. The algorithm requires that you then make the non-deterministic automaton deterministic, through the powerset construction.

We'll explore a construction which combines the reversing and powersetting in one go. First off, consider what the shape of output we want is, since this will guide our type-checking as we hone in towards a solution. We are starting off with a category $\mathsf{DA}_{\mathcal{L}}$ of deterministic automata, and we are trying to get some way of taking each object and obtaining the reverse automata.

Notice that the reverse automata will necessarily recognize the reverse language. So the output will be an automaton which recognizes $\mathsf{rev}(\mathcal{L})$, in other words an object in $\mathsf{DA}_{\mathsf{rev}(\mathcal{L})}$! This makes it necessary for us to provide a *functor* with this signature:

$$\mathsf{DA}_{\mathcal{L}} \to \mathsf{DA}_{\mathsf{rev}(\mathcal{L})} \tag{15}$$

As always, we need to find a functor, because the rules governing morphisms will restrict the way the objects can even be mapped to only well-behaved maps. Otherwise, we could produce some degenerate maps. This will also give us more benefits, as we will see later.

Time to define action on objects. For some arbitrary object $Q \in \mathsf{DA}_{\mathcal{L}}$, we'd like to produce the reverse automaton. We can start with what we expect $\delta_{Q'}$, the $\delta$ of the transformed automaton to be. It should be some sort of reversing function, so we should be taking the pre-image of $\delta_Q$:

$$\delta_{Q'} :\equiv (q, s) \mapsto \{q' \in Q \mid \delta(q', s) = q\} \tag{16}$$

But this doesn't type-check. The result is a set, since multiple $q'$s could map to the same $q$ under the old $\delta$. This means our state space fundamentally needs to be a powerset of $Q$. In that case, $\delta_{Q'}$'s type would need to be $\mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$. We can update the function to look like this:

$$\delta_{Q'} :\equiv (q, s) \mapsto \{q' \in Q \mid \delta(q', s) \in q\} \tag{17}$$

We can essentially fill in the rest of the definitions now:

- $Q'$'s state space is $\mathcal{P}(Q)$
- $\delta_{Q'}$ was defined above
- $i_{Q'}$ is the set containing all the final states in $Q$
- $F_{Q'}$ is the set containing the initial state in $Q$

Here is where the magic happens. The function $\delta_Q$ is essentially a function over sets. Even though it's doing $Q \times \Sigma \to Q$, this is equivalent to $Q \to (\Sigma \to Q)$, which is $Q \to Q^{\Sigma}$. Over in our reverse language, we have $\delta_{Q'}$ having type $\mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$, which is the same as $\mathcal{P}(Q) \to \mathcal{P}(Q)^{\Sigma}$.

**This is exactly the behavior of the contravariant power functor!**

Let's step back one step and build up to this insight. If we just considered the category of sets $\mathbf{Set}$, the *powerset functor* $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ maps every set $X$ to its power set $\mathcal{P}(X)$. In particular, for morphisms, it has type $\mathcal{P}\left(X \xrightarrow{f} Y\right) : \mathcal{P}(X) \to \mathcal{P}(Y)$. For any subset of $X$, it maps $f$ restricted to that subset to obtain an image that is in the subset of $Y$.

There is also a contravariant version, which we will denote $\mathcal{P}^* : \mathbf{Set}^{\mathsf{op}} \to \mathbf{Set}$. In particular, the action on morphisms has type $\mathcal{P}^*\left(X \xrightarrow{f} Y\right) : \mathcal{P}(Y) \to \mathcal{P}(X)$. This action maps a subset of $Y$ into the *pre-image* under $f$, which is a subset of $X$.

This is exactly what our $\delta_{Q'}$ is doing! In essence, our desired transformation is performing the covariant powerset functor on each of the components of the automata. We can concretize this idea by constructing a forgetful functor $U : \mathsf{DA}_{\mathcal{L}} \to \mathbf{Set}$ which simply takes the set of states and forgets all the other structure. Then, we can see clearly that:

$$U(Q) \xrightarrow{\mathcal{P}^*} U(Q') \tag{18}$$

Here we introduce a powerful property of the contravariant powerset functor:

**Lemma 3.3.1**

The contravariant powerset functor $\mathcal{P}^*$ is left adjoint to its own opposite functor $\mathcal{P}^{*\mathsf{op}}$.

*Proof.* The contravariant powerset functor is $\mathcal{P}^* : \mathbf{Set}^{\mathsf{op}} \to \mathbf{Set}$. Its opposite functor would be $\mathcal{P}^{*\mathsf{op}} : \mathbf{Set} \to \mathbf{Set}^{\mathsf{op}}$. This amounts to showing that for any $X, Y \in \mathbf{Set}$:

$$\begin{aligned} \mathbf{Hom}_{\mathbf{Set}}(\mathcal{P}^*(X), Y) &\cong \mathbf{Hom}_{\mathbf{Set}}(X, \mathcal{P}^{*\mathsf{op}}(Y)) \\ \mathbf{Hom}_{\mathbf{Set}}(Y, \mathcal{P}^*(X)) &\cong \mathbf{Hom}_{\mathbf{Set}}(X, \mathcal{P}^{*\mathsf{op}}(Y)) \end{aligned} \tag{19}$$

and really, both $\mathcal{P}^*$ and $\mathcal{P}^*$ have the same action on sets, which is to turn them into their power sets. So really, we are trying to show:

$$\mathbf{Hom}(Y, \mathcal{P}(X)) \cong \mathbf{Hom}(X, \mathcal{P}(Y)) \tag{20}$$

We can show this via creating a bijection. First, the forward direction: define $\mathsf{fwd} : \mathbf{Hom}(Y, \mathcal{P}(X)) \to \mathbf{Hom}(X, \mathcal{P}(Y))$ with the function $\lambda(f : Y \to \mathcal{P}(X)).\lambda(x : X).\{y \in Y \mid x \in f(y)\}$.[3]

Then, define the inverse direction $\mathsf{bwd} : \mathbf{Hom}(X, \mathcal{P}(Y)) \to \mathbf{Hom}(Y, \mathcal{P}(X))$ with the function $\lambda(g : X \to \mathcal{P}(Y)).\lambda(y : Y).\{x \in X \mid y \in g(x)\}$.

It's easily to show the inverses compose to the identity:

$$\begin{aligned} \forall f.\quad f &= \mathsf{bwd}(\mathsf{fwd}(f)) \\ &= \lambda(y : Y).\{x \in X \mid y \in \mathsf{fwd}(f)(x)\} \\ &= \lambda(y : Y).\{x \in X \mid y \in (\lambda(x : X).\{y \in Y \mid x \in f(y)\})(x)\} \\ &= \lambda(y : Y).\{x \in X \mid y \in \{y \in Y \mid x \in f(y)\}\} \\ &= \lambda(y : Y).\{x \in X \mid x \in f(y)\}\} \\ &= \lambda(y : Y).f(y)\}\} \\ &= f \end{aligned} \tag{21}$$

and the other way:

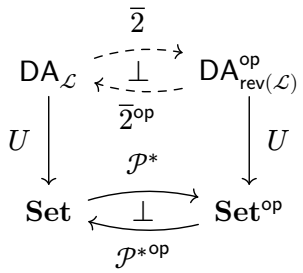
$$\begin{aligned} \forall g.\quad g &= \mathsf{fwd}(\mathsf{bwd}(g)) \\ &= \lambda(x : X).\{y \in Y \mid x \in \mathsf{bwd}(g)(y)\} \\ &= \lambda(x : X).\{y \in Y \mid x \in (\lambda(y : Y).\{x \in X \mid y \in g(x)\})(y)\} \\ &= \lambda(x : X).\{y \in Y \mid x \in \{x \in X \mid y \in g(x)\}\} \\ &= \lambda(x : X).\{y \in Y \mid y \in g(x)\} \\ &= \lambda(x : X).g(x) \\ &= g \end{aligned} \tag{22}$$

□

[3] Sorry about the sudden lambda notation, I needed to see the type of the argument inline for a second...

Lemma 3.3.1 is essentially saying that the contravariant powerset functor is self-dual. The strategy now is: if we can somehow lift this self-duality into $\mathsf{DA}_{\mathcal{L}}$, we will be able to get an operation that turns reachable automata into observable automata and vice versa.

We will devise a functor called $\overline{2}$, which is self-dual, that makes this diagram commute:




This essentially lifts the self-duality of $\mathcal{P}^*$ into $\mathsf{DA}_{\mathcal{L}}$. The paper (Bonchi et al. 2014) notes that this lifting is true generically

**Definition 3.3.2 (Self-dual powerset functor lift)**

Define the functor $\overline{2} : \mathsf{DA}_{\mathcal{L}} \to \mathsf{DA}^{\mathsf{op}}_{\mathsf{rev}(\mathcal{L})}$ as:

- action on objects: $\overline{2}(Q)$ applies the contravariant powerset functor to all components of the automaton:
  - $Q$ is transformed into the power set of states $\mathcal{P}(Q)$
  - $\delta_Q$ is transformed into the pre-image map as described in Equation 17. Note that this recognizes the reverse language.
  - $i_Q$ is transformed into the set of all final states
  - $F_Q$ is transformed into the set of all initial states
- action on morphisms maps a morphism $Q_1 \overset{f}{\to} Q_2$ to its inverse mapping $\mathcal{P}(Q_2) \overset{\overline{\mathcal{P}}(f)}{\to} \mathcal{P}(Q_1)$

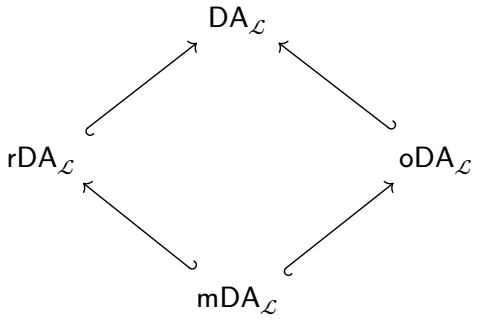
*Proof.* Omitting for brevity. See (Bonchi et al. 2014, Proposition 9.1) for the full proof. □

## 3.4 (Co-)reflective subcategories

In order to see how all of this machinery comes together, we will introduce some subcategories of $\mathsf{DA}_{\mathcal{L}}$:

- $\mathsf{rDA}_{\mathcal{L}}$ is the category of *only* reachable automata in $\mathsf{DA}_{\mathcal{L}}$
- $\mathsf{oDA}_{\mathcal{L}}$ is the category of *only* observable automata in $\mathsf{DA}_{\mathcal{L}}$
- $\mathsf{mDA}_{\mathcal{L}}$ is the category of *only* minimal automata in $\mathsf{DA}_{\mathcal{L}}$

Since minimal automata are ones that are both reachable and observable, it must also be true that $\mathsf{mDA}_{\mathcal{L}}$ is a subcategory of both $\mathsf{rDA}_{\mathcal{L}}$ and $\mathsf{oDA}_{\mathcal{L}}$. This means the subcategory relationships can be represented as a sort of diamond diagram:




The arrows between them represent inclusion functors. In fact, these are a special kind of subcategory known as a *reflective* (and *co-reflective*) subcategory. For reflective subcategories, this means that the inclusion functor has a left adjoint. Dually, for co-reflective subcategories, this means that the inclusion functor has a right adjoint. Let us see why this holds.

**Theorem 3.4.1**

$\mathsf{rDA}_{\mathcal{L}}$ is a co-reflective subcategory of $\mathsf{DA}_{\mathcal{L}}$. In other words, supposing $I : \mathsf{rDA}_{\mathcal{L}} \hookrightarrow \mathsf{DA}_{\mathcal{L}}$ is the inclusion functor, there exists a functor $R : \mathsf{DA}_{\mathcal{L}} \to \mathsf{rDA}_{\mathcal{L}}$ such that

$$I \dashv R \tag{23}$$

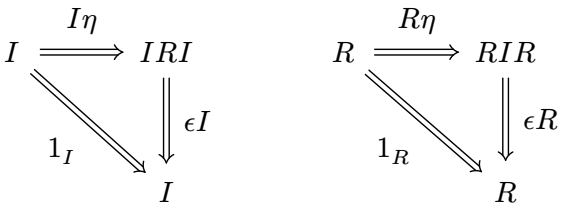
*Proof.* For our action on objects, assume we are dealing with an object $Q : \mathsf{DA}_{\mathcal{L}}$. We are looking to construct a functor $R$ that is right-adjoint to $I$.

To do this, we must first define $R$, then construct a unit $\eta : \mathsf{id}_{\mathsf{rDA}_{\mathcal{L}}} \Rightarrow R \circ I$ and a counit $\epsilon : I \circ R \Rightarrow \mathsf{id}_{\mathsf{DA}_{\mathcal{L}}}$ that satisfies the triangle inequalities below. Since we are looking for a functor that takes an automaton with possibly unreachable states into its closest reachable approximation, the most obvious thing to do is simply discarding all unreachable states. Using the machinery we used above to define reachability, we can say that $R(Q)$ takes the codomain of the morphism $\mathsf{Init} \to Q$. Thus, $R$ is obviously reachable, and lies in $\mathsf{rDA}_{\mathcal{L}}$.

Next, define the unit: for some $Q' : \mathsf{rDA}_{\mathcal{L}}$, we must define $\eta_{Q'} : Q' \to R(I(Q'))$. We can define this as the *identity* morphism, since this round trip is a no-op both ways.

For the counit, for some $Q : \mathsf{DA}_{\mathcal{L}}$, we must define $\epsilon_Q : I(R(Q)) \to Q$. Since $R(Q)$ contains the subset of $Q$'s states that are reachable, it's actually a subset. So $\epsilon_Q$ can just be a subset inclusion (notice that unlike above, $I(R(Q))$ may be a different object than just $Q$).

Let's check that our definitions for unit and counit satisfy the triangle identities:




For the first triangle, we can write this equationally as:

$$\epsilon I \circ I\eta = 1_I \tag{24}$$

Instantiating this with our $Q' : \mathsf{rDA}_{\mathcal{L}}$:

$$\begin{gathered}(\epsilon I)_{Q'} \circ (I\eta)_{Q'} = (1_I)_{Q'} \\ I(Q') \overset{(I\eta)_{Q'}}{\longrightarrow} I(R(I(Q'))) \overset{(\epsilon I)_{Q'}}{\longrightarrow} I(Q') = I(Q') \overset{(1_I)_{Q'}}{\longrightarrow} I(Q')\end{gathered} \tag{25}$$

Well, we have defined $\eta$ to be a no-op, so the left side $(I\eta)_{Q'}$ is the identity. Since $\epsilon$ is a subset inclusion, for reachable automata, $(\epsilon I)_{Q'}$ is *also* the identity. Thus, identity compose identity is the identity.

For the second triangle, we can write this equationally as:

$$\epsilon R \circ R\eta = 1_R \tag{26}$$

Instantiating this with $Q : \mathsf{DA}_{\mathcal{L}}$:

$$\begin{gathered}(\epsilon R)_Q \circ (R\eta)_Q = (1_R)_Q \\ R(Q) \overset{(R\eta)_Q}{\longrightarrow} R(I(R(Q))) \overset{(\epsilon R)_Q}{\longrightarrow} R(Q) = R(Q) \overset{(1_R)_Q}{\longrightarrow} R(Q)\end{gathered} \tag{27}$$

Again, we have $\eta$ is a no-op, so the left side is the identity. The right side goes from reachable to reachable, so the subset inclusion $\epsilon$ is simply restricted to the identity by default here.

Both triangle inequalities check out, so our adjunction is defined.

□

Similarly, for $\mathsf{oDA}_{\mathcal{L}}$:

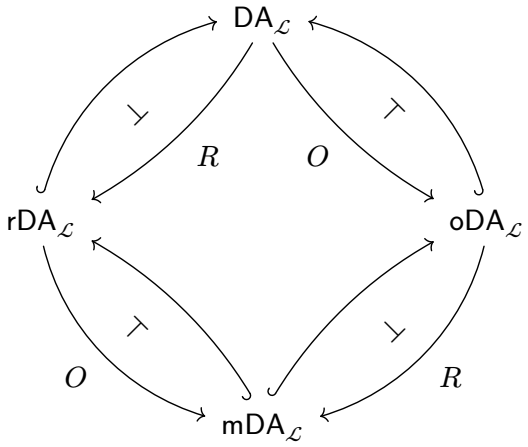
**Theorem 3.4.2**

$\mathsf{oDA}_{\mathcal{L}}$ is a reflective subcategory of $\mathsf{DA}_{\mathcal{L}}$. In other words, supposing $I : \mathsf{oDA}_{\mathcal{L}} \hookrightarrow \mathsf{DA}_{\mathcal{L}}$ is the inclusion functor, there exists a functor $\mathrm{O} : \mathsf{DA}_{\mathcal{L}} \to \mathsf{oDA}_{\mathcal{L}}$ such that

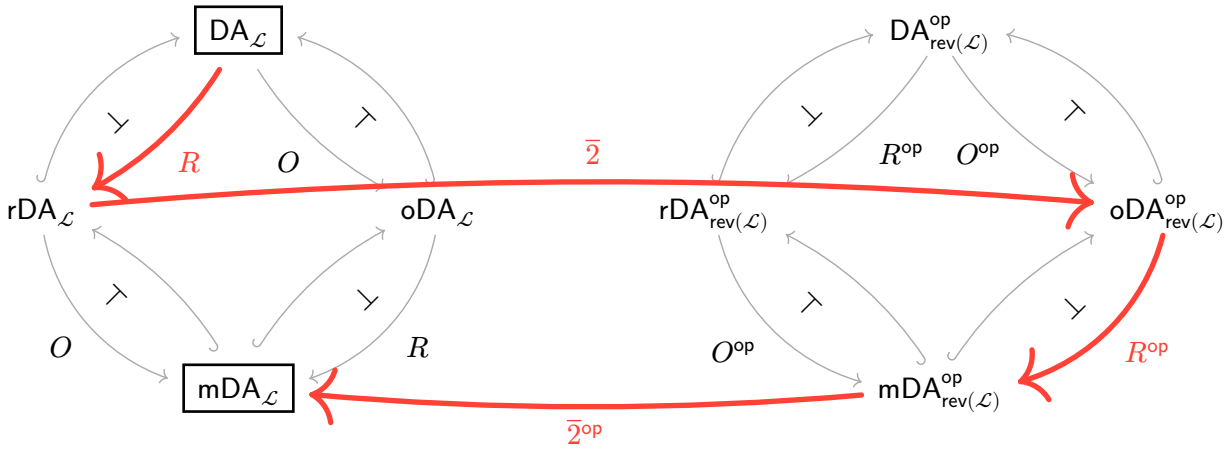
$$O \dashv I \tag{28}$$

**Exercise 3.4.1.** *Prove Theorem 3.4.2. The proof should look mostly like a dualization of the proof of Theorem 3.4.1.*

The corresponding adjoint functor for $\mathsf{rDA}_{\mathcal{L}}$, called $R$, then restricts automata in the input category to their reachable subset. Similarly, the corresponding adjoint functor for $\mathsf{oDA}_{\mathcal{L}}$, called $O$, restricts automata in *its* input category to their observable subset. And then at the bottom, $\mathsf{mDA}_{\mathcal{L}}$ is both a reflective subcategory and co-reflective subcategory of $\mathsf{oDA}_{\mathcal{L}}$ and $\mathsf{rDA}_{\mathcal{L}}$, respectively. Let us update the diamond diagram to see these new functors:




Finally, due to our self-dual functor $\overline{2}$, we have a functor between $\mathsf{rDA}_{\mathcal{L}}$ and $\mathsf{oDA}_{\mathcal{L}}$, that reflects reachable automata into observable automata in the *reverse* language: Thus, our final minimization morphism is simply a concatenation of a series of functors:




**Theorem 3.4.3 (Brzozowski minimization algorithm)**

The functor:

$$\left(\overline{2}^{\mathsf{op}} \circ R^{\mathsf{op}} \circ \overline{2} \circ R\right) : \mathsf{DA}_{\mathcal{L}} \to \mathsf{mDA}_{\mathcal{L}} \tag{29}$$

minimizes deterministic automata.

*Proof.* Due to the way the functors were constructed, we get the following two properties "for free" (ignoring all the work that went into constructing the functors):

- the series of functors preserves the language being recognized (notice the subscript $\mathcal{L}$)
- the resulting automaton is minimal (by definition of $\mathsf{mDA}_{\mathcal{L}}$)

Thus, the theorem holds by construction. □

# 4 Exercises

Here is a table of some of the exercises that have been distributed throughout the explainer.

# Bibliography

Bezhanishvili, Nick, Clemens Kupke, and Prakash Panangaden. 2012. "Minimization via Duality." In *Logic, Language, Information and Computation*, edited by Luke Ong and Ruy de Queiroz, *Logic, Language, Information and Computation*. Springer. https://doi.org/10.1007/978-3-642-32621-9_14.

Bonchi, Filippo, Marcello M. Bonsangue, Helle H. Hansen, Prakash Panangaden, Jan J. M. M. Rutten, and Alexandra Silva. 2014. "Algebra-Coalgebra Duality in Brzozowski's Minimization Algorithm." *ACM Transactions on Computational Logic* 15 (1): 1–29. https://doi.org/10.1145/2490818.

# 22
# *Bibliography*


[1] Julian Asilis. *Probability Monads*. PhD thesis, Ph. D. thesis, PhD thesis, Harvard University Cambridge, Massachusetts, 2020.

[2] Robert Atkey, Patricia Johann, and Neil Ghani. Refining inductive types. *Logical Methods in Computer Science*, Volume 8, Issue 2, Jun 2012.

[3] Robert J Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5(4):614–630, 1961.

[4] Andrej Bauer. Notes on realizability. February 2025.

[5] Nick Benton and Chung-Kil Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical Report MSR-TR-2010-62, April 2010.

[6] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):24–es, 2007.

[7] Rahul Chhabra. Modest Sets are Equivalent to PERs, 2024.

[8] Guy Cousineau, P-L Curien, and Michel Mauny. The categorical abstract machine. *Science of computer programming*, 8(2):173–202, 1987.

[9] Roy L Crole. *Categories for types*. Cambridge University Press, 1993.

[10] Haskell B. Curry and Robert Feys. *Combinatory logic*, volume 1 of *Studies in logic and the foundations of mathematics*. North-Holland, Amsterdam, 1958.

[11] Brijesh Dongol, Ian J Hayes, and Georg Struth. Convolution as a unifying concept: Applications in separation logic, interval

calculi, and concurrency. *ACM Transactions on Computational Logic (TOCL)*, 17(3):1–25, 2016.

[12] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 71–80. IEEE, 2009.

[13] Neil Ghani, Patricia Johann, and Clément Fumex. Fibrational induction rules for initial algebras. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic*, pages 336–350, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[14] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2017.

[15] Pieter J W Hofstra. Partial Combinatory Algebras and Realizability Toposes.

[16] Bart Jacobs. Comprehension categories and the semantics of type dependency. *Theor. Comput. Sci.*, 107(2):169–207, January 1993.

[17] Bart Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

[18] S. C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10(4):109–124, December 1945.

[19] Stephen Cole Kleene. *Introduction to metamathematics*, volume 1 of *Bibliotheca mathematica*. North-Holland, Amsterdam, 1 edition, 1952.

[20] Joachim Lambek and Philip J Scott. *Introduction to higher-order categorical logic*, volume 7. Cambridge University Press, 1988.

[21] F William Lawvere and Stephen H Schanuel. *Conceptual mathematics: a first introduction to categories*. Cambridge University Press, 2009.

[22] John M Li, Amal Ahmed, and Steven Holtzen. Lilac: a modal separation logic for conditional probability. *Proceedings of the ACM on Programming Languages*, 7(PLDI):148–171, 2023.

[23] John Longley. Matching typed and untyped realizability. *Electronic Notes in Theoretical Computer Science*, 23(1):74–100, 1999.

[24] John Longley. Unifying typed and untyped realizability. May 1999.

[25] John R. Longley. *Realizability Toposes and Language Semantics*. PhD thesis, University of Edinburgh, 1994.

[26] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 1998.

[27] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic journal of philosophical logic*, 1(1):11–60, 1996.

[28] Max S. New. Category theory for computer scientists. Lecture notes, 2023. Scribed by Pranav Srinivasan and Kevin Wang.

[29] Daniel Baker Patterson. *Interoperability through realizability: expressing high-level abstractions using low-level code*. PhD thesis, Northeastern University, 2022.

[30] Andrew M Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge University Press, 2013.

[31] Gordon Plotkin and John Power. Notions of computation determine monads. In *International Conference on Foundations of Software Science and Computation Structures*, pages 342–356. Springer, 2002.

[32] John C Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*, pages 513–523, 1983.

[33] John C Reynolds. The coherence of languages with intersection types. In *International Symposium on Theoretical Aspects of Computer Software*, pages 675–700. Springer, 1991.

[34] Michael B Smyth and Gordon D Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, 1982.

[35] Sam Staton. Commutative semantics for probabilistic programming. In *European Symposium on Programming*, pages 855–879. Springer, 2017.

[36] Philip Wadler. Recursive types for free! Unpublished draft, 1990.