The Dao of Functional Programming

Bartosz Milewski

(Last updated: 2024 年 8 月 23 日)

目录

目录 **i**

序言

大多数编程教材，遵循 Brian Kernighan 的做法，都是以 “Hello World!’’ 开始的。这很自然，因为我们想立即通过让计算机执行命令并打印出这些著名的词语来获得满足感。但计算机编程的真正掌握远比这更深刻，急于求成可能只会给你一种虚假的力量感，而实际上你只是在模仿大师。如果你的目标只是学习一项有用且薪水丰厚的技能，那么，尽管去写你的 “Hello World!’’ 程序吧。有大量的书籍和课程可以教你如何用你选择的任何语言编写代码。然而，如果你真的想深入编程的本质，你需要耐心和坚持。

范畴论是数学的一个分支，它提供了与编程实践经验相符的抽象。借用克劳塞维茨的一句话：编程只是以其他方式继续数学的过程。当用数据类型和函数来解释范畴论的许多复杂概念时，程序员们可能比专业数学家更容易理解它们。

当我面对新的范畴概念时，我常常会在 Wikipedia 或 nLab 上查找它们，或者重新阅读 Mac Lane 或 Kelly 的某一章。这些都是很好的资源，但它们要求你对主题有一定的熟悉度，并且有能力填补其中的空白。本书的一个目标是提供必要的引导，以继续学习范畴论。

在范畴论和计算机科学中，有许多民间知识在文献中是找不到的。当你通过干巴巴的定义和定理来学习时，很难获得有用的直觉。我尽可能地提供了那些缺失的直觉，并解释了不仅是什么，还解释了为什么。

本书的标题暗示了 Benjamin Hoff 的 “The Tao of Pooh’’ 和 Robert Pirsig 的 “Zen and the Art of Motorcycle Maintenance’’，这两本书都是西方人尝试吸收东方哲学元素的作品。粗略地说，范畴论之于编程，正如道（Dao）[1]之于西方哲学。许多范畴论的定义在第一次阅读时显得毫无意义，但随着时间的推移，你会逐渐理解其中深刻的智慧。如果要用一句话总结范畴论，那就是：‘‘事物通过它们与宇宙的关系来定义。”

集合论

传统上，集合论被认为是数学的基础，尽管最近类型论也在争夺这一头衔。从某种意义上说，集合论是数学的汇编语言，包含了许多实现细节，这些细节往往会掩盖高级思想的表达。

范畴论并不是要取代集合论，通常它被用来构建抽象，然后用集合来建模。事实上，范畴论的基本定理尤内达引理将范畴与集合论中的模型联系起来。我们可以在计算机图形学中找到有用的直觉，在那里我们构建和操作抽象世界，直到最后一刻才将它们投影并采样到数字显示器上。

为了学习范畴论，不必精通集合论。但一些基本知识是必要的。例如，集合包含元素的概念。我们说，给定一个集合 S 和一个元素 a，可以问 a 是否属于 S。这个陈述写作 a Ë S （a 是 S 的成员）。也可以有一个不包含任何元素的空集合。

[1] Dao 是 Tao 的现代拼写

集合元素的重要属性是它们可以比较是否相等。给定两个元素 a Ë S 和 b Ë S，我们可以问：a 是否等于 b? 或者我们可以强加条件 $a = b$，如果 a 和 b 是通过不同的方法从集合 S 中选择的元素的结果。集合元素的相等性是所有范畴论中交换图的本质。

两个集合的笛卡尔积 $S \quad T$ 被定义为一组所有的元素对 $\langle s, t\rangle$，其中 s Ë S 和 t Ë T。

一个从源集合（称为 f 的*定义域*）到目标集合（称为 f 的*值域*）的函数 $f\colon S \to T$ 也被定义为一组对。这些对的形式为 $\langle s, t\rangle$，其中 $t = fs$。这里 fs 是函数 f 作用在参数 s 上的结果。你可能更熟悉函数应用的符号 $f(s)$，但这里我将遵循 Haskell 的惯例，省略括号（以及多个变量的逗号）。

在编程中，我们习惯于通过一系列指令来定义函数。我们提供一个参数 s 并应用指令，最终产生结果 t。我们经常担心计算结果可能需要多长时间，或者算法是否会终止。在数学中，我们假设对于任何给定的参数 s Ë S，结果 t Ë T 是立即可得的，并且是唯一的。在编程中，我们称这种函数为纯函数且是全函数。

约定

我尽量使全书的符号保持一致。它大致基于 nLab 中流行的风格。

特别地，我决定使用小写字母，如 a 或 b 来表示范畴中的对象，并使用大写字母如 S 来表示集合（尽管集合是集合与函数范畴中的对象）。通用范畴有名称如 $\mathcal{C}$ 或 $\mathcal{D}$，而特定的范畴有名称如 Set 或 Cat。

编程示例以 Haskell 语言编写。尽管这不是一本 Haskell 手册，但语言构造的引入是逐渐进行的，以帮助读者理解代码。Haskell 语法通常基于数学符号，这是一个额外的优势。程序片段以如下格式编写：

Chapter 1

Clean Slate
白板

Programming starts with types and functions. You probably have some preconceptions about what types and functions are: get rid of them! They will cloud your mind.

编程从类型和函数开始。你可能对类型和函数有一些先入为主的观念：抛开它们! 它们会混淆你的思维。

Don’t think about how things are implemented in hardware. What computers are is just one of the many models of computation. We shouldn’t get attached to it. You can perform computations in your mind, or with pen and paper. The physical substrate is irrelevant to the idea of programming.

不要去想事情是如何在硬件中实现的。计算机只是众多计算模型中的一种。我们不应该执着于此。你可以在脑海中进行计算，或者使用笔和纸进行计算。物理载体与编程的思想无关。

1.1 Types and Functions
类型与函数

Paraphrasing Lao Tzu[1]: *The type that can be described is not the eternal type*. In other words, type is a primitive notion. It cannot be defined.

引用老子的话[2]：*可描述的类型不是永恒的类型*。换句话说，类型是一种原始概念。它无法被定义。

[1] The modern spelling of Lao Tzu is Laozi, but I’ll be using the traditional one. Lao Tzu was the semi-legendary author of Tao Te Ching (or Daodejing), a classic text on Daoism.

[2] 老子的现代拼写是 Laozi，但我会使用传统的拼写。老子是道德经的半传奇作者，一本关于道教的经典著作。

Instead of calling it a *type*, we could as well call it an *object* or a *proposition*. These are the words that are used to describe it in different areas of mathematics (type theory, category theory, and logic, respectively).

我们可以不称之为*类型*，而是称之为*对象*或*命题*。这些是数学中不同领域用来描述它的词汇（分别是类型论、范畴论和逻辑）。

There may be more than one type, so we need a way to name them. We could do it by pointing fingers at them, but since we want to effectively communicate with other people, we usually name them. So we'll talk about type a, b, c; or , , , and so on. These are just names.

可能有不止一种类型，所以我们需要一种命名它们的方式。我们可以用手指指向它们，但由于我们希望有效地与他人交流，我们通常为它们命名。因此，我们会谈到类型 a、b、c；或者 、、等等。这些只是名称。

A type by itself has no meaning. What makes it special is how it connects to other types. The connections are described by arrows. An arrow has one type as its source and one type as its target. The target could be the same as the source, in which case the arrow loops around.

类型本身没有意义。使它特别的是它如何与其他类型连接。这些连接由箭头描述。一个箭头有一个类型作为其源，另一个类型作为其目标。目标类型可以与源类型相同，在这种情况下，箭头会回环。

An arrow between types is called a *function*. An arrow between objects is called a *morphism*. An arrow between propositions is called an *entailment*. These are just words that are used to describe arrows in different areas of mathematics. You can use them interchangeably.

类型之间的箭头称为*函数*。对象之间的箭头称为*态射*。命题之间的箭头称为*蕴涵*。这些只是用来描述不同数学领域中的箭头的词汇。你可以互换使用它们。

A proposition is something that may be true. In logic, we interpret an arrow between two objects as a entails b, or b is derivable from a.

命题是可能为真的东西。在逻辑中，我们将两个对象之间的箭头解释为 a 蕴涵 b，或者 b 可以从 a 推导出来。

There may be more than one arrow between two types, so we need to name them. For instance, here's an arrow called f that goes from type a to type b

在两个类型之间可能有多个箭头，所以我们需要为它们命名。例如，这里有一个从类型 a 到类型 b 的箭头，叫做 f。

$$a \xrightarrow{f} b$$

One way to interpret this is to say that the function f takes an argument of type a and produces a result of type b. Or that f is a proof that if a is true then b is also true.

解释这一点的一种方式是说函数 f 接受一个类型为 a 的参数，并生成一个类型为 b 的结果。或者说 f 是一种证明，如果 a 为真，那么 b 也为真。

Note: The connection between type theory, lambda calculus (which is the foundation of programming), logic, and category theory is known as the Curry-Howard-Lambek correspondence.

注意：类型论、lambda 演算（它是编程的基础）、逻辑和范畴论之间的联系被称为 Curry-Howard-Lambek 对应。

1.2 Yin and Yang
阴阳

An object is defined by its connections. An arrow is a proof, a witness, of the fact that two objects are connected. Sometimes there's no proof, the objects are disconnected; sometimes there are many proofs; and sometimes there's a single proof—a unique arrow between two objects.

一个对象由它的连接来定义。箭头是两个对象连接的证明，一个证据。有时没有证明，对象是断开的；有时有很多证明；有时只有一个证明——两个对象之间的唯一箭头。

What does it mean to be *unique*? It means that if you can find two of those, then they must be equal.

什么是*唯一的*？这意味着如果你能找到两个这样的东西，那么它们一定是相等的。

An object that has a unique outgoing arrow to every object is called the *initial object*.

一个对每个对象都有唯一出射箭头的对象称为*初始对象*。

Its dual is an object that has a unique incoming arrow from every object. It's called the *terminal object*.

它的对偶是一个从每个对象都有唯一入射箭头的对象。它被称为*终端对象*。

In mathematics, the initial object is often denoted by 0 and the terminal object by 1.

在数学中，初始对象通常表示为 0，终端对象表示为 1。

The arrow from 0 to any object a is denoted by $¡_a$, often abbreviated to $¡$.

从 0 到任何对象 a 的箭头表示为 $¡_a$，通常缩写为 $¡$。

The arrow from any object a to 1 is denoted by $!_a$, often abbreviated to $!$.

从任何对象 a 到 1 的箭头表示为 $!_a$，通常缩写为 $!$。

The initial object is the source of everything. As a type it's known in Haskell as . It symbolizes the chaos from which everything arises. Since there is an arrow from to everything, there is also an arrow from to itself.

初始对象是所有事物的源头。作为一种类型，它在 Haskell 中被称为 。它象征着一切事物源于其中的混沌。由于有一个从 到一切事物的箭头，也有一个从 到它自身的箭头。

$$\begin{tikzcd} \mathtt{Void} \arrow[loop above] \end{tikzcd}$$

Thus begets and everything else.

因此，产生了 和其他一切。

The terminal object unites everything. As a type it's known as Unit. It symbolizes the ultimate order.

终端对象将一切联合起来。作为一种类型，它被称为 Unit。它象征着终极秩序。

In logic, the terminal object signifies the ultimate truth, symbolized by T or $\top$. The fact that there's an arrow to it from any object means that $\top$ is true no matter what your assumptions are.

在逻辑中，终端对象象征着终极真理，用 T 或 $\top$ 表示。事实是，从任何对象到它都有一个箭头，这意味着无论你的假设是什么，$\top$ 都是正确的。

Dually, the initial object signifies logical falsehood, contradiction, or a counterfactual. It's written as False and symbolized by an upside down T, $\bot$. The fact that there is an arrow from it to any object means that you can prove anything starting from false premises.

相对地，初始对象象征逻辑谬误、矛盾或反事实。它写作 False，用倒 T 符号 $\bot$ 表示。事实是，从它到任何对象都有一个箭头，这意味着你可以从错误的前提开始证明任何事情。

In English, there is a special grammatical construct for counterfactual implications. When we say, "If wishes were horses, beggars would ride," we mean that the equality between wishes and horses implies that beggars be able to ride. But we know that the premise is false.

在英语中，有一种特殊的语法结构用于反事实蕴涵。当我们说，"如果愿望是马，乞丐会骑马"，我们的意思是愿望与马的等价性意味着乞丐能够骑马。但我们知道这个前提是错误的。

A programming language lets us communicate with each other and with computers. Some languages are easier for the computer to understand, others are closer to the theory. We will use Haskell as a compromise.

编程语言让我们可以相互交流，也可以与计算机交流。有些语言对计算机来说更容易理解，另一些则更接近理论。我们将使用 Haskell 作为折中。

In Haskell, the name for the terminal type is , a pair of empty parentheses, pronounced Unit. This notation will make sense later.

在 Haskell 中，终端类型的名称是 , 一对空括号，读作 Unit。这种符号稍后会更有意义。

There are infinitely many types in Haskell, and there is a unique function/arrow from to each one of them. All these functions are known under the same name: .

在 Haskell 中有无限多的类型，并且从 到每个类型都有一个唯一的函数/箭头。所有这些函数都有相同的名称：。

Programming	Category theory	Logic
type	object	proposition
function	morphism (arrow)	implication
	initial object, 0	False ⊥
	terminal object, 1	True ⊤

1.3 Elements
元素

An object has no parts but it may have structure. The structure is defined by the arrows pointing at the object. We can *probe* the object with arrows.

一个对象没有部分，但它可能有结构。结构由指向对象的箭头定义。我们可以用箭头探测对象。

In programming and in logic we want our initial object to have no structure. So we'll assume that it has no incoming arrows (other than the one that's looping back from it). Therefore has no structure.

在编程和逻辑中，我们希望初始对象没有结构。所以我们假设它没有入射箭头（除了从它自身回环的那个）。因此 没有结构。

The terminal object has the simplest structure. There is only one incoming arrow from any object to it: there is only one way of probing it from any direction. In this respect, the terminal object behaves like an indivisible point. Its only property is that it exists, and the arrow from any other object proves it.

终端对象有最简单的结构。只有一个从任何对象到它的入射箭头：只有一种方式可以从任何方向探测它。在这方面，终端对象表现得像一个不可分割的点。它唯一的属性是它存在，并且从任何其他对象到它的箭头证明了这一点。

Because the terminal object is so simple, we can use it to probe other, more complex objects.

由于终端对象非常简单，我们可以用它来探测其他更复杂的对象。

If there is more than one arrow coming from the terminal object to some object a, it means that a has some structure: there is more than one way of looking at it. Since the terminal object behaves like a point, we can visualize each arrow from it as picking a different point or element of its target.

如果有多个箭头从终端对象指向某个对象 a，这意味着 a 有某种结构：有多种方式可以看待它。由于终端对象表现得像一个点，我们可以将每个箭头视为从它选取目标的不同点或元素。

In category theory we say that x is a *global element* of a if it's an arrow

在范畴论中，如果 x 是一个箭头，我们说它是 a 的*全局元素*

$$1 \xrightarrow{x} a$$

We'll often simply call it an element (omitting "global").

我们经常简单地称之为元素（省略"全局"）。

In type theory, $x\colon A$ means that x is of type A.

在类型论中，$x\colon A$ 意味着 x 是 A 类型的。

In Haskell, we use the double-colon notation instead:

(Haskell uses capitalized names for concrete types, and lower-cased names for type variables.)

在 Haskell 中，我们使用双冒号符号：

（Haskell 对具体类型使用大写名称，对类型变量使用小写名称。）

We say that is a term of type but, categorically, we'll interpret it as an arrow $x : 1 \to A$, a global element of . [3]

我们说 是 类型的一个项，但在范畴论中，我们将其解释为箭头 $x : 1 \to A$，即 的全局元素。[4]

In logic, such x is called the proof of A, since it corresponds to the implication $\top \to A$ (if **True** is true then **A** is true). Notice that there may be many different proofs of A.

在逻辑中，这样的 x 被称为 A 的证明，因为它对应于蕴涵 $\top \to A$（如果 **True** 为真，那么 **A** 也为真）。请注意，A 可能有许多不同的证明。

Since we have mandated there be no arrows from any other object to , there is no arrow from the terminal object to it. Therefore has no elements. This is why we think of as empty.

由于我们规定不允许有任何其他对象到 的箭头，因此从终端对象到它没有箭头。因此，没有元素。这就是为什么我们认为 是空的。

The terminal object has just one element, since there is a unique arrow coming from it to itself, $1 \to 1$. This is why we sometimes call it a singleton.

终端对象只有一个元素，因为只有一个从它到它自身的箭头，$1 \to 1$。这就是为什么我们有时称它为单一元素集。

[3] The Haskell type system distinguishes between and . However, they denote the same thing in categorical semantics.

[4] Haskell 类型系统区分 和 。然而，在范畴语义中，它们表示相同的东西。

Note: In category theory there is no prohibition against the initial object having incoming arrows from other objects. However, in cartesian closed categories that we're studying here, this is not allowed.

注意：在范畴论中，没有禁止初始对象从其他对象接收箭头。然而，在我们这里研究的笛卡尔闭范畴中，这是不允许的。

1.4 The Object of Arrows
箭头的对象

Arrows between any two objects form a set[5]. This is why some knowledge of set theory is a prerequisite to the study of category theory.

任何两个对象之间的箭头形成一个集合[6]。这就是为什么集合论的一些知识是学习范畴论的先决条件。

In programming we talk about the *type* of functions from to . In Haskell we write:

```
```

meaning that is of the type "function from to ". Here, is just the name we are giving to this type.

在编程中，我们谈论从 到 的函数的*类型*。在 Haskell 中我们写：

```
```

意思是 是 ''从 到 的函数'' 类型。在这里，只是我们给这个类型起的名字。

If we want function types to be treated the same way as other types, we need an *object* that would represent a set of arrows from to .

如果我们希望函数类型与其他类型被相同对待，我们需要一个*对象*，它可以表示从 到 的箭头集合。

To fully define this object, we would have to describe its relation to other objects, in particular to and . We don't have the tools to do that yet, but we'll get there.

要完全定义这个对象，我们必须描述它与其他对象的关系，特别是与 和 的关系。我们还没有工具来做到这一点，但我们会到达那里。

For now, let's keep in mind the following distinction: On the one hand we have arrows which connect two objects and . These arrows form a set. On the other hand we have an *object of arrows* from to . An "element" of this object is defined as an arrow from the terminal object to the object we call .

[5] Strictly speaking, this is true only in a *locally small* category.

[6] 严格来说，这只在*局部小*范畴中成立。

目前，让我们记住以下区别：一方面，我们有连接两个对象 和 的箭头。这些箭头形成一个集合。另一方面，我们有一个从 到 的箭头对象。这个对象的 ‘‘元素’’ 被定义为从终端对象 到我们称之为 的对象的箭头。

The notation we use in programming tends to blur this distinction. This is why in category theory we call the object of arrows an *exponential* and write it as b^a (the source object is in the exponent). So the statement:

is equivalent to

$$1\xrightarrow{f} b^a$$

In logic, an arrow $A \to B$ is an implication: it states the fact that “if A then B.” An exponential object B^A is the corresponding proposition. It could be true or it could be false, we don’t know. You have to prove it. Such a proof is an element of B^A.

在编程中我们使用的符号倾向于模糊这个区别。这就是为什么在范畴论中我们称箭头对象为指数对象并将其写作 b^a（源对象在指数中）。因此，以下语句：

等同于

$$1\xrightarrow{f} b^a$$

在逻辑中，箭头 $A \to B$ 是一个蕴涵：它陈述了 ‘‘如果 A，那么 B’’ 的事实。指数对象 B^A 是相应的命题。它可能为真，也可能为假，我们不知道。你必须证明它。这样的证明是 B^A 的一个元素。

Show me an element of B^A and I’ll know that B follows from A.

给我看 B^A 的一个元素，我就知道 B 是从 A 推导出来的。

Consider again the statement, “If wishes were horses, beggars would ride”—this time as an object. It’s not an empty object, because you can point at a proof of it—something along the lines: “A person who has a horse rides it. Beggars have wishes. Since wishes are horses, beggars have horses. Therefore beggars ride.” But, even though you have a proof of this statement, it’s of no use to you, because you can never prove its premise: “wish = horse”.

再考虑一下这个语句，“如果愿望是马，乞丐会骑马”——这次作为一个对象。它不是一个空对象，因为你可以指出它的证明——类似于：“一个有马的人骑马。乞丐有愿望。因为愿望是马，所以乞丐有马。因此，乞丐骑马。”但即使你有这个语句的证明，对你来说也没有用，因为你永远无法证明它的前提：“愿望 = 马”。

Chapter 2

Composition

2.1 Composition

Programming is about composition. Paraphrasing Wittgenstein, one could say: “Of that which cannot be decomposed one should not speak.” This is not a prohibition, it’s a statement of fact. The process of studying, understanding, and describing is the same as the process of decomposing; and our language reflects this.

The reason we have built the vocabulary of objects and arrows is precisely to express the idea of composition.

Given an arrow f from a to b and an arrow g from b to c, their composition is an arrow that goes directly from a to c. In other words, if there are two arrows, the target of one being the same as the source of the other, we can always compose them to get a third arrow.

$$\begin{tikzcd}
a \arrow[r, "f"'] \arrow[rr, bend left, "h"] & b \arrow[r, "g"'] & c
\end{tikzcd}$$

In math we denote composition using a little circle

$$h = g \circ f$$

We read this: “h is equal to g after f.” The choice of the word “after” suggests temporal ordering of actions, which in most cases is a useful intuition.

The order of composition might seem backward, but this is because we think of functions as taking arguments on the right. In Haskell we replace the circle with a dot:

This is every program in a nutshell. In order to accomplish , we decompose it into simpler problems, and . These, in turn, can be decomposed further, and so on.

Now suppose that we were able to decompose g itself into $j \circ k$. We have

$$h = (j \circ k) \circ f$$

We want this decomposition to be the same as

$$h = j\circ(k\circ f)$$

We want to be able to say that we have decomposed h into three simpler problems

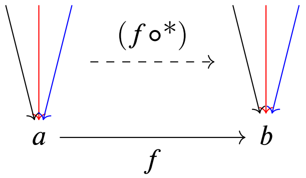
$$h = j\circ k\circ f$$

and not have to keep track which decomposition came first. This is called *associativity* of composition, and we will assume it from now on.

Composition is the source of two mappings of arrows called pre-composition and post-composition.

When you *post-compose* an arrow h with an arrow f, it produces the arrow $f\circ h$ (the arrow f is applied *after* the arrow h). Of course, you can post-compose h only with arrows whose source is the target of h. Post-composition by f is written as $(f\circ *)$, leaving a hole for h. As Lao Tzu would say, "Usefulness of post-composition comes from what is not there."

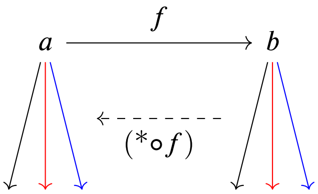
Thus an arrow $f\colon a \to b$ induces a mapping of arrows $(f\circ *)$ that maps arrows which are probing a to arrows which are probing b.



Since objects have no *internal* structure, when we say that f transforms a to b, this is exactly what we mean.

Post-composition lets us shift focus from one object to another.

Dually, you can *pre-compose* by f, or apply $(*\circ f)$ to arrows originating in b to map them to arrows originating in a (notice the change of direction).



Pre-composition lets us shift the perspective from observer b to observer a.

Pre- and post-composition are mappings of arrows. Since arrows form sets, these are *functions* between sets.

Another way of looking at pre- and post-composition is that they are the result of partial application of the two-hole composition operator $(*\circ *)$, in which we can pre-fill one hole or the other with a fixed arrow.

In programming, an outgoing arrow is interpreted as extracting data from its source. An incoming arrow is interpreted as producing or constructing the target. Outgoing arrows define the interface, incoming arrows define the constructors.

For Haskell programmers, here's the implementation of post-composition as a higher-order function:

And similarly for pre-composition:

Do the following exercises to convince yourself that shifts in focus and perspective are composable.

Exercise 2.1.1. *Suppose that you have two arrows, $f : a \to b$ and $g : b \to c$. Their composition $g \circ f$ induces a mapping of arrows $((g \circ f) \circ *)$. Show that the result is the same if you first apply $(f \circ *)$ and follow it by $(g \circ *)$. Symbolically:*

$$((g \circ f) \circ *) = (g \circ *) \circ (f \circ *)$$

Hint: Pick an arbitrary object x and an arrow $h : x \to a$ and see if you get the same result. Note that $\circ$ is overloaded here. On the right, it means regular function composition when put between two post-compositions.

Exercise 2.1.2. *Convince yourself that the composition from the previous exercise is associative. Hint: Start with three composable arrows.*

Exercise 2.1.3. *Show that pre-composition $(* \circ f)$ is composable, but the order of composition is reversed:*

$$(* \circ (g \circ f)) = (* \circ f) \circ (* \circ g)$$

2.2 Function application

We are ready to write our first program. There is a saying: "A journey of a thousand miles begins with a single step." Consider a journey from 1 to b. Our single step can be an arrow from the terminal object 1 to some a. It's an element of a. We can write it as:

$$1 \xrightarrow{x} a$$

The rest of the journey is the arrow:

$$a \xrightarrow{f} b$$

These two arrows are composable (they share the object a in the middle) and their composition is the arrow y from 1 to b. In other words, y is an *element* of b:

$$\begin{tikzcd}
1 \arrow[r, "x"'] \arrow[rr, bend left, "y"] & a \arrow[r, "f"'] & b
\end{tikzcd}$$

We can write it as:

$$y = f \circ x$$

We used f to map an *element* of a to an *element* of b. Since this is something we do quite often, we call it the *application* of a function f to x, and use the shorthand notation

$$y = fx$$

Let's translate it to Haskell. We start with an element x of a (a shorthand for)

We declare a function f as an element of the "object of arrows" from a to b

with the understanding (which will be elaborated upon later) that it corresponds to an arrow from to . The result is an element of b

and it is defined as

We call this the application of a function to an argument, but we were able to express it purely in terms of function composition. (Note: In other programming languages function application requires the use of parentheses, e.g., .)

2.3 Identity

You may think of arrows as representing change: object a becomes object b. An arrow that loops back represents a change in an object itself. But change has its dual: lack of change, inaction or, as Lao Tzu would say *wu wei*.

Every object has a special arrow called the identity, which leaves the object unchanged. It means that, when you compose this arrow with any other arrow, either incoming or outgoing, you get that other arrow back. Thought of as an action, the identity arrow does nothing and takes no time.

An identity arrow on the object a is called id_a. So if we have an arrow $f : a \to b$, we can compose it with identities on either side

$$id_b \circ f = f = f \circ id_a$$

or, pictorially:

$$\begin{tikzcd}
a \arrow[loop above, "id_a"] \arrow[r, "f"] & b \arrow[loop above, "id_b"]
\end{tikzcd}$$

We can easily check what an identity does to elements. Let's take an element $x\colon 1 \to a$ and compose it with id_a. The result is:

$$id_a \circ x = x$$

which means that identity leaves elements unchanged.

In Haskell, we use the same name for all identity functions (we don't subscript it with the type it's acting on). The above equation, which specifies the action of id on elements, translates directly to:

and it becomes the definition of the function .

We've seen before that both the initial object and the terminal object have unique arrows circling back to them. Now we are saying that every object has an identity arrow circling back to it. Remember what we said about uniqueness: If you can find two of those, then they must be equal. We must conclude that these unique looping arrows we talked about must be the identity arrows. We can now label these diagrams:

$$\begin{tikzcd}
\texttt{Void} \arrow[loop above, "id"] & & \texttt{()} \arrow[loop above, "id"]
\end{tikzcd}$$

In logic, identity arrow translates to a tautology. It's a trivial proof that, "if a is true then a is true." It's also called the *identity rule*.

If identity does nothing then why do we care about it? Imagine going on a trip, composing a few arrows, and finding yourself back at the starting point. The question is: Have you done anything, or have you wasted your time? The only way to answer this question is to compare your path to the identity arrow.

Some round trips bring change, others don't.

More importantly, identity arrows will allow us to compare objects. They are an integral part of the definition of an isomorphism.

Exercise 2.3.1. *What does $(id_a \circ *)$ do to arrows terminating in a? What does $(* \circ id_a)$ do to arrows originating from a?*

2.4 Monomorphisms

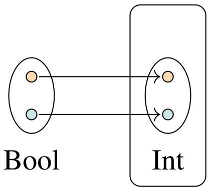
Consider the function that tests whether its input is divisible by two:

It's a many-to-one function: all even numbers are mapped to and all odd numbers to . Most nformation about the input is discarded, we are only interested in its evenness, not its actual value. By discarding information we arrive at an abstraction[1]. Functions (and later functors) epitomize abstractions.

Contrast this with the function :

This function doesn't discard information. You can recover its argument from its result.

Functions that don't discard information are also useful: they can be thought of as injecting their source into their target. You may imagine the type of the source as a *shape* that is being embedded in the target. Here, we are embedding a two-element shape into the type of integers.

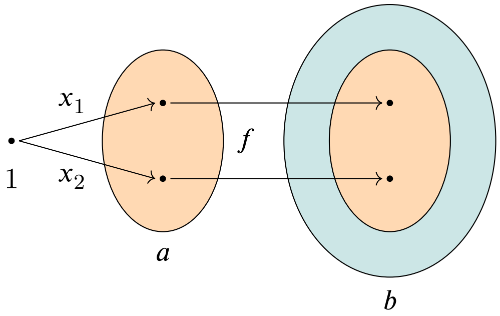


Injective functions, or *injections* are defined to always assign different values to different arguments. In other words, they don't collapse multiple elements into one.

Here's another, slightly convoluted, way of saying this: An injection maps two elements to one only when the two elements are equal.

We could translate this definition to the categorical language by replacing "elements" with arrows from the terminal object. We would say that $f : a \to b$ is an injection if, for any pair of global elements $x_1 : 1 \to a$ and $x_2 : 1 \to a$, the following implication holds:

$$f \circ x_1 = f \circ x_2 \implies x_1 = x_2$$



[1] to abstract literally means to draw away

The problem with this definition is that not every category has a terminal object. A better definition would replace global elements with arbitrary shapes. Thus the notion of injectivity is generalized to that of monomorphism.

An arrow $f \colon a \to b$ is *monomorphic* if, for any choice of an object c and a pair of arrows $g_1 \colon c \to a$ and $g_2 \colon c \to a$ we have the following implication:

$$f \circ g_1 = f \circ g_2 \implies g_1 = g_2$$

To show that an arrow $f \colon a \to b$ is *not* a monomorphism, it's enough to find a counterexample: two different shapes in a, such that f maps them to the same shape in b.

Monomorphisms, or "monos" for short, are often denoted using special arrows, as in $a \hookrightarrow b$ or $a \rightarrowtail b$.

In category theory objects are indivisible, so we can only talk about sub-objects using arrows. We say that monomorphism $a \hookrightarrow b$ picks a subobject of b in the shape of a.

Exercise 2.4.1. *Show that any arrow from the terminal object is a monomorphism.*

2.5 Epimorphisms

The function is injective (hence a monomorphism), but it only covers a small subset of its target — just two integers out of infinitely many.

In contrast, the function covers the whole of (it can produce both and). A function that covers the whole of its target is called a *surjection*.

To generalize injections we used additional mappings-in. To generalize surjections, we'll use mappings-out. The categorical counterpart of a surjection is called an epimorphism.

An arrow $f \colon a \to b$ is an *epimorphism* if for any choice of an object c and a pair of arrows $g_1 \colon b \to c$ and $g_2 \colon b \to c$ we have the following implication:

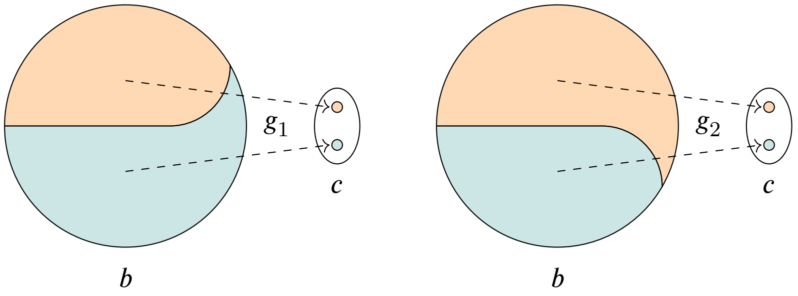
$$g_1 \circ f = g_2 \circ f \implies g_1 = g_2$$

Conversely, to show that f is not an epimorphism, it's enough to pick an object c and two different arrows g_1 and g_2 that agree when precomposed with f.

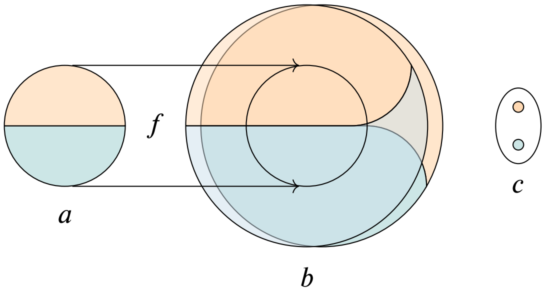
To get some insight into this definition, we have to visualize mappings out. Just as a mapping into an object can be thought of as defining a shape, mapping out of an object can be thought of as defining properties of that object.

This is clear when dealing with sets, especially if the target set is finite. You may think of an element of the target set as defining a color. All elements of the source that are mapped to that element are "painted" a particular color. For instance, the function paints all even integers the color, and all odd ones the color.

In the definition of an epimorphism we have two such mapping, g_1 and g_2. Suppose that they differ only slightly. Most of the object b is painted alike by both of them.



If f is *not* and epimorphism, it's possible that its image only covers the part that is painted alike by g_1 and g_2. The two arrows then agree on painting a when precomposed by f, even though they are different on the whole.



Of course, this is just an illustration. In an actual category there is no peeking inside objects.

Epimorphisms, or "epis" for short, are often denoted by a special arrow $a \twoheadrightarrow b$.

In sets, a function that is both injective and surjective is called a bijection. It provides a one-to-one invertible mapping between elements of two sets. This role is played by isomorphisms in category theory. However, in general it's not true that an arrow that is both mono and epi is an isomorphism.

Exercise 2.5.1. *Show that any arrow to the terminal object is an epimorphism.*

Chapter 3

Isomorphisms

When we say that:

$$f\circ(g\circ h) = (f\circ g)\circ h$$

or:

$$f = f\circ id$$

we are asserting the *equality* of arrows. The arrow on the left is the result of one operation, and the arrow on the right is the result of another. But the results are *equal*.

We often illustrate such equalities by drawing *commuting* diagrams, e.g.,

$$\begin{tikzcd}
a \arrow[r, "h"] \arrow[rr, bend left, "g\circ h"] \arrow[rrr, bend left=60, "f\circ(g\circ h)"] \arrow[rrr, bend right=60, "(f\circ g)\circ h"'] & b \arrow[r, "g"] \arrow[rr, bend right, "f\circ g"'] & c \arrow[r, "f"] & d
\end{tikzcd}
\qquad
\begin{tikzcd}
a \arrow[loop above, "id"] \arrow[r, "f"] & b
\end{tikzcd}$$

Thus we compare arrows for equality.

We do *not* compare objects for equality[1]. We see objects as confluences of arrows, so if we want to compare two objects, we look at the arrows.

3.1 Isomorphic Objects

The simplest relation between two objects is an arrow.

The simplest round trip is a composition of two arrows going in opposite directions.

[1] Half-jokingly, invoking equality of objects is considered "evil" in category theory.

$$\begin{tikzcd}
a \arrow[r, bend left, "f"] & b \arrow[l, bend left, "g"]
\end{tikzcd}$$

There are two possible round trips. One is $g \circ f$, which goes from a to a. The other is $f \circ g$, which goes from b to b.

If both of them result in identities, then we say that g is the *inverse* of f

$$g \circ f = id_a$$

$$f \circ g = id_b$$

and we write it as $g = f^{*1}$ (pronounced f *inverse*). The arrow f^{*1} undoes the work of the arrow f.

Such a pair of arrows is called an *isomorphism* and the two objects are called *isomorphic*.

What does the existence of an isomorphism tell us about the two objects it connects?

We have said that objects are described by their interactions with other objects. So let's consider what the two isomorphic objects look like from the perspective of an observer x. Take an arrow h coming from x to a.

$$\begin{tikzcd}
& x \arrow[dl, "h"'] & \\
a \arrow[rr, "f"] & & b \arrow[ll, bend left, "f^{*1}"]
\end{tikzcd}$$

There is a corresponding arrow coming from x to b. It's just the composition of $f \circ h$, or the action of $(f \circ *)$ on h.

$$\begin{tikzcd}
& x \arrow[dl, "h"'] \arrow[dr, "f \circ h"] & \\
a \arrow[rr, "f"] & & b \arrow[ll, bend left, "f^{*1}"]
\end{tikzcd}$$

Similarly, for any arrow probing b there is a corresponding arrow probing a. It is given by the action of $(f^{*1} \circ *)$.

We can move focus back and forth between a and b using the mappings $(f \circ *)$ and $(f^{*1} \circ *)$.

We can combine these two mappings (see exercise 2.1.1) to form a round trip. The result is the same as if we applied the composite $((f^{*1} \circ f) \circ *)$. But this is equal to $(id_a \circ *)$ which, as we know from exercise 2.3.1, leaves the arrows unchanged.

Similarly, the round trip induced by $f \circ f^{*1}$ leaves the arrows $x \to b$ unchanged.

This creates a "buddy system" between the two groups of arrows. Imagine each arrow sending a message to its buddy, as determined by f or f^{*1}. Each arrow would then receive exactly one message, and that would be a message from its buddy. No arrow would be left behind, and no arrow would receive more than one message. Mathematicians call this kind of buddy system a *bijection* or one-to-one correspondence.

Therefore, arrow by arrow, the two objects a and b look exactly the same from the perspective of x. Arrow-wise, there is no difference between the two objects.

Two isomorphic objects have exactly the same properties.

In particular, if you replace x with the terminal object 1, you'll see that the two objects have the same elements. For every element $x\colon 1 \to a$ there is a corresponding element $y\colon 1 \to b$, namely $y = f \circ x$, and vice versa. There is a bijection between the elements of isomorphic objects.

Such indistinguishable objects are called *isomorphic* because they have "the same shape." You've seen one, you've seen them all.

We write this isomorphism as:

$$a \text{ ö } b$$

When dealing with objects, we use isomorphism in place of equality.

In programming, two isomorphic types have the same external behavior. One type can be implemented in terms of the other and vice versa. One can be replaced by the other without changing the behavior of the system (except, possibly, the performance).

In classical logic, if B follows from A and A follows from B then A and B are logically equivalent. We often say that B is true "if and only if" A is true. However, unlike previous parallels between logic and type theory, this one is not as straightforward if you consider proofs to be relevant. In fact, it led to the development of a new branch of fundamental mathematics called homotopy type theory, or HoTT for short.

Exercise 3.1.1. *Make an argument that there is a bijection between arrows that are* outgoing *from two isomorphic objects. Draw the corresponding diagrams.*

Exercise 3.1.2. *Show that every object is isomorphic to itself*

Exercise 3.1.3. *If there are two terminal objects, show that they are isomorphic*

Exercise 3.1.4. *Show that the isomorphism from the previous exercise is unique.*

3.2 Naturality

We've seen that, when two objects are isomorphic, we can switch focus from one to another using post-composition: either $(f \circ *)$ or $(f^{*1} \circ *)$.

Conversely, to switch between different observers, we would use pre-composition.

Indeed, an arrow h probing a from x is related to the arrow $h\circ g$ probing the same object from y.

$$\begin{tikzcd}
x \arrow[d, "h"'] & & y \arrow[ll, dashed, "g"'] \arrow[dll, "h\circ g"'] \\
a \arrow[rr, "f"] & & b \arrow[ll, bend left, "f^{*1}"]
\end{tikzcd}$$

Similarly, an arrow $h^{\ddot{}}$ probing b from x corresponds to the arrow $h^{\ddot{}}\circ g$ probing it from y.

$$\begin{tikzcd}
x \arrow[drr, "h^{\ddot{}}"] & & y \arrow[ll, dashed, "g"'] \arrow[d, "h^{\ddot{}}\circ g"] \\
a \arrow[rr, "f"] & & b \arrow[ll, bend left, "f^{*1}"]
\end{tikzcd}$$

In both cases, we change perspective from x to y by applying precomposition $(*\circ g)$.

The important observation is that the change of perspective preserves the buddy system established by the isomorphism. If two arrows were buddies from the perspective of x, they are still buddies from the perspective of y. This is as simple as saying that it doesn't matter if you first pre-compose with g (switch perspective) and then post-compose with f (switch focus), or first post-compose with f and then pre-compose with g. Symbolically, we write it as:

$$(*\circ g)\circ(f\circ *) = (f\circ *)\circ(*\circ g)$$

and we call it the *naturality* condition.

The meaning of this equation is revealed when you apply it to a morphism $h\colon x\to a$. Both sides evaluate to $f\circ h\circ g$.

$$\begin{tikzcd}
h \arrow[r, mapsto, "(*\circ g)"] \arrow[d, mapsto, "(f\circ *)"'] & h\circ g \arrow[d, mapsto, "(f\circ *)"] \\
f\circ h \arrow[r, mapsto, "(*\circ g)"] & f\circ h\circ g
\end{tikzcd}$$

Here, the naturality condition is satisfied automatically due to associativity, but we'll soon see it generalized to less trivial circumstances.

Arrows are used to broadcast information about an isomorphism. Naturality tells us that all objects get a consistent view of it, independent of the path.

We can also reverse the roles of observers and subjects. For instance, using an arrow $h\colon a\to x$, the object a can probe an arbitrary object x. If there is an arrow $g\colon x\to y$, it can switch focus to y. Switching the perspective to b is done by precomposition with f^{*1}.

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[d, "h"'] \arrow[dr, red, "g\circ h"] & b \arrow[l, bend right, "f^{*1}"'] \\
x \arrow[r, dashed, "g"'] & y
\end{tikzcd}$$

Again, we have the naturality condition, this time from the point of view of the isomorphic pair:

$$(*\circ f^{*1})\circ(g\circ *) = (g\circ *)\circ(*\circ f^{*1})$$

This situation when we have to take two steps to move from one place to another is typical in category theory. Here, the operations of pre-composition and post-composition can be done in any order—we say that they *commute*. But in general the order in which we take steps leads to different outcomes. We often impose commutation conditions and say that one operation is compatible with another if these conditions hold.

Exercise 3.2.1. *Show that both sides of the naturality condition for f^{*1}, when acting on h, reduce to:*

$$b \xrightarrow{f^{*1}} a \xrightarrow{h} x \xrightarrow{g} y$$

3.3 Reasoning with Arrows

Master Yoneda says: “At the arrows look!”

If two objects are isomorphic, they have the same sets of incoming arrows.

If two objects are isomorphic, they also have the same sets of outgoing arrows.

If you want to see if two objects are isomorphic, at the arrows look!

When two objects a and b are isomorphic, any isomorphism f induces a one-to-one mapping $(f\circ *)$ between corresponding sets of arrows.

$$\begin{tikzcd}
& x \arrow[ddl, bend right, green] \arrow[ddl, bend right=15] \arrow[ddl, blue] \arrow[ddr, bend left, green] \arrow[ddr, bend left=15] \arrow[ddr, blue] & \\
{} \arrow[rr, dashed, red, "(f\circ *)"] & & {} \\
a \arrow[rr, "f"'] & & b
\end{tikzcd}$$

The function $(f\circ *)$ maps every arrow $h\colon x \to a$ to an arrow $f\circ a\colon x \to b$. It’s inverse $(f^{*1}\circ *)$ maps every arrow $h^{\ddot{}}\colon x \to b$ to an arrow $(f^{*1}\circ h^{\ddot{}})$.

Suppose that we don’t know if the objects are isomorphic, but we know that there is an invertible mapping, α_x, between sets of arrows impinging on a and b from every object x. In other words, for every x, α_x is a bijection of arrows.

$$\begin{tikzcd}
 & & x \arrow[ddll, bend right=40] \arrow[ddll, bend right=20] \arrow[ddll] \arrow[ddrr, bend left=40] \arrow[ddrr, bend left=20] \arrow[ddrr] & & \\
 & {} \arrow[rr, dashed, "\alpha_x"] & & {} & \\
a & & & & b
\end{tikzcd}$$

Before, the bijection of arrows was generated by the isomorphism f. Now, the bijection of arrows is given to us by α_x. Does it mean that the two objects are isomorphic? Can we construct the isomorphism f from the family of mappings α_x? The answer is “yes”, as long as the family α_x satisfies the naturality condition.

Here’s the action of α_x on a particular arrow h.

$$\begin{tikzcd}
x \arrow[d, "h"'] \arrow[dr, "\alpha_x h"] & \\
a & b
\end{tikzcd}$$

This mapping, along with its inverse α_x^{*1}, which takes arrows $x \to b$ to arrows $x \to a$, would play the role of $(f \circ *)$ and $(f^{*1} \circ *)$, if there was indeed an isomorphism f. The family of maps α describes an “artificial” way of switching focus from a to b.

Here’s the same situation from the point of view of another observer y:

$$\begin{tikzcd}
x & y \arrow[dl, "h^{\cdot\cdot}"'] \arrow[d, "\alpha_y h^{\cdot\cdot}"] \\
a & b
\end{tikzcd}$$

Notice that y is using a different mapping α_y from the same family.

These two mappings, α_x and α_y, become entangled whenever there is a morphism $g \colon y \to x$. In that case, pre-composition with g allows us to switch perspective from x to y (notice the direction)

$$\begin{tikzcd}
x \arrow[d, "h"'] & y \arrow[l, dashed, "g"'] \arrow[dl, "h \circ g"] \\
a & b
\end{tikzcd}$$

We have separated the switching of focus from the switching of perspective. The former is done by α, the latter by pre-composition. Naturality imposes a compatibility condition between those two.

Indeed, starting with some h, we can either apply $(* \circ g)$ to switch to y’s point of view, and then apply α_y to switch focus to b:

$$\alpha_y \circ (* \circ g)$$

or we can first let x switch focus to b using α_x, and then switch perspective using $(*\circ g)$:

$$(*\circ g)\circ\alpha_x$$

In both cases we end up looking at b from y. We've done this exercise before, when we had an isomorphism between a and b, and we've found out that the results were the same. We called it the naturality condition.

If we want the α's to give us an isomorphism, we have to impose the equivalent naturality condition:

$$\alpha_y\circ(*\circ g) = (*\circ g)\circ\alpha_x$$

When acting on some arrow $h\colon x \to a$, we want this diagram to commute:

$$\begin{tikzcd}
h \arrow[r, mapsto, "(*\circ g)"] \arrow[d, mapsto, "\alpha_x"] & h\circ g \arrow[d, mapsto, "\alpha_y"] \\
\alpha_x h \arrow[r, mapsto, "(*\circ g)"] & (\alpha_x h)\circ g = \alpha_y(h\circ g)
\end{tikzcd}$$

This way we know that replacing all α's with $(f\circ *)$ will work. But does such f exist? Can we reconstruct f from the α's? The answer is yes, and we'll use the Yoneda trick to accomplish that.

Since α_x is defined for any object x, it is also defined for a itself. By definition, α_a takes a morphism $a \to a$ to a morphism $a \to b$. We know for sure that there is at least one morphism $a \to a$, namely the identity id_a. It turns out that the isomorphism f we are seeking is given by:

$$f = \alpha_a(id_a)$$

or, pictorially:

$$\begin{tikzcd}
a \arrow[d, "id_a"'] \arrow[rd, "f=\alpha_a(id_a)"] & \\
a & b
\end{tikzcd}$$

Let's verify this. If f is indeed our isomorphism then, for any x, α_x should be equal to $(f\circ *)$. To see that, let's rewrite the naturality condition replacing x with a. We get:

$$\alpha_y(h\circ g) = (\alpha_a h)\circ g$$

as illustrated in the following diagram:

$$\begin{tikzcd}
a \arrow[d, "h"'] \arrow[rd, "\alpha_a(h)"'] & y \arrow[l, "g"'] \arrow[d, "\alpha_y(h\circ g)"] \\
a & b
\end{tikzcd}$$

Since both the source and the target of h is a, this equality must also be true for $h = id_a$

$$\alpha_y(id_a \circ g) = (\alpha_a(id_a)) \circ g$$

But $id_a \circ g$ is equal to g and $\alpha_a(id_a)$ is our f, so we get:

$$\alpha_y g = f \circ g = (f \circ *)g$$

In other words, $\alpha_y = (f \circ *)$ for every object y and every morphism $g \colon y \to a$.

Notice that, even though α_x was defined individually for every x and every arrow $x \to a$, it turned out to be completely determined by its value at a single identity arrow. This is the power of naturality!

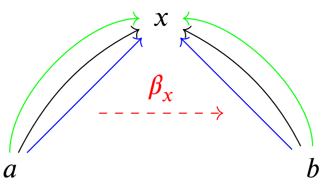
Reversing the Arrows

As Lao Tzu would say, the duality between the observer and the observed cannot be complete unless the observer is allowed to switch roles with the observed.

Again, we want to show that two objects a and b are isomorphic, but this time we want to treat them as observers. An arrow $h \colon a \to x$ probes an arbitrary object x from the perspective of a. Previously, when we knew that the two objects were isomorphic, we were able to switch perspective to b using $(* \circ f^{*1})$. This time we have at our disposal a transformation β_x instead. It establishes the bijection between arrows impinging on x.

$$\begin{tikzcd}
x & \\
a \arrow[u, "h"] & b \arrow[ul, "\beta_x h"']
\end{tikzcd}$$

If we want to observe another object, y, we will use β_y to switch perspectives between a and b, and so on.



If the two objects x and y are connected by an arrow $g \colon x \to y$ then we also have an option of switching focus using $(g \circ *)$. If we want to do both: switch perspective and switch focus, there are two ways of doing it. Naturality demands that the results be equal:

$$(g \circ *) \circ \beta_x = \beta_y \circ (g \circ *)$$

Indeed, if we replace β with $(* \circ f^{*1})$, we recover the naturality condition for an isomorphism.

Exercise 3.3.1. *Use the trick with the identity morphism to recover f^{*1} from the family of mappings β.*

Exercise 3.3.2. *Using f^{*1} from the previous exercise, evaluate $\beta_y g$ for an arbitrary object y and an arbitrary arrow $g \colon a \to y$.*

As Lao Tzu would say: To show an isomorphism, it is often easier to define a natural transformation between ten thousand arrows than it is to find a pair of arrows between two objects.

Chapter 4

Sum Types

4.1 Bool

We know how to compose arrows. But how do we compose objects?

We have defined 0 (the initial object) and 1 (the terminal object). What is 2 if not 1 plus 1?

A 2 is an object with two elements: two arrows coming from 1. Let's call one arrow and the other . Don't confuse those names with the logical interpretations of the initial and the terminal objects. These two are *arrows*.

$$\begin{tikzcd}
1 \arrow[d, bend right, "\text{True}"'] \arrow[d, bend left, "\text{False}"] \\
2
\end{tikzcd}$$

This simple idea can be immediately expressed in Haskell[1] as the definition of a type, traditionally called , after its inventor George Boole (1815-1864).

It corresponds to the same diagram (only with some Haskell renamings):

$$\begin{tikzcd}
\texttt{()} \arrow[d, bend right, "\texttt{True}"'] \arrow[d, bend left, "\texttt{False}"] \\
\texttt{Bool}
\end{tikzcd}$$

As we've seen before, there is a shortcut notation for elements, so here's a more compact version:

[1] This style of definition is called the Generalized Algebraic Data Types or in Haskell

We can now define a term of the type , for instance

The first line declares to be an element of (secretly a function), and the second line tells us which one of the two.

The functions and that we used in the definition of are called *data constructors*. They can be used to construct specific terms, like in the example above. As a side note, in Haskell, function names start with lower-case letters, except when they are data constructors.

Our definition of the type is still incomplete. We know how to construct a term, but we don't know what to do with it. We have to be able to define arrows that go out of —the *mappings out* of .

The first observation is that, if we have an arrow from to some concrete type then we automatically get two arrows and from unit to , just by composition. The following two (distorted) triangles commute:

$$\begin{tikzcd}
() \arrow[d, bend right, "\text{True}"'] \arrow[d, bend left, "\text{False}"] \arrow[dd, bend right=80, "x"'] \arrow[dd, bend left=80, "y"] \\
\text{Bool} \arrow[d, dashed, "h"] \\
A
\end{tikzcd}$$

In other words, every function produces a pair of elements of .

Given a concrete type :

we have:

where

Notice the use of the shorthand notation for the application of a function to an element:

We are now ready to complete our definition of by adding the condition that any function from to not only produces but *is equivalent* to a pair of elements of . In other words, a pair of elements uniquely determines a function from .

What this means is that we can interpret the diagram above in two ways: Given , we can easily get and . But the converse is also true: a pair of elements and uniquely *defines* .

We have a bijection at work here. This time it's a one-to-one mapping between a pair of elements (x, y) and an arrow h.

In Haskell, this definition of is encapsulated in the , , construct. Given

we define the mapping out

Here, is a term of the type .

In general, a data type is created using *introduction* rules and deconstructed using *elimination* rules. The data type has two introduction rules, one using and another using . The , , construct defines the elimination rule.

The fact that, given the above definition of , we can retrieve the two terms that were used to define it, is called the *computation* rule. It tells us how to compute the result of . If we call with , the result is ; if we call it with , the result is .

We should never lose sight of the purpose of programming: to decompose complex problems into a series of simpler ones. The definition of illustrates this idea. Whenever we have to construct a mapping out of , we decompose it into two smaller tasks of constructing a pair of elements of the target type. We traded one larger problem for two simpler ones.

Examples

Let's do a few examples. We haven't defined many types yet, so we'll be limited to mappings out of to either , , or . Such edge cases, however, may offer new insights into well known results.

We have decided that there can be no functions (other than identity) with as a target, so we don't expect any functions from to . And indeed, we have zero pairs of elements of .

What about functions from to ? Since is terminal, there can be only one function from to it. And, indeed, this function corresponds to the single possible pair of functions from to —both being identities. So far so good.

$$\begin{tikzcd}
\mathtt{()} \arrow[d, bend right, "\text{True}"'] \arrow[d, bend left, "\text{False}"] \arrow[dd, bend right=70, "\mathtt{id}"'] \arrow[dd, bend left=70, "\mathtt{id}"] \\
\text{Bool} \arrow[d, dashed, "h"] \\
\mathtt{()}
\end{tikzcd}$$

The interesting case is functions from to . Let's plug in place of :

$$\begin{tikzcd}
 & () \arrow[d, bend right, "\text{True}"'] \arrow[d, bend left, "\text{False}"] \arrow[dd, bend right=80, "x"'] \arrow[dd, bend left=80, "y"] & \\
 & \text{Bool} \arrow[d, dashed, "h"] & \\
 & \text{Bool} &
\end{tikzcd}$$

How many pairs (x, y) of functions from to do we have at our disposal? There are only two such functions, and , so we can form four pairs. These are $(True, True)$, $(False, False)$, $(True, False)$, and $(False, True)$. Therefore there can only be four functions from to .

We can write them in Haskell using the , , construct. For instance, the last one, which we'll call is defined as:

We can also look at functions from to as elements of the object of arrows, or the exponential object A^2, where 2 is the object. According to our count, we have zero elements in 0^2, one element in 1^2, and four elements in 2^2. This is exactly what we'd expect from high-school algebra, where numbers actually mean numbers.

Exercise 4.1.1. *Write the implementations of the three other functions .*

4.2 Enumerations

What comes after 0, 1, and 2? An object with three data constructors. For instance:

If you're tired of redundant syntax, there is a shorthand for this type of definition:

This introduction rule allows us to construct terms of the type , for instance:

To define mappings out of , we need a more general elimination pattern. Just like a function from was determined by two elements, a function from to is determined by a triple of elements of : , , and . We write such a function using *pattern matching* syntax:

This is just one function whose definition is split into three cases.

It's possible to use the same syntax for as well, in place of , , :

In fact, there is a third way of writing the same thing using the statement:

or even

You can use any of these at your convenience when programming.

These patterns will also work for types with four, five, and more data constructors. For instance, a decimal digit is one of:

There is a giant enumeration of Unicode characters called . Their constructors are given special names: you write the character itself between two apostrophes, e.g.,

As Lao Tzu would say, a pattern of ten thousand things would take many years to complete, therefore people came up with the wildcard pattern, the underscore, which matches everything.

Because the patterns are matched in order, you should use the wildcard pattern as the last one in a series:

But why should we stop at that? The type could be thought of as an enumeration of integers in the range between *2^{29} and 2^{29} (or more, depending on the implementation). Of course, exhaustive pattern matching on such ranges is out of the question, but the principle holds.

In practice, the types for Unicode characters, for fixed-precision integers, for double-precision floating point numbers, and several others, are built into the language.

These are not infinite types. Their elements can be enumerated, even if it would take ten thousand years. The type is infinite, though.

Short Haskell Digression

Since we are going to write more Haskell code, we have to establish some preliminaries. To define data types using functions, we need to use the language pragma called (it stands for *Generalized Algebraic Data Types*). The pragma has to be put at the top of the source file. For instance:

The data type can be defined as:

with the empty clause (no data constructor!).

The function works with any type as its target (it's a *polymorphic* function), so it is parameterized by a *type variable*. Unlike concrete types, type variables must start with a lowercase letter. Here, is such a type variable:

We use to placate the compiler. In this case, we are absolutely sure that the function can never be called, because it's impossible to construct an argument of type .

You may use when you're only interested in compiling, as opposed to running, your code. For instance, you may need to plug a function to check if your definitions work together:

If you want to experiment with defining your own versions of standard types, like , you have to tell the compiler to hide the originals that are defined in the standard library called the . Put this line at the top of the file, after the language pragmas:

4.3 Sum Types

The type could be seen as the sum $2 = 1 + 1$. But nothing stops us from replacing 1 with another type, or even replacing each of the 1s with different types. We can define a new type $a + b$ by using two arrows. Let's call them and . The defining diagram is the introduction rule:

$$\begin{tikzcd}
a \arrow[dr, "\text{Left}"'] & & b \arrow[dl, "\text{Right}"] \\
& a + b &
\end{tikzcd}$$

In Haskell, the type $a + b$ is called . By analogy with , we can define it as

(Note the use of lower-case letters for type variables.)

Similarly, the mapping out from $a + b$ to some type c is determined by this commuting diagram:

$$\begin{tikzcd}
a \arrow[dr, bend left, "\text{Left}"] \arrow[ddr, bend right, "f"'] & & b \arrow[dl, bend right, "\text{Right}"'] \arrow[ddl, bend left, "g"] \\
& a + b \arrow[d, dashed, "h"] & \\
& c &
\end{tikzcd}$$

Given a function h, we get a pair of functions f and g just by composing it with and . Conversely, such a pair of functions uniquely determines h. This is the elimination rule.

When we want to translate this diagram to Haskell, we need to select elements of the two types. We can do it by defining the arrows a and b from the terminal object.

$$\begin{tikzcd}
 & 1 \arrow[dl, "a"'] \arrow[dr, "b"] & \\
a \arrow[dr, bend left, "\text{Left}"] \arrow[ddr, bend right, "f"'] & & b \arrow[dl, bend right, "\text{Right}"'] \arrow[ddl, bend left, "g"] \\
 & a+b \arrow[d, dashed, "h"] & \\
 & c &
\end{tikzcd}$$

Follow the arrows in this diagram to get:

$$h \circ \text{Left} \circ a = f \circ a$$

$$h \circ \text{Right} \circ b = g \circ b$$

Haskell syntax repeats these equations almost literally, resulting in this pattern-matching syntax for the definition of :

(Again, notice the use of lower-case letters for type variables and the same letters for terms of that type. Unlike humans, the compilers don't get confused by this.)

You can also read these equations right to left, and you will see the computation rules for sum types: The two functions that were used to define can be recovered by applying to and .

You can also use the syntax to define :

So what is the essence of a data type? It is but a recipe for manipulating arrows.

Maybe

A very useful data type, is defined as a sum $1 + a$, for any a. This is its definition in Haskell:

The data constructor is an arrow from the unit type, and constructs from . is isomorphic to . It can also be defined using the shorthand notation

is mostly used to encode the return type of partial functions: ones that are undefined for some values of their arguments. In that case, instead of failing, such functions return . In other programming languages partial functions are often implemented using exceptions (or core dumps).

Logic

In logic, the proposition $A + B$ is called the alternative, or *logical or*. You can prove it by providing the proof of A or the proof of B. Either one will suffice.

If you want to prove that C follows from $A + B$, you have to be prepared for two eventualities: either somebody proved $A + B$ by proving A (and B could be false) or by proving B (and A could be false). In the first case, you have to show that C follows from A. In the second case you need a proof that C follows from B. These are exactly the arrows in the elimination rule for $A + B$.

4.4 Cocartesian Categories

In Haskell, we can define a sum of any two types using . A category in which all sums exist, and the initial object exists, is called *cocartesian*, and the sum is called a *coproduct*. You might have noticed that sum types mimic addition of numbers. It turns out that the initial object plays the role of zero.

One Plus Zero

Let's first show that $1 + 0$ ö 1, meaning the sum of the terminal object and the initial object is isomorphic to the terminal object. The standard procedure for this kind of proofs is to use the Yoneda trick. Since sum types are defined by mapping out, we should compare arrows coming *out* of either side.

The Yoneda argument says that two objects are isomorphic if there is a bijection β_a between the sets of arrows coming out of them to an arbitrary object a, and this bijection is natural.

Let's look at the definition of $1 + 0$ and it's mapping out to any object a. This mapping is defined by a pair $(x, ¡)$, where x is an element of a and ¡ is the unique arrow from the initial object to a (the function in Haskell).

$$\begin{tikzcd}
1 \arrow[dr, bend left, "\text{Left}"'] \arrow[ddr, bend right, "x"'] & & 0 \arrow[dl, bend right, "\text{Right}"] \arrow[ddl, bend left, "¡"] & & 1 \arrow[dd, "x"] \\
& 1+0 \arrow[d, dashed, "h"] & & & \\
& a & & & a
\end{tikzcd}$$

We want to establish a one-to-one mapping between arrows originating in $1 + 0$ and the ones originating in 1. The arrow h is determined by the pair $(x, ¡)$. Since there is only one ¡, there is a bijection between h's and x's.

We define β_a to map any h defined by a pair $(x, ¡)$ to x. Conversely, β_a^{*1} maps x to the pair $(x, ¡)$. But is it a natural transformation?

To answer that, we need to consider what happens when we change focus from a to some b that is connected to it through an arrow $g \colon a \to b$. We have two options now:

- Make h switch focus by post-composing both x and ¡ with g. We get a new pair $(y = g \circ x, ¡)$. Follow it by β_b.
- Use β_a to map $(x, ¡)$ to x. Follow it with the post-composition $(g \circ *)$.

In both cases we get the same arrow $y = g \circ x$. So the mapping β is natural. Therefore $1 + 0$ is isomorphic to 1.

In Haskell, we can define the two functions that form the isomorphism, but there is no way of directly expressing the fact that they are the inverse of each other.

The underscore wildcard in a function definition means that the argument is ignored. The second clause in the definition of is redundant, since there are no terms of the type .

Something Plus Zero

A very similar argument can be used to show that $a + 0$ ö a. The following diagram explains it.

$$\begin{tikzcd}
a \arrow[dr, bend left, "\text{Left}"] \arrow[ddr, bend right, "f"'] & & 0 \arrow[dl, bend right, "\text{Right}"'] \arrow[ddl, bend left, "¡"] \\
& a + 0 \arrow[d, dashed, "h"] & \\
& x &
\end{tikzcd}
\qquad
\begin{tikzcd}
a \arrow[d, "f"] \\
x
\end{tikzcd}$$

We can translate this argument to Haskell by implementing a (polymorphic) function that works for any type .

Exercise 4.4.1. *Implement, in Haskell, the two functions that form the isomorphism between and .*

We could use a similar argument to show that $0 + a$ ö a, but there is a more general property of sum types that obviates that.

Commutativity

There is a nice left-right symmetry in the diagrams that define the sum type, which suggests that it satisfies the commutativity rule, $a + b$ ö $b + a$.

Let’s consider mappings out of both sides of this formula. You can easily see that, for every h that is determined by a pair (f, g) on the left, there is a corresponding $h^{\text{¨}}$ given by a pair (g, f) on the right. That establishes the bijection of arrows.

$$\begin{tikzcd}
a \arrow[dr, bend left, "\text{Left}"] \arrow[ddr, bend right, "f"'] & & b \arrow[dl, bend right, "\text{Right}"'] \arrow[ddl, bend left, "g"] \\
& a+b \arrow[d, dashed, "h"] & \\
& x &
\end{tikzcd}
\qquad
\begin{tikzcd}
b \arrow[dr, bend left, "\text{Left}"] \arrow[ddr, bend right, "g"'] & & a \arrow[dl, bend right, "\text{Right}"'] \arrow[ddl, bend left, "f"] \\
& b+a \arrow[d, dashed, "h^{\text{¨}}"] & \\
& x &
\end{tikzcd}$$

Exercise 4.4.2. *Show that the bijection defined above is natural. Hint: Both f and g change focus by post-composition with $k \colon x \to y$.*

Exercise 4.4.3. *Implement, in Haskell, the function that witnesses the isomorphism between and . Notice that this function is its own inverse.*

Associativity

Just like in arithmetic, the sum that we have defined is associative:

$$(a+b)+c \text{ ö } a+(b+c)$$

It’s easy to write the mapping out for the left-hand side:

Notice the use of nested patterns like , etc. The mapping is fully defined by a triple of functions. The same functions can be used to define the mapping out of the right-hand side:

This establishes a one-to-one mapping between triples of functions that define the two mappings out. This mapping is natural because all changes of focus are done using post-composition. Therefore the two sides are isomorphic.

This code can also be displayed in diagrammatical form. Here’s the diagram for the left hand side of the isomorphism:

$$\begin{tikzcd}
a \arrow[dr, "L"'] \arrow[dddrr, bend right, red, "f_1"'] & & b \arrow[dl, "R"] \arrow[ddd, bend left, red, "f_2"] & & c \arrow[ddll, "R"'] \arrow[dddll, bend left, red, "f_3"] \\
& a+b \arrow[dr, "L"'] & & & \\
& & (a+b)+c \arrow[d, dashed, "h"] & & \\
& & x & &
\end{tikzcd}$$

Functoriality

Since the sum is defined by the mapping out property, it was easy to see what happens when we change focus: it changes “naturally” with the foci of the arrows that define the product. But what happens when we move the sources of those arrows?

Suppose that we have arrows that map a and b to some a' and b':

$$f \colon a \to a'$$
$$g \colon b \to b'$$

The composition of these arrows with the constructors Left and Right, respectively, can be used to define the mapping between the sums:

$$\begin{tikzcd}
a \arrow[d, "f"] \arrow[rd, bend left, "\text{Left}"'] & & b \arrow[ld, bend right, "\text{Right}"] \arrow[d, "g"'] \\
a' \arrow[rd, "\text{Left}"'] & a+b \arrow[d, dashed, "h"] & b' \arrow[ld, "\text{Right}"] \\
& a'+b' &
\end{tikzcd}$$

The pair of arrows, $(\text{Left} \circ f, \text{Right} \circ g)$ uniquely defines the arrow $h \colon a+b \to a'+b'$.

This property of the sum is called *functoriality*. You can imagine it as allowing you to transform the two objects *inside* the sum and get a new sum. We also say that functoriality lets us *lift* a pair of arrows in order to operate on sums.

Exercise 4.4.4. *Show that functoriality preserves composition. Hint: take two composable arrows, $g \colon b \to b'$ and $g' \colon b' \to b''$ and show that applying $g' \circ g$ gives the same result as first applying g to transform $a+b$ to $a+b'$ and then applying g' to transform $a+b'$ to $a+b''$.*

Exercise 4.4.5. *Show that functoriality preserves identity. Hint: use id_b and show that it is mapped to id_{a+b}.*

Symmetric Monoidal Category

When a child learns addition we call it arithmetic. When a grownup learns addition we call it a cocartesian category.

Whether we are adding numbers, composing arrows, or constructing sums of objects, we are re-using the same idea of decomposing complex things into their simpler components.

As Lao Tzu would say, when things come together to form a new thing, and the operation is associative, and it has a neutral element, we know how to deal with ten thousand things.

The sum type we have defined satisfies these properties:

$$a + 0 \text{ ö } a$$

$$a + b \text{ ö } b + a$$

$$(a + b) + c \text{ ö } a + (b + c)$$

and it's functorial. A category with this type of operation is called *symmetric monoidal*. When the operation is the sum (coproduct), it's called *cocartesian*. In the next chapter we'll see another monoidal structure that's called *cartesian* without the "co."

Chapter 5

Product Types

We can use sum types to enumerate possible values of a given type, but the encoding can be wasteful. We needed ten constructors just to encode numbers between zero and nine.

But if we combine two digits into a single data structure, a two-digit decimal number, we'll be able to encode a hundred numbers. Or, as Lao Tzu would say, with just four digits you can encode ten thousand numbers.

A data type that combines two types in this manner is called a product, or a *cartesian* product. Its defining quality is the elimination rule: there are two arrows coming from $a \quad b$; one called "fst" goes to a, and another called "snd" goes to b. They are called (cartesian) *projections*. They let us retrieve a and b from the product $a \quad b$.

$$\begin{tikzcd}
 & a \quad b \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\
a & & b
\end{tikzcd}$$

Suppose that somebody gave you an element of a product, that is an arrow h from the terminal object 1 to $a \quad b$. You can easily retrieve a pair of elements, just by using composition: an element of a given by

$$x = \text{fst} \circ h$$

and an element of b given by

$$y = \text{snd} \circ h$$

$$\begin{tikzcd}
& 1 \arrow[d, dashed, "h"] \arrow[ddl, bend right, "x"'] \arrow[ddr, bend left, "y"] & \\
& a\ \ b \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\
a & & b
\end{tikzcd}$$

In fact, given an arrow from an arbitrary object c to $a\ \ b$, we can define, by composition, a pair of arrows $f : c \to a$ and $g : c \to b$

$$\begin{tikzcd}
& c \arrow[d, dashed, "h"] \arrow[ddl, bend right, "f"'] \arrow[ddr, bend left, "g"] & \\
& a\ \ b \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\
a & & b
\end{tikzcd}$$

As we did before with the sum type, we can turn this idea around, and use this diagram to *define* the product type: We impose the condition that a pair of functions f and g be in one-to-one correspondence with a *mapping in* from c to $a\ \ b$. This is the *introduction* rule for the product.

In particular, the mapping out of the terminal object is used in Haskell to define a product type. Given two elements, and , we construct the product

The built-in syntax for products is just that: a pair of parentheses and a comma in between. It works both for defining the product of two types and the data constructor that takes two elements and pairs them together.

We should never lose sight of the purpose of programming: to decompose complex problems into a series of simpler ones. We see it again in the definition of the product. Whenever we have to construct a mapping *into* the product, we decompose it into two smaller tasks of constructing a pair of functions, each mapping into one of the components of the product. This is as simple as saying that, in order to implement a function that returns a pair of values, it's enough to implement two functions, each returning one of the elements of the pair.

Logic

In logic, a product type corresponds to logical conjunction. In order to prove $A\ \ B$ (A and B), you need to provide the proofs of *both* A and B. These are the arrows targeting A and B. The elimination rule says that if you have a proof of $A\ \ B$, then you automatically get the proof of A (through fst) and the proof of B (through snd).

Tuples and Records

As Lao Tzu would say, a product of ten thousand objects is just an object with ten thousand projections.

We can form arbitrary products in Haskell using the tuple notation. For instance, a product of three types is written as . A term of this type can be constructed from three elements: .

In what mathematicians call "abuse of notation", a product of zero types is written as , an empty tuple, which happens to be the same as the terminal object, or unit type. This is because the product behaves very much like multiplication of numbers, with the terminal object playing the role of one.

In Haskell, rather than defining separate projections for all tuples, we use the pattern-matching syntax. For instance, to extract the third component from a triple we would write

We use wildcards for the components that we want to ignore.

Lao Tzu said that "Naming is the origin of all particular things." In programming, keeping track of the meaning of the components of a particular tuple is difficult without giving them names. Record syntax allows us to give names to projections. This is the definition of a product written in record style:

 is the data constructor and and are the projections.

This is how it could be used to declare and initialize a particular pair:

5.1 Cartesian Category

In Haskell, we can define a product of any two types. A category in which all products exist, and the terminal object exists, is called *cartesian*.

Tuple Arithmetic

The identities satisfied by the product can be derived using the mapping-in property. For instance, to show that $a \quad b$ ö $b \quad a$ consider the following two diagrams:

$$\begin{tikzcd}
& x \arrow[dl, bend right, "f"'] \arrow[d, dashed, "h"] \arrow[dr, bend left, "g"] & \\
& a \quad b \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\
a & & b
\end{tikzcd}
\qquad
\begin{tikzcd}
& x \arrow[dl, bend right, "g"'] \arrow[d, dashed, "h\text{¨}"] \arrow[dr, bend left, "f"] & \\
& b \quad a \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\
b & & a
\end{tikzcd}$$

They show that, for any object x the arrows to $a \;\; b$ are in one-to-one correspondence with arrows to $b \;\; a$. This is because each of these arrows is determined by the same pair f and g.

You can check that the naturality condition is satisfied because, when you shift the perspective using $k \colon x^{¨} \to x$, all arrows originating in x are shifted by pre-composition $(* \circ k)$.

In Haskell, this isomorphism can be implemented as a function which is its own inverse:

Here's the same function written using pattern matching:

It's important to keep in mind that the product is symmetric only "up to isomorphism." It doesn't mean that swapping the order of pairs won't change the behavior of a program. Symmetry means that the information content of a swapped pair is the same, but access to it needs to be modified.

The terminal object is the unit of the product, $1 \;\; a$ ö a. The arrow that witnesses the isomorphism between $1 \;\; a$ and a is called the *left unitor*:

$$\lambda \colon 1 \;\; a \to a$$

It can be implemented as $\lambda = \text{snd}$. Its inverse λ^{*1} is defined as the unique arrow in the following diagram:

$$\begin{tikzcd} & a \arrow[d, dashed, "\lambda^{*1}"] \arrow[ddl, bend right, "!"'] \arrow[ddr, bend left, "\text{id}"] & \\ & 1 \;\; a \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\ 1 & & a \end{tikzcd}$$

The arrow from a to 1 is called ! (pronounced, *bang*). This indeed shows that

$$\text{snd} \circ \lambda^{*1} = \text{id}$$

We still have to prove that λ^{*1} is the left inverse of snd. Consider the following diagram:

$$\begin{tikzcd} & 1 \;\; a \arrow[d, dashed, "h"] \arrow[ddl, bend right, "!"'] \arrow[ddr, bend left, "\text{snd}"] & \\ & 1 \;\; a \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\ 1 & & a \end{tikzcd}$$

It obviously commutes for $h = id$. It also commutes for $h = \lambda^{*1} \circ \text{snd}$, because we have:

$$\text{snd} \circ \lambda^{*1} \circ \text{snd} = \text{snd}$$

Since h is supposed to be unique, we conclude that:

$$\lambda^{*1} \circ \text{snd} = id$$

This kind of reasoning with universal constructions is pretty standard.

Here are some other isomorphisms written in Haskell (without proofs of having the inverse). This is associativity:

And this is the right unit

These two functions correspond to the *associator*

$$\alpha \colon (a \quad b) \quad c \to a \quad (b \quad c)$$

and the *right unitor*:

$$\rho \colon a \quad 1 \to a$$

Exercise 5.1.1. *Show that the bijection in the proof of left unit is natural. Hint, change focus using an arrow $g \colon a \to b$.*

Exercise 5.1.2. *Construct an arrow*

$$h \colon b + a \quad b \to (1 + a) \quad b$$

Is this arrow unique?

Hint: It's a mapping into a product, so it's given by a pair of arrow. These arrows, in turn, map out of a sum, so each is given by a pair of arrows.

Hint: The mapping $b \to 1 + a$ is given by $(Left \circ !)$

Exercise 5.1.3. *Redo the previous exercise, this time treating h as a mapping* out *of a sum.*

Exercise 5.1.4. *Implement a Haskell function . Is this function uniquely defined by its type signature or is there some leeway?*

Functoriality

Suppose that we have arrows that map a and b to some $a^{\ddot{}}$ and $b^{\ddot{}}$:

$$f \colon a \to a^{\ddot{}}$$
$$g \colon b \to b^{\ddot{}}$$

The composition of these arrows with the projections fst and snd, respectively, can be used to define the mapping h between the products:

$$\begin{tikzcd}
 & a\ \ b \arrow[ld, "\text{fst}"'] \arrow[rd, "\text{snd}"] \arrow[d, dashed, "h"] & \\
a \arrow[d, "f"'] & a^{\cdot\cdot}\ \ b^{\cdot\cdot} \arrow[ld, "\text{fst}"] \arrow[rd, "\text{snd}"'] & b \arrow[d, "g"] \\
a^{\cdot\cdot} & & b^{\cdot\cdot}
\end{tikzcd}$$

The shorthand notation for this diagram is:

$$a\ \ b \xrightarrow{f\ g} a^{\cdot\cdot}\ \ b^{\cdot\cdot}$$

This property of the product is called *functoriality*. You can imagine it as allowing you to transform the two objects *inside* the product to get the new product. We also say that functoriality lets us *lift* a pair of arrows in order to operate on products.

5.2 Duality

When a child sees an arrow, it knows which end points at the source, and which points at the target

$$a \to b$$

But maybe this is just a preconception. Would the Universe be very different if we called b the source and a the target?

We would still be able to compose this arrow with this one

$$b \to c$$

whose "target" b is the same as the same as the "source" of $a \to b$, and the result would still be an arrow

$$a \to c$$

only now we would say that it goes from c to a.

In this dual Universe, the object that we call "initial" would be called "terminal," because it's the "target" of unique arrows coming from all objects. Conversely, the terminal object would be called initial.

Now consider this diagram that we used to define the sum object:

$$\begin{tikzcd}
a \arrow[rd, bend left, "\text{Left}"'] \arrow[rdd, bend right, "f"'] & & b \arrow[ld, bend right, "\text{Right}"] \arrow[ldd, bend left, "g"] \\
 & a+b \arrow[d, dashed, "h"] & \\
 & c &
\end{tikzcd}$$

In the new interpretation, the arrow h would go “from” an arbitrary object c “to” the object we call $a + b$. This arrow is uniquely defined by a pair of arrows (f, g) whose “source” is c. If we rename Left to fst and Right to snd, we will get the defining diagram for a product.

A product is the sum with arrows reversed.

Conversely, a sum is the product with arrows reversed.

Every construction in category theory has its dual.

If the direction of arrows is just a matter of interpretation, then what makes sum types so different from product types, in programming? The difference goes back to one assumption we made at the start: There are no incoming arrows to the initial object (other than the identity arrow). This is in contrast with the terminal object having lots of outgoing arrows, arrows that we used to define (global) elements. In fact, we assume that every object of interest has elements, and the ones that don’t are isomorphic to .

We’ll see an even deeper difference when we talk about function types.

5.3 Monoidal Category

We have seen that the product satisfies these simple rules:

$$1 \quad a \text{ ö } a$$

$$a \quad b \text{ ö } b \quad a$$

$$(a \quad b) \quad c \text{ ö } a \quad (b \quad c)$$

and is functorial.

A category in which an operation with these properties is defined is called *symmetric monoidal*[1]. We’ve seen a similar structure before, when working with sums and the initial object.

A category can have multiple monoidal structures at the same time. When you don’t want to name your monoidal structure, you replace the plus sign or the product sign with a tensor sign, and the neutral element with the letter I. The rules of a symmetric monoidal category can then be written as:

$$I \otimes a \text{ ö } a$$

$$a \otimes b \text{ ö } b \otimes a$$

$$(a \otimes b) \otimes c \text{ ö } a \otimes (b \otimes c)$$

[1] Strictly speaking, a product of two objects is defined up to isomorphism, whereas the product in a monoidal category must be defined on the nose. But we can get a monoidal category by making a choice of a product

These isomorphisms are often written as families of invertible arrows called associators and unitors. If the monoidal category is not symmetric, there is a separate left and right unitor.

$$\alpha\colon (a \otimes b) \otimes c \to a \otimes (b \otimes c)$$
$$\lambda\colon I \otimes a \to a$$
$$\rho\colon a \otimes I \to a$$

The symmetry is witnessed by:

$$\gamma : a \otimes b \to b \otimes a$$

Functoriality lets us *lift* a pair of arrows:

$$f : a \to a^{\cdot\cdot}$$
$$g : b \to b^{\cdot\cdot}$$

to operate on tensor products:

$$a \otimes b \xrightarrow{f \otimes g} a^{\cdot\cdot} \otimes b^{\cdot\cdot}$$

If we think of morphisms as actions, their tensor product corresponds to performing two actions in parallel. Contrast this with the serial composition of morphisms, which suggests their temporal ordering.

You may think of a tensor product as the lowest common denominator of product and sum. It still has an introduction rule, which requires both objects a and b; but it has no elimination rule. Once created, a tensor product "forgets" how it was created. Unlike a cartesian product, it has no projections.

Some interesting examples of tensor products are not even symmetric.

Monoids

Monoids are very simple structures equipped with a binary operation and a unit. Natural numbers with addition and zero form a monoid. So do natural numbers with multiplication and one.

The intuition is that a monoid lets you combine two things to get another thing. There is also one special thing, such that combining it with anything else gives back the same thing. That's the unit. And the combining must be associative.

What's not assumed is that the combining is symmetric, or that there is an inverse element.

The rules that define a monoid are reminiscent of the rules of a category. The difference is that, in a monoid, any two things are composable, whereas in a category this is usually not the case: You can only compose two arrows if the target of one is the source of another. Except, that is, when the category contains only one object, in which case all arrows are composable.

A category with a single object is called a monoid. The combining operation is the composition of arrows and the unit is the identity arrow.

This is a perfectly valid definition. In practice, however, we are often interested in monoids that are embedded in larger categories. In particular, in programming, we want to be able to define monoids inside the category of types and functions.

However, in a category, rather than looking at individual elements, we prefer to define operations in bulk. So we start with an object m. A binary operation is a function of two arguments. Since elements of a product are pairs of elements, we can characterize a binary operation as an arrow from a product $m \quad m$ to m:

$$\mu\colon m \quad m \to m$$

The unit element can be defined as an arrow from the terminal object 1:

$$\eta\colon 1 \to m$$

We can translate this description directly to Haskell by defining a class of types equipped with two methods, traditionally called and :

```
```

The two arrows μ and η have to satisfy monoid laws but, again, we have to formulate them in bulk, without any recourse to elements.

To formulate the left unit law, we first create the product $1 \quad m$. We then use η to "pick the unit element in m" or, in terms of arrows, turn 1 into m. Since we are operating on a product $1 \quad m$, we have to lift the pair $\langle \eta, id_m \rangle$, which ensures that we "do not touch" the m. Finally we perform the "multiplication" using μ.

We want the result to be the same as the original element of m, but without mentioning elements. So we just use the left unitor λ to go from $1 \quad m$ to m without "stirring things up."

$$\begin{tikzcd}
1 \quad m \arrow[r, "\eta \quad id_m"] \arrow[dr, "\lambda"'] & m \quad m \arrow[d, "\mu"] \\
& m
\end{tikzcd}$$

Here is the analogous law for the right unit:

$$\begin{tikzcd}
m \quad m \arrow[d, "\mu"] & m \quad 1 \arrow[l, "id_m \quad \eta"'] \arrow[dl, "\rho"] \\
m &
\end{tikzcd}$$

To formulate the law of associativity, we have to start with a triple product and act on it in bulk. Here, α is the associator that rearranges the product without “stirring things up.”

$$\begin{tikzcd}
(m\ \ m)\ \ m \arrow[rr, "\alpha"] \arrow[d, "\mu\ id"] & & m\ \ (m\ \ m) \arrow[d, "id\ \mu"] \\
m\ \ m \arrow[dr, "\mu"] & & m\ \ m \arrow[dl, "\mu"'] \\
& m &
\end{tikzcd}$$

Notice that we didn’t have to assume a lot about the categorical product that we used with the objects m and 1. In particular we never had to use projections. This suggests that the above definition will work equally well for a tensor product in an arbitrary monoidal category. It doesn’t even have to be symmetric. All we have to assume is that: there is a unit object, that the product is functorial, and that it satisfies the unit and associativity laws up to isomorphism.

Thus if we replace   with $\otimes$ and 1 with I, we get a definition of a monoid in an arbitrary monoidal category.

A *monoid* in a monoidal category is an object m equipped with two morphisms:

$$\mu\colon m \otimes m \to m$$

$$\eta\colon I \to m$$

satisfying the unit and associativity laws:

$$\begin{tikzcd}
1 \otimes m \arrow[r, "\eta\otimes id_m"] \arrow[dr, "\lambda"'] & m \otimes m \arrow[d, "\mu"'] & m \otimes 1 \arrow[l, "id_m\otimes\eta"'] \arrow[dl, "\rho"] \\
& m &
\end{tikzcd}$$

$$\begin{tikzcd}
(m \otimes m) \otimes m \arrow[rr, "\alpha"] \arrow[d, "\mu\otimes id_m"] & & m \otimes (m \otimes m) \arrow[d, "id_m\otimes\mu"] \\
m \otimes m \arrow[dr, "\mu"] & & m \otimes m \arrow[dl, "\mu"'] \\
& m &
\end{tikzcd}$$

We used the functoriality of $\otimes$ to lift pairs of arrows, as in $\eta \otimes id_m$, $\mu \otimes id_m$, etc.

Chapter 6

Function Types

There is another kind of composition that is at the heart of functional programming. It happens when you pass a function as an argument to another function. The outer function can then use this argument as a pluggable part of its own machinery. It lets you implement, for instance, a generic sorting algorithm that accepts an arbitrary comparison function.

If we model functions as arrows between objects, then what does it mean to have a function as an argument?

We need a way to objectify functions in order to define arrows that have an "object of arrows" as a source or as a target. A function that takes a function as an argument or returns a function is called a *higher-order* function. Higher-order functions are the work-horses of functional programming.

Elimination rule

The defining quality of a function is that it can be applied to an argument to produce the result. We have defined function application in terms of composition:

$$\begin{tikzcd}
1 \arrow[d, "x"'] \arrow[dr, "y"] & \\
a \arrow[r, "f"'] & b
\end{tikzcd}$$

Here f is represented as an arrow from a to b, but we would like to be able to replace f with an element of the object of arrows or, as mathematicians call it, the exponential object b^a; or as we call it in programming, a function type .

Given an element of b^a and an element of a, function application should produce an element of b. In other words, given a pair of elements:

$$f \colon 1 \to b^a$$

$$x \colon 1 \to a$$

it should produce an element:

$$y \colon 1 \to b$$

Keep in mind that, here, f denotes an element of b^a. Previously, it was an arrow from a to b.

We know that a pair of elements (f, x) is equivalent to an element of the product $b^a \quad a$. We can therefore define function application as a single arrow:

$$\varepsilon_{ab} \colon b^a \quad a \to b$$

This way y, the result of the application, is defined by this commuting diagram:

$$\begin{tikzcd}
1 \arrow[d, "(f{,}x)"'] \arrow[dr, "y"] & \\
b^a \quad a \arrow[r, "\varepsilon_{ab}"'] & b
\end{tikzcd}$$

Function application is the *elimination rule* for function type.

When somebody gives you an element of the function object, the only thing you can do with it is to apply it to an element of the argument type using ε.

Introduction rule

To complete the definition of the function object, we also need the introduction rule.

First, suppose that there is a way of constructing a function object b^a from some other object c. It means that there is an arrow

$$h \colon c \to b^a$$

We know that we can eliminate the result of h using ε_{ab}, but we have to first multiply it by a. So let's first multiply c by a and the use functoriality to map it to $b^a \quad a$.

Functoriality lets us apply a pair of arrows to a product to get another product. Here, the pair of arrows is (h, id_a) (we want to turn c into b^a, but we're not interested in modifying a)

$$c \quad a \xrightarrow{h \quad id_a} b^a \quad a$$

We can now follow this with function application to get to b

$$c \quad a \xrightarrow{h \quad id_a} b^a \quad a \xrightarrow{\varepsilon_{ab}} b$$

This composite arrow defines a mapping we'll call f:

$$f : c \quad a \to b$$

Here's the corresponding diagram

$$\begin{tikzcd}
c \quad a \arrow[d, dashed, "h \quad id_a"'] \arrow[dr, "f"] & \\
b^a \quad a \arrow[r, "\varepsilon"'] & b
\end{tikzcd}$$

This commuting diagram tells us that, given an h, we can construct an f; but we can also demand the converse: Every mapping out, $f : c \quad a \to b$, should uniquely define a mapping into the exponential, $h : c \to b^a$.

We can use this property, this one-to-one correspondence between two sets of arrows, to define the exponential object. This is the *introduction rule* for the function object b^a.

We've seen that product was defined using its mapping-in property. Function application, on the other hand, is defined as a *mapping out* of a product.

Currying

There are several ways of looking at this definition. One is to see it as an example of currying.

So far we've been only considering functions of one argument. This is not a real limitation, since we can always implement a function of two arguments as a (single-argument) function from a product. The f in the definition of the function object is such a function:

on the other hand is a function that returns a function:

Currying is the isomorphism between these two sets of arrows.

This isomorphism can be represented in Haskell by a pair of (higher-order) functions. Since, in Haskell, currying works for any types, these functions are written using type variables—they are *polymorphic*:

In other words, the h in the definition of the function object can be written as

$$h = curry\, f$$

Of course, written this way, the types of and correspond to function objects rather than arrows. This distinction is usually glossed over because there is a one-to-one correspondence between the *elements* of the exponential and the *arrows* that define them. This is easy to see when we replace the arbitrary object c with the terminal object. We get:

$$\begin{tikzcd}
1 \quad a \arrow[d, dashed, "h \; id_a"'] \arrow[dr, "f"] & \\
b^a \quad a \arrow[r, "\varepsilon_{ab}"'] & b
\end{tikzcd}$$

In this case, h is an element of the object b^a, and f is an arrow from $1 \quad a$ to b. But we know that $1 \quad a$ is isomorphic to a so, effectively, f is an arrow from a to b.

Therefore, from now on, we'll call an arrow an arrow →, without making much fuss about it. The correct incantation for this kind of phenomenon is to say that the category is self-enriched.

We can write ε_{ab} as a Haskell function :

but it's just a syntactic trick: function application is built into the language: means applied to . Other programming languages require the arguments to a function to be enclosed in parentheses, not so in Haskell.

Even though defining function application as a separate function may seem redundant, Haskell library does provide an infix operator for that purpose:

The trick, though, is that regular function application binds to the left, e.g., is the same as ; but the dollar sign binds to the right, so that

is the same as . In the first example, must be a function of (at least) two arguments; in the second, it could be a function of one argument.

In Haskell, currying is ubiquitous. A function of two arguments is almost always written as a function returning a function. Because the function arrow binds to the right, there is no need to parenthesize such types. For instance, the pair constructor has the signature:

You may think of if as a function of two arguments returning a pair, or a function of one argument returning a function of one argument, . This way it's okay to partially apply such a function, the result being another function. For instance, we can define:

Relation to lambda calculus

Another way of looking at the definition of the function object is to interpret c as the type of the environment in which f is defined. In that case it's customary to call the environment Γ. The arrow is interpreted as an expression that uses the variables defined in Γ.

Consider a simple example, the expression:

$$ax^2 + bx + c$$

You may think of it as being parameterized by a triple of real numbers (a, b, c) and a variable x, taken to be, let's say, a complex number. The triple is an element of a product $\mathbb{R}\ \ \mathbb{R}\ \ \mathbb{R}$. This product is the environment Γ for our expression.

The variable x is an element of $\mathbb{C}$. The expression is an arrow from the product $\Gamma\ \ \mathbb{C}$ to the result type (here, also $\mathbb{C}$)

$$f : \Gamma\ \ \mathbb{C} \to \mathbb{C}$$

This is a mapping-out from a product, so we can use it to construct a function object $\mathbb{C}^{\mathbb{C}}$ and define a mapping $h \colon \Gamma \to \mathbb{C}^{\mathbb{C}}$

$$\begin{tikzcd}
\Gamma\ \ \mathbb{C} \arrow[d, dashed, "h\ id_{\mathbb{C}}"'] \arrow[dr, "f"] & \\
\mathbb{C}^{\mathbb{C}}\ \ \mathbb{C} \arrow[r, "\varepsilon"'] & \mathbb{C}
\end{tikzcd}$$

This new mapping h can be seen as a constructor of the function object. The resulting function object represents all functions from $\mathbb{C}$ to $\mathbb{C}$ that have access to the environment Γ; that is, to the triple of parameters (a, b, c).

Corresponding to our original expression $ax^2 + bx + c$ there is a particular function in $\mathbb{C}^{\mathbb{C}}$ that we write as:

$$\lambda x.\, ax^2 + bx + c$$

or, in Haskell, with the backslash replacing λ,

The arrow $h \colon \Gamma \to \mathbb{C}^{\mathbb{C}}$ is uniquely determined by the arrow f. This mapping produces a function that we call $\lambda x.f$.

In general, the defining diagram for the function object becomes:

$$\begin{tikzcd}
\Gamma\ \ a \arrow[d, dashed, "h\ id_a"'] \arrow[dr, "f"] & \\
b^a\ \ a \arrow[r, "\varepsilon"'] & b
\end{tikzcd}$$

The environment Γ that provides free parameters for the expression f is a product of multiple objects representing the types of the parameters (in our example, it was $\mathbb{R}\ \ \mathbb{R}\ \ \mathbb{R}$).

An empty environment is represented by the terminal object 1, the unit of the product. In that case, f is just an arrow $a \to b$, and h simply picks an element from the function object b^a that corresponds to f.

It's important to keep in mind that, in general, a function object represents functions that depend on external parameters. Such functions are called *closures*. Closures are functions that capture values from their environment.

Here's our example translated to Haskell. Corresponding to f we have an expression:

If we use to approximate ℝ, our environment is a product . The type is parameterized by another type—here we used again:

The conversion from to is done by setting the imaginary part to zero, as in .

The corresponding arrow h takes the environment and produces a closure of the type :

Modus ponens

In logic, the function object corresponds to an implication. An arrow from the terminal object to the function object is the proof of that implication. Function application ε corresponds to what logicians call *modus ponens*: if you have a proof of the implication $A \Rightarrow B$ and a proof of A then this constitutes the proof of B.

6.1 Sum and Product Revisited

When functions gain the same status as elements of other types, we have the tools to directly translate diagrams into code.

Sum types

Let's start with the definition of the sum.

$$\begin{tikzcd}
a \arrow[rd, bend left, "\text{Left}"] \arrow[rdd, bend right, "f"'] & & b \arrow[ld, bend right, "\text{Right}"'] \arrow[ldd, bend left, "g"] \\
& a + b \arrow[d, dashed, "h"] & \\
& c &
\end{tikzcd}$$

We said that the pair of arrows (f, g) uniquely determines the mapping h out of the sum. We can write it concisely using a higher-order function:

where:

This function takes a pair of functions as an argument and it returns a function.

First, we pattern-match the pair to extract and . Then we construct a new function using a lambda. This lambda takes an argument of the type , which we call , and does the case analysis on it. If it was constructed using , we apply to its contents, otherwise we apply .

Note that the function we are returning is a closure. It captures and from its environment.

The function we have implemented closely follows the diagram, but it's not written in the usual Haskell style. Haskell programmers prefer to curry functions of multiple arguments. Also, if possible, they prefer to eliminate lambdas.

Here's the version of the same function taken from the Haskell standard library, where it goes under the name (lower-case) :

The other direction of the bijection, from h to the pair (f, g), also follows the arrows of the diagram.

Product types

Product types are dually defined by their mapping-in property.

$$\begin{tikzcd}
 & c \arrow[d, dashed, "h"] \arrow[ddl, bend right, "f"'] \arrow[ddr, bend left, "g"] & \\
 & a \quad b \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\
a & & b
\end{tikzcd}$$

Here's the direct Haskell reading of this diagram

And this is the stylized version written in Haskell style as an infix operator

The other direction of the bijection is given by:

which also closely follows the reading of the diagram.

Functoriality revisited

Both sum and product are functorial, which means that we can apply functions to their contents. We are ready to translate those diagrams into code.

This is the functoriality of the sum type:

$$\begin{tikzcd}
a \arrow[d, "f"] \arrow[dr, bend left, "\text{Left}"] & & b \arrow[dl, bend right, "\text{Right}"'] \arrow[d, "g"] \\
a^{\cdot\cdot} \arrow[dr, "\text{Left}"'] & a + b \arrow[d, dashed, "h"] & b^{\cdot\cdot} \arrow[dl, "\text{Right}"] \\
 & a^{\cdot\cdot} + b^{\cdot\cdot} &
\end{tikzcd}$$

Reading this diagram we can immediately write h using :

Or we could expand it and call it :

Similarly for the product type:

$$\begin{tikzcd}
 & a\ b \arrow[ld, "\text{fst}"'] \arrow[d, dashed, "h"] \arrow[rd, "\text{snd}"] & \\
a \arrow[d, "f"'] & a'\ b' \arrow[ld, "\text{fst}"] \arrow[rd, "\text{snd}"'] & b \arrow[d, "g"] \\
a' & & b'
\end{tikzcd}$$

h can be written as:

Or it could be expanded to

In both cases we call this higher-order function since, in Haskell, both the sum and the product are instances of a more general class called .

6.2 Functoriality of the Function Type

The function type, or the exponential, is also functorial, but with a twist. We are interested in a mapping from b^a to $b'^{a'}$, where the primed objects are related to the non-primed ones through some arrows—to be determined.

The exponential is defined by its mapping-in property, so if we're looking for

$$k \colon b^a \to b'^{a'}$$

we should draw the diagram that has k as a mapping into $b'^{a'}$. We get this diagram from the original definition by substituting b^a for c and primed objects for the non-primed ones:

$$\begin{tikzcd}
b^a\ a' \arrow[d, dashed, "k\ id_a"'] \arrow[rd, "g"] & \\
b'^{a'}\ a' \arrow[r, "\varepsilon"'] & b'
\end{tikzcd}$$

The question is: can we find an arrow g to complete this diagram?

$$g \colon b^a\ \ a' \to b'$$

If we find such a g, it will uniquely define our k.

The way to think about this problem is to consider how we would implement g. It takes the product $b^a \;\; a^{\cdot\cdot}$ as its argument. Think of it as a pair: an element of the function object from a to b and an element of $a^{\cdot\cdot}$. The only thing we can do with the function object is to apply it to something. But b^a requires an argument of type a, and all we have at our disposal is $a^{\cdot\cdot}$. We can't do anything unless somebody gives us an arrow $a^{\cdot\cdot} \to a$. This arrow applied to $a^{\cdot\cdot}$ will generate the argument for b^a. However, the result of the application is of type b, and g is supposed to produce a $b^{\cdot\cdot}$. Again, we'll need an arrow $b \to b^{\cdot\cdot}$ to complete our assignment.

This may sound complicated, but the bottom line is that we require two arrows between the primed and non-primed objects. The twist is that the first arrow goes from $a^{\cdot\cdot}$ to a, which feels backward from the usual functoriality considerations. In order to map b^a to $b^{\cdot\cdot a^{\cdot\cdot}}$ we need a pair of arrows:

$$f : a^{\cdot\cdot} \to a$$
$$g : b \to b^{\cdot\cdot}$$

This is somewhat easier to explain in Haskell. Our goal is to implement a function , given a function .

This new function takes an argument of the type so, before we can pass it to , we need to convert to . That's why we need a function .

Since produces a , and we want to return a , we need another function . All this fits nicely into one higher-order function:

Similar to being an interface to the typeclass , is a member of the typeclass .

6.3 Bicartesian Closed Categories

A category in which both the product and the exponential are defined for any pair of objects, and which has a terminal object, is called *cartesian closed*. The idea is that hom-sets are not something alien to the category in question: the category is "closed" under the operation of forming hom-sets.

If the category also has sums (coproducts) and the initial object, it's called *bicartesian closed*.

This is the minimum structure for modeling programming languages.

Data types constructed using these operations are called *algebraic data types*. We have addition, multiplication, and exponentiation (but not subtraction or division) of types; with all the familiar laws we know from high-school algebra. They are satisfied up to isomorphism. There is one more algebraic law that we haven't discussed yet.

Distributivity

Multiplication of numbers distributes over addition. Should we expect the same in a bicartesian closed category?

$$b \quad a + c \quad a \text{ ö } (b+c) \quad a$$

The left to right mapping is easy to construct, since it's simultaneously a mapping out of a sum and a mapping into a product. We can construct it by gradually decomposing it into simpler mappings. In Haskell, this means implementing a function

A mapping out of the sum on the left is given by a pair of arrows:

$$f\colon b \quad a \to (b+c) \quad a$$
$$g\colon c \quad a \to (b+c) \quad a$$

$$\begin{tikzcd}
b \quad a \arrow[dr, bend left, "\text{Left}"] \arrow[ddr, bend right, "f"'] & & c \quad a \arrow[dl, bend right, "\text{Right}"'] \arrow[ddl, bend left, "g"] \\
& b \quad a + c \quad a \arrow[d, dashed, "\text{dist}"] & \\
& (b+c) \quad a &
\end{tikzcd}$$

We write it in Haskell as:

The clause is used to introduce the definitions of sub-functions.

Now we need to implement f and g. They are mappings into the product, so each of them is equivalent to a pair of arrows. For instance, the first one is given by the pair:

$$f^{¨}\colon b \quad a \to (b+c)$$
$$f^{¨¨}\colon b \quad a \to a$$

$$\begin{tikzcd}
& b \quad a \arrow[ddl, bend right, "f^{¨}"'] \arrow[ddr, bend left, "f^{¨¨}"] \arrow[d, dashed, "f"] & \\
& (b+c) \quad a \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\
b+c & & a
\end{tikzcd}$$

In Haskell:

The first arrow can be implemented by projecting the first component *b* and then using Left to construct the sum. The second is just the projection snd:

$$
\begin{aligned}
f^{\text{¨}} &= \text{Left}\circ\text{fst} \\
f^{\text{¨¨}} &= \text{snd}
\end{aligned}
$$

Similarly, we decompose *g* into a pair $g^{\text{¨}}$ and $g^{\text{¨¨}}$:

$$\begin{tikzcd}
& c \quad a \arrow[d, dashed, "g"] \arrow[ddl, bend right, "g^{\text{¨}}"'] \arrow[ddr, bend left, "g^{\text{¨¨}}"] & \\
& b + c \quad a \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\
(b + c) & & a
\end{tikzcd}$$

Combining all these together, we get:

These are the type signatures of the helper functions:

They can also be inlined to produce this terse form:

This style of programming is called *point free* because it omits the arguments (points). For readability reasons, Haskell programmers prefer a more explicit style. The above function would normally be implemented as:

Notice that we have only used the definitions of sums and products. The other direction of the isomorphism requires the use of the exponential, so it's only valid in a bicartesian *closed* category. This is not immediately clear from the straightforward Haskell implementation:

but that's because currying is implicit in Haskell.

Here's the point-free version of this function:

This may not be the most readable implementation, but it underscores the fact that we need the exponential: we use both and to implement the mapping.

We'll come back to this identity later, when we are equipped with more powerful tools: adjunctions.

Exercise 6.3.1. *Show that:*

$$2 \quad a \text{ ö } a + a$$

where 2 *is the Boolean type. Do the proof diagrammatically first, and then implement two Haskell functions witnessing the isomorphism.*

Chapter 7

Recursion

When you step between two mirrors, you see your reflection, the reflection of your reflection, the reflection of that reflection, and so on. Each reflection is defined in terms of the previous reflection, but together they produce infinity.

Recursion is a decomposition pattern that splits a single task into many steps, the number of which is potentially unbounded.

Recursion is based on suspension of disbelief. You are faced with a task that may take arbitrarily many steps. You tentatively assume that you know how to solve it. Then you ask yourself the question: "How would I make the last step if I had the solution to everything *but* the last step?"

7.1 Natural Numbers

An object of natural numbers N does not contain numbers. Objects have no internal structure. Structure is defined by arrows.

We can use an arrow from the terminal object to define one special element. By convention, we'll call this arrow Z for "zero."

$$Z \colon 1 \to N$$

But we have to be able to define infinitely many arrows to account for the fact that, for every natural number, there is another number that is one larger than it.

We can formalize this statement by saying: Suppose that we know how to create a natural number $n \colon 1 \to N$. How do we make the next step, the step that will point us to the next number—its successor?

This next step doesn't have to be any more complex than just post-composing *n* with an arrow that loops back from N to N. This arrow should not be the identity, because we want the successor of a number to be different from that number. But a single such arrow, which we'll call S for "successor" will suffice.

The element corresponding to the successor of n is given by the composition:

$$1 \xrightarrow{n} N \xrightarrow{S} N$$

(We sometimes draw the same object multiple times in a single diagram, if we want to straighten the looping arrows.)

In particular, we can define One as the successor of Z:

$$\begin{tikzcd} 1 \arrow[r, "Z"] \arrow[rr, bend left, "One"] & N \arrow[r, "S"] & N \end{tikzcd}$$

and Two as the successor of the successor of Z

$$\begin{tikzcd} 1 \arrow[r, "Z"] \arrow[rrr, bend left, "Two"] & N \arrow[r, "S"] & N \arrow[r, "S"] & N \end{tikzcd}$$

and so on.

Introduction Rules

The two arrows, Z and S, serve as the introduction rules for the natural number object N. The twist is that one of them is recursive: S uses N as its source as well as its target.

$$\begin{tikzcd} 1 \arrow[r, "Z"] & N \arrow[loop above, "S"] \end{tikzcd}$$

The two introduction rules translate directly to Haskell

```
```

They can be used to define arbitrary natural numbers; for instance:

```
```

This definition of natural number type is not very useful in practice. However, it's often used in defining type-level naturals, where each number is its own type.

You may encounter this construction under the name of *Peano arithmetic*.

Elimination Rules

The fact that the introduction rules are recursive complicates the matters slightly when it comes to defining elimination rules. We will follow the pattern from previous chapters of first assuming that we are given a mapping out of N:

$$h \colon N \to a$$

and see what we can deduce from there.

Previously, we were able to decompose such an h into simpler mappings (pairs of mappings for sum and product; a mapping out of a product for the exponential).

The introduction rules for N look similar to those for the sum (it's either Z or the successor), so we would expect that h could be split into two arrows. And, indeed, we can easily get the first one by composing $h \circ Z$. This is an arrow that picks an element of a. We call it init:

$$\text{init} \colon 1 \to a$$

But there is no obvious way to find the second one.

To see that, let's expand the definition of N:

$$\begin{tikzcd}
1 \arrow[r, "Z"] & N \arrow[r, "S"] & N \arrow[r, "S"] & N & \dots
\end{tikzcd}$$

and plug h and init into it:

$$\begin{tikzcd}
1 \arrow[r, "Z"] \arrow[dr, "\text{init}"'] & N \arrow[r, "S"] \arrow[d, dashed, "h"] & N \arrow[r, "S"] \arrow[d, dashed, "h"] & N \arrow[d, dashed, "h"] & \dots \\
& a & a & a &
\end{tikzcd}$$

The intuition is that an arrow from N to a represents a *sequence* a_n of elements of a. The zeroth element is given by

$$a_0 = \text{init}$$

The next element is

$$a_1 = h \circ S \circ Z$$

followed by

$$a_2 = h \circ S \circ S \circ Z$$

and so on.

We have thus replaced one arrow h with infinitely many arrows a_n. Granted, the new arrows are simpler, since they represent elements of a, but there are infinitely many of them.

The problem is that, no matter how you look at it, an arbitrary mapping out of N contains infinite amount of information.

We have to drastically simplify the problem. Since we used a single arrow S to generate all natural numbers, we can try to use a single arrow $a \to a$ to generate all the elements a_n. We'll call this arrow step:

$$\begin{tikzcd}
1 \arrow[r, "Z"] \arrow[dr, "\text{init}"'] & N \arrow[r, "S"] \arrow[d, dashed, "h"] & N \arrow[d, dashed, "h"] \\
& a \arrow[r, "\text{step}"] & a
\end{tikzcd}$$

The mappings out of N that are generated by such pairs, init and step, are called *recursive*. Not all mappings out of N are recursive. In fact very few are; but recursive mappings are enough to define the object of natural numbers.

We use the above diagram as the elimination rule. We decree that every recursive mapping h out of N is in one-to-one correspondence with a pair init and step.

This means that the *evaluation rule* (extracting $(\mathrm{init}, \mathrm{step})$ for a given h) cannot be formulated for an arbitrary arrow $h \colon N \to a$, only for those arrows that have been previously recursively defined using a pair $(\mathrm{init}, \mathrm{step})$.

The arrow init can be always recovered by composing $h \circ Z$. The arrow step is a solution to the equation:

$$\mathrm{step} \circ h = h \circ S$$

If h was defined using some init and step, then this equation obviously has a solution.

The important part is that we demand that this solution be *unique*.

Intuitively, the pair init and step generate the sequence of elements a_0, a_1, a_2, ... If two arrows h and $h^{\ddot{}}$ are given by the same pair $(\mathrm{init}, \mathrm{step})$, it means that the sequences they generate are the same.

So if h were somehow different from $h^{\ddot{}}$, it would mean that N contains more than just the sequence of elements Z, SZ, $S(SZ)$, ... For instance, if we added $*1$ to N (that is, made Z somebody's successor), we could have h and $h^{\ddot{}}$ differ at $*1$ and yet be generated by the same init and step. Uniqueness means there are no natural number before, after, or in between the numbers generated by Z and S.

The elimination rule we've discussed here corresponds to *primitive recursion*. We'll see a more advanced version of this rule, corresponding to the induction principle, in the chapter on dependent types.

In Programming

The elimination rule can be implemented as a recursive function in Haskell:

This single function, which is called a *recursor*, is enough to implement all recursive functions of natural numbers. For instance, this is how we could implement addition:

This function takes as an argument and produces a function (a closure) that takes another number and adds to it.

In practice, programmers prefer to implement recursion directly—an approach that is equivalent to inlining the recursor . The following implementation is arguably easier to understand:

It can be read as: If is zero then the result is . Otherwise, if is a successor of some , then the result is the successor of . This is exactly the same as saying that and .

In imperative languages recursion is often replaced by iteration. Conceptually, iteration seems to be easier to understand, as it corresponds to sequential decomposition. The steps in the sequence usually follow some natural order. This is in contrast with recursive decomposition, where we assume that we have done all the work up to the n'th step, and we combine that result with the next consecutive step.

On the other hand, recursion is more natural when processing recursively defined data structures, such as lists or trees.

The two approaches are equivalent, and compilers often convert recursive functions to loops in what is called *tail recursion optimization*.

Exercise 7.1.1. *Implement a curried version of addition as a mapping out of N into the function object N^N. Hint: use these types in the recursor:*

7.2 Lists

A list of things is either empty or a thing followed by a list of things.

This recursive definition translates into two introduction rules for the type L_a, the list of a:

$$\text{Nil} \colon 1 \to L_a$$

$$\text{Cons} \colon a \quad L_a \to L_a$$

The Nil element describes an empty list, and Cons constructs a list from a head and a tail.

The following diagram depicts the relationship between projections and list constructors. The projections extract the head and the tail of the list that was constructed using Cons.

$$\begin{tikzcd}
& a \quad L_a \arrow[dl, "fst"'] \arrow[dr, "snd"] \arrow[dr, bend left, "\text{Cons}"] & & 1 \arrow[dl, "\text{Nil}"] \\
a & & L_a &
\end{tikzcd}$$

This description can be immediately translated to Haskell:

Elimination Rule

Suppose that we have a mapping out from a list of a to some arbitrary type c:

$$h \colon L_a \to c$$

This is how we would plug it into our definition of the list:

$$\begin{tikzcd}
1 \arrow[r, "\text{Nil}"] \arrow[dr, "\text{init}"'] & L_a \arrow[d, dashed, "h"] & a\ \ L_a \arrow[l, "\text{Cons}"'] \arrow[d, dashed, "id_a\ \ h"] \\
& c & a\ \ c \arrow[l, "\text{step}"]
\end{tikzcd}$$

We used the functoriality of the product to apply the pair (id_a, h) to the product $a\ \ L_a$.

Similar to the natural number object, we can try to define two arrows, $\text{init} = h \circ \text{Nil}$ and step. The arrow step is a solution to:

$$\text{step} \circ (id_a\ \ h) = h \circ \text{Cons}$$

Again, not every h can be reduced to such a pair of arrows.

However, given init and step, we can define an h. Such a function is called a *fold*, or a list catamorphism.

This is the list recursor in Haskell:

Given and , it produces a mapping out of a list.

A list is such a basic data type that Haskell has a built-in syntax for it. The type is written as . The constructor is an empty pair of square brackets, , and the constructor is an infix colon .

We can pattern-match on these constructors. A generic mapping out of a list has the form:

Corresponding to the list recursor, , here's the type signature of the function (fold *right*), which you can find in the standard library:

Here's one possible implementation:

As an example, we can use to calculate the sum of the elements of a list of natural numbers:

Exercise 7.2.1. *Consider what happens when you replace a in the definition of a list with the terminal object. Hint: What is base-one encoding of natural numbers?*

Exercise 7.2.2. *How many mappings $h \colon L_a \to 1 + a$ are there? Can we get all of them using a list recursor? How about Haskell functions of the signature:*

Exercise 7.2.3. *Implement a function that extracts the third element from a list, if the list is long enough. Hint: Use for the result type.*

7.3 Functoriality

Functoriality means, roughly, the ability to transform the "contents" of a data structure. The contents of a list L_a is of the type a. Given an arrow $f\colon a \to b$, we need to define a mapping of lists $h\colon L_a \to L_b$.

Lists are defined by the mapping out property, so let's replace the target c of the elimination rule by L_b. We get:

$$\begin{tikzcd}
1 \arrow[r, "\text{Nil}_a"] \arrow[dr, "\text{init}"'] & L_a \arrow[d, dashed, "h"] & a \quad L_a \arrow[l, "\text{Cons}_a"'] \arrow[d, dashed, "id_a \ h"] \\
 & L_b & a \quad L_b \arrow[l, "\text{step}"]
\end{tikzcd}$$

Since we are dealing with two different lists here, we have to distinguish between their constructors. For instance, we have:

$$\text{Nil}_a\colon 1 \to L_a$$
$$\text{Nil}_b\colon 1 \to L_b$$

and similarly for Cons.

The only candidate for init is Nil_b, which is to say that h acting on an empty list of a's produces an empty list of b's:

$$h \circ \text{Nil}_a = \text{Nil}_b$$

What remains is to define the arrow:

$$\text{step}\colon a \quad L_b \to L_b$$

We can guess:

$$\text{step} = \text{Cons}_b \circ (f \quad id_{L_b})$$

This corresponds to the Haskell function:

or, using the built-in list syntax and inlining the recursor,

You might wonder what prevents us from choosing $\text{step} = \text{snd}$, resulting in:

We'll see, in the next chapter, why this is a bad choice. (Hint: What happens when we apply to id?)

Chapter 8

Functors

8.1 Categories

So far we've only seen only one category—that of types and functions. So let's quickly gather the essential info about a category.

A category is a collection of objects and arrows that go between them. Every pair of composable arrows can be composed. The composition is associative, and there is an identity arrow looping back on every object.

The fact that types and functions form a category can be expressed in Haskell by defining composition as:

The composition of two functions after is a new function that first applies to its argument and then applies to the result.

The identity is a polymorphic "do nothing" function:

You can easily convince yourself that such composition is associative, and composing with does nothing to a function.

Based on the definition of a category, we can come up with all kinds of weird categories. For instance, there is a category that has no objects and no arrows. It satisfies all the condition of a category vacuously. There's another one that contains a single object and a single arrow (can you guess what arrow it is?). There's one with two unconnected objects, and one where the two objects are connected by a single arrow (plus two identity arrows), and so on. These are example of what I call stick-figure categories—categories with a small handful of objects and arrows.

Category of sets

We can also strip a category of all arrows (except for the identity arrows). Such a bare-object category is called a *discrete* category or a set[1]. Since we associate arrows with structure, a set is a category with no structure.

Sets form their own category called $\mathbf{Set}$[2]. The objects in that category are sets, and the arrows are functions between sets. Such functions are defined as special kind of relations, which themselves are defined as sets of pairs.

To the lowest approximation, we can model programming in the category of sets. We often think of types as sets of values, and functions as set-theoretical functions. There's nothing wrong with that. In fact all of categorical constructions we've described so far have their set-theoretical roots. The categorical product is a generalization of the cartesian product of sets, the sum is the disjoint union, and so on.

What category theory offers is more precision: the fine distinction between the structure that is absolutely necessary, and the superfluous details.

A set-theoretical function, for instance, doesn't fit the definition of a function we work with as programmers. Our functions must have underlying algorithms because they have to be computable by some physical systems, be it computers or a human brains.

Opposite categories

In programming, the focus is on the category of types and functions, but we can use this category as a starting point to construct other categories.

One such category is called the *opposite* category. This is the category in which all the original arrows are inverted: what is called the source of an arrow in the original category is now called its target, and vice versa.

The opposite of a category C is called C^{op}. We've had a glimpse of this category when we discussed duality. The objects of C^{op} are the same as those of C.

Whenever there is an arrow $f : a \to b$ in C, there is a corresponding arrow $f^{op} : b \to a$ in C^{op}.

The composition $g^{op} \circ f^{op}$ of two such arrows $f^{op} : a \to b$ and $g^{op} : b \to c$ is given by the arrow $(f \circ g)^{op}$ (notice the reversed order).

The terminal object in C is the initial object in C^{op}, the product in C is the sum in C^{op}, and so on.

[1] Ignoring "size" issues.

[2] Again, ignoring "size" issues, in particular the non-existence of the set of all sets.

Product categories

Given two categories $\mathcal{C}$ and $\mathcal{D}$, we can construct a product category $\mathcal{C} \times \mathcal{D}$. The objects in this category are pairs of objects $\langle c, d \rangle$, and the arrows are pairs of arrows.

If we have an arrow $f \colon c \to c'$ in $\mathcal{C}$ and an arrow $g \colon d \to d'$ in $\mathcal{D}$ then there is a corresponding arrow $\langle f, g \rangle$ in $\mathcal{C} \times \mathcal{D}$. This arrow goes from $\langle c, d \rangle$ to $\langle c', d' \rangle$, both being objects in $\mathcal{C} \times \mathcal{D}$. Two such arrows can be composed if their components are composable in, respectively, $\mathcal{C}$ and $\mathcal{D}$. An identity arrow is a pair of identity arrows.

The two product categories we're most interested in are $\mathcal{C} \times \mathcal{C}$ and $\mathcal{C}^{op} \times \mathcal{C}$, where $\mathcal{C}$ is our familiar category of types and functions.

In both of these categories, objects are pairs of objects from $\mathcal{C}$. In the first category, $\mathcal{C} \times \mathcal{C}$, a morphism from $\langle a, b \rangle$ to $\langle a', b' \rangle$ is a pair $\langle f \colon a \to a', g \colon b \to b' \rangle$. In the second category, $\mathcal{C}^{op} \times \mathcal{C}$, a morphism is a pair $\langle f \colon a' \to a, g \colon b \to b' \rangle$, in which the first arrow goes in the opposite direction.

Slice categories

In a neatly organized universe, objects are always objects and arrows are always arrows. Except that sometimes sets of arrows can be thought of as objects. But slice categories break this neat separation: they turn individual arrows into objects.

A slice category $\mathcal{C}/c$ describes how a particular object c is seen from the perspective of its category $\mathcal{C}$. It's the totality of arrows pointing at c. But to specify an arrow we need to specify both of its ends. Since one of these ends is fixed to be c, we only have to specify the other.

An object in the *slice category* $\mathcal{C}/c$ (also known as an over-category) is a pair $\langle e, p \rangle$, with $p \colon e \to c$.

An arrow between two objects $\langle e, p \rangle$ and $\langle e', p' \rangle$ is an arrow $f \colon e \to e'$ of $\mathcal{C}$, which makes the following triangle commute:

$$\begin{tikzcd}
e \arrow[rr, "f"] \arrow[dr, "p"'] & & e' \arrow[dl, "p'"] \\
& c &
\end{tikzcd}$$

Coslice categories

There is a dual notion of a coslice category $c/\mathcal{C}$, also known as an under-category. It's a category of arrows emanating from a fixed object c. Objects in this category are pairs $\langle a, i \colon c \to a \rangle$.

Morphisms in c/C are arrows that make the relevant triangles commute.

$$\begin{tikzcd}
& c \arrow[dl, "i"'] \arrow[dr, "j"] & \\
a \arrow[rr, "f"'] & & b
\end{tikzcd}$$

In particular, if the category C has a terminal object 1, then the coslice $1/C$ has, as objects, global elements of all the objects of C.

Morphisms of $1/C$ that correspond to arrows $f : a \to b$ map the set of global elements of a to the set of global elements of b.

$$\begin{tikzcd}
& 1 \arrow[dl, "x"'] \arrow[dr, "y"] & \\
a \arrow[rr, "f"'] & & b
\end{tikzcd}$$

In particular, the construction of a coslice category from the category of types and functions justifies our intuition of types as sets of values, with values represented by global elements of types.

8.2 Functors

We've seen examples of functoriality when discussing algebraic data types. The idea is that such a data type "remembers" the way it was created, and we can manipulate this memory by applying an arrow to its "contents."

In some cases this intuition is very convincing: we think of a product type as a pair that "contains" its ingredients. After all, we can retrieve them using projections.

This is less obvious in the case of function objects. You can visualize a function object as secretly storing all possible results and using the function argument to index into them. A function from is obviously equivalent to a pair of values, one for and one for . It's a known programming trick to implement some functions as lookup tables. It's called *memoization*.

Even though it's not practical to memoize functions that take, say, natural numbers as arguments; we can still conceptualize them as (infinite, or even uncountable) lookup tables.

If you can think of a data type as a container of values, it makes sense to apply a function to transform all these values, and create a transformed container. When this is possible, we say that the data type is *functorial*.

Again, function types require some more suspension of disbelief. You visualize a function object as a lookup table, keyed by some type. If you want to use another, related type as your key, you need a function that translates the new key to the original key. This is why functoriality of the function object has one of the arrows reversed:

You are applying the transformation to a function that has a “receptor” that responds to values of type , and you want to use it to process input of type . This is only possible if you have a converter from to , namely .

The idea of a data type “containing” values of another type can be also expressed by saying that one data type is paremeterized by another. For instance, the type is parameterized by the type .

In other words, maps the type to the type . by itself, without the argument, is called a *type constructor*.

Functors between categories

In category theory, a type constructor is modeled as a mapping of objects to objects. It’s a function on objects. This is not to be confused with arrows between objects, which are part of the structure of the category.

In fact, it’s easier to imagine a mapping *between* categories. Every object in the source category is mapped to an object in the target category. If a is an object in $\mathcal{C}$, there is a corresponding object Fa in $\mathcal{D}$.

A functorial mapping, or a *functor*, not only maps objects but also arrows between them. Every arrow

$$f : a \to b$$

in the first category has a corresponding arrow in the second category:

$$Ff : Fa \to Fb$$

$$\begin{tikzcd}
a \arrow[r, dashed] \arrow[d, "f"] & Fa \arrow[d, "Ff"] \\
b \arrow[r, dashed] & Fb
\end{tikzcd}$$

We use the same letter, here F, to name both, the mapping of objects and the mapping of arrows.

If categories distill the essence of *structure*, then functors are mappings that preserve this structure. Objects that are related in the source category are related in the target category.

The structure of a category is defined by arrows and their composition. Therefore a functor must preserve composition. What is composed in one category:

$$h = g \circ f$$

should remain composed in the second category:

$$Fh = F(g \circ f) = Fg \circ Ff$$

We can either compose two arrows in $\mathcal{C}$ and map the composite to $\mathcal{D}$, or we can map individual arrows and then compose them in $\mathcal{D}$. We demand that the result be the same.

$$\begin{tikzcd}
a \arrow[r, dashed] \arrow[d, "f"] \arrow[dd, bend right=60, "g\circ f"'] & Fa \arrow[d, "Ff"] \arrow[dd, bend right=60, "F(g\circ f)"'] \arrow[dd, bend left=60, "Fg\circ Ff"] \\
b \arrow[d, "g"] & Fb \arrow[d, "Fg"] \\
c \arrow[r, dashed] & Fc
\end{tikzcd}$$

Finally, a functor must preserve identity arrows:

$$F\,id_a = id_{Fa}$$

$$\begin{tikzcd}
a \arrow[loop above, "id_a"] \arrow[r, dashed] & Fa \arrow[loop above, "F\,id_a"] \arrow[loop above, distance=4em, "id_{Fa}"]
\end{tikzcd}$$

These conditions taken together define what it means for a functor to preserve the structure of a category.

It's also important to realize what conditions are *not* part of the definition. For instance, a functor is allowed to map multiple objects into the same object. It can also map multiple arrows into the same arrow, as long as the endpoints match.

In the extreme, any category can be mapped to a singleton category with one object and one arrow.

Also, not all object or arrows in the target category must be covered by a functor. In the extreme, we can have a functor from the singleton category to any (non-empty) category. Such a functor picks a single object together with its identity arrow.

A *constant functor* Δ_c is an example of a functor that maps all objects from the source category to a single object c in the target category, and all arrows from the source category to a single identity arrow id_c.

In category theory, functors are often used to create models of one category inside another. The fact that they can merge multiple objects and arrows into one means that they produce simplified views of the source category. They "abstract" some aspects of the source category.

The fact that they may only cover parts of the target category means that the models are embedded in a larger environment.

Functors from some minimalistic, stick-figure, categories can be used to define patterns in larger categories.

Exercise 8.2.1. *Describe a functor whose source is the "walking arrow" category. It's a stick-figure category with two objects and a single arrow between them (plus the mandatory identity arrows).*

$$\begin{tikzcd}
a \arrow[loop above, "id_a"] \arrow[r, "f"] & b \arrow[loop above, "id_b"]
\end{tikzcd}$$

Exercise 8.2.2. *The "walking iso" category is just like the "walking arrow" category, plus one more arrow going back from b to a. Show that a functor from this category always picks an isomorphism in the target category.*

8.3 Functors in Programming

Endofunctors are the class of functors that are the easiest to express in a programming language. These are functors that map a category (here, the category of types and functions) to itself.

Endofunctors

The first part of the endofunctor is the mapping of types to types. This is done using type constructors, which are type-level functions.

The list type constructor, , maps an arbitrary type to the type .

The type constructor maps to .

The second part of an endofunctor is the mapping of arrows. Given a function , we want to be able to define a function , or . This is the "functoriality" property of these data types that we have discussed before. Functoriality lets us *lift* an arbitrary function to a function between transformed types.

Functoriality can be expressed in Haskell using a *typeclass*. In this case, the typeclass is parameterized by a type constructor (in Haskell we use lower case names for type-constructor variables). We say that is a if there is a corresponding mapping of functions called :

The compiler knows that is a type constructor because it's applied to types, as in and .

To prove to the compiler that a particular type constructor is a , we have to provide the implementation of for it. This is done by defining an *instance* of the typeclass . For example:

A functor must also satisfy some laws: it must preserve composition and identity. These laws cannot be expressed in Haskell, but should be checked by the programmer. We have previously seen a definition of that didn't satisfy the identity laws, yet it would be accepted by the compiler. It would define an "unlawful" instance of for the list type constructor .

Exercise 8.3.1. *Show that is a functor*

There are some elementary functors that might seem trivial, but they serve as building blocks for other functors.

We have the identity endofunctor that maps all objects to themselves, and all arrows to themselves.

Exercise 8.3.2. *Show that is a . Hint: implement the instance for it.*

We also have a constant functor Δ_c that maps all objects to a single object c, and all arrows to the identity arrow on this object. In Haskell, it's a family of functors parameterized by the target object :

This type constructor ignores its second argument.

Exercise 8.3.3. *Show that is a . Hint: The type constructor takes two arguments, but in the instance it's partially applied to the first argument. It is functorial in the second argument.*

Bifunctors

We have also seen data constructors that take two types as arguments: the product and the sum. They were functorial as well, but instead of lifting a single function, they lifted a pair of functions. In category theory, we would define these as functors from the product category $\mathcal{C}$ $\mathcal{C}$ to $\mathcal{C}$.

Such functors map a pair of objects to an object, and a pair of arrows to an arrow.

In Haskell, we treat such functors as members of a separate class called .

Again, the compiler deduces that is a two-argument type constructor because it sees it applied to two types, e.g., .

To prove to the compiler that a particular type constructor is a , we define an instance. For example, bifunctoriality of a pair can be defined as:

Exercise 8.3.4. *Show that is a bifunctor.*

Contravariant functors

Functors from the opposite category C^{op} are called *contravariant*. They have the property of lifting arrows that go in the opposite direction. Regular functors are sometimes called *covariant*.

In Haskell, contravariant functors form the typeclass :

It's often convenient to think of functors in terms of producers and consumers. In this analogy, a (covariant) functor is a producer. You can turn a producer of 's into a producer of 's by applying (using) a function . Conversely, to turn a consumer of 's to a consumer of 's you need a function going in the opposite direction, .

Example: A predicate is a function returning or :

It's easy to see that it's a contravariant functor:

In practice, the only non-trivial examples of contravariant functors are variations on the theme of function objects.

One way to tell if a given function type is covariant or contravariant in one of the type arguments is by assigning polarities to the types used in its definition. We say that the return type of a function is in a *positive* position, so it's covariant; and the argument type is in the *negative* position, so it's contravariant. But if you put the whole function object in the negative position of another function, then its polarities get reversed.

Consider this data type:

It has in a double-negative, therefore a positive position. This is why it's a covariant . It acts as a producer of 's:

Notice that parentheses are important here. A similar function has in a *negative* position. That's because it's a function of returning a function . Equivalently, you may uncurry it to get a function that takes a pair: . Either way, ends up in the negative position.

Profunctors

We've seen before that the function type is functorial. It lifts two functions at a time, just like , except that one of the functions goes in the opposite direction.

In category theory this corresponds to a functor from a product of two categories, one of them being the opposite category: it's a functor from C^{op} C. Functors from C^{op} C to Set are called *profunctors*.

In Haskell, profunctors form a typeclass:

You may think of a profunctor as a type that's simultaneously a producer and a consumer. It consumes one type and produces another.

The function type, which can be written as an infix operator , is an instance of

This is in accordance with our intuition that a function consumes arguments of the type and produces results of the type .

In programming, all non-trivial profunctors are variations on the function type.

8.4 The Hom-Functor

Arrows between any two objects form a set. This set is called a hom-set and is usually written using the name of the category followed by the names of the objects:

$$\mathcal{C}(a, b)$$

We can interpret the hom-set $\mathcal{C}(a, b)$ as all the ways b can be observed from a.

Another way of looking at hom-sets is to say that they define a mapping that assigns a set $\mathcal{C}(a, b)$ to every pair of objects. Sets themselves are objects in the category $\mathbf{Set}$. So we have a mapping between categories.

This mapping is functorial. To see that, let's consider what happens when we transform the two objects a and b. We are interested in a transformation that would map the set $\mathcal{C}(a, b)$ to the set $\mathcal{C}(a', b')$. Arrows in $\mathbf{Set}$ are regular functions, so it's enough to define their action on individual elements of a set.

An element of $\mathcal{C}(a, b)$ is an arrow $h \colon a \to b$ and an element of $\mathcal{C}(a', b')$ is an arrow $h' \colon a' \to b'$. We know how to transform one into another: we need to pre-compose h with an arrow $g' \colon a' \to a$ and post-compose it with an arrow $g \colon b \to b'$.

In other words, the mapping that takes a pair $\langle a, b \rangle$ to the set $\mathcal{C}(a, b)$ is a *profunctor*:

$$\mathcal{C}^{op} \quad \mathcal{C} \to \mathbf{Set}$$

Frequently, we are interested in varying only one of the objects, keeping the other fixed. When we fix the source object and vary the target, the result is a functor that is written as:

$$\mathcal{C}(a, *) \colon \mathcal{C} \to \mathbf{Set}$$

The action of this functor on an arrow $g \colon b \to b'$ is written as:

$$\mathcal{C}(a, g) \colon \mathcal{C}(a, b) \to \mathcal{C}(a, b')$$

and is given by post-composition:

$$C(a, g) = (g \circ *)$$

Varying b means switching focus from one object to another, so the complete functor $C(a, *)$ combines all the arrows emanating from a into a coherent view of the category from the perspective of a. It is "the world according to a."

Conversely, when we fix the target and vary the source of the hom-functor, we get a contravariant functor:

$$C(*, b) \colon C^{op} \to \mathrm{Set}$$

whose action on an arrow $g^{\cdot\cdot} : a^{\cdot\cdot} \to a$ is written as:

$$C(g^{\cdot\cdot}, b) \colon C(a, b) \to C(a^{\cdot\cdot}, b)$$

and is given by pre-composition:

$$C(g^{\cdot\cdot}, b) = (* \circ g^{\cdot\cdot})$$

The functor $C(*, b)$ organizes all the arrows pointing at b into one coherent view. It is the picture of b "as it's seen by the world."

We can now reformulate the results from the chapter on isomorphisms. If two objects a and b are isomorphic, then their hom-sets are also isomorphic. In particular:

$$C(a, x) \text{ ö } C(b, x)$$

and

$$C(x, a) \text{ ö } C(x, b)$$

We'll discuss naturality conditions in the next chapter.

Another way of looking at the hom-functor $C(a, *)$ is as an oracle that provides answers to the question: "Is a connected to me?" If the set $C(a, x)$ is empty, the answer is negative: "a is not connected to x." Otherwise, every element of the set $C(a, x)$ is a proof that such connection exists.

Conversely, the contravariant functor $C(*, a)$ answers the question: "Am I connected to a?"

Taken together, the profunctor $C(x, y)$ establishes a *proof-relevant* relation between objects. Every element of the set $C(x, y)$ is a proof that x is connected to y. If the set is empty, the two objects are unrelated.

8.5 Functor Composition

Just like we can compose functions, we can compose functors. Two functors are composable if the target category of one is the source category of the other.

On objects, functor composition of G after F first applies F to an object, then applies G to the result; and similarly on arrows.

Obviously, you can only compose composable functors. However all *endofunctors* are composable, since their target category is the same as the source category.

In Haskell, a functor is a parameterized data type, so the composition of two functors is again a parameterized data type. On objects, we define:

The compiler figures out that and must be type constructors because they are applied to types: is applied to the type parameter , and is applied to the resulting type.

Alternatively, you can tell the compiler that the first two arguments to are type constructors. You do this by providing a *kind signature*, which requires a language extension that you put at the top of the source file:

You should also import the library that defines :

A kind signature is just like a type signature, except that it can be used to describe functions operating on types.

Regular types have the kind . Type constructors have the kind , since they map types to types.

takes two type constructors and produces a type constructor, so its kind signature is:

and the full definition is:

Any two type constructors can be composed this way. There is no requirement, at this point, that they be functors.

However, if we want to lift a function using the composition of type constructors, after , then they must be functors. This requirement is encoded as a constraint in the instance declaration:

The constraint expresses the condition that both type constructors be instances of the class. The constraints are followed by a double arrow.

The type constructor whose functoriality we are establishing is , which is a partial application of to two functors.

In the implementation of , we pattern match on the data constructor . Its argument is of the type . We use one to “get under” . Then we use to get under . The compiler knows which to use by analyzing the types.

You may visualize a composite functor as a container of containers. For instance, the composition of with is a list of optional values.

Exercise 8.5.1. *Define a composition of a after . Hint: You can reuse , but you have to provide a different instance declaration.*

Category of categories

We can view functors as arrows between categories. As we've just seen, functors are composable and it's easy to check that this composition is associative. We also have an identity (endo-) functor for every category. So categories themselves seem to form a category, let's call it Cat.

And this is where mathematicians start worrying about "size" issues. It's a shorthand for saying that there are paradoxes lurking around. So the correct incantation is that Cat is a category of *small* categories. But as long as we are not engaged in proofs of existence, we can ignore size problems.

Chapter 9

Natural Transformations

We've seen that, when two objects a and b are isomorphic, they generate bijections between sets of arrows, which we can now express as isomorphisms between hom-sets. For all x, we have:

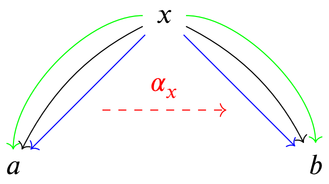
$$C(a, x) \text{ ö } C(b, x)$$
$$C(x, a) \text{ ö } C(x, b)$$

The converse is not true, though. An isomorphism between hom-sets does not result in an isomorphism between object *unless* additional naturality conditions are satisfied. We'll now re-formulate these naturality conditions in progressively more general settings.

9.1 Natural Transformations Between Hom-Functors

One way an isomorphism between two objects can be established is by directly providing two arrows—one the inverse of the other. But quite often it's easier to do it indirectly, by defining bijections between arrows, either the ones impinging on the two objects, or the ones emanating from the two objects.

For instance, as we've seen before, we may have, for every x, an invertible mapping of arrows α_x.



In other words, for every x, there is a mapping of hom-sets:

$$\alpha_x \colon C(x, a) \to C(x, b)$$

When we vary x, the two hom-sets become two (contravariant) functors, $C(*, a)$ and $C(*, b)$, and α can be seen as a mapping between them. Such a mapping of functors, called a transformation, is really a family of individual mappings α_x, one per each object x in the category C.

The functor $C(*, a)$ describes the way the worlds sees a, and the functor $C(*, b)$ describes the way the world sees b.

The transformation α switches back and forth between these two views. Every component of α, the bijection α_x, shows that the view of a from x is isomorphic to the view of b from x.

The naturality condition we discussed before was the condition:

$$\alpha_y \circ (* \circ g) = (* \circ g) \circ \alpha_x$$

It relates components of α taken at different objects. In other words, it relates the views from two different observers x and y, who are connected by an arrow $g \colon y \to x$.

Both sides of this equation are acting on the hom-set $C(x, a)$. The result is in the hom-set $C(y, b)$. We can rewrite the two sides as:

$$C(x, a) \xrightarrow{(* \circ g)} C(y, a) \xrightarrow{\alpha_y} C(y, b)$$

$$C(x, a) \xrightarrow{\alpha_x} C(x, b) \xrightarrow{(* \circ g)} C(y, b)$$

Precomposition with $g \colon y \to x$ is also a mapping of hom-sets. In fact it is the lifting of g by the contravariant hom-functor. We can write it as $C(g, a)$ and $C(g, b)$, respectively.

$$C(x, a) \xrightarrow{C(g,a)} C(y, a) \xrightarrow{\alpha_y} C(y, b)$$

$$C(x, a) \xrightarrow{\alpha_x} C(x, b) \xrightarrow{C(g,b)} C(y, b)$$

The naturality condition can therefore be rewritten as:

$$\alpha_y \circ C(g, a) = C(g, b) \circ \alpha_x$$

It can be illustrated by this commuting diagram:

$$\begin{tikzcd}
C(x, a) \arrow[r, "C(g,a)"] \arrow[d, "\alpha_x"] & C(y, a) \arrow[d, "\alpha_y"] \\
C(x, b) \arrow[r, "C(g,b)"] & C(y, b)
\end{tikzcd}$$

We can now re-formulate our previous result: An invertible transformation α between the functors $C(*, a)$ and $C(*, b)$ that satisfies the naturality condition is equivalent to an isomorphism between a and b.

We can follow exactly the same reasoning for the outgoing arrows. This time we start with a transformation β whose components are:

$$\beta_x \colon \mathcal{C}(a, x) \to \mathcal{C}(b, x)$$

The two (covariant) functors $\mathcal{C}(a, *)$ and $\mathcal{C}(b, *)$ describe the view of the world from the perspective of a and b, respectively. The invertible transformation β tells us that these two views are equivalent, and the naturality condition

$$(g \circ *) \circ \beta_x = \beta_y \circ (g \circ *)$$

tells us that they behave nicely when we switch focus.

Here's the commuting diagram that illustrates the naturality condition:

$$\begin{tikzcd}
\mathcal{C}(a, x) \arrow[r, "\mathcal{C}(a,g)"] \arrow[d, "\beta_x"] & \mathcal{C}(a, y) \arrow[d, "\beta_y"] \\
\mathcal{C}(b, x) \arrow[r, "\mathcal{C}(b,g)"] & \mathcal{C}(b, y)
\end{tikzcd}$$

Again, such an invertible natural transformation β establishes the isomorphism between a and b.

9.2 Natural Transformation Between Functors

The two hom-functors from the previous section were

$$Fx = \mathcal{C}(a, x)$$
$$Gx = \mathcal{C}(b, x)$$

They both map the category $\mathcal{C}$ to $\mathbf{Set}$, because that's where the hom-sets live. We can say that they create two different *models* of $\mathcal{C}$ inside $\mathbf{Set}$.

A natural transformation is a structure-preserving mapping between two such models.

$$\begin{tikzcd}
& \mathcal{C}(a, x) \arrow[dd, "\beta_x"] \\
x \arrow[ur, dashed, "\mathcal{C}(a,*)"] \arrow[dr, dashed, "\mathcal{C}(b,*)"'] & \\
& \mathcal{C}(b, x)
\end{tikzcd}$$

This idea naturally extends to functors between any pair of categories. Any two functors

$$F \colon \mathcal{C} \to \mathcal{D}$$
$$G \colon \mathcal{C} \to \mathcal{D}$$

may be seen as two different models of $\mathcal{C}$ inside $\mathcal{D}$.

To transform one model into another we connect the corresponding dots using arrows in $\mathcal{D}$. For every object x in $\mathcal{C}$ we pick an arrow that goes from Fx to Gx:

$$\alpha_x \colon Fx \to Gx$$

A natural transformation thus maps objects to arrows.

$$\begin{tikzcd}
 & Fx \arrow[dd, "\alpha_x"] \\
x \arrow[ur, dashed, "F"] \arrow[dr, dashed, "G"'] & \\
 & Gx
\end{tikzcd}$$

The structure of a model, though, has as much to do with objects as it does with arrows, so let's see what happens to arrows. For every arrow $f \colon x \to y$ in $\mathcal{C}$, we have two corresponding arrows in $\mathcal{D}$:

$$Ff \colon Fx \to Fy$$
$$Gf \colon Gx \to Gy$$

These are the two liftings of f. We can use them to move within the bounds of each of the two models. Then there are the components of α which let us switch *between* models.

Naturality says that it shouldn't matter whether you first move inside the first model and then jump to the second one, or first jump to the second one and then move within it. This is illustrated by the commuting *naturality square*:

$$\begin{tikzcd}
Fx \arrow[r, "Ff"] \arrow[d, "\alpha_x"] & Fy \arrow[d, "\alpha_y"] \\
Gx \arrow[r, "Gf"] & Gy
\end{tikzcd}$$

Such a family of arrows α_x that satisfies the naturality condition is called a *natural transformation*.

This is a diagram that shows a pair of categories, two functors between them, and a natural transformation α between the functors:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=50, "F"{name=U}] \arrow[r, bend right=50, "G"'{name=D}] & \mathcal{D}
\arrow[Rightarrow, from=U, to=D, "\alpha"']
\end{tikzcd}$$

Since for every arrow in C there is a corresponding naturality square, we can say that a natural transformation maps objects to arrows, and arrows to commuting squares.

If every component α_x of a natural transformation is an isomorphism, α is called a *natural isomorphism*.

We can now restate the main result about isomorphisms: Two objects are isomorphic if and only if there is a natural isomorphism between their hom-functors (either the covariant, or the contravariant ones—either one will do).

Natural transformations provide a very convenient high-level way of expressing commuting conditions in a variety of situations. We'll use them in this capacity to reformulate the definitions of algebraic data types.

9.3 Natural Transformations in Programming

A natural transformation is a family of arrows parameterized by objects. In programming, this corresponds to a family of functions parameterized by types, that is a *polymorphic function*.

The type of the argument to a natural transformation is described using one functor, and the return type using another.

In Haskell, we can define a data type that accepts two type constructors representing two functors, and produces a type of natural transformations:

The quantifier tells the compiler that the function is polymorphic—that is, it's defined for every type . As long as and are functors, this formula defines a natural transformation.

The types defined by are very special, though. The are polymorphic in the sense of *parametric polymorphism*. It means that a single formula is used for all types. We've seen the example of the identity function, which can be written as:

The body of this function is very simple, just the variable . It doesn't matter what type is, the formula remains the same.

This is in contrast to *ad-hoc polymorphism*. An ad-hoc polymorphic function may use different implementations for different types. An example of such a function is , the member function of the typeclass. There is one implementation of for lists, a different one for , and so on, case by case.

The standard definition of a (parametric) natural transformation in Haskell uses a *type synonym:*

A declaration introduces an alias, a shorthand, for the right-hand-side.

It turns out that limiting the type of a natural transformation to adhere to parametric polymorphism has far-reaching consequences. Such a function automatically satisfies naturality conditions. It's an example of parametricity producing so called *theorems for free*.

We can't express equalities of arrows in Haskell, but we can use naturality to transform programs. In particular, if is a natural transformation, we can replace:

with:

Here, the compiler will automatically figure out what versions of and which components of to use.

We can also use more advanced language options to make the choices explicit. We can express naturality using a pair of functions:

The annotations and specify the components of the parametrically polymorphic function , and the annotations and specify the functors for which the ad-hoc polymorphic is instantiated.

The following Haskell extensions have to be specified at the top of the file:

Here's an example of a useful function that is a natural transformation between the list functor and the functor:

(The standard library function is "unsafe" in that it faults when given an empty list.)

Another example is the function , which reverses a list. It's a natural transformation from the list functor to the list functor:

Incidentally, this is a very inefficient implementation. The actual library function uses an optimized algorithm.

A useful intuition for understanding natural transformations builds on the idea that functors acts like containers for data. There are two completely orthogonal things that you can do with a container: You can transform the data it contains, without changing the shape of the container. This is what does. Or you can transfer the data, without modifying it, to another container. This is what a natural transformation does: It's a procedure for moving "stuff" between containers without knowing what kind of "stuff" it is.

In other words, a natural transformation repackages the contents of one container into another container. It does it in a way that is agnostic of the type of the contents, which means it cannot inspect, create, or modify the contents. All it can do is to move it to a new location, or drop it.

Naturality condition enforces the orthogonality of these two operations. It doesn't matter if you first modify the data and then move it to another container; or first move it, and then modify.

This is another example of successfully decomposing a complex problem into a sequence of simpler ones. Keep in mind, though, that not every operation with containers of data can be decomposed in that way. Filtering, for instance, requires both examining the data, as well as changing the size or even the shape of the container.

On the other hand, almost every parametrically polymorphic function is a natural transformation. In some cases you may have to consider the identity or the constant functor as either source or the target. For instance, the polymorphic identity function can be thought of as a natural transformation between two identity functors.

Vertical composition of natural transformations

Natural transformations can only be defined between *parallel* functors, that is functors that share the same source category and the same target category. Such parallel functors form a *functor category*. The standard notation for a functor category between two categories $\mathcal{C}$ and $\mathcal{D}$ is $[\mathcal{C}, \mathcal{D}]$. You just put the names of the two categories between square brackets.

The objects in $[\mathcal{C}, \mathcal{D}]$ are functors, the arrows are natural transformations.

To show that this is indeed a category, we have to define the composition of natural transformations. This is easy if we keep in mind that components of natural transformations are regular arrows in the target category. These arrows compose.

Indeed, suppose that we have a natural transformation α between two functors F and G. We want to compose it with another natural transformation β that goes from G to H.

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=60, "F"] \arrow[r, "G"] \arrow[r, bend right=60, "H"'] \arrow[r, phantom, bend left=30, "\alpha \Downarrow"] \arrow[r, phantom, bend right=30, "\beta \Downarrow"] & \mathcal{D}
\end{tikzcd}$$

Let's look at the components of these transformations at some object x

$$\alpha_x \colon F\,x \to G\,x$$

$$\beta_x \colon G\,x \to H\,x$$

These are just two arrows in $\mathcal{D}$ that are composable. So we can define a composite natural transformation γ as follows:

$$\gamma \colon F \to H$$

$$\gamma_x = \beta_x \circ \alpha_x$$

This is called the *vertical composition* of natural transformations. You'll see it written using a dot $\gamma = \beta \cdot \alpha$ or a simple juxtaposition $\gamma = \beta\alpha$.

Naturality condition for γ can be shown by pasting together (vertically) two naturality squares for α and β:

$$\begin{tikzcd}
Fx \arrow[r, "Ff"] \arrow[d, "\alpha_x"] \arrow[dd, bend right=60, "\gamma_x"'] & Fy \arrow[d, "\alpha_y"] \arrow[dd, bend left=60, "\gamma_y"] \\
Gx \arrow[r, "Gf"] \arrow[d, "\beta_x"] & Gy \arrow[d, "\beta_y"] \\
Hx \arrow[r, "Hf"] & Hy
\end{tikzcd}$$

In Haskell, vertical composition of natural transformation is just regular function composition applied to polymorphic functions. Using the intuition that natural transformations move items between containers, vertical composition combines two such moves, one after another.

Functor categories

Since the composition of natural transformations is defined in terms of composition of arrows, it is automatically associative.

There is also an identity natural transformation id_F defined for every functor F. Its component at x is the usual identity arrow at the object Fx:

$$(id_F)_x = id_{Fx}$$

To summarize, for every pair of categories $\mathcal{C}$ and $\mathcal{D}$ there is a category of functors $[\mathcal{C}, \mathcal{D}]$ with natural transformations as arrows.

The hom-set in that category is the set of natural transformations between two functors F and G. Following the standard notational convention, we write it as:

$$[\mathcal{C}, \mathcal{D}](F, G)$$

with the name of the category followed by the names of the two objects (here, functors) in parentheses.

In category theory objects and arrows are drawn differently. Objects are dots and arrows are pointy lines.

In $\mathbf{Cat}$, the category of categories, functors are drawn as arrows. But in a functor category $[\mathcal{C}, \mathcal{D}]$ functors are dots and natural transformations are arrows.

What is an arrow in one category could be an object in another.

Exercise 9.3.1. *Prove the naturality condition of the composition of natural transformations:*

$$\gamma_y \circ F f = H f \circ \gamma_x$$

Hint: Use the definition of γ and the two naturality conditions for α and β.

Horizontal composition of natural transformations

The second kind of composition of natural transformations is induced by the composition of functors. Suppose that we have a pair of composable functors

$$F \colon \mathcal{C} \to \mathcal{D} \qquad\qquad G \colon \mathcal{D} \to \mathcal{E}$$

and, in parallel, another pair of composable functors:

$$F^{\cdot\cdot} \colon \mathcal{C} \to \mathcal{D} \qquad\qquad G^{\cdot\cdot} \colon \mathcal{D} \to \mathcal{E}$$

We also have two natural transformations:

$$\alpha \colon F \to F^{\cdot\cdot} \qquad\qquad \beta \colon G \to G^{\cdot\cdot}$$

Pictorially:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=50, "F"{name=U1}] \arrow[r, bend right=50, "F^{\cdot\cdot}"'{name=D1}] & \mathcal{D} \arrow[r, bend left=50, "G"{name=U2}] \arrow[r, bend right=50, "G^{\cdot\cdot}"'{name=D2}] & \mathcal{E}
\arrow[Rightarrow, from=U1, to=D1, "\alpha"']
\arrow[Rightarrow, from=U2, to=D2, "\beta"']
\end{tikzcd}$$

The *horizontal composition* $\beta \circ \alpha$ maps $G \circ F$ to $G^{\cdot\cdot} \circ F^{\cdot\cdot}$.

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=50, "G \circ F"{name=U}] \arrow[r, bend right=50, "G^{\cdot\cdot} \circ F^{\cdot\cdot}"'{name=D}] & \mathcal{E}
\arrow[Rightarrow, from=U, to=D, "\beta \circ \alpha"']
\end{tikzcd}$$

Let's pick an object x in $\mathcal{C}$ and try to define the component of the composite $(\beta \circ \alpha)$ at x. It should be a morphism in $\mathcal{E}$:

$$(\beta \circ \alpha)_x \colon G(F x) \to G^{\cdot\cdot}(F^{\cdot\cdot} x)$$

We can use α to map x to an arrow

$$\alpha_x \colon F x \to F^{\cdot\cdot} x$$

We can lift this arrow using G

$$G(\alpha_x) \colon G(F x) \to G(F^{\cdot\cdot} x)$$

To get from there to $G^{¨}(F^{¨}x)$ we can use the appropriate component of β

$$\beta_{F^{¨}x}\colon G(F^{¨}x) \to G^{¨}(F^{¨}x)$$

Altogether, we have

$$(\beta \circ \alpha)_x = \beta_{F^{¨}x} \circ G(\alpha_x)$$

But there is another equally plausible candidate:

$$(\beta \circ \alpha)_x = G^{¨}(\alpha_x) \circ \beta_{Fx}$$

Fortunately, they are equal due to naturality of β.

$$\begin{tikzcd}
 & & G(Fx) \arrow[r, "\beta_{Fx}"] \arrow[d, "G(\alpha_x)"] & \\
 & Fx \arrow[ur, dashed] \arrow[rr, dashed, bend right] \arrow[dd, "\alpha_x"] & & G^{¨}(Fx) \arrow[dd, "G^{¨}(\alpha_x)"] \\
x \arrow[ur, dashed] \arrow[dr, dashed] & & G(F^{¨}x) \arrow[dr, "\beta_{F^{¨}x}"] & \\
 & F^{¨}x \arrow[ur, dashed] \arrow[rr, dashed, bend right] & & G^{¨}(F^{¨}x)
\end{tikzcd}$$

The proof of naturality of $\beta \circ \alpha$ is left as an exercise to a dedicated reader.

We can translate this directly to Haskell. We start with two natural transformations:

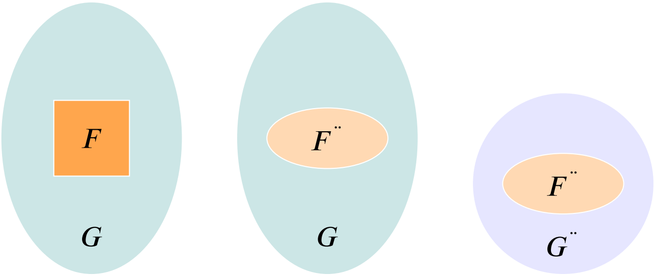
Their horizontal composition has the following type signature:

It has two equivalent implementations. The first one is:

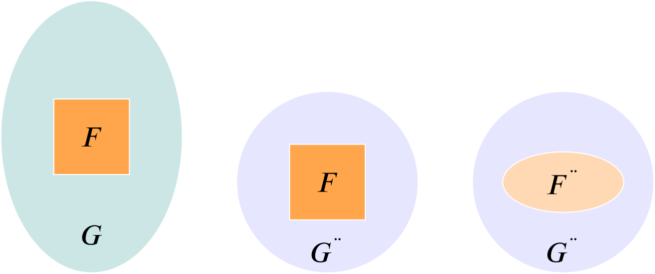
The compiler will automatically pick the correct version of , the one for the functor . The second implementation is:

Here, the compiler will pick the version of for the functor .

What's the intuition for horizontal composition? We've seen before that a natural transformation can be seen as repackaging data between two containers–functors. Here we are dealing with nested containers. We start with the outer container described by that is filled with inner containers, each described by . We have two natural transformations, for transfering the contents of to , and for moving the contents of to . There are two ways of moving data from to . We can use to repackage all inner containers, and then use to repackage the outer container.



Or we can first use to repackage the outer container, and then apply to repackage all the inner containers. The end result is the same.



Exercise 9.3.2. *Implement two versions of horizontal composition of after . Compare their results acting on various arguments.*

Exercise 9.3.3. *Do the same with the horizontal composition of after .*

Whiskering

Quite often horizontal composition is used with one of the natural transformations being the identity. There is a shorthand notation for such composition. For instance, $\alpha \circ id_F$ is written as $\alpha \circ F$.

Because of the characteristic shape of the diagram, such composition is called "whiskering".

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "F"] & \mathcal{D} \arrow[r, bend left=50, "G"{name=U}] \arrow[r, bend right=50, "G''"'{name=D}] & \mathcal{E} \arrow[from=U, to=D, Rightarrow, "\alpha"']
\end{tikzcd}$$

In components, we have:

$$(\alpha \circ F)_x = \alpha_{Fx}$$

Let's consider how we would translate this to Haskell. A natural transformation is a polymorphic function. Because of parametricity, it's defined by the same formula for all types. So whiskering on the right doesn't change the formula, it changes the function signature.

For instance, if this is the declaration of :

then its whiskered version would be:

Because of Haskell's type inference, this shift is implicit. When a polymorphic function is called, we don't have to specify which component of the natural transformation is executed–the type checker figures it out by looking at the type of the argument.

The intuition in this case is that we are repackaging the outer container leaving the inner containers intact.

Similarly, $id_H \circ \alpha$ is written as $H \circ \alpha$.

$$\begin{tikzcd} \mathcal{D} \arrow[r, bend left=50, "G"] \arrow[r, bend right=50, "G^{\cdot\cdot}"'] \arrow[r, phantom, "\alpha \Downarrow"] & \mathcal{E} \arrow[r, "H"] & \mathcal{F} \end{tikzcd}$$

In components:

$$(H \circ \alpha)_x = H(\alpha_x)$$

In Haskell, the lifting of α_x by H is done using , so given:

the whiskered version would be:

Again, Haskell's type inference engine figures out which version of to use (here, it's the one from the instance of).

The intuition is that we are repackaging the contents of the inner containers leaving the outer container intact.

Finally, in many applications a natural transformation is whiskered on both sides:

$$\begin{tikzcd} \mathcal{C} \arrow[r, "F"] & \mathcal{D} \arrow[r, bend left=50, "G"] \arrow[r, bend right=50, "G^{\cdot\cdot}"'] \arrow[r, phantom, "\alpha \Downarrow"] & \mathcal{E} \arrow[r, "H"] & \mathcal{F} \end{tikzcd}$$

In components, we have:

$$(H \circ \alpha \circ F)x = H(\alpha_{Fx})$$

and in Haskell:

Here the intuition is that we have a triple layer of containers; and we are rearranging the middle one, leaving the outer container and all the inner containers intact.

Interchange law

We can combine vertical composition with horizontal composition, as seen in the following diagram:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=60, "F"{name=F}] \arrow[r, "G"{name=G}] \arrow[r, bend right=60, "H"'{name=H}] & \mathcal{D} \arrow[r, bend left=60, "F^{\cdot\cdot}"{name=F2}] \arrow[r, "G^{\cdot\cdot}"{name=G2}] \arrow[r, bend right=60, "H^{\cdot\cdot}"'{name=H2}] & \mathcal{E}
\arrow[Rightarrow, from=F, to=G, "\alpha"']
\arrow[Rightarrow, from=G, to=H, "\beta"']
\arrow[Rightarrow, from=F2, to=G2, "\alpha^{\cdot\cdot}"']
\arrow[Rightarrow, from=G2, to=H2, "\beta^{\cdot\cdot}"']
\end{tikzcd}$$

The interchange law states that the order of composition doesn't matter: we can first do vertical compositions and then the horizontal one, or first do the horizontal compositions and then the vertical one.

9.4 Universal Constructions Revisited

Lao Tzu says, the simplest pattern is the clearest.

We've seen definitions of sums, products, exponentials, natural numbers, and lists.

The old-school approach to defining such data types is to explore their internals. It's the set-theory way: we look at how the elements of new sets are constructed from the elements of old sets. An element of a sum is either an element of the first set, or the second set. An element of a product is a pair of elements. And so on. We are looking at objects from the engineering point of view.

In category theory we take the opposite approach. We are not interested in what's inside the object or how it's implemented. We are interested in the purpose of the object, how it can be used, and how it interacts with other objects. We are looking at objects from the utilitarian point of view.

Both approaches have their advantages. The categorical approach came later, because you need to study a lot of examples before clear patterns emerge. But once you see the patterns, you discover unexpected connections between things, like the duality between sums and products.

Defining particular objects through their connections requires looking at possibly infinite numbers of objects with which they interact.

"Tell me your relation to the Universe, and I'll tell you who you are."

Defining an object by its mappings-out or mappings-in with respect to all objects in the category is called a *universal construction*.

Why are natural transformations so important? It's because most categorical constructions involve commuting diagrams. If we can re-cast these diagrams as naturality squares, we move one level up the abstraction ladder and gain new valuable insights.

Being able to compress a lot of facts into small elegant formulas helps us see new patterns. We'll see, for instance, that natural isomorphisms between hom-sets pop up all over category theory and eventually lead to the idea of an adjunction.

But first we'll study several examples in greater detail to get some understanding of the terse language of category theory. We'll try, for instance, to decode the statement that the sum, or the coproduct of two objects, is defined by the following natural isomorphism:

$$[2, \mathcal{C}](D, \Delta_x) \text{ ö } \mathcal{C}(a+b, x)$$

Picking objects

Even such a simple task as pointing at an object has a special interpretation in category theory. We have already seen that pointing at an element of a set is equivalent to selecting a function from the singleton set to it. Similarly, picking an object in a category is equivalent to selecting a functor from the single-object category. Alternatively, it can be done using a constant functor from another category.

Quite often we want to pick a pair of objects. That, too, can be accomplished by selecting a functor from a two-object stick-figure category. Similarly, picking an arrow is equivalent to selecting a functor from the "walking arrow" category, and so on.

By judiciously selecting our functors and natural transformations between them, we can reformulate all the universal constructions we've seen so far.

Cospans as natural transformations

The definition of a sum requires the selection of two objects to be summed; and a third one to serve as the target of the mapping out.

$$\begin{tikzcd}
a \arrow[dr, bend left, "\text{Left}"'] \arrow[ddr, bend right, "f"'] & & b \arrow[dl, bend right, "\text{Right}"] \arrow[ddl, bend left, "g"] \\
& a+b \arrow[d, dashed, "h"] & \\
& c &
\end{tikzcd}$$

This diagram can be further decomposed into two simpler shapes called *cospans*:

$$\begin{tikzcd}
a \arrow[dr] & & b \arrow[dl] \\
& x &
\end{tikzcd}$$

To construct a cospan we first have to pick a pair of objects. To do that we'll start with a two-object category 2. We'll call its objects 1 and 2. We'll use a functor

$$D \colon 2 \to \mathcal{C}$$

to select the objects a and b:

$$D\,1 = a$$
$$D\,2 = b$$

(D stands for "diagram", since the two objects form a very simple diagram consisting of two dots in $\mathcal{C}$.)

We'll use the constant functor

$$\Delta_x \colon 2 \to \mathcal{C}$$

to select the object x. This functor maps both 1 and 2 to x (and the two identity arrows to id_x).

Since both functors go from 2 to $\mathcal{C}$, we can define a natural transformation α between them. In this case, it's just a pair of arrows:

$$\alpha_1 \colon D\,1 \to \Delta_x 1$$
$$\alpha_2 \colon D\,2 \to \Delta_x 2$$

These are exactly the two arrows in the cospan.

Naturality condition for α is trivial, since there are no arrows (other than identities) in 2.

There may be many cospans sharing the same three objects—meaning: there may be many natural transformations between the two functors D and Δ_x. These natural transformations form a hom-set in the functor category $[2, \mathcal{C}]$, namely:

$$[2, \mathcal{C}](D, \Delta_x)$$

Functoriality of cospans

Let's consider what happens when we start varying the object x in a cospan. We have a mapping F that takes x to the set of cospans over x:

$$F x = [2, \mathcal{C}](D, \Delta_x)$$

This mapping turns out to be functorial in x.

To see that, consider an arrow $m \colon x \to y$. The lifting of this arrow is a mapping between two sets of natural transformations:

$$[2, \mathcal{C}](D, \Delta_x) \to [2, \mathcal{C}](D, \Delta_y)$$

This might look very abstract until you remember that natural transformations have components, and these components are just regular arrows. An element of the left-hand side is a natural transformation:

$$\mu \colon D \to \Delta_x$$

It has two components corresponding to the two objects in 2. For instance, we have

$$\mu_1 \colon D\,1 \to \Delta_x 1$$

or, using the definitions of D and Δ:

$$\mu_1 \colon a \to x$$

This is just the left leg of our cospan.

Similarly, the element of the right-hand side is a natural transformation:

$$\nu \colon D \to \Delta_y$$

Its component at 1 is an arrow

$$\nu_1 \colon a \to y$$

We can get from μ_1 to ν_1 simply by post-composing it with $m \colon x \to y$. So the lifting of m is a component-by-component post-compositon $(m \circ *)$:

$$\nu_1 = m \circ \mu_1$$
$$\nu_2 = m \circ \mu_2$$

Sum as a universal cospan

Of all the cospans that you can build on the pair a and b, the one with the arrows we called Left and Right converging on $a + b$ is very special. There is a unique mapping out of it to any other cospan—a mapping that makes two triangles commute.

$$\begin{tikzcd}
a \arrow[dr, bend left, "\text{Left}"] \arrow[ddr, bend right, "f"'] & & b \arrow[dl, bend right, "\text{Right}"'] \arrow[ddl, bend left, "g"] \\
& a + b \arrow[d, dashed, "h"] & \\
& x &
\end{tikzcd}$$

We are now in the position to translate this condition to a statement about natural transformations and hom-sets. The arrow h is an element of the hom-set

$$C(a + b, x)$$

A cospan over x is a natural transformation, that is an element of the hom-set in the functor category:

$$[2, C](D, \Delta_x)$$

Both are hom-sets in their respective categories. And both are just sets, that is objects in the category Set. This category forms a bridge between the functor category $[2, C]$ and a "regular" category C, even though, conceptually, they seem to be at very different levels of abstraction.

Paraphrasing Sigmund Freud, "Sometimes a set is just a set."

Our universal construction is the bijection or the isomorphism of sets:

$$[2, C](D, \Delta_x) \text{ ö } C(a + b, x)$$

Moreover, if we vary the object x, the two sides behave like functors from C to Set. Therefore it makes sense to ask if this mapping of functors is a natural isomorphism.

Indeed, it can be shown that the naturality condition for this isomorphism translates into commuting conditions for the triangles in the definition of the sum. So the definition of the sum can be replaced by a single equation.

Product as a universal span

An analogous argument can be made about the universal construction for the product. Again, we start with the stick-figure category 2 and the functor D. But this time we use a natural transformation going in the opposite direction

$$\alpha \colon \Delta_x \to D$$

Such a natural transformation is a pair of arrows that form a *span*:

$$\begin{tikzcd} & x \arrow[dl, "f"'] \arrow[dr, "g"] & \\ a & & b \end{tikzcd}$$

Collectively, these natural transformations form a hom-set in the functor category :

$$[2, C](\Delta_x, D)$$

Every element of this hom-set is in one-to-one correspondence with a unique mapping h into the product $a\ b$. Such a mapping is a member of the hom-set $C(x, a\ b)$. This correspondence is expressed as the isomorphism:

$$[2, C](\Delta_x, D) \text{ ö } C(x, a\quad b)$$

It can be shown that the naturality of this isomorphism guarantees that the triangles in this diagram commute:

$$\begin{tikzcd}
 & x \arrow[d, dashed, "h"] \arrow[ddl, bend right, "f=\alpha_1"'] \arrow[ddr, bend left, "g=\alpha_2"] & \\
 & a\ \ b \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\
a = D1 & & b = D2
\end{tikzcd}$$

Exponentials

The exponentials, or function objects, are defined by this commuting diagram:

$$\begin{tikzcd}
x\ \ a \arrow[d, dashed, "h\ id_a"'] \arrow[dr, "f"] & \\
b^a\ \ a \arrow[r, "\varepsilon_{ab}"'] & b
\end{tikzcd}$$

Here, f is an element of the hom-set $C(x\quad a, b)$ and h is an element of $C(x, b^a)$.

The isomorphism between these sets, natural in x, defines the exponential object.

$$C(x\quad a, b) \text{ ö } C(x, b^a)$$

The f in the diagram above is an element of the left-hand side, and h is the corresponding element of the right-hand side. The transformation α_x (which also depends on a and b) maps f to h.

$$\alpha_x \colon C(x\quad a, b) \to C(x, b^a)$$

In Haskell, we call it . Its inverse, α^{*1} is known as .

Unlike in the previous examples, here both hom-sets are in the same category, and it's easy to analyze the isomorphism in more detail. In particular, we'd like to see how the commuting condition:

$$f = \varepsilon_{ab} \circ (h\quad id_a)$$

arises from naturality.

The standard Yoneda trick is to make a substitution for x that would reduce one of the hom-sets to an endo-hom-set, that is a hom-set whose source is the same the target. This will allow us to pick a canonical element of that hom-set, that is the identity arrow.

In our case, substituting b^a for x will allow us to pick $h = id_{(b^a)}$.

$$\begin{tikzcd}
b^a\ \ a \arrow[d, dashed, "id_{(b^a)}\ id_a"'] \arrow[dr, "f"] & \\
b^a\ \ a \arrow[r, "\varepsilon_{ab}"'] & b
\end{tikzcd}$$

The commuting condition in this case tells us that $f = \varepsilon_{ab}$. In other words, we get the formula for ε_{ab} in terms of α:

$$\varepsilon_{ab} = \alpha_x^{*1}(id_x)$$

where x is equal to b^a.

Since we recognize α^{*1} as , and ε as function application, we can write it in Haskell as:

This my be surprising at first, until you realize that the currying of leads to .

We can also encode the two sides of the main isomorphism as Haskell functors:

They are both contravariant functors in the type variable .

This says that the lifting of $g\colon x \to y$ is given by the following pre-composition:

$$C(y \quad a, b) \xrightarrow{(*\circ(g \; id_a))} C(x \quad a, b)$$

Similarly:

translates to:

$$C(y, b^a) \xrightarrow{(*\circ g)} C(x, b^a)$$

The natural transformation α is just a thin encapsulation of ; and its inverse is :

Using the two formulas for the lifting of $g\colon x \to y$, here's the naturality square:

$$\begin{tikzcd}
C(y \quad a, b) \arrow[r, "(*\circ(g \; id_a))"] \arrow[d, "\alpha_y"] & C(x \quad a, b) \arrow[d, "\alpha_x"] \\
C(y, b^a) \arrow[r, "(*\circ g)"] & C(x, b^a)
\end{tikzcd}$$

Let's now apply the Yoneda trick to it and replace y with b^a. This also allows us to substitute g—which now goes for x to b^a—with h.

$$\begin{tikzcd}
C(b^a \quad a, b) \arrow[r, "(*\circ(h \; id_a))"] \arrow[d, "\alpha_{(b^a)}"] & C(x \quad a, b) \arrow[d, "\alpha_x"] \\
C(b^a, b^a) \arrow[r, "(*\circ h)"] & C(x, b^a)
\end{tikzcd}$$

We know that the hom-set $C(b^a, b^a)$ contains at least the identity arrow, so we can pick the element $id_{(b^a)}$ in the lower left corner.

Reversing the arrow on the left, we know that α^{*1} acting on identity produces ε_{ab} in the upper left corner (that's the trick).

Pre-composition with h acting on identity produces h in the lower right corner.

α^{*1} acting on h produces f in the upper right corner.

$$\begin{tikzcd}
\varepsilon_{ab} \arrow[r, mapsto, "(*\circ(h\ id_a))"] & f \\
id_{(b^a)} \arrow[u, mapsto, "\alpha^{*1}"] \arrow[r, mapsto, "(*\circ h)"] & h \arrow[u, mapsto, "\alpha^{*1}"']
\end{tikzcd}$$

(The ↦ arrows denote the action of functions on elements of sets.)

So the selection of $id_{(b^a)}$ in the lower left corner fixes the other three corners. In particular, we can see that the upper arrow applied to ε_{ab} produces f, which is exactly the commuting condition:

$$\varepsilon_{ab}\circ(h \quad id_a) = f$$

the one that we set out to derive.

9.5 Limits and Colimits

In the previous section we defined the sum and the product using natural transformations. These were transformations between diagrams defined as functors from a very simple stick-figure category 2, one of the functors being the constant functor.

Nothing prevents us from replacing the category 2 with something more complex. For instance, we could try categories that have non-trivial arrows between objects, or categories with infinitely many objects.

There is a whole vocabulary built around such constructions.

We used objects in the category 2 for indexing objects in the category $\mathcal{C}$. We can replace 2 with an arbitrary indexing category $\mathcal{J}$. A diagram in $\mathcal{C}$ is still defined as a functor $D\colon \mathcal{J} \to \mathcal{C}$. It picks objects in $\mathcal{C}$, but it also picks arrows between them.

As the second functor we'll still use the constant functor $\Delta_x\colon \mathcal{J} \to \mathcal{C}$.

A natural transformation, that is an element of the hom-set

$$[\mathcal{J}, \mathcal{C}](\Delta_x, D)$$

is now called a *cone*. Its dual, an element of

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x)$$

is called a *cocone*. They generalize the span and the cospan, respectively.

Diagramatically, cones and cocones look like this:

$$\begin{tikzcd}
& x \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[dd, "h"] & \\
D1 \arrow[rr] \arrow[dr] & & D2 \arrow[dl] \\
& D3 &
\end{tikzcd}
\qquad
\begin{tikzcd}
D1 \arrow[rr] \arrow[dr] \arrow[ddr, "f"'] & & D2 \arrow[dl] \arrow[ddl, "g"] \\
& D3 \arrow[d, "h"] & \\
& x &
\end{tikzcd}$$

Since the indexing category may now contain arrows, the naturality conditions for these diagrams are no longer trivial. The constant functor Δ_x shrinks all vertices to one, so naturality squares shrink to triangles. Naturality means that all triangles with x in their apex must now commute.

The universal cone, if it exists, is called the *limit* of the diagram D, and is written as $\text{Lim}D$. Universality means that it satisfies the following isomorphism, natural in x:

$$[\mathcal{J}, \mathcal{C}](\Delta_x, D) \text{ ö } \mathcal{C}(x, \text{Lim}D)$$

For each cone with the apex x there is a unique mapping from x into the limit $\text{Lim}D$.

A limit of a $\mathbf{Set}$-valued functor has a particularly simple characterization. It's a set of cones with the singleton set at the apex. Indeed, elements of the limit, that is functions from the singleton set to it, are in one-to-one correspondence with such cones:

$$[\mathcal{J}, \mathcal{C}](\Delta_1, D) \text{ ö } \mathcal{C}(1, \text{Lim}D)$$

Dually, the universal cocone is called a *colimit*, and is described by the following natural isomorphism:

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x) \text{ ö } \mathcal{C}(\text{Colim}D, x)$$

We can now say that a product is a limit (and a sum, a colimit) of a diagram from the indexing category $\mathbf{2}$.

Limits and colimits distill the essence of a pattern.

A limit, like a product, is defined by its mapping-in property. A colimit, like a sum, is defined by its mapping-out property.

There are many interesting limits and colimits, and we'll see some when we discuss algebras and coalgebras.

Exercise 9.5.1. *Show that the limit of a "walking arrow" category, that is a two-object category with an arrow connecting the two objects, has the same elements as the first object in the diagram ("elements" are the arrows from the terminal object).*

Equalizers

A lot of high-school math involves learning how to solve equations or systems of equations. An equation equates the outcomes of two different ways of producing something. If we are allowed to subtract things, we usually shove everything to one side and simplify the problem to the one of calculating the zeros of some expression. In geometry, the same idea is expressed as the intersection of two geometric objects.

In category theory all these patterns are embodied in a single construction called an equalizer. An equalizer is a limit of a diagram whose pattern is given by a stick-figure category with two parallel arrows:

$$\begin{tikzcd} i \arrow[r, shift left] \arrow[r, shift right] & j \end{tikzcd}$$

The two arrows represent two ways of producing something.

A functor from this category picks a pair of objects and a pair of morphisms in the target category. The limit of this diagram embodies an intersection of the two outcomes. It is an object e with two arrows $p\colon e \to a$ and $p^{\cdot\cdot}\colon e \to b$.

$$\begin{tikzcd} & e \arrow[dl, "p"'] \arrow[dr, "p^{\cdot\cdot}"] & \\ a \arrow[rr, shift left, "f"] \arrow[rr, shift right, "g"'] & & b \end{tikzcd}$$

We have two commuting conditions:

$$p^{\cdot\cdot} = f \circ h$$
$$p^{\cdot\cdot} = g \circ h$$

It means that $p^{\cdot\cdot}$ is fully determined by one of the equations, while the other turns into the constraints:

$$f \circ p = g \circ p$$

Since the equalizer is the limit, it is the universal such pair, as illustrated in this diagram:

$$\begin{tikzcd} x \arrow[d, dashed, "h"'] \arrow[dr] & & \\ e \arrow[r, "p"'] & a \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & b \end{tikzcd}$$

To develop the intuition for equalizers it's instructive to consider how it works for sets. As usual, the trick is to replace x with the singleton set 1:

$$\begin{tikzcd} 1 \arrow[d, dashed, "e"'] \arrow[dr, "a"] & & \\ E \arrow[r, "p"'] & A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \end{tikzcd}$$

In this case a is an element of A such that $fa = ga$. That's just a way of saying that a is the solution of a pair of equations. Universality means that there is a unique element e of E such that $p \circ e = a$. In other words, elements of E are in one-to-one correspondence with the solutions of the system of equations.

Coequalizers

What's dual to the idea of equating or intersecting? It's the process of discovering commonalities and organizing things into buckets. For instance, we can distribute integers into even and odd buckets. In category theory, this process of bucketizing is described by coequalizers.

A coequalizer is the colimit of the same diagram that we used to define the equalizer:

$$\begin{tikzcd}
a \arrow[rr, shift left, "f"] \arrow[rr, shift right, "g"'] \arrow[dr, "q^{\cdot\cdot}"'] & & b \arrow[dl, "q"] \\
& c &
\end{tikzcd}$$

This time, the arrow $q^{\cdot\cdot}$ is fully determined by q; and q must satisfy the equation:

$$q \circ f = q \circ g$$

Again, we can gain some intuition by considering a coequalizer of two functions acting on sets.

$$\begin{tikzcd}
A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \arrow[r, "q"] & C
\end{tikzcd}$$

An x Ë A is mapped to two elements fx and gx in B, but then q maps them back to a single element of C. This element represents the bucket. Universality means that C is a copy of B in which the elements that were produced from the same x have been identified.

Consider an example where A is a set of pairs of integers , such that either both are even or both are odd. We want to coequalize two functions that are the two projections . The equalizer set C will have two elements corresponding to two buckets. We'll represent it as . The equalizing function selects the bucket:

Any function that cannot distinguish between the components of our pairs can be uniquely factorized through the function :

Exercise 9.5.2. *Run a few tests that show that, in the example above, the factorization gives the same result as given by the following definition:*

Exercise 9.5.3. *What is the coequalizer of the pair , both of the type ? Test its universality by factorizing the following function:*

The existence of the terminal object

Lao Tzu says: great acts are made up of small deeds.

So far we've been studying limits of tiny diagrams, that is functors from simple stick-figure categories. Nothing, however, prevents us from defining limits and colimits where the patterns are taken to be infinite categories. But there is a gradation of infinities. When the objects in a category form a proper set, we call such a category *small*. Unfortunately, the very basic example, the category Set of sets, is not small. We know that there is no set of all sets. Set is a *large* category. But at least all the hom-sets in Set are sets. We say that Set is *locally small*. In what follows we'll be always working with locally small categories.

A small limit is a limit of a small diagram, that is a functor from a category whose objects and morphisms form sets. A category in which all small limits exist is called small complete, or just *complete*. In particular, in such a category, a product of an arbitrary *set* of objects exists. You can also equalize an arbitrary set of arrows between two objects. If such category is locally small, that means all equalizers exist.

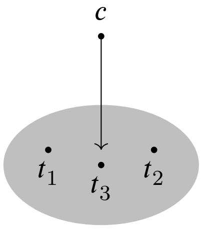
Conversely, a (small) cocomplete category has all small colimits. In particular, such a category has all small coproducts and coequalizers.

The category Set is both complete and cocomplete.

In a cocomplete locally small category there is a simple criterion for the existence of the terminal object: It's enough that a weakly terminal set exists.

A *weakly terminal* object, just like the terminal object, has an arrow coming from any object; except that such an arrow is not necessarily unique.

A *weakly terminal set* is a family of objects t_i indexed by a set I such that, for any object c in C there exists an i and an arrow $c \to t_i$. Such a set is also called a *solution set*.



In a cocomplete category we can always construct a coproduct $\coprod_{iËI} t_i$. This coproduct is a weakly terminal object, because there is an arrow to it from every c. This arrow is the composite of the arrow to some t_i followed by the injection $\iota_i \colon t_i \to \coprod_{jËI} t_j$.

Given a weakly terminal object, we can construct the (strongly) terminal object. We first define a subcategory $\mathcal{T}$ of $\mathcal{C}$ whose objects are t_i. Morphisms in $\mathcal{T}$ are all the morphisms in $\mathcal{C}$ that go between the objects of $\mathcal{T}$. This is called a *full* subcategory of $\mathcal{C}$. By our construction, $\mathcal{T}$ is small.

There is an obvious inclusion functor F that embeds $\mathcal{T}$ in $\mathcal{C}$. This functor defines a small diagram in $\mathcal{C}$. It turns out that the colimit of this diagram is the terminal object in $\mathcal{C}$.

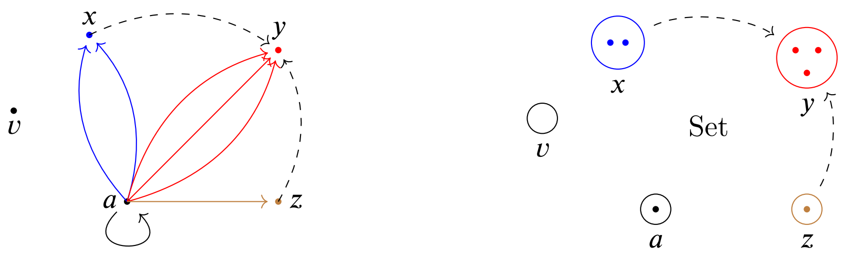
Dually, a similar construction can be used to define an initial object as a limit of a weakly initial set.

This property of solution sets will come handy in the proof of the Freyd's adjoint functor theorem.

9.6 The Yoneda Lemma

A functor from some category $\mathcal{C}$ to the category of sets can be thought of as a model of this category in $\mathbf{Set}$. Modeling, in general, is a lossy process: it discards some information. A constant $\mathbf{Set}$-valued functor is an extreme example: it maps the whole category to a single set and its identity function.

A hom-functor produces a model of the category as viewed from a certain vantage point. The functor $\mathcal{C}(a, *)$, for instance, offers the panorama of $\mathcal{C}$ from the vantage point of a. It organizes all the arrows emanating from a into neat packages that are connected by images of arrows that go between them, all in accordance with the original structure of the source category.



Some vantage points are better than others. For instance, the view from the initial object is quite sparse. Every object x is mapped to a singleton set $\mathcal{C}(0, x)$ corresponding to the unique mapping $0 \to x$.

The view from the terminal object is more interesting: it maps all objects to their sets of (global) elements $\mathcal{C}(1, x)$.

The Yoneda lemma may be considered one of the most profound statements, or one of the most trivial statements in category theory. Let's start with the profound version.

Consider two models of $\mathcal{C}$ in Set. The first one is given by the hom-functor $\mathcal{C}(a, *)$. It's the panoramic, very detailed view of $\mathcal{C}$ from the vantage point of a. The second is given by some arbitrary functor $F \colon \mathcal{C} \to \mathrm{Set}$. Any natural transformation between them embeds one model in the other. It turns out that the set of all such natural transformations is fully determined by the set Fa.

Since the set of natural transformation is the hom-set in the functor category $[\mathcal{C}, \mathrm{Set}]$, the formal statement of the Yoneda lemma takes the form:

$$[\mathcal{C}, \mathrm{Set}](\mathcal{C}(a, *), F) \text{ ö } Fa$$

Moreover, this isomorphism is natural in both a and F.

The reason this works is because all the mappings involved in this theorem are bound by the requirements of preserving the structure of the category $\mathcal{C}$ and the structure of its models. In particular, naturality conditions impose a huge set of constraints on the way the components of a natural transformation propagate from one point to another.

The proof of the Yoneda lemma starts with a single identity arrow and lets naturality propagate it across the whole category.

Here's the sketch of the proof. It consists of two parts: First, given a natural transformation we construct an element of Fa. Second, given an element of Fa we construct the corresponding natural transformation.

First, let's pick an arbitrary element on the left-hand side of the Yoneda lemma: a natural transformation α. Its component at x is a function:

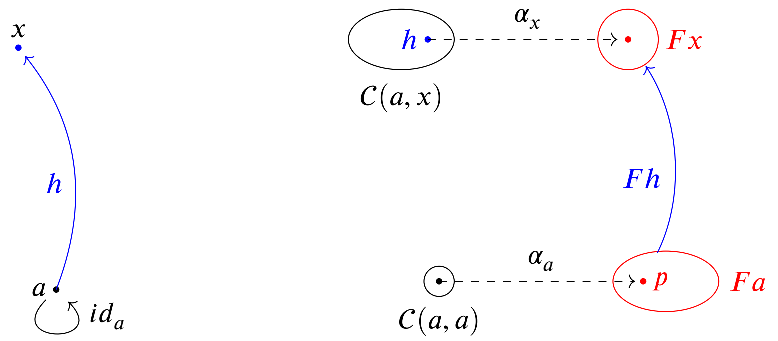
$$\alpha_x \colon \mathcal{C}(a, x) \to Fx$$

We can now apply the Yoneda trick: substitute a for x:

$$\alpha_a \colon \mathcal{C}(a, a) \to Fa$$

and then pick the identity id_a as the canonical element of $\mathcal{C}(a, a)$. The result is an element $\alpha_a(id_a)$ in the set Fa. This defines a mapping in one direction, from natural transformations to elements of the set Fa.

Now the other way around. Given an element p of the set Fa we want to construct a natural transformation α. First, we assign p to be the action of α_a on id_a Ë $\mathcal{C}(a, a)$.



Now let's take an arbitrary object x and an arbitrary element of $C(a,x)$. The latter is an arrow $h\colon a \to x$. Our natural transformation must map it to an element of Fx. We can do it by lifting the arrow h using F. We get a function:

$$Fh\colon Fa \to Fx$$

We can apply this function to p and get an element of Fx. We take this element to be the action of α_x on h:

$$\alpha_x h = (Fh)p$$

The isomorphism in the Yoneda lemma is natural both in a and in F. The latter means that you can "move" from the functor F to another functor G by applying an arrow in the functor category, that is a natural transformation. This is quite a leap in the levels of abstraction, but all the definitions of functoriality and naturality work equally well in the functor category, where objects are functors, and arrows are natural transformations.

Exercise 9.6.1. *Fill in the gap in the proof when Fa is empty.*

Exercise 9.6.2. *Show that the mapping*

$$C(a,x) \to Fx$$

defined above is a natural transformation. Hint: Vary x using some $f : x \to y$.

Exercise 9.6.3. *Show that the formula for α_x can be derived from the assumption that $\alpha_a(id_a) = p$ and the naturality condition. Hint: The lifting of h by the hom-functor $C(a,h)$ is given by post-composition.*

Yoneda lemma in programming

Now for the trivial part: The proof of the Yoneda lemma translates directly to Haskell code. We start with the type of natural transformation between the hom-functor and some functor , and show that it's equivalent to the type of acting on .

We produce a value of the type using the standard Yoneda trick

Here's the inverse mapping:

Note that we are cheating a little by mixing types and sets. The Yoneda lemma in the present formulation works with Set-valued functors. Again, the correct incantation is to say that we use the enriched version of the Yoneda lemma in a self-enriched category.

The Yoneda lemma has some interesting applications in programming. For instance, let's consider what happens when we apply the Yoneda lemma to the identity functor. We get the isomorphism between the type (the identity functor acting on) and

We interpret this as saying that any data type can be replaced by a higher order polymorphic function. This function takes another function—called a handler, a callback, or a *continuation*—as an argument.

This is the standard continuation passing transformation that's used a lot in distributed programming, for instance when the value of type has to be retrieved from a remote server. It's also useful as a program transformation that turns recursive algorithms into tail-recursive functions.

Continuation-passing style is difficult to work with because the composition of continuations is highly nontrivial, resulting in what programmers often call a "callback hell." Fortunately continuations form a monad, which means their composition can be hidden behind the notation.

The contravariant Yoneda lemma

By reversing a few arrow, the Yoneda lemma can be applied to contravariant functors as well. It works on natural transformations between the contravariant hom-functor $C(*, a)$ and a contravariant functor F:

$$[C^{op}, \mathrm{Set}](C(*, a), F) \text{ ö } Fa$$

This is the Haskell implementation of the mapping:

And this is the inverse transformation:

9.7 Yoneda Embedding

In a closed category, we have exponential objects that serve as stand-ins for hom-sets. This is obviously a thing in Set, where hom-sets, being sets, are automatically objects in Set.

On the other hand, in the category of categories Cat, hom-sets are *sets* of functors, and it's not immediately obvious that they can be promoted to objects in Cat—that is, categories. But, as we've seen before, they can! Functors between any two categories form a functor *category*.

Because of that, it's possible to curry functors just like we curried functions. A functor from a product category can be viewed as a functor returning a functor. In other words, Cat is a closed (symmetric) monoidal category.

In particular, we can apply currying to the hom-functor $\mathcal{C}(a, b)$. It is a profunctor, or a functor from the product category:

$$\mathcal{C}^{op} \quad \mathcal{C} \to \mathrm{Set}$$

But it's also a contravariant functor in the first argument a. And for every a in $\mathcal{C}^{op}$ it produces a covariant functor $\mathcal{C}(a, *)$, which is an object in the functor category $[\mathcal{C}, \mathrm{Set}]$. We can write this mapping as:

$$\mathcal{C}^{op} \to [\mathcal{C}, \mathrm{Set}]$$

Alternatively, we can fix b and produce a contravariant functor $\mathcal{C}(*, b)$. This mapping can be written as

$$\mathcal{C} \to [\mathcal{C}^{op}, \mathrm{Set}]$$

Both mappings are functorial, which means that, for instance, an arrow in $\mathcal{C}$ is mapped to a natural transformation in $[\mathcal{C}^{op}, \mathrm{Set}]$.

These Set-valued functor categories are common enough that they have special names. The functors in $[\mathcal{C}^{op}, \mathrm{Set}]$ are called *presheaves*, and the ones in $[\mathcal{C}, \mathrm{Set}]$ are called *co-presheaves*. (The names come from algebraic topology.)

Let's focus our attention on the following reading of the hom-functor:

$$\mathcal{Y} \colon \mathcal{C} \to [\mathcal{C}^{op}, \mathrm{Set}]$$

It takes an object x and maps it to a presheaf

$$\mathcal{Y}_x = \mathcal{C}(*, x)$$

which can be visualized as the totality of views of x from all possible directions.

Let's also review its action on arrows. The functor $\mathcal{Y}$ lifts an arrow $f : x \to y$ to a mapping of presheaves:

$$\alpha \colon \mathcal{C}(*, x) \to \mathcal{C}(*, y)$$

The component of this natural transformation at some z is a function between hom-sets:

$$\alpha_z \colon \mathcal{C}(z, x) \to \mathcal{C}(z, y)$$

which is simply implemented as the post-composition $(f \circ *)$.

Such a functor $\mathcal{Y}$ can be thought of as creating a model of $\mathcal{C}$ in the presheaf category. But this is no run-of-the-mill model—it's an *embedding* of one category inside another. This particular one is called the *Yoneda embedding* and the functor $\mathcal{Y}$ is called the *Yoneda functor*.

First of all, every object of $\mathcal{C}$ is mapped to a *different* object (presheaf) in $[\mathcal{C}^{op}, \mathrm{Set}]$. We say that it's *injective on objects*.

But that's not all: every arrow in $\mathcal{C}$ is mapped to a *different* arrow. We say that the embedding functor is *faithful*.

If that weren't enough, the mapping of hom-sets is also *surjective*, meaning that every arrow between objects in $[\mathcal{C}^{op}, \mathrm{Set}]$ comes from some arrow in $\mathcal{C}$. We say that the functor is *full*.

Altogether, the embedding is *fully faithful*, that is the mapping of arrows is one-to-one. However, in general, the Yoneda embedding is *not* surjective on objects, hence the word "embedding."

The fact that the embedding is fully faithful is the direct consequence of the Yoneda lemma. Indeed, we know that, for any functor $F \colon \mathcal{C}^{op} \to \mathrm{Set}$, we have a natural isomorphism:

$$[\mathcal{C}^{op}, \mathrm{Set}](\mathcal{C}(*, x), F) \text{ ö } Fx$$

In particular, we can substitute another hom-functor $\mathcal{C}(*, y)$ for F to get:

$$[\mathcal{C}^{op}, \mathrm{Set}](\mathcal{C}(*, x), \mathcal{C}(*, y)) \text{ ö } \mathcal{C}(x, y)$$

The left-hand side is the hom-set in the presheaf category and the right-hand side is the hom-set in $\mathcal{C}$. They are isomorphic, which proves that the embedding is fully faithful. In fact, the Yoneda lemma tells us that the isomorphism is natural in x and y.

Let's have a closer look at this isomorphism. Let's pick an element of the right-hand set $\mathcal{C}(x, y)$— an arrow $f \colon x \to y$. The isomorphism maps it to a natural transformation whose component at z is a function:

$$\mathcal{C}(z, x) \to \mathcal{C}(z, y)$$

This mapping is implemented as post-composition $(f \circ *)$.

In Haskell, we would write it as:

In fact, this syntax works too:

The inverse mapping is:

(Notice the use of the Yoneda trick again.)

This isomorphism maps identity to identity and composition to composition. That's because it's implemented as post-composition, and post-composition preserves both identity and composition. We've shown this fact before, in the chapter on isomorphisms:

$$((f \circ g) \circ *) = (f \circ *) \circ (g \circ *)$$

Because it preserves composition and identity, this isomorphism also preserves *isomorphisms*. So if x is isomorphic to y then the presheaves $\mathcal{C}(*, x)$ and $\mathcal{C}(*, y)$ are isomorphic, and vice versa.

This is exactly the result that we've been using all along to prove numerous isomorphisms in previous chapters. If the hom-sets are naturally isomorphic, then the objects are isomorphic.

The Yoneda embedding builds on the idea that there is nothing more to an object than its relationships with other objects. The presheaf $\mathcal{C}(*, a)$, like a hologram, encodes the totality of views of a from the perspective of the whole category $\mathcal{C}$. The Yoneda embedding tells us that, when we combine all these individual holograms, we get a perfect hologram of the whole category.

9.8 Representable Functors

Objects in a co-presheaf category are functors that assign sets to objects in $\mathcal{C}$. Some of these functors work by picking a reference object a and assigning, to all objects x, their hom-sets $\mathcal{C}(a, x)$:

$$Fx = \mathcal{C}(a, x)$$

Such functors, and all the functors isomorphic to those, are called *representable*. The whole functor is "represented" by a single object a.

In a closed category, the functor which assigns to every object x the set of elements of the exponential object x^a is represented by a. This is because the set of elements of x^a is isomorphic to $\mathcal{C}(a, x)$:

$$\mathcal{C}(1, x^a) \text{ ö } \mathcal{C}(1 \quad a, x) \text{ ö } \mathcal{C}(a, x)$$

Seen this way, the representing object a is like a logarithm of a functor.

The analogy goes deeper: just like a logarithm of a product is a sum of logarithms, a representing object for a product data type is a sum. For instance, the functor that squares its argument using a product, $Fx = x \quad x$, is represented by 2, which is the sum $1 + 1$. Indeed, we've seen before that $x \quad x$ ö x^2.

Representable functors play a very special role in the category of Set-valued functors. Notice that the Yoneda embedding maps all objects of $\mathcal{C}$ to representable presheaves. It maps an object x to a presheaf represented by x:

$$\mathcal{Y}\colon x \mapsto \mathcal{C}(*, x)$$

We can find the entire category $\mathcal{C}$, objects and morphisms, embedded inside the presheaf category as representable functors. The question is, what else is there in the presheaf category "in between" representable functors?

Just like rational numbers are dense among real numbers, so representables are "dense" among (co-) presheaves. Every real number may be approximated by rational numbers. Every presheaf is a colimit of representables (and every co-presheaf, a limit). We'll come back to this topic when we talk about (co-) ends.

Exercise 9.8.1. *Describe limits and colimits as representing objects. What are the functors they represent?*

Exercise 9.8.2. *Consider a* singleton functor $F\colon \mathcal{C} \to \mathrm{Set}$ *that assigns to each object c a singleton set ^c‘ that contains just that object (that is, a different singleton for every object). Define the action of F on arrows. Show that F being representable is equivalent to C having an initial object.*

The guessing game

The idea that objects can be described by the way they interact with other objects is sometimes illustrated by playing an imaginary guessing game. One category theorist picks a secret object in a category, and the other has to guess which object it is (up to isomorphism, of course).

The guesser is allowed to point at objects, and use them as "probes" into the secret object. The opponent is supposed to respond each time with a set: the set of arrows from the probing object a to the secret object x. This, of course, is the hom-set $\mathcal{C}(a, x)$.

The totality of these answers, as long as the opponent is not cheating, will define a presheaf $F\colon \mathcal{C}^{op} \to \mathrm{Set}$, and the object they are hiding is its representing object.

But how do we know that the second category theorist in not cheating? To test that, we ask questions about arrows. For every arrow we select, they should give us a function between two sets—the sets they gave us for its endpoints. We can then check if all identity arrows are mapped to identity functions, and whether compositions of arrows map to compositions of functions. In other words, we'll be able to verify that F is indeed a functor.

However, a clever enough opponent may still fool us. The presheaf they are revealing to us may describe a fantastic object—a figment of their imagination—and we won't be able to tell. It turns out that such fantastic beasts are often as interesting as the real ones.

Representable functors in programming

In Haskell, we define a class of representable functors using two functions that witness the isomorphism:

$$Fx = \mathcal{C}(a, x)$$

The first one, , turns a function into a lookup table; and the second, , uses the representing type to index into it.

Algebraic data types that use sums are not representable (there is no formula for taking a logarithm of a sum). For instance, the list type is defined as a sum, so it's not representable. However, an infinite stream is.

Conceptually, a stream is like an infinite tuple, which is technically a product. Such a stream is represented by the type of natural numbers. In other words, an infinite stream is equivalent to a mapping out of natural numbers.

Here's the instance definition:

Representable types are useful for implementing memoization of functions.

Exercise 9.8.3. *Implement the instance for :*

Exercise 9.8.4. *Is the constant functor that maps everything to the terminal object representable? Hint: what's the logarithm of 1?*

In Haskell, such a functor could be implemented as:

Implement the instance of for it.

Exercise 9.8.5. *The list functor is not representable. But can it be considered a sum or representables?*

9.9 2-category Cat

In the category of categories, Cat, the hom-sets are not "just" sets. Each of them can be promoted to a functor category, with natural transformations playing the role of arrows. This kind of structure is called a 2-category.

In the language of 2-categories, objects are called 0-cells, arrows between them are called 1-cells, and arrows between arrows are called 2-cells.

The obvious generalization of that picture would be to have 3-cells that go between 2-cells and so on. An *n*-category has cells going up to the *n*-th level.

But why not have arrows going all the way? Enter infinity categories. Far from being a curiosity, Ø-categories have practical applications. For instance they are used in algebraic topology to describe points, paths between points, surfaces swiped by paths, volumes swiped by surfaces, and so on, ad infinitum.

9.10 Useful Formulas

- Yoneda lemma for covariant functors:

$$[\mathcal{C}, \mathrm{Set}](\mathcal{C}(a, *), F) \text{ ö } Fa$$

- Yoneda lemma for contravariant functors:

$$[\mathcal{C}^{op}, \mathrm{Set}](\mathcal{C}(*, a), F) \text{ ö } Fa$$

- Corollaries to the Yoneda lemma:

$$[\mathcal{C}, \mathrm{Set}](\mathcal{C}(x, *), \mathcal{C}(y, *)) \text{ ö } \mathcal{C}(y, x)$$
$$[\mathcal{C}^{op}, \mathrm{Set}](\mathcal{C}(*, x), \mathcal{C}(*, y)) \text{ ö } \mathcal{C}(x, y)$$

Chapter 10

Adjunctions

A sculptor subtracts irrelevant stone until a sculpture emerges. A mathematician abstracts irrelevant details until a pattern emerges.

We were able to define a lot of constructions using their mapping-in and mapping-out properties. Those, in turn, could be compactly written as isomorphisms between hom-sets. This pattern of natural isomorphisms between hom-sets is called an adjunction and, once recognized, pops up virtually everywhere.

10.1 The Currying Adjunction

The definition of the exponential is the classic example of an adjunction that relates mappings-out and mappings-in. Every mapping out of a product corresponds to a unique mapping into the exponential:

$$\mathcal{C}(e \quad a, b) \text{ ö } \mathcal{C}(e, b^a)$$

The object b takes the role of the focus on the left hand side; the object e becomes the observer on the right hand side.

We can spot two functors at play. They are both parameterized by a. On the left we have the product functor $(* \quad a)$ applied to e. On the right we have the exponential functor $(*)^a$ applied to b.

If we write these functors as:

$$L_a e = e \quad a$$

$$R_a b = b^a$$

then the natural isomorphism

$$\mathcal{C}(L_a e, b) \text{ ö } \mathcal{C}(e, R_a b)$$

is called the adjunction between them.

In components, this isomorphism tells us that, given a mapping $\phi \in \mathcal{C}(L_a e, b)$, there is a unique mapping $\phi^T \in \mathcal{C}(e, R_a b)$ and vice versa. These mappings are sometimes called the *transpose* of each other—the nomenclature taken from matrix algebra.

The shorthand notation for the adjunction is $L \dashv R$. Substituting the product functor for L and the exponential functor for R, we can write the currying adjunction concisely as:

$$(- \times a) \dashv (-)^a$$

The exponential object b^a is sometimes called the *internal hom* and is written as $[a, b]$. This is in contrast to the *external hom*, which is the set $\mathcal{C}(a, b)$. The external hom is *not* an object in $\mathcal{C}$ (except when $\mathcal{C}$ itself is Set). With this notation, the currying adjunction can be written as:

$$\mathcal{C}(e \times a, b) \cong \mathcal{C}(e, [a, b])$$

A category in which this adjunction holds is called cartesian closed.

Since functions play central role in every programming language, cartesian closed categories form the basis of all models of programming. We interpret the exponential b^a as the function type $a \to b$.

Here e plays the role of the external environment—the Γ of the lambda calculus. The morphism in $\mathcal{C}(\Gamma \times a, b)$ is interpreted as an expression of type b in the environment Γ extended by a variable of type a. The function type $a \to b$ therefore represents a closure that may capture a value of type e from its environment.

Incidentally, the category of (small) categories Cat is also cartesian closed, as reflected in this adjunction between product categories and functor categories that uses the same internal-hom notation:

$$\mathrm{Cat}(\mathcal{A} \times \mathcal{B}, \mathcal{C}) \cong \mathrm{Cat}(\mathcal{A}, [\mathcal{B}, \mathcal{C}])$$

Here, both sides are sets of natural transformations.

10.2 The Sum and the Product Adjunctions

The currying adjunction relates two endofunctors, but an adjunction can be easily generalized to functors that go between different categories. Let's see some examples first.

The diagonal functor

The sum and the product types were defined using bijections where one of the sides was a single arrow and the other was a pair of arrows. A pair of arrows can be seen as a single arrow in the product category.

To explore this idea, we need to define the diagonal functor Δ, which is a special mapping from $\mathcal{C}$ to $\mathcal{C} \times \mathcal{C}$. It takes an object x and duplicates it, producing a pair of objects $\langle x, x \rangle$. It also takes an arrow f and duplicates it $\langle f, f \rangle$.

Interestingly, the diagonal functor is related to the constant functor we've seen previously. The constant functor can be though of as a functor of two variables—it just ignores the second one. We've seen this in the Haskell definition:

To see the connection, let's look at the product category $\mathcal{C} \times \mathcal{C}$ as a functor category $[2, \mathcal{C}]$, in other words, the exponential object $\mathcal{C}^2$ in Cat. Indeed, a functor from 2 (the stick-figure category with two objects) picks a pair of objects—which is equivalent to a single object in the product category.

A functor $\mathcal{C} \to [2, \mathcal{C}]$ can be uncurried to $\mathcal{C} \times 2 \to \mathcal{C}$. The diagonal functor ignores the second argument, the one coming from 2: it does the same thing whether the second argument is 1 or 2. That's exactly what the constant functor does as well. This is why we use the same symbol Δ for both.

Incidentally, this argument can be easily generalized to any indexing category, not just 2.

The sum adjunction

Recall that the sum is defined by its mapping out property. There is a one-to one correspondence between the arrows coming out of the sum $a + b$ and pairs of arrows coming from a and b separately. In terms of hom-sets, we can write it as:

$$\mathcal{C}(a + b, x) \cong \mathcal{C}(a, x) \times \mathcal{C}(b, x)$$

where the product on the right-hand side is just a cartesian product of sets, that is the set of pairs. Moreover, we've seen earlier that this bijection is natural in x.

We know that a pair of arrows is a single arrow in the product category. We can, therefore, look at the elements on the right-hand side as arrows in $\mathcal{C} \times \mathcal{C}$ going from the object $\langle a, b \rangle$ to the object $\langle x, x \rangle$. The latter can be obtained by acting with the diagonal functor Δ on x. We have:

$$\mathcal{C}(a + b, x) \cong (\mathcal{C} \times \mathcal{C})(\langle a, b \rangle, \Delta x)$$

This is a bijection between hom-sets in two different categories. It satisfies naturality conditions, so it's a natural isomorphism.

We can spot a pair of functors here as well. On the left we have the functor that takes a pair of objects $\langle a, b \rangle$ and produces their sum $a + b$:

$$(+) \colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$$

On the right-hand side, we have the diagonal functor Δ going in the opposite direction:

$$\Delta\colon \mathcal{C} \to \mathcal{C} \quad \mathcal{C}$$

Altogether, we have a pair of functors between a pair of categories:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend right, "\Delta"'] & \mathcal{C} \quad \mathcal{C} \arrow[l, bend right, "(+)"']
\end{tikzcd}$$

and an isomorphism between the hom-sets:

$$\begin{tikzcd}
a + b \arrow[d, dashed, red, bend right] \arrow[d, dashed] \arrow[d, dashed, blue, bend left] & & \langle a, b \rangle \arrow[ll, bend right, "(+)"'] \arrow[d, dashed, red, bend right] \arrow[d, dashed] \arrow[d, dashed, blue, bend left] \\
x \arrow[rr, bend right, "\Delta"'] & & \langle x, x \rangle
\end{tikzcd}$$

In other words, we have the adjunction:

$$(+) \dashv \Delta$$

The product adjunction

We can apply the same reasoning to the definition of a product. This time we have a natural isomorphism between pairs of arrows and a mapping *into* the product.

$$\mathcal{C}(x, a) \quad \mathcal{C}(x, b) \text{ ö } \mathcal{C}(x, a \quad b)$$

Replacing pairs of arrows with arrows in the product category we get:

$$(\mathcal{C} \quad \mathcal{C})(\Delta x, \langle a, b \rangle) \text{ ö } \mathcal{C}(x, a \quad b)$$

These are the two functors going in the opposite direction:

$$\begin{tikzcd}
\mathcal{C} \quad \mathcal{C} \arrow[r, bend right, "(\quad)"'] & \mathcal{C} \arrow[l, bend right, "\Delta"']
\end{tikzcd}$$

and this is the isomorphism of hom-sets:

$$\begin{tikzcd}
\langle x, x \rangle \arrow[d, dashed, red, bend right] \arrow[d, dashed] \arrow[d, dashed, blue, bend left] & & x \arrow[ll, bend right, "\Delta"'] \arrow[d, dashed, red, bend right] \arrow[d, dashed] \arrow[d, dashed, blue, bend left] \\
\langle a, b \rangle \arrow[rr, bend right, "(\quad)"'] & & a \quad b
\end{tikzcd}$$

In other words, we have the adjunction:

$$\Delta \dashv (\ )$$

Distributivity

In a bicartesian closed category products distribute over sums. We've seen one direction of the proof using universal constructions. Adjunctions combined with the Yoneda lemma give us more powerful tools to tackle this problem.

We want to show the natural isomorphism:

$$(b+c)\quad a \text{ ö } b\quad a+c\quad a$$

Instead of proving this identity directly, we'll show that the mappings out from both sides to an arbitrary object x are isomorphic:

$$\mathcal{C}((b+c)\quad a, x) \text{ ö } \mathcal{C}(b\quad a+c\quad a, x)$$

The left hand side is a mapping out of a product, so we can apply the currying adjunction to it:

$$\mathcal{C}((b+c)\quad a, x) \text{ ö } \mathcal{C}(b+c, x^a)$$

This gives us a mapping out of a sum which, by the sum adjunction is isomorphic to the product of two mappings:

$$\mathcal{C}(b+c, x^a) \text{ ö } \mathcal{C}(b, x^a)\quad \mathcal{C}(c, x^a)$$

We can now apply the inverse of the currying adjunction to both components:

$$\mathcal{C}(b, x^a)\quad \mathcal{C}(c, x^a) \text{ ö } \mathcal{C}(b\quad a, x)\quad \mathcal{C}(c\quad a, x)$$

Using the inverse of the sum adjunction, we arrive at the final result:

$$\mathcal{C}(b\quad a, x)\quad \mathcal{C}(c\quad a, x) \text{ ö } \mathcal{C}(b\quad a+c\quad a, x)$$

Every step in this proof was a natural isomorphism, so their composition is also a natural isomorphism. By Yoneda lemma, the two objects that form the left- and the right-hand side of distributivity law are therefore isomorphic.

A much shorter proof of this statement follows from the property of left adjoints that we'll discuss soon.

10.3 Adjunction between functors

In general, an adjunction relates two functors going in opposite directions between two categories. The left functor

$$L\colon \mathcal{D} \to \mathcal{C}$$

and the right functor:

$$R\colon \mathcal{C} \to \mathcal{D}$$

The adjunction $L \dashv R$ is defined as a natural isomorphism between two hom-sets.

$$\mathcal{C}(Lx, y) \text{ ö } \mathcal{D}(x, Ry)$$

In other words, we have a family of invertible functions between sets:

$$\phi_{xy}\colon \mathcal{C}(Lx, y) \to \mathcal{D}(x, Ry)$$

natural in both x and y. For instance, naturality in y means that, for any $f\colon\ y \to y^{\ddot{}}$ the following diagram commutes:

$$\begin{tikzcd}
\mathcal{C}(Lx, y) \arrow[r, "\mathcal{C}(Lx{,}f)"] \arrow[d, leftrightarrow, "\phi_{xy}"] & \mathcal{C}(Lx, y^{\ddot{}}) \arrow[d, leftrightarrow, "\phi_{xy^{\ddot{}}}"] \\
\mathcal{D}(x, Ry) \arrow[r, "\mathcal{D}(x{,}Rf)"] & \mathcal{D}(x, Ry^{\ddot{}})
\end{tikzcd}$$

or, considering that a lifting of arrows by hom-functors is the same as post-composition:

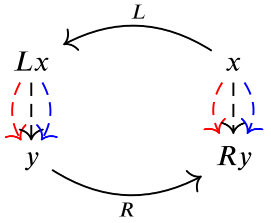
$$\begin{tikzcd}
\mathcal{C}(Lx, y) \arrow[r, "f \circ *"] \arrow[d, leftrightarrow, "\phi_{xy}"] & \mathcal{C}(Lx, y^{\ddot{}}) \arrow[d, leftrightarrow, "\phi_{xy^{\ddot{}}}"] \\
\mathcal{D}(x, Ry) \arrow[r, "Rf \circ *"] & \mathcal{D}(x, Ry^{\ddot{}})
\end{tikzcd}$$

The double-headed arrows can be traversed in either direction (using ϕ_{xy}^{*1} when going up), since they are the components of an isomorphism.

Pictorially, we have two functors:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend right, "R"'] & \mathcal{D} \arrow[l, bend right, "L"']
\end{tikzcd}$$

and, for any pair x and y, two isomorphic hom-sets:



These hom-sets come from two different categories, but sets are just sets. We say that L is the left adjoint of R, or that R is the right adjoint of L

In Haskell, the simplified version of this could be encoded as a multi-parameter type class:

It requires the following pragma at the top of the file:

Therefore, in a bicartesian category, the sum is the left adjoint to the diagonal functor; and the product is its right adjoint. We can write this very concisely (or we could impress it in clay, in a modern version of cuneiform):

$$(+) \dashv \Delta \dashv (\;)$$

Exercise 10.3.1. *Draw the commuting square witnessing the naturality of the adjunction function ϕ_{xy} in x.*

Exercise 10.3.2. *The hom-set $\mathcal{C}(Lx, y)$ on the left-hand side of the adjunction formula suggests that Lx could be seen as a representing object for some functor (a co-presheaf). What is this functor? Hint: It maps y to a set. What set is it?*

Exercise 10.3.3. *Conversely, a representing object a for a presheaf P is defined by:*

$$Px \text{ ö } \mathcal{D}(x, a)$$

What is the presheaf for which Ry, in the adjunction formula, is the representing object.

10.4 Limits and Colimits as Adjunctions

The definition of a limit also involves a natural isomorphism between hom-sets:

$$[\mathcal{J}, \mathcal{C}](\Delta_x, D) \text{ ö } \mathcal{C}(x, \mathrm{Lim} D)$$

The hom-set on the left is in the functor category. Its elements are cones, or natural transformations between the constant functor Δ_x and the diagram functor D. The one on the right is a hom-set in $\mathcal{C}$.

In a category where all limits exist, we have the adjunction between these two functors:

$$\Delta_{(*)} : \mathcal{C} \to [\mathcal{J}, \mathcal{C}]$$

and:

$$\mathrm{Lim}(*) : [\mathcal{J}, \mathcal{C}] \to \mathcal{C}$$

Dually, the colimit is described by the following natural isomorphism:

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x) \text{ ö } \mathcal{C}(\mathrm{Colim} D, x)$$

We can write both adjunctions using one terse formula:

$$\mathrm{Colim} \dashv \Delta \dashv \mathrm{Lim}$$

In particular, since the product category $\mathcal{C} \quad \mathcal{C}$ is equivalent to $\mathcal{C}^2$, or the functor category $[2, \mathcal{C}]$, we can rewrite a product and a coproduct as a limit and a colimit:

$$[2, \mathcal{C}](\Delta_x, \langle a, b\rangle) \text{ ö } \mathcal{C}(x, a \quad b)$$

$$\mathcal{C}(a + b, x) \text{ ö } [2, \mathcal{C}](\langle a, b\rangle, \Delta_x)$$

where $\langle a, b\rangle$ denotes a diagram that is the action of a functor $D\colon 2 \to \mathcal{C}$ on the two objects of 2.

10.5 Unit and Counit of an Adjunction

We compare arrows for equality, but we prefer to use isomorphisms for comparing objects.

We have a problem when it comes to functors, though. On the one hand, they are objects in the functor category, so isomorphisms are the way to go; on the other hand, they are arrows in Cat so maybe it's okay to compare them for equality?

To shed some light on this dilemma, we should ask ourselves *why* we use equality for arrows. It's not because we like equality, but because there's nothing else for us to do in a set but to compare elements for equality. Two elements of a hom-set are either equal or not, period.

That's not the case in Cat which, as we know, is a 2-category. Here, hom-sets themselves have the structure of a category—the functor category. In a 2-category we have arrows between arrows so, in particular, we can define isomorphisms between arrows. In Cat these would be natural isomorphisms between functors.

However, even though we have the option of replacing arrow equalities with isomorphisms, categorical laws in Cat are still expressed as equalities. For instance, the composition of a functor F with the identity functor is *equal* to F, and the same for associativity. A 2-category in which the laws are satisfied "on the nose" is called *strict*, and Cat is an example of a strict 2-category.

But as far as comparing categories goes, we have more options. Categories are objects in Cat, so it's possible to define an isomorphism of categories as a pair of functors L and R:

$$\begin{tikzcd}
\mathcal{C} \arrow[loop above, "\mathrm{Id}_{\mathcal{C}}"] \arrow[r, bend right, "R"'] & \mathcal{D} \arrow[loop above, "\mathrm{Id}_{\mathcal{D}}"] \arrow[l, bend right, "L"']
\end{tikzcd}$$

such that:

$$L \circ R = \mathrm{Id}_{\mathcal{C}}$$
$$\mathrm{Id}_{\mathcal{D}} = R \circ L$$

This definition involves equality of functors, though. What's worse, acting on objects, it involves equality of objects:

$$L(Rx) = x$$
$$y = R(Ly)$$

This is why it's more proper to talk about a weaker notion of *equivalence* of categories, where equalities are replaced by isomorphisms:

$$L \circ R \text{ ö } \mathrm{Id}_{\mathcal{C}}$$
$$\mathrm{Id}_{\mathcal{D}} \text{ ö } R \circ L$$

On objects, an equivalence of categories means that a round trip produces an object that is isomorphic, rather than equal, to the original one. In most cases, this is exactly what we want.

An adjunction is also defined as a pair of functors going in opposite directions, so it makes sense to ask what the result of a round trip is.

The isomorphism that defines an adjunction works for any pair of objects x and y

$$\mathcal{C}(Lx, y) \text{ ö } \mathcal{D}(x, Ry)$$

so, in particular, it works if we replace y with Lx

$$\mathcal{C}(Lx, Lx) \text{ ö } \mathcal{D}(x, R(Lx))$$

We can now use the Yoneda trick and pick the identity arrow id_{Lx} on the left. The isomorphism maps it to a unique arrow on the right, which we'll call η_x:

$$\eta_x \colon x \to R(Lx)$$

Not only is this mapping defined for every x, but it's also natural in x. The natural transformation η is called the *unit* of the adjunction. If we observe that the x on the left is the action of the identity functor on x, we can write:

$$\eta \colon \mathrm{Id}_{\mathcal{D}} \to R \circ L$$

As an example, let's evaluate the unit of the coproduct adjunction:

$$\mathcal{C}(a + b, x) \text{ ö } (\mathcal{C} \quad \mathcal{C})(\langle a, b \rangle, \Delta x)$$

by replacing x with $a+b$. We get:

$$\eta_{\langle a,b\rangle} \colon \langle a, b\rangle \to \Delta(a+b)$$

This is a pair of arrows that are exactly the two injections $\langle \text{Left}, \text{Right}\rangle$.

We can do a similar trick by replacing x with Ry:

$$\mathcal{C}(L(Ry), y) \text{ ö } \mathcal{D}(Ry, Ry)$$

Corresponding to id_{Ry} on the right, we get an arrow on the left:

$$\varepsilon_y \colon L(Ry) \to y$$

These arrows form another natural transformation called the *counit* of the adjunction:

$$\varepsilon \colon L \circ R \to \mathrm{Id}_{\mathcal{C}}$$

Notice that, if those two natural transformations were invertible, they would witness the equivalence of categories. But even if they're not, this kind of "half-equivalence" is still very interesting in the context of category theory.

As an example, let's evaluate the counit of the product adjunction:

$$(\mathcal{C}\quad\mathcal{C})(\Delta x, \langle a, b\rangle) \text{ ö } \mathcal{C}(x, a\quad b)$$

by replacing x with $a\quad b$. We get:

$$\varepsilon_{\langle a,b\rangle} \colon \Delta(a\quad b) \to \langle a, b\rangle$$

This is a pair of arrows that are exactly the two projections $\langle \text{fst}, \text{snd}\rangle$.

Exercise 10.5.1. *Derive the counit of the coproduct adjunction and the unit of the product adjunction.*

Triangle identities

We can use the unit/counit pair to formulate an equivalent definition of an adjunction. To do that, we start with a pair of natural transformations:

$$\eta \colon \mathrm{Id}_{\mathcal{D}} \to R \circ L$$
$$\varepsilon \colon L \circ R \to \mathrm{Id}_{\mathcal{C}}$$

and impose additional *triangle identities*.

These identities can be derived from the standard definition of the adjunction by noticing that η can be used to replace an identity functor with the composite $R \circ L$, effectively letting us insert $R \circ L$ anywhere an identity functor would work.

Similarly, ε can be used to eliminate the composite $L \circ R$ (i.e., replace it with identity). So, for instance, starting with L:

$$L = L \circ \mathrm{Id}_{\mathcal{D}} \xrightarrow{L \circ \eta} L \circ R \circ L \xrightarrow{\varepsilon \circ L} \mathrm{Id}_{\mathcal{C}} \circ L = L$$

Here, we used the horizontal composition of natural transformation, with one of them being the identity transformation (a.k.a., whiskering).

The first triangle identity is the condition that this chain of transformations result in the identity natural transformation. Pictorially:

$$\begin{tikzcd}
L \arrow[r, "L \circ \eta"] \arrow[dr, "id_L"'] & L \circ R \circ L \arrow[d, "\varepsilon \circ L"] \\
& L
\end{tikzcd}$$

Similarly, we want the following chain of natural transformations to also compose to identity:

$$R = \mathrm{Id}_{\mathcal{D}} \circ R \xrightarrow{\eta \circ R} R \circ L \circ R \xrightarrow{R \circ \varepsilon} R \circ \mathrm{Id}_{\mathcal{C}} = R$$

or, pictorially:

$$\begin{tikzcd}
R \arrow[r, "\eta \circ R"] \arrow[dr, "id_R"'] & R \circ L \circ R \arrow[d, "R \circ \varepsilon"] \\
& R
\end{tikzcd}$$

It turns out that an adjunction can be alternatively defined in terms of the two natural transformations, η and ε, satisfying the triangle identities:

$$(\varepsilon \circ L) \cdot (L \circ \eta) = id_L$$

$$(R \circ \varepsilon) \cdot (\eta \circ R) = id_R$$

From those, the mapping of hom-sets can be easily recovered. For instance, let's start with an arrow $f \colon x \to Ry$, which is an element of $\mathcal{D}(x, Ry)$. We can lift it to

$$Lf \colon Lx \to L(Ry)$$

We can then use η to collapse the composite $L \circ R$ to identity. The result is an arrow $Lx \to y$, which is an element of $\mathcal{C}(Lx, y)$.

The definition of the adjunction using unit and counit is more general in the sense that it can be translated to an arbitrary 2-category setting.

Exercise 10.5.2. *Given an arrow $g \colon Lx \to y$ implement an arrow $x \to Ry$ using ε and the fact that R is a functor. Hint: Start with the object x and see how you can get from there to Ry with one stopover.*

The unit and counit of the currying adjunction

Let’s calculate the unit and the counit of the currying adjunction:

$$\mathcal{C}(e \quad a, b) \text{ ö } \mathcal{C}(e, b^a)$$

If we replace b with $e \quad a$, we get

$$\mathcal{C}(e \quad a, e \quad a) \text{ ö } \mathcal{C}(e, (e \quad a)^a)$$

Corresponding to the identity arrow on the left, we get the unit of the adjunction on the right:

$$\eta \colon e \to (e \quad a)^a$$

This is a curried version of the product constructor. In Haskell, we write it as:

The counit is more interesting. Replacing e with b^a we get:

$$\mathcal{C}(b^a \quad a, b) \text{ ö } \mathcal{C}(b^a, b^a)$$

Corresponding to the identity arrow on the right, we get:

$$\varepsilon \colon b^a \quad a \to b$$

which is the function application arrow.

In Haskell:

When the adjunction is between two endofunctors, we can write an alternative Haskell definition of it using the unit and the counit:

The additional two clauses and tell the compiler that, when using an instance of the adjunction, one functor can be derived from the other. This definition requires the following compile extensions:

The two functors that form the currying adjunction can be written as:

and the instance for currying is:

The first triangle identity states that the following polymorphic function:

is the identity, and so is the second one:

Notice that these two functions require the use of functional dependencies to be well-defined. Triangle identities cannot be expressed in Haskell, so it's up to the implementor of the adjunction to prove them.

Exercise 10.5.3. *Test a few examples of the first triangle identity for the currying adjunction. Here's an example:*

Exercise 10.5.4. *How would you test the second triangle identity for the currying adjunction? Hint: the result of is a function, so you can't display it, but you could call it.*

10.6 Adjunctions Using Universal Arrows

We've seen the definition of an adjunction using the isomorphism of hom-sets, and another one using the pair of unit/counit. It turns out that we can define an adjunction using just one element of this pair, as long as it satisfies certain universality condition. To see that, we will construct a new category whose objects are arrows.

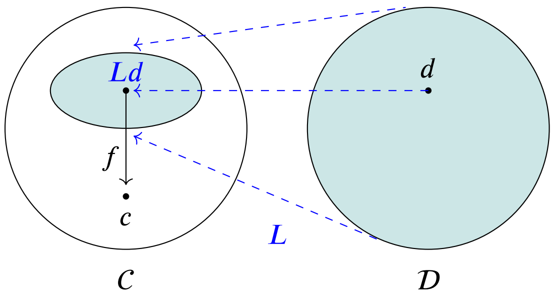
We've seen before an example of such a category—the slice category $\mathcal{C}/c$ that collects all the arrows that converge on c. Such a category describes the view of the object c from every possible angle in $\mathcal{C}$.

Comma category

When dealing with an adjunction:

$$\mathcal{C}(Ld, c) \text{ ö } \mathcal{D}(d, Rc)$$

we are observing the object c from a narrower perspective defined by the functor L. Think of L as defining a model of the category $\mathcal{D}$ inside $\mathcal{C}$. We are interested in the view of c from the perspective of this model. The arrows that describe this view form the comma category L/c.



An object in the *comma category* L/c is a pair $\langle d, f\rangle$, where d is an object of $\mathcal{D}$ and $f \colon Ld \to c$ is an arrow in $\mathcal{C}$.

A morphism from $\langle d, f\rangle$ to $\langle d'', f''\rangle$ is an arrow $h\colon d \to d''$ that makes the diagram on the left commute:

$$\begin{tikzcd}
Ld \arrow[rr, "Lh"] \arrow[dr, "f"'] & & Ld'' \arrow[dl, "f''"] \\
& c &
\end{tikzcd}
\qquad
\begin{tikzcd}
d \arrow[r, "h"] & d''
\end{tikzcd}$$

Universal arrow

The universal arrow from L to c is defined as the terminal object in the comma category L/c. Let's unpack this definition. The terminal object in L/c is a pair $\langle t, \tau\rangle$ with a unique morphism from any object $\langle d, f\rangle$. Such a morphism is an arrow $h\colon d \to t$ that satisfies the commuting condition:

$$\begin{tikzcd}
Ld \arrow[rr, dashed, "Lh"] \arrow[dr, "f"'] & & Lt \arrow[dl, "\tau"] \\
& c &
\end{tikzcd}$$

In other words, for any f in the hom-set $\mathcal{C}(Ld, c)$ there is a unique element h in the hom-set $\mathcal{D}(d, t)$ such that:

$$f = \tau \circ Lh$$

Such a one-to-one mapping between elements of two hom-sets hints at the underlying adjunction.

Universal arrows from adjunctions

Let's first convince ourselves that, when the functor L has a right adjoint R, then for every c there exists a universal arrow from L to c. Indeed, this arrow is given by the pair $\langle Rc, \varepsilon_c\rangle$, where ε is the counit of the adjunction. First of all, the component of the counit has the right signature for the object in the comma category L/c:

$$\varepsilon_c \colon L(Rc) \to c$$

We'd like to show that $\langle Rc, \varepsilon_c\rangle$ is the terminal object in L/c. That is, for any object $\langle d, f \colon Ld \to c\rangle$ there is a unique $h\colon d \to Rc$ such that $f = \varepsilon_c \circ Lh$:

$$\begin{tikzcd}
Ld \arrow[rr, dashed, "Lh"] \arrow[dr, "f"'] & & L(Rc) \arrow[dl, "\varepsilon_c"] \\
& c &
\end{tikzcd}$$

To prove this, let's write one of the naturality conditions for ϕ_{dc} as the function of d:

$$\phi_{dc} \colon \mathcal{C}(Ld, c) \to \mathcal{D}(d, Rc)$$

For any arrow $h \colon d \to d^{\ddot{}}$ the following diagram must commute:

$$\begin{tikzcd}
\mathcal{C}(Ld^{\ddot{}}, c) \arrow[r, "* \circ Lh"] \arrow[d, leftrightarrow, "\phi_{d^{\ddot{}}, c}"] & \mathcal{C}(Ld, c) \arrow[d, leftrightarrow, "\phi_{d,c}"] \\
\mathcal{D}(d^{\ddot{}}, Rc) \arrow[r, "* \circ h"] & \mathcal{D}(d, Rc)
\end{tikzcd}$$

We can use the Yoneda trick by setting $d^{\ddot{}}$ to Rc.

$$\begin{tikzcd}
\mathcal{C}(L(Rc), c) \arrow[r, "* \circ Lh"] \arrow[d, leftrightarrow, "\phi_{Rc,c}"] & \mathcal{C}(Ld, c) \arrow[d, leftrightarrow, "\phi_{d,c}"] \\
\mathcal{D}(Rc, Rc) \arrow[r, "* \circ h"] & \mathcal{D}(d, Rc)
\end{tikzcd}$$

We can now pick the special element of the hom-set $\mathcal{D}(Rc, Rc)$, namely the identity arrow id_{Rc} and propagate it through the rest of the diagram. The upper left corner becomes ε_c, the lower right corner becomes h, and the upper right corner becomes the adjoint to h, which we called f:

$$\begin{tikzcd}
\varepsilon_c \arrow[r, mapsto, "* \circ Lh"] \arrow[d, leftrightarrow, "\phi_{Rc,c}"] & f \arrow[d, leftrightarrow, "\phi_{d,c}"] \\
id_{Rc} \arrow[r, mapsto, "* \circ h"] & h
\end{tikzcd}$$

The upper arrow then gives us the sought after equality $f = (* \circ Lh)\varepsilon_c = \varepsilon_c \circ Lh$.

Adjunction from universal arrows

The converse result is even more interesting. If, for every c, we have a universal arrow from L to c, that is a terminal object $\langle t_c, \varepsilon_c \rangle$ in the comma category L/c, then we can construct a functor R that is the right adjoint to L. The action of this functor on objects is given by $Rc = t_c$, and the family ε_c is automatically natural in c, and it forms the counit of the adjunction.

There is also a dual statement: An adjunction can be constructed starting from a family of universal arrows η_d, which form initial objects in the comma category d/R.

These results will help us prove the Freyd's adjoint functor theorem.

10.7 Properties of Adjunctions

Left adjoints preserve colimits

We defined colimits as universal cocones. For every cocone—that is a natural transformation from the diagram $D \colon \mathcal{J} \to \mathcal{C}$ to the constant functor Δ_x—there's supposed to be a unique factorizing morphism from the colimit Colim D to x. This condition can be written as a one-to-one correspondence between the set of cocones and a particular hom-set:

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x) \text{ ö } \mathcal{C}(\text{Colim } D, x)$$

The factorizing condition is encoded in the naturality of this isomorphism.

It turns out that the set of cocones, which is an object in Set, is itself a *limit* of the following Set-valued functor $F \colon \mathcal{J} \to \mathrm{Set}$:

$$F j = \mathcal{C}(D j, x)$$

To show this, we'll start with the limit of F and end up with the set of cocones. You may recall that a limit of a Set-valued functor is equal to a set of cones with the apex 1 (the singleton set). In our case, each such cone describes a selection of morphisms from the corresponding hom-set $\mathcal{C}(D j, x)$:

$$\begin{tikzcd}
& 1 \arrow[ld] \arrow[rd] \arrow[dd] & \\
\mathcal{C}(Dj_1, x) \arrow[rr] \arrow[rd] & & \mathcal{C}(Dj_2, x) \arrow[ld] \\
& \mathcal{C}(Dj_3, x) &
\end{tikzcd}$$

Each of these morphisms has as target the same object x, so they form the sides of a cocone with the apex x.

$$\begin{tikzcd}
Dj_1 \arrow[rr] \arrow[rd] \arrow[rdd] & & Dj_2 \arrow[ld] \arrow[ldd] \\
& Dj_3 \arrow[d] & \\
& x &
\end{tikzcd}$$

The commuting conditions for the cone with the apex 1 are simultaneously the commuting condition for this cocone with the apex x. But these are exactly the cocones in the set $[\mathcal{J}, \mathcal{C}](D, \Delta_x)$.

We can therefore replace the original set of cocones with the limit of $\mathcal{C}(D^*, x)$ to get:

$$\text{Lim } \mathcal{C}(D^*, x) \text{ ö } \mathcal{C}(\text{Colim } D, x)$$

The contravariant hom-functor is sometimes notated as:

$$h_x = \mathcal{C}(*, x)$$

In this notation we can write:

$$Lim\,(h_x \circ D) \text{ ö } h_x(Colim\,D)$$

The limit of a hom-functor acting on a diagram D is isomorphic to the hom-functor acting on a colimit of this diagram. This is usually abbreviated to: The hom-functor preserves colimits. (With the understanding that the contravariant hom-functor turns colimits into limits.)

A functor that preserves colimits is called co-continuous. Thus the contravariant hom-functor is co-continuous.

Now suppose that we have the adjunction $L \dashv R$, where $L\colon \mathcal{C} \to \mathcal{D}$ and R goes in the opposite direction. We want to show that the left functor L preserves colimits, that is:

$$L(\text{Colim}\,D) \text{ ö } \text{Colim}(L \circ D)$$

for any diagram $D\colon \mathcal{J} \to \mathcal{C}$ for which the colimit exists.

We'll use the Yoneda lemma to show that the mappings out of both sides to an arbitrary x are isomorphic:

$$\mathcal{D}(L(\text{Colim}\,D), x) \text{ ö } \mathcal{D}(\text{Colim}(L \circ D), x)$$

We apply the adjunction to the left hand side to get:

$$\mathcal{D}(L(\text{Colim}\,D), x) \text{ ö } \mathcal{C}(\text{Colim}\,D, Rx)$$

Preservation of colimits by the hom-functor gives us:

$$\text{ö Lim } \mathcal{C}(D^*, Rx)$$

Using the adjunction again, we get:

$$\text{ö Lim } \mathcal{D}((L \circ D)^*, x)$$

And the second application of preservation of colimits gives us the desired result:

$$\text{ö } \mathcal{D}((\text{Colim }(L \circ D), x)$$

Since this is true for any x, we get our result.

We can use this result to reformulate our earlier proof of distributivity in a cartesian closed category. We use the fact that the product is the left adjoint of the exponential. Left adjoints preserve colimits. A coproduct is a colimit, therefore:

$$(b + c) \quad a \text{ ö } b \quad a + c \quad a$$

Here, the left functor is $Lx = x \quad a$, and the diagram D selects a pair of objects b and c.

Right adjoints preserve limits

Using a dual argument, we can show that right adjoints preserve limits, that is:

$$R(\text{Lim}\, D) \text{ ö } \text{Lim}\,(R \circ D)$$

We start by showing that the (covariant) hom-functor preserves limits.

$$\text{Lim}\; \mathcal{C}(x, D^*) \text{ ö } \mathcal{C}(x, \text{Lim}\, D)$$

This follows from the argument that a set of cones that defines the limit is isomorphic to the limit of the $\mathbf{Set}$-valued functor:

$$F j = \mathcal{C}(x, D j)$$

A functor that preserves limits is called continuous.

To show that, given the adjunction $L \dashv R$, the right functor $R \colon \mathcal{D} \to \mathcal{C}$ preserves limits, we use the Yoneda argument:

$$\mathcal{C}(x, R(\text{Lim}\, D)) \text{ ö } \mathcal{C}(x, \text{Lim}\,(R \circ D))$$

Indeed, we have:

$$\mathcal{C}(x, R(\text{Lim}\, D)) \text{ ö } \mathcal{D}(Lx, \text{Lim}\, D) \text{ ö } \text{Lim}\; \mathcal{D}(Lx, D^*) \text{ ö } \mathcal{C}(x, \text{Lim}\,(R \circ D))$$

10.8 Freyd's adjoint functor theorem

In general functors are lossy—they are not invertible. In some cases we can make up the lost information by replacing it with the "best guess." If we do it in an organized manner, we end up with an adjunction. The question is: given a functor between two categories, what are the conditions under which we can construct its adjoint.

The answer to this question is given by the Freyd's adjoint functor theorem. At first it might seem like this is a technical theorem involving a very abstract construction called the solution set condition. We'll see later that this condition translates directly to a programming technique called defunctionalization.

In what follows, we'll focus our attention on constructing the right adjoint to a functor $L \colon \mathcal{D} \to \mathcal{C}$. A dual reasoning can be used to solve the converse problem of finding the left adjoint to a functor $R \colon \mathcal{C} \to \mathcal{D}$.

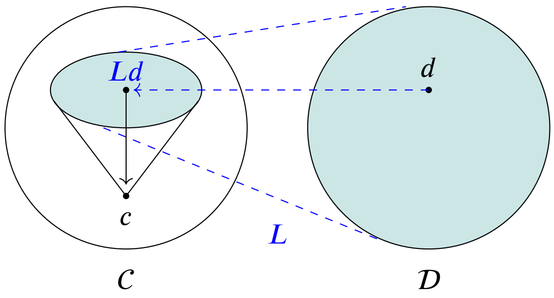
The first observation is that, since the left functor in an adjunction preserves colimits, we have to postulate that our functor L preserves colimits. This gives us a hint that the construction of the right adjoint relies on the ability to construct colimits in $\mathcal{D}$, and being able to somehow transport them back to $\mathcal{C}$ using L.

We could demand that all colimits, large and small, exist in $\mathcal{D}$ but this condition is too strong. Even a small category that has all colimits is automatically a preorder—that is, it can't have more than one morphism between any two objects.

But let's ignore size problems for a moment, and see how one would construct the right adjoint to a colimit-preserving functor L, whose source category $\mathcal{D}$ is small and has all colimits, large and small (thus it is a preorder).

Freyd's theorem in a preorder

The easiest way to define the right adjoint to L is to construct, for every object c, a universal arrow from L to c. Such an arrow is the terminal object in the comma category L/c—the category of arrows which originate in the image of L and converge on the object c.

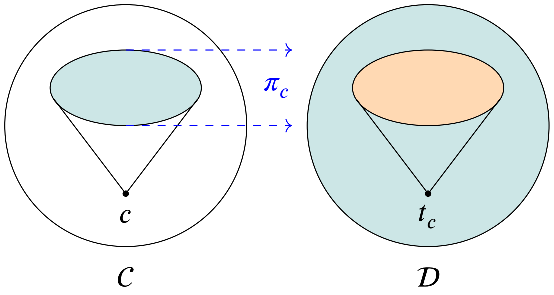


The important observation is that this comma category describes a cocone in $\mathcal{C}$. The base of this cocone is formed by those objects in the image of L that have an unobstructed view of c. The arrows in the base of the cocone are the morphisms in L/c. These are exactly the arrows that make the sides of the cocone commute.

$$\begin{tikzcd}
Ld \arrow[rr, "Lh"] \arrow[dr, "f"'] & & Ld'' \arrow[dl, "f''"] \\
& c &
\end{tikzcd}
\qquad
\begin{tikzcd}
d \arrow[r, "h"] & d''
\end{tikzcd}$$

The base of this cocone can then be projected back to $\mathcal{D}$. There is a projection π_c which maps every pair (d, f) in L/c back to d, thus forgetting the arrow f. It also maps every morphism in L/c to an arrow in $\mathcal{D}$ that gave rise to it. This way π_c defines a diagram in $\mathcal{D}$. The colimit of this diagram exists, because we have assumed that all colimits exist in $\mathcal{D}$. Let's call this colimit t_c:

$$t_c = \text{colim } \pi_c$$



Let's see if we can use this t_c to construct a terminal object in L/c. We have to find an arrow, let's call it $\varepsilon_c \colon Lt_c \to c$, such that the pair $\langle t_c, \varepsilon_c \rangle$ is terminal in L/c.

Notice that L maps the diagram generated by π_c back to the base of the cocone defined by L/c. The projection π_c did nothing more than to ignore the sides of this cocone, leaving its base intact.

We now have two cocones in $\mathcal{C}$ with the same base: the original one with the apex c and the new one obtained by applying L to the cocone in $\mathcal{D}$. Since L preserves colimits, the colimit of the new cocone is Lt_c—the image of the colimit t_c:

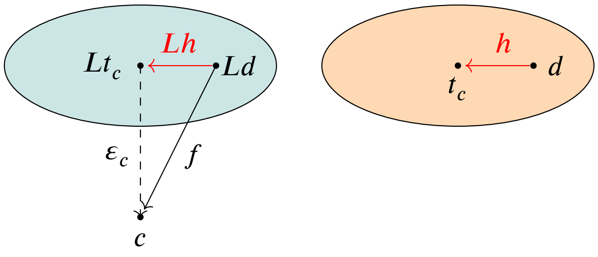
$$\text{colim } (L \circ \pi_c) = L(\text{colim } \pi_c) = Lt_c$$

By universal construction, we deduce that there must be a unique cocone morphism from the colimit Lt_c to c. That morphism, which we'll call ε_c, makes all the relevant triangles commute.

What remains to be shown is that $\langle t_c, \varepsilon_c \rangle$ is terminal in L/c, that is, for any $\langle d, f \colon Ld \to c \rangle$ there is a unique comma-category morphism $h \colon d \to t_c$ that makes the following triangle commute:

$$\begin{tikzcd}
Ld \arrow[rr, dashed, "Lh"] \arrow[dr, "f"'] & & Lt_c \arrow[dl, "\varepsilon_c"] \\
& c &
\end{tikzcd}$$

Notice that any such d is automatically part of the diagram produced by π_c (it's the result of π_c acting on $\langle d, f \rangle$). We know that t_c is the limit of the π_c diagram. So there must be a wire from d to t_c in the limiting cocone. We pick this wire as our h.



The commuting condition then follows from ε_c being a morphism between two cocones. It is a unique cocone morphism simply because $\mathcal{D}$ is a preorder.

This proves that there is a universal arrow $\langle t_c, \varepsilon_c \rangle$ for every c, therefore we have a functor R defined on objects as $Rc = t_c$ that is the right adjoint to L.

Solution set condition

The problem with the previous proof is that comma categories in most practical cases are large: their objects don't form a set. But maybe we can approximate the comma category by selecting a smaller but representative set of objects and arrows?

To select the objects, we'd use a mapping from some indexing set I. We define a set of objects d_i where i Ë I. Since we are trying to approximate the comma category L/c, we select objects together with arrows $f_i \colon Ld_i \to c$.

The relevant part of the comma category is encoded in morphism between objects satisfying the commuting condition. We could try to specialize this condition to only apply inside our family of objects, but that would not be enough. We have to find a way to probe all other objects of the comma category.

To do this, we reinterpret the commuting condition as a recipe for factorizing an arbitrary $f \colon Ld \to c$ through some pair $\langle d_i, f_i \rangle$:

$$\begin{tikzcd}
Ld \arrow[rr, "Lh"] \arrow[dr, "f"'] & & Ld_i \arrow[dl, "f_i"] \\
& c &
\end{tikzcd}$$

A *solution set* is a family of pairs $\langle d_i, f_i \colon Ld_i \to c \rangle$ indexed by a set I that can be used to factor any pair $\langle d, f \colon Ld \to c \rangle$. It means that there exists an index i Ë I and an arrow $h \colon d \to d_i$ that factorizes f:

$$f = f_i \circ Lh$$

Another way of expressing this property is to say that there exists a *weakly terminal* set of object in the comma category L/c. A weakly terminal set has the property that for any object in the category there is a morphism to at least one object in the set.

Previously we've seen that having the terminal object in the comma category L/c for every c is enough to define the adjunction. It turns out that we can achieve the same goal using the solution set.

The assumptions of the Freyd's adjoint functor theorem state that we have a colimit-preserving functor $L \colon \mathcal{D} \to \mathcal{C}$ from a small co-complete category. Both these conditions relate to *small* diagrams. If we can pick a solution set $\langle d_i, f_i \colon Ld_i \to c \rangle$ for every c, then the right adjoint R exists. Solution sets for different c's may be different.

We've seen before that in a cocomplete category the existence of a weakly terminal set is enough to define a terminal object. In our case it means that, for any c, we can construct the universal arrow from L to c. And this is enough to define the whole adjunction.

A dual version of the adjoint functor theorem can be used to construct the left adjoint.

Defunctionalization

Every programming language lets us define functions, but not all languages support higher level functions (functions taking functions as arguments, functions returning functions, or data types constructed from functions) or anonymous functions (a.k.a., lambdas). It turns out that, even in such languages, higher order functions can be implemented using the process called defunctionalization. This technique is based on the adjoint functor theorem. Moreover, defunctionalization can be used whenever passing functions around is impractical, for instance in a distributed system.

The idea behind defunctionalization is that the function type is defined as the right adjoint to the product.

$$\mathcal{C}(e \quad a, b) \text{ ö } \mathcal{C}(e, b^a)$$

The adjoint functor theorem can be used to approximate this adjoint.

In general, any finite program can only have a finite number of function definitions. These functions (together with the environments they capture) form the solution set that we can use to construct the function type. In practice, we do it only for a small subset of functions which occur as arguments to, or are returned from, other functions.

A typical example of the usage of higher order functions is in continuation passing style. For instance, here's a function that calculates the sum of the elements of a list. But instead of returning the sum it calls a continuation with the result:

If the list is empty, the function calls the continuation with zero. Otherwise it calls itself recursively, with two arguments: the tail of the list , and a new continuation:

This new continuation calls the previous continuation , passing it the sum of the head of the list and its argument (which is the accumulated sum).

Notice that this lambda is a closure: It's a function of one variable , but it also has access to and from its environment.

To extract the final sum, we call our recursive function with the trivial continuation, the identity:

Anonymous functions are convenient, but nothing prevents us from using named functions. However, if we want to factor out the continuations, we have to be explicit about passing in the environments.

For instance, we can replace our first lambda:

with the function , but we have to explicitly pass the pair as the environment of the type :

The other lambda, the identity, uses the empty environment, so it becomes:

Here's the implementation of our algorithm using these two named functions:

In fact, if all we are interested in is calculating the sum, we can replace the polymorphic type with with no other changes.

This implementation still uses higher order functions. In order to eliminate them, we have to analyze what it means to pass a function as an argument. Such a function can only be used in one way: it can be applied to its arguments. This property of a function type is expressed as the counit of the currying adjunction:

$$\varepsilon : b^a \quad a \to b$$

or, in Haskell, as a higher-order function:

This time we are interested in constructing the counit from first principles. We've seen that this can be accomplished using the comma category. In our case, an object of the comma category for the product functor $L_a = (*) \quad a$ is a pair

$$(e, f : (e \quad a) \to b)$$

or, in Haskell:

A morphism in this category between (e, f) and $(e^{\text{¨}}, f^{\text{¨}})$ is an arrow $h : e \to e^{\text{¨}}$, which satisfies the commuting condition:

$$f^{\text{¨}} \circ h = f$$

We interpret this morphism as "reducing" the environment e down to $e^{\text{¨}}$. The arrow $f^{\text{¨}}$ is able to produce the same output of the type b using a potentially smaller environment given by $h(e)$. For instance e may contain variables that are irrelevant for computing b from a, and h projects them out.

$$\begin{tikzcd}
e \ a \arrow[rr, "h \ a"] \arrow[dr, "f"'] & & e^{\cdot\cdot} \ a \arrow[dl, "f^{\cdot\cdot}"] \\
& b &
\end{tikzcd}
\qquad
\begin{tikzcd}
e \arrow[r, "h"] & e^{\cdot\cdot}
\end{tikzcd}$$

In fact, we performed this kind of reduction when defining and . In principle, we could have passed the tail to both functions, since it's accessible at the point of call. But we knew that they didn't need it.

Using the Freyd's theorem, we could define the function object $a \to b$ as the colimit of the diagram defined by the comma category. Such a colimit is essentially a giant coproduct of all environments modulo identifications given by comma-category morphisms. This identification does the job of reducing the environment needed by $a \to b$ to the bare minimum.

In our example, the continuations we're interested in are functions . In fact we are not interested in generating the generic function type ; just the minimal one that would accommodate our two functions and . We can do it by creating a very small solution set.

In our case the solution set consists of pairs $(e_i, f_i : e_i \quad a \to b)$ such that any pair $(e, f : e \quad a \to b)$ can be factorized through one of the f_i's. More precisely, the only two environments we're interested in are for , and the empty environment for .

In principle, our solution set should allow for the factorization of every object of the comma category, that is a pair of the type:

but here we are only interested in two specific functions. Also, we are not concerned with the uniqueness of the representation so, instead of using a colimit (as we did for the adjoint functor theorem), we'll just use a coproduct of all the environments of interest. We end up with the following data type that is the sum of the two environments, and , we're interested in. We end up with the type:

Notice that we have recursively encoded the part of the environment as . Thus we have also removed the need to use functions as arguments to data constructors.

If you look at this definition carefully, you will discover that it's the definition of a list of , modulo some renamings. Every call to pushes another integer on the stack. This interpretation agrees with our intuition that recursive algorithms require some kind of a runtime stack.

We are now ready to implement our approximation to the counit of the adjunction. It's composed from the bodies of the two functions, with the understanding that recursive calls also go through :

Compare this with our earlier:

The main algorithm can now be rewritten without any higher order functions or lambdas:

The main advantage of defunctionalization is that it can be used in distributed environments. Arguments to remote functions, as long as they are data structures and not functions, can be serialized and sent along the wire. All that's needed is for the receiver to have access to .

10.9 Free/Forgetful Adjunctions

The two functors in the adjunction play different roles: the picture of the adjunction is not symmetric. Nowhere is this illustrated better than in the case of the free/forgetful adjunctions.

A forgetful functor is a functor that "forgets" some of the structure of its source category. This is not a rigorous definition but, in most cases, it's pretty obvious what structure is being forgotten. Very often the target category is just the category of sets, which is considered the epitome of structurelessness. The result of the forgetful functor in that case is called the "underlying" set, and the functor itself is often called U.

More precisely, we say that a functor forgets *structure* if the mapping of hom-sets is not surjective, that is, there are arrows in the target hom-set that have no corresponding arrows in the source hom-set. Intuitively, it means that the arrows in the source have some structure to preserve, so there are fewer of them; and that structure is absent in the target.

The left adjoint to a forgetful functor is called a *free functor*.

$$\begin{tikzcd}
Fx \arrow[d, dashed, red, bend right] \arrow[d, dashed] \arrow[d, dashed, blue, bend left] & x \arrow[l, bend right, "F"'] \arrow[d, dashed, red, bend right] \arrow[d, dashed] \arrow[d, dashed, blue, bend left] \\
y \arrow[r, bend right, "U"'] & Uy
\end{tikzcd}$$

A classic example of a free/forgetful adjunction is the construction of the free monoid.

The category of monoids

Monoids in a monoidal category $\mathcal{C}$ form their own category $\mathbf{Mon}(\mathcal{C})$. Its objects are monoids, and its arrows are the arrows of $\mathcal{C}$ that preserve the monoidal structure.

The following diagram explains what it means for f to be a monoid morphism, going from a monoid (M_1, η_1, μ_1) to a monoid (M_2, η_2, μ_2):

$$\begin{tikzcd}
 & M_1 \arrow[dd, "f"] & M_1 \otimes M_1 \arrow[l, "\mu_1"'] \arrow[dd, "f \otimes f"] \\
I \arrow[ur, "\eta_1"] \arrow[dr, "\eta_2"'] & & \\
 & M_2 & M_2 \otimes M_2 \arrow[l, "\mu_2"']
\end{tikzcd}$$

A monoid morphism f must map unit to unit, which means that:

$$f \circ \eta_1 = \eta_2$$

and it must map multiplication to multiplication:

$$f \circ \mu_1 = \mu_2 \circ (f \otimes f)$$

Remember, the tensor product $\otimes$ is functorial, so it can lift pairs of arrows, as in $f \otimes f$.

In particular, the category Set is monoidal, with cartesian product and the terminal object providing the monoidal structure.

In particular, monoids in Set are sets with additional structure. They form their own category $\mathrm{Mon}(\mathrm{Set})$ and there is a forgetful functor U that simply maps the monoid to the set of its elements. When we say that a monoid is a set, we mean the underlying set.

Free monoid

We want to construct the free functor

$$F \colon \mathrm{Set} \to \mathrm{Mon}(\mathrm{Set})$$

that is adjoint to the forgetful functor U.

We start with an arbitrary set X and an arbitrary monoid m. On the right-hand side of the adjunction we have the set of functions from X to Um. On the left-hand side, we have a set of highly constrained structure-preserving monoid morphisms from FX to m. How can these two hom-sets be isomorphic?

In $\mathrm{Mon}(\mathrm{Set})$, monoids are sets of elements, and monoid morphisms are functions between such sets, satisfying additional constraints: preserving unit and multiplication.

Arrows in Set, on the other hand, are just functions with no additional constraints. So, in general, there are fewer arrows between monoids than there are between their underlying sets.

$$\begin{tikzcd}
FX \arrow[d, dashed, red, bend right] \arrow[d, dashed] \arrow[d, dashed, blue, bend left] & X \arrow[l, bend right, "F"'] \arrow[d, dashed, red, bend right] \arrow[d, dashed] \arrow[d, dashed, blue, bend left] \\
m \arrow[r, bend right, "U"'] & Um
\end{tikzcd}$$

Here's the idea: if we want to have a one to one matching between arrows, we want FX to be much larger than X. This way, there will be many more functions from it to m—so many that, even after rejecting the ones that don't preserve the structure, we'll still have enough to match every function $f \colon X \to Um$.

We'll construct the monoid FX starting from the set X, and adding more and more elements as we go. We'll call the initial set X the *generators* of FX. We'll construct a monoid morphism $g \colon FX \to m$ starting with the original function f and extending it to act on more and more elements.

On generators, x Ë X, g works the same as f:

$$gx = fx$$

Since FX is supposed to be a monoid, it has to have a unit. We can't pick one of the generators to be the unit, because it would impose constraints on the part of g that is already fixed by f—it would have to map it to the unit $e^{\text{¨}}$ of m. So we'll just add an extra element e to FX and call it the unit. We'll define the action of g on it by saying that it is mapped to the unit $e^{\text{¨}}$ of m:

$$ge = e^{\text{¨}}$$

We also have to define monoidal multiplication in FX. Let's start with a product of two generators a and b. The result of the multiplication cannot be another generator because, again, that would constrain the part of g that's fixed by f—products must be mapped to products. So we have to make all products of generators new elements of FX. Again, the action of g on those products is fixed:

$$g(a \cdot b) = ga \cdot gb$$

Continuing with this construction, any new multiplication produces a new element of FX, except when it can be reduced to an existing element by applying monoid laws. For instance, the new unit e times a generator a must be equal to a. But we have made sure that e is mapped to the unit of m, so the product $ge \cdot ga$ is automatically equal to ga.

Another way of looking at this construction is to think of the set X as an alphabet. The elements of FX are then strings of characters from this alphabet. The generators are single-letter strings, "*a*", "*b*", and so on. The unit is an empty string "". Multiplication is string concatenation, so "*a*" times "*b*" is a new string "*ab*". Concatenation is automatically associative and unital, with the empty string as the unit.

The intuition behind free functors is that they generate structure "freely," as in "with no additional constraints." They also do it lazily: instead of performing operations, they just record them. They create generic domain-specific programs that can be executed later by specific interpreters.

The free monoid "remembers to do the multiplication" at a later time. It stores the arguments to multiplication in a string, but doesn't perform the multiplication. It's only allowed to simplify its records based on generic monoidal laws. For instance, it doesn't have to store the command to multiply by the unit. It can also "skip the parentheses" because of associativity.

Exercise 10.9.1. *What is the unit and the counit of the free monoid adjunction $F \dashv U$?*

Free monoid in programming

In Haskell, monoids are defined using the following typeclass:

Here, is the curried form of the mapping from the product: . The element corresponds to the arrow from the terminal object (unit of the monoidal category), or simply an element of .

A free monoid generated by some type , which serves as a set of generators, is represented by a list type . An empty list serves as the unit; and monoid multiplication is implemented as list concatenation, traditionally written in infix form:

A list is an instance of a :

To show that it's a free monoid, we have to be able to construct a monoid morphism from the list of to an arbitrary monoid , provided we have an (unconstrained) mapping from to (the underlying set of) . We can't express all of this in Haskell, but we can define the function:

This function transforms the elements of the list to monoidal values using and then folds them using , starting with the unit .

It's easy to see that an empty list is mapped to the monoidal unit. It's not too hard to see that a concatenation of two lists is mapped to the monoidal product of the results. So, indeed, is a monoid morphism.

Following the intuition of a free monoid being a domain-specific program for multiplying stuff, provides an *interpreter* for this program. It performs all the multiplications that have been postponed. Note that the same program may be interpreted in many different ways, depending on the choice of the concrete monoid and the function .

We'll come back to free monoids as lists in the chapter on algebras.

Exercise 10.9.2. *Write a program that takes a list of integers and interprets it in two different ways: once using the additive and once using the multiplicative monoid of integers.*

10.10 The Category of Adjunctions

We can define composition of adjunctions by taking advantage of the composition of functors that define them. Two adjunctions, $L \dashv R$ and $L' \dashv R'$, are composable if they share the category in the middle:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend right, "R'"'] & \mathcal{D} \arrow[l, bend right, "L'"'] \arrow[r, bend right, "R"'] & \mathcal{E} \arrow[l, bend right, "L"']
\end{tikzcd}$$

By composing the functors we get a new adjunction $(L' \circ L) \dashv (R \circ R')$.

Indeed, let's consider the hom-set:

$$\mathcal{C}(L'(Le), c)$$

Using the $L' \dashv R'$ adjunction, we can transpose L' to the right, where it becomes R':

$$\mathcal{D}(Le, R'c)$$

and using $L \dashv R$ we can similarly transpose L:

$$\mathcal{E}(e, R(R'c))$$

Combining these two isomorphisms, we get the composite adjunction:

$$\mathcal{C}((L' \circ L)e, c) \cong \mathcal{E}(e, (R \circ R')c)$$

Because functor composition is associative, the composition of adjunctions is also associative. It's easy to see that a pair of identity functors forms a trivial adjunction that serves as the identity with respect to composition of adjunctions. Therefore we can define a category $\mathrm{Adj}(\mathrm{Cat})$ in which objects are categories and arrows are adjunctions (by convention, pointing in the direction of the left adjoint).

Adjunctions can be defined purely in terms of functors and natural transformations, that is 1-cells and 2-cells in the 2-category Cat. There is nothing special about Cat, and in fact adjunctions can be defined in any 2-category. Moreover, the category of adjunctions is itself a 2-category.

10.11 Levels of Abstraction

Category theory is about structuring our knowledge. In particular, it can be applied to the knowledge of category theory itself. Hence we see a lot of mixing of abstraction levels in category theory. The structures that we see at one level can be grouped into higher-level structures which exhibit even higher levels of structure, and so on.

In programming we are used to building hierarchies of abstractions. Values are grouped into types, types into kinds. Functions that operate on values are treated differently than functions that operate on types. We often use different syntax to separate levels of abstractions. Not so in category theory.

A set, categorically speaking, can be described as a discrete category. Elements of the set are objects of this category and, other than the obligatory identity morphisms, there are no arrows between them.

The same set can then be seen as an object in the category $\mathbf{Set}$. Arrows in this category are functions between sets.

The category $\mathbf{Set}$, in turn, is an object in the category $\mathbf{Cat}$. Arrows in $\mathbf{Cat}$ are functors.

Functors between any two categories $\mathcal{C}$ and $\mathcal{D}$ are objects in the functor category $[\mathcal{C}, \mathcal{D}]$. Arrows in this category are natural transformations.

We can define functors between functor categories, product categories, opposite categories, and so on, ad infinitum.

Completing the circle, hom-sets in every category are sets. We can define mappings and isomorphisms between them, reaching across disparate categories. Adjunctions are possible because we can compare hom-sets that live in different categories.

Chapter 11

Dependent Types

We've seen types that depend on other types. They are defined using type constructors with type parameters, like or . Most programming languages have some support for generic data types—data types parameterized by other data types.

Categorically, such types are modeled as functors [1].

A natural generalization of this idea is to have types that are parameterized by values. For instance, it's often advantageous to encode the length of a list in its type. A list of length zero would have a different type than a list of length one, and so on.

Obviously, you cannot change the length of such a list, since it would change its type. This is not a problem in functional programming, where all data types are immutable anyway. When you prepend an element to a list, you create a new list, at least conceptually. With a length-encoded list, this new list is of a different type, that's all!

Types parameterized by values are called *dependent types*. There are languages like Idris or Agda that have full support for dependent types. It's also possible to implement dependent types in Haskell, but support for them is still rather patchy.

The reason for using dependent types in programming is to make programs provably correct. In order to do that, the compiler must be able to check the assumptions made by the programmer.

Haskell, with its strong type system, is able to uncover a lot of bugs at compile time. For instance, it won't let you write (infix notation for), unless you provide the instance for the type of your variables.

However, within Haskell's type system, there is no way to express or, much less enforce, the unit and associativity laws for the monoid. For that, the instance of the type class would have to carry with itself proofs of equality (not actual code):

[1] A type constructor that has no instance can be thought of as a functor from a discrete category—a category with no arrows other than identities

Dependent types, and equality types in particular, pave the way towards this goal.

The material in this chapter is more advanced, and not used in the rest of the book, so you may safely skip it on first reading. Also, to avoid confusion between fibers and functions, I decided to use capital letters for objects in parts of this chapter.

11.1 Dependent Vectors

We'll start with the standard example of a counted list, or a vector:

The compiler will recognize this definition as dependently typed if you include the following language pragmas:

The first argument to the type constructor is a natural number . Notice: this is a value, not a type. The type checker is able to figure this out from the usage of in the two data constructors. The first one creates a vector of the type , and the second creates a vector of the type , where and are defined as the constructors of natural numbers:

We can be more explicit about the parameters if we use the pragma:

and import the library:

We can then specify that is a , whereas is a :

Using one of these definitions we can, for instance, construct a vector (of integers) of length zero:

It has a different type than a vector of length one:

and so on.

We can now define a dependently typed function that returns the first element of a vector:

This function is guaranteed to work exclusively with non-zero-length vectors. These are the vectors whose size matches , which cannot be . If you try to call this function with , the compiler will flag the error.

Another example is a function that zips two vectors together. Encoded in its type signature is the requirement that the two vectors be of the same size (the result is also of the size):

Dependent types are especially useful when encoding the shapes of containers. For instance, the shape of a list is encoded in its length. A more advanced example would encode the shape of a tree as a runtime value.

Exercise 11.1.1. *Implement the function that returns the tail of the non-zero-length vector. Try calling it with .*

11.2 Dependent Types Categorically

The easiest way to visualize dependent types is to think of them as families of types indexed by elements of a set. In the case of counted vectors, the indexing set would be the set of natural numbers $\mathbb{N}$.

The zeroth type would be the unit type representing an empty vector. The type corresponding to would be ; then we'd have a pair , a triple and so on, with higher and higher powers of .

If we want to talk about the whole family as one big set, we can take the sum of all these types. For instance, the sum of all powers of a is the familiar list type, a.k.a., a free monoid:

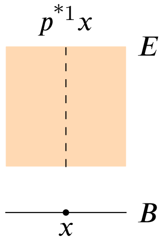
$$\text{List}(a) = 1 + a + a \quad a + a \quad a \quad a + ... = \sum_{n:\mathbb{N}} a^n$$

Fibrations

Although intuitively easy to visualize, this point of view doesn't generalize nicely to category theory, where we don't like mixing sets with objects. So we turn this picture on its head and instead of talking about injecting family members into the sum, we consider a mapping that goes in the opposite direction.

This, again, we can first visualize using sets. We have one big set E describing the *entire* family, and a function p called the projection, or a *display map*, that goes from E down to the indexing set B (also called the *base*).

This function will, in general, map multiple elements to one. We can then talk about the inverse image of a particular element x Ë B as the set of elements that get mapped down to it by p. This set is called the *fiber* and is written $p^{*1}x$ (even though, in general, p is not invertible in the usual sense). Seen as a collection of fibers, E is often called a *fiber bundle* or just a bundle.



Now forget about sets. A *fibration* in an arbitrary category is a pair of objects e and b and an arrow $p\colon e \to b$.

So this is really just an arrow, but the context is everything. When an arrow is called a fibration, we use the intuition from sets, and imagine its source e as a collection of fibers, with p projecting each fiber down to a single point in the base b.

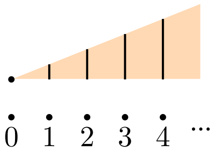
We can go even further: since (small) categories form a category $\mathbf{Cat}$ with functors as arrows, we can define a fibration of a category, taking another category as its base.

Type families as fibrations

We will therefore model type families as fibrations. For instance, our counted-vector family can be represented as a fibration whose base is the type of natural numbers. The whole family is a sum (coproduct) of consecutive powers (products) of a:

$$\text{List}(a) = a^0 + a^1 + a^2 + ... = \sum_{n\colon \mathbb{N}} a^n$$

with the zeroth power—the initial object—representing the vector of size zero.



The projection $p\colon \text{List}(a) \to \mathbb{N}$ is the familiar length function.

In category theory we like to describe things in bulk—defining internal structure of things by structure-preserving maps between them. Such is the case with fibrations. If we fix the base object b and consider all possible source objects in the category C, and all possible projections down to b, we get a *slice category* C/b. This category represents all the ways we can slice the objects of C over the base b.

Recall that the objects in the slice category are pairs $\langle e, p\colon e \to b\rangle$, and a morphism between two objects $\langle e, p\rangle$ and $\langle e^{¨}, p^{¨}\rangle$ is an arrow $f\colon e \to e^{¨}$ that commutes with the projections, that is:

$$p^{¨} \circ f = p$$

The best way to visualize this is to notice that such a morphism maps fibers of p to fibers of $p^{\cdot\cdot}$. It's a "fiber-preserving" mapping between bundles.

$$\begin{tikzcd}
e \arrow[rr, "f"] \arrow[dr, "p"'] & & e^{\cdot\cdot} \arrow[dl, "p^{\cdot\cdot}"] \\
& b &
\end{tikzcd}$$

Our counted vectors can be seen as objects in the slice category $\mathcal{C}/\mathbb{N}$ given by pairs $\langle \mathrm{List}(a), \mathrm{length} \rangle$. A morphism in this category maps vectors of length n to vectors of the same length n.

Pullbacks

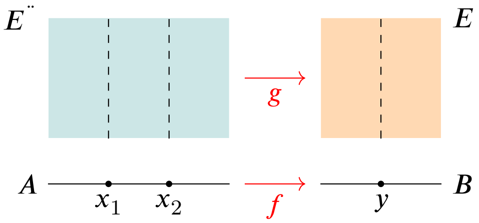
We've seen a lot of examples of commuting squares. Such a square is a graphical representation of an equation: two path between opposite corners of a square, each a result of a composition of two morphisms, are equal.

Like with every equality we may want to replace one or more of its components with an unknown, and try to solve the resulting equation. For instance, we may ask the question: Is there an object together with two arrows that would complete a commuting square? If many such objects exist, is there a universal one? If the missing piece of the puzzle is the upper left corner of a square (the source), we call it a pullback. If it's the lower right corner (the target), we call it a pushout.

$$\begin{tikzcd}
? \arrow[r, dashed, "?"] \arrow[d, dashed, "?"'] & E \arrow[d, "p"] \\
A \arrow[r, "f"] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
E \arrow[r, "f"] \arrow[d, "p"'] & E^{\cdot\cdot} \arrow[d, dashed, "?"] \\
B \arrow[r, dashed, "?"] & ?
\end{tikzcd}$$

Let's start with a particular fibration $p \colon E \to B$ and ask ourselves the question: what happens when we change the base from B to some A that is related to it through a mapping $f \colon A \to B$. Can we "pull the fibers back" along f?

Again, let's think about sets first. Imagine picking a fiber in E over some point y Ë B that is in the image of f. If f were invertible, there would be an element $x = f^{*1}y$. We'd plant our fiber over it. In general, though, f is not invertible. It means that there could be more elements of A that are mapped to our y. In the picture below you see two such elements, x_1 and x_2. We'll just duplicate the fiber above y and plant it over all elements that map to y. This way, every point in A will have a fiber sticking out of it. The sum of all these fibers will form a new bundle $E^{\cdot\cdot}$.



We have thus constructed a new fibration with the base A. Its projection $p^{\cdot\cdot} : E^{\cdot\cdot} \to A$ maps each point in a given fiber to the point over which this fiber was planted. There is also an obvious mapping $g \colon E^{\cdot\cdot} \to E$ that maps fibers to their corresponding fibers.

By construction, this new fibration $\langle E^{\cdot\cdot}, p^{\cdot\cdot} \rangle$ satisfies the condition:

$$p \circ g = f \circ p^{\cdot\cdot}$$

which can be represented as a commuting square:

$$\begin{tikzcd} E^{\cdot\cdot} \arrow[r, "g"] \arrow[d, "p^{\cdot\cdot}"'] & E \arrow[d, "p"] \\ A \arrow[r, "f"] & B \end{tikzcd}$$

In Set, we can explicitly construct $E^{\cdot\cdot}$ as a *subset* of the cartesian product $A \ E$ with $p^{\cdot\cdot} = \pi_1$ and $g = \pi_2$ (the two cartesian projections). An element of $E^{\cdot\cdot}$ is a pair $\langle a, e \rangle$, such that:

$$f(a) = p(e)$$

This commuting square is the starting point for the categorical generalization. However, even in Set there are many different fibrations over A that make this diagram commute. We have to pick the universal one. Such a universal construction is called a *pullback*, or a *fibered product*.

In category theory, a pullback of $p \colon e \to b$ along $f \colon a \to b$ is an object $e^{\cdot\cdot}$ together with two arrows $p^{\cdot\cdot} : e^{\cdot\cdot} \to a$ and $g \colon e^{\cdot\cdot} \to e$ that makes the following diagram commute

$$\begin{tikzcd} e^{\cdot\cdot} \arrow[r, "g"] \arrow[d, "p^{\cdot\cdot}"'] & e \arrow[d, "p"] \\ a \arrow[r, "f"] & b \end{tikzcd}$$

and that satisfies the universal condition.

The universal condition says that, for any other candidate object x with two arrows $q^{\cdot\cdot} : x \to e$ and $q \colon x \to a$ such that $p \circ q^{\cdot\cdot} = f \circ q$ (making the bigger "square" commute), there is a unique arrow $h \colon x \to e^{\cdot\cdot}$ that makes the two triangles commute, that is:

$$q = p^{\cdot\cdot} \circ h$$
$$q^{\cdot\cdot} = g \circ h$$

Pictorially:

$$\begin{tikzcd}
x \arrow[rd, dashed, "h"] \arrow[rrd, bend left, "q^{¨}"] \arrow[rdd, bend right, "q"'] & & \\
& e^{¨} \arrow[r, "g"] \arrow[d, "p^{¨}"'] \arrow[rd, phantom, very near start, "\lrcorner"] & e \arrow[d, "p"] \\
& a \arrow[r, "f"] & b
\end{tikzcd}$$

The angle symbol in the upper corner of the square is used to mark pullbacks.

If we look at the pullback through the prism of sets and fibrations, e is a bundle over b, and we are constructing a new bundle $e^{¨}$ out of the fibers taken from e. Where we plant these fibers over a is determined by (the inverse image of) f. This procedure makes $e^{¨}$ a bundle over both a and b, the latter with the projection $p \circ g = f \circ p^{¨}$.

The x in this picture is some other bundle over a with the projection q. It is simultaneously a bundle over b with the projection $f \circ q = p \circ q^{¨}$. The unique mapping h maps the fibers of x given by q^{*1} to fibers of $e^{¨}$ given by $p^{¨*1}$.

All mappings in this picture work on fibers. Some of them rearrange fibers over new bases—that's what a pullback does. Other mappings modify individual fibers—the mapping $h \colon x \to e^{¨}$ works like this.

If you think of bundles as containers of fibers, the rearrangements of fibers corresponds to natural transformations, and the modifications of fibers correspond to the action of .

The universal condition then tells us that $q^{¨}$ can be factored into a modification of fibers h, followed by the rearrangement of fibers g.

It's worth noting that picking the terminal object or the singleton set as the pullback target gives us automatically the definition of the cartesian product:

$$\begin{tikzcd}
b\ e \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] \arrow[rd, phantom, very near start, "\lrcorner"] & e \arrow[d, "!"] \\
b \arrow[r, "!"] & 1
\end{tikzcd}$$

Alternatively, we can think of this picture as planting as many copies of e as there are elements in b. We'll use this analogy when we talk about the dependent sum and product.

Notice also that a single fiber can be extracted from a fibration by pulling it back to the terminal object. In this case the mapping $x \colon 1 \to b$ picks an element of the base, and the pullback along it extracts a single fiber φ:

$$\begin{tikzcd}
\varphi \arrow[r, "g"] \arrow[d, "!"'] \arrow[rd, phantom, very near start, "\lrcorner"] & e \arrow[d, "p"] \\
1 \arrow[r, "x"] & b
\end{tikzcd}$$

The arrow g injects this fiber back into e. By varying x we can pick different fibers in e.

Exercise 11.2.1. *Show that the pullback with the terminal object as the target is the product.*

Exercise 11.2.2. *Show that a pullback can be defined as a limit of the diagram from a stick-figure category with three objects:*

$$a \to b \leftarrow c$$

Exercise 11.2.3. *Show that a pullback in C with the target b is a product in the slice category C/b. Hint: Define two projections as morphisms in the slice category. Use universality of the pullback to show the universality of the product.*

Substitution

We have two alternative descriptions of dependent types: one as fibrations and another as type families. It's in the latter framework that the pullback along a morphism f can be interpreted as a substitution. When we have a type family $T\,y$ parameterized by elements $y \colon B$ and we always define a new type family by substituting $f\,x$ for y.

$$\begin{tikzcd}
T(f\,x) & T\,y \\
x \arrow[u, mapsto] \arrow[r, mapsto, "f"] & y \arrow[u, mapsto]
\end{tikzcd}$$

The new type family is thus parameterized by different shapes.

Dependent environments

When modeling lambda calculus, we used the objects of a cartesian closed category to serve both as types and environments. An empty environment was modeled as the terminal object (unit type), and we were building more complex environments using products. The order in which we multiply types doesn't matter since the product is symmetric (up to isomorphism).

When dealing with dependent types, we have to take into account that the type we are adding to the environment may depend on the values of the types already present in the environment. As before, we start with an empty environment modeled as the terminal object.

Weakening

Base-change functor

We used a cartesian closed category as a model for programming. To model dependent types, we need to impose an additional condition: We require the category to be *locally cartesian closed*. This is a category in which all slice categories are cartesian closed.

In particular, such categories have all pullbacks, so it's always possible to change the base of any fibration. Base change induces a mapping between slice categories that is functorial.

Given two slice categories C/b and C/a and an arrow between bases $f : b \to a$ the base-change functor $f^< : C/a \to C/b$ maps a fibration $\langle e, p\rangle$ to the fibration $f^<\langle e, p\rangle = \langle f^< e, f^< p\rangle$, which is given by the pullback:

$$\begin{tikzcd}
f^< e \arrow[r, "g"] \arrow[d, "f^< p"'] \arrow[dr, phantom, "\lrcorner", very near start] & e \arrow[d, "p"] \\
b \arrow[r, "f"] & a
\end{tikzcd}$$

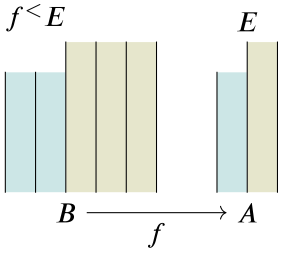
Notice that the functor $f^<$ goes in the opposite direction to the arrow f.

To visualize the base-change functor let's consider how it works on sets.

$$\begin{tikzcd}
f^< E \arrow[r, "g"] \arrow[d, "f^< p"'] \arrow[dr, phantom, "\lrcorner", very near start] & E \arrow[d, "p"] \\
B \arrow[r, "f"] & A
\end{tikzcd}$$

We have the intuition that the fibration p decomposes the set E into fibers over each point of A.

We can think of f as another fibration that similarly decomposes B. Let's call these fibers in B "patches." For instance, if A is just a two-element set, then the fibration given by f splits B into two patches. The pullback takes a fiber from E and plants it over the whole patch in B. The resulting set $f^< E$ looks like a patchwork, where each patch is planted with clones of a single fiber from E.



Since we have a function from B to A that may map many elements to one, the fibration over B has finer grain than the coarser fibration over A. The simplest, least-effort way to turn the fibration of E over A to a fibration over B, is to spread the existing fibers over the patches defined by (the inverse of) f. This is the essence of the universal construction of the pullback.

In particular, if A is a singleton set (the terminal object), then we have only one fiber (the whole of E) and the bundle $f^< E$ is a cartesian product $B \quad E$. Such bundle is called a *trivial bundle*.

A non-trivial bundle is not a product, but it can be *locally* decomposed into products. Just as B is a sum of patches, so $f^< E$ is a sum of products of these patches and the corresponding fibers of E.

You may also think of A as providing an *atlas* that enumerates all the patches in the *base* B. Imagine that A is a set of countries and B is a set of cities. The mapping f assigns a country to each city.

Continuing with this example, let E be the set of languages fibrated by the country. If we assume that in each city the language of the given country is spoken, the base-change functor replants the country's languages over each of its cities.

By the way, this idea of using local patches and an atlas goes back to differential geometry and general relativity, where we often glue together local coordinate systems to describe topologically nontrivial bundles, like Möbius strips or Klein bottles.

As we'll see soon, in a locally cartesian closed category, the base change functor has both the left and the right adjoints. The left adjoint to $f^<$ is called $f_!$ (sometimes pronounced "f down-shriek") and the right adjoint is called $f_<$ ("f down-star"):

$$f_! \dashv f^< \dashv f_<$$

In programming, the left adjoint is called the dependent sum, and the right adjoint is called the dependent product or dependent function:

$$\Sigma_f \dashv f^< \dashv \Pi_f$$

Exercise 11.2.4. *Define the action of the base-change functor on morphisms in C/a, that is, given a morphism h construct its counterpart $f^<h$*

$$\begin{tikzcd}
f^<e^{\cdot\cdot} \arrow[r, "f^<h"] \arrow[dr, "f^<p^{\cdot\cdot}"'] & f^<e \arrow[d, "f^<p"] & & e^{\cdot\cdot} \arrow[r, "h"] \arrow[d, "p^{\cdot\cdot}"] & e \arrow[dl, "p"] \\
& b \arrow[rr, "f"'] & & a &
\end{tikzcd}$$

Hint: Use the universality of the pullback and the commuting condition: $g^{\cdot\cdot} \circ h \circ p = f^<p^{\cdot\cdot} \circ f$.

$$\begin{tikzcd}
f^<e^{\cdot\cdot} \arrow[dr, dashed, "f^<h"] \arrow[drr, bend left, "g^{\cdot\cdot} \circ h"] \arrow[ddr, bend right, "f^<p^{\cdot\cdot}"'] & & \\
& f^<e \arrow[r, "g"] \arrow[d, "f^<p"'] \arrow[dr, phantom, "\lrcorner", very near start] & e \arrow[d, "p"] \\
& b \arrow[r, "f"] & a
\end{tikzcd}
\qquad
\begin{tikzcd}
f^<e^{\cdot\cdot} \arrow[r, "g^{\cdot\cdot}"] \arrow[d, "f^<p^{\cdot\cdot}"'] \arrow[dr, phantom, "\lrcorner", very near start] & e^{\cdot\cdot} \arrow[r, "h"] \arrow[d, "p^{\cdot\cdot}"'] & e \arrow[dl, "p"] \\
b \arrow[r, "f"] & a &
\end{tikzcd}$$

11.3 Dependent Sum

In type theory, the dependent sum, or the sigma type $\Sigma_{x:B}T(x)$, is defined as a type of pairs in which the *type* of the second component depends on the *value* of the first component.

Conceptually, the sum type is defined using its mapping-out property. The mapping out of a sum is a pair of mappings, as illustrated in this adjunction:

$$\mathcal{C}(F_1 + F_2, F) \text{ ö } (\mathcal{C} \quad \mathcal{C})(\langle F_1, F_2 \rangle, \Delta F)$$

Here, we have a pair of arrows $(F_1 \to F, F_2 \to F)$ that define the mapping out of the sum $S = F_1 + F_2$. In Set, the sum is a tagged union. A dependent sum is a sum that is tagged by elements of another set.

Our counted vector type can be thought of as a dependent sum tagged by natural numbers. An element of this type is a natural number (a value) paired with an element of the n-tuple type . Here are some counted vectors of integers written in this representation:

More generally, the introduction rule for the dependent sum assumes that there is a family of types $T(x)$ indexed by elements of the base type B. Then an element of $\Sigma_{x:B}T(x)$ is constructed from a pair of elements $x\colon B$ and $y\colon T(x)$.

Categorically, dependent sum is modeled as the left adjoint of the base-change functor.

To see this, let's first revisit the definition of a pair, which is an element of a product. We've noticed before that a product can be written as a pullback from the singleton set—the terminal object. Here's the universal construction for the product/pullback (the notation anticipates the target of this construction):

$$\begin{tikzcd}
S \arrow[rrd, bend left, "\phi"] \arrow[rd, dashed, "\phi^T"] \arrow[rdd, bend right, "q"'] & & \\
& B \quad F \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] \arrow[rd, phantom, "\lrcorner", very near start] & F \arrow[d, "!"] \\
& B \arrow[r, "!"] & 1
\end{tikzcd}$$

We have also seen that the product can be defined using an adjunction. We can spot this adjunction in our diagram: for every pair of arrows $\langle \phi, q \rangle$ there is a unique arrow ϕ^T that makes the triangles commute.

Notice that, if we keep q fixed, we get a one-to-one correspondence between the arrows ϕ and ϕ^T. This will be the adjunction we're interested in.

We can now put our fibrational glasses on and notice that $\langle S, q \rangle$ and $\langle B \quad F, \pi_1 \rangle$ are two fibrations over the same base B. The commuting triangle makes ϕ^T a morphism in the slice category $\mathcal{C}/B$, or a fiber-wise mapping. In other words ϕ^T is a member of the hom-set:

$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} B \quad F \\ \pi_1 \end{matrix} \right\rangle\right)$$

Since ϕ is a member of the hom-set $C(S, F)$, we can rewrite the one-to-one correspondence between ϕ^T and ϕ as an isomorphism of hom-sets:

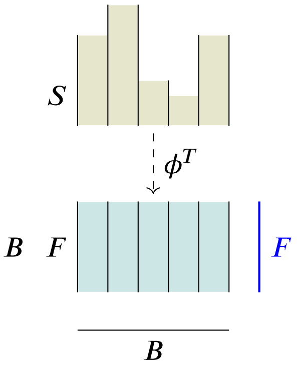
$$(C/B)\left(\left\langle {S \atop q} \right\rangle, \left\langle {B \quad F \atop \pi_1} \right\rangle\right) \text{ ö } C(S, F)$$

In fact, it's an adjunction in which we have the forgetful functor $U : C/B \to C$ mapping $\langle S, q\rangle$ to S, thus forgetting the fibration.

If you squint at this adjunction hard enough, you can see the outlines of the definition of S as a categorical sum (coproduct).

Firstly, on the right you have a mapping out of S. Think of S as the sum of fibers that are defined by the fibration $\langle S, q\rangle$.

Secondly, recall that the fibration $\langle B \quad F, \pi_1\rangle$ can be though of as producing many copies of F planted over points in B. This is a generalization of the diagonal functor Δ that duplicates F—here, we make "B copies" of F. The left hand side of the adjunction is just a bunch of arrows, each mapping a different fiber of S to the target fiber F.



Applying this idea to our counted-vector example, ϕ^T stands for infinitely many functions, one per every natural number. In practice, we define these functions using recursion. For instance, here's a mapping out of a vector of integers:

Adding the atlas

We can generalize our diagram by replacing the terminal object with an arbitrary base A (an atlas). Instead of a single fiber, we now have a fibration $\langle F, p\rangle$, and we use the pullback square that defines the base-change functor $f^{<}$:

$$\begin{tikzcd}
S \arrow[drr, bend left, "\phi"] \arrow[dr, dashed, "\phi^T"] \arrow[ddr, bend right, "q"'] & & \\
& f^{<}F \arrow[r, "g"] \arrow[d, "f^{<}p"'] \arrow[dr, phantom, "\lrcorner", very near start] & F \arrow[d, "p"] \\
& B \arrow[r, "f"] & A
\end{tikzcd}$$

We can imagine that the fibration over B is finer grain, since f may map multiple points to one. Think, for instance, of a function that creates two bunches of even and odd numbers. In this picture, f defines a coarser "resampling" of the original S.

The universality of the pullback results in the following isomorphism of hom-sets:

$$(C/B)\left(\left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, f^{<}\left\langle \begin{matrix} F \\ p \end{matrix} \right\rangle\right) \ddot{o}\ (C/A)\left(\left\langle \begin{matrix} S \\ f\circ q \end{matrix} \right\rangle, \left\langle \begin{matrix} F \\ p \end{matrix} \right\rangle\right)$$

Here, ϕ^T is an element of the left-hand side, and ϕ is the corresponding element of the right-hand side.

We interpret this isomorphism as the adjunction between the base change functor $f^{<}$ on the left and the dependent sum functor on the right.

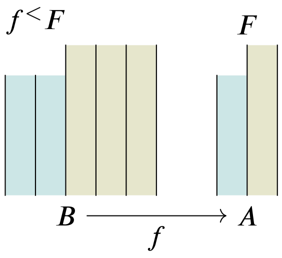
$$(C/B)\left(\left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, f^{<}\left\langle \begin{matrix} F \\ p \end{matrix} \right\rangle\right) \ddot{o}\ (C/A)\left(\Sigma_f \left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} F \\ p \end{matrix} \right\rangle\right)$$

The dependent sum is thus given by this formula:

$$\Sigma_f \left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle = \left\langle \begin{matrix} S \\ f\circ q \end{matrix} \right\rangle$$

This says that, if S is fibered over B using q, and there is a mapping f from B to A, then S is automatically (more coarsely) fibered over A, the projection being the composition $f\circ q$.

We've seen before that, in Set, f defines patches within B. Fibers of F are replanted in these patches to form $f^{<}F$. Locally—that is within each patch—$f^{<}F$ looks like a cartesian product.



S itself is fibered in two ways: coarsely chopped over A using $f\circ q$ and finely julienned over B using q.

In category theory, the dependent sum, which is the left adjoint to the base change functor $f^{<}$, is denoted by $f_!$. For a given $f\colon b \to a$, it's a functor:

$$f_!\colon \mathcal{C}/b \to \mathcal{C}/a$$

Its action on an object $(s, q\colon s \to b)$ is given by post-composition by f:

$$f_!(s, q) = (s, f \circ q)$$

Existential quantification

In the *propositions as types* interpretation, type families correspond to families of propositions. The dependent sum type $\Sigma_{x:B}\, T(x)$ corresponds to the proposition: There exists an x for which $T(x)$ is true:

$$\text{Ç}_{x:B}\, T(x)$$

Indeed, a term of the type $\Sigma_{x:B}\, T(x)$ is a pair of an element $x\colon B$ and an element $y\colon T(x)$—which shows that $T(x)$ is inhabited for some x.

11.4 Dependent Product

In type theory, the dependent product, or dependent function, or pi-type $\Pi_{x:B} T(x)$, is defined as a function whose return *type* depends on the *value* of its argument.

It's called a function, because you can evaluate it. Given a dependent function $f\colon \Pi_{x:B} T(x)$, you may apply it to an argument $x\colon B$ to get a value $f(x)\colon T(x)$.

Dependent product in Haskell

A simple example of a dependent product is a function that constructs a vector of a given size and fills it with copies of a given value:

At the time of this writing, Haskell's support for dependent types is limited, so the implementation of dependent functions requires the use of singleton types. In this case, the number that is the argument to is passed as a singleton natural:

(Note that is a function of two arguments, so it can be either considered a dependent function of a pair, or a regular function returning a dependent function.)

Dependent product of sets

Before we describe the categorical model of dependent functions, it's instructive to consider how they work on sets. A dependent function selects one element from each set $T(x)$.

You may visualize the totality of this selection as a giant tuple—an element of a cartesian product. For instance, in the trivial case of B a two-element set ⌢1, 2‘, a dependent function type is just a cartesian product $T(1) \quad T(2)$. In general, you get one tuple component per every value of x. It's a giant tuple indexed by elements of B. This is the meaning of the product notation, $\Pi_{x:B}T(x)$.

In our example, picks a particular counted vector for each value of . Counted vectors are equivalent to tuples so, for equal zero, returns an empty tuple ; for it returns a single value ; for equal two, it duplicates returning ; etc.

The function , evaluated at some , is equivalent to an infinite tuple of tuples:

$$((), x, (x, x), (x, x, x), ...)$$

which is a specific element of the type:

$$((), a, (a, a), (a, a, a), ...)$$

Dependent product categorically

In order to build a categorical model of dependent functions, we need to change our perspective from a family of types to a fibration. We start with a bundle E/B fibered by the projection $p \colon E \to B$. A dependent function is called a *section* of this bundle.

If you visualize the bundle as a bunch of fibers sticking out from the base B, a section is like a haircut: it cuts through each fiber to produce a corresponding value. In physics, such sections are called fields—with spacetime as the base.

Just like we talked about a function object representing a set of functions, we can talk about an object $S(E)$ that represents a set of sections of a given bundle E.

Just like we defined function application as a mapping out of the product:

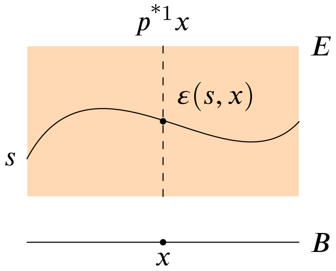
$$\varepsilon_{BC} \colon C^B \quad B \to C$$

we can define the dependent function application as a mapping:

$$\varepsilon \colon S(E) \quad B \to E$$

We can visualize it as picking a section s in $S(E)$ and an element x of the base B and producing a value in the bundle E. (In physics, this would correspond to measuring a field at a particular point in spacetime.)

But this time we have to insist that this value be in the correct fiber. If we project the result of applying ε to (s, x), it should fall back to the x.



In other words, this diagram must commute:

$$\begin{tikzcd}
S(E)\ \ B \arrow[rr, "\varepsilon"] \arrow[dr, "\pi_2"'] & & E \arrow[dl, "p"] \\
& B &
\end{tikzcd}$$

This makes ε a morphism in the slice category C/B.

And just like the exponential object was universal, so is the object of sections. The universality condition has the same form: For any other object G with an arrow $\phi\colon G\ \ B \to E$ there is a unique arrow $\phi^T : G \to S(E)$ that makes the following diagram commute:

$$\begin{tikzcd}
G\ \ B \arrow[d, dashed, "\phi^T\ B"'] \arrow[dr, "\phi"] & \\
S(E)\ \ B \arrow[r, "\varepsilon"] & E
\end{tikzcd}$$

The difference is that both ε and ϕ are now morphisms in the slice category C/B.

The one-to-one correspondence between ϕ and ϕ^T forms the adjunction:

$$(C/B)\left(\left\langle {G\ \ B \atop \pi_2} \right\rangle, \left\langle {E \atop p} \right\rangle\right) \text{ ö } C\left(G, S(E)\right)$$

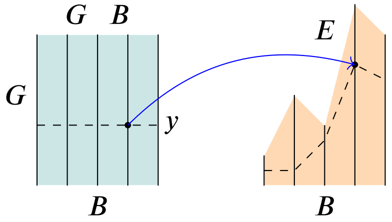
which we can use as the definition of the object of sections $S(E)$. The counit of this adjunction is the dependent-function application. We get it by replacing G with $S(E)$ and selecting the identity morphism on the right. The counit is thus a member of the hom-set:

$$(C/B)\left(\left\langle {S(E)\ \ B \atop \pi_2} \right\rangle, \left\langle {E \atop p} \right\rangle\right)$$

Compare the above adjunction with the currying adjunction that defines the function object E^B:

$$C(G\ \ B, E) \text{ ö } C(G, E^B)$$

Now recall that, in Set, we interpret the product $G\ \ B$ as planting copies of G as identical fibers over each element of B. So a single element of the left-hand side of our adjunction is a family of functions, one per fiber. Any given y Ë G cuts a horizontal slice through $G \;\; B$. These are the pairs (y, b) for all b Ë B. Our family of functions maps this slice to the corresponding fibers of E thus creating a section of E.



The adjunction tells us that this family of mappings uniquely determines a function from G to $S(E)$. Every y Ë G is thus mapped to a different element s of $S(E)$. Therefore elements of $S(E)$ are in one-to-one correspondence with sections of E .

These are all set-theoretical intuitions. We can generalize them by first noticing that the right hand side of the adjunction can be easily expressed as a hom-set in the slice category $C/1$ over the terminal object.

Indeed, there is one-to-one correspondence between objects X in C and objects $\langle X, !\rangle$ in $C/1$ (here ! is the unique arrow to the terminal object). Arrows in $C/1$ are arrows of C with no additional constraints. We therefore have:

$$(C/B)\left(\left\langle {G \;\; B \atop \pi_2} \right\rangle, \left\langle {E \atop p} \right\rangle\right) \text{ ö } (C/1)\left(\left\langle {G \atop !} \right\rangle, \left\langle {S(E) \atop !} \right\rangle\right)$$

Adding the atlas

The next step is to "blur the focus" by replacing the terminal object with a more general base A, serving as the atlas.

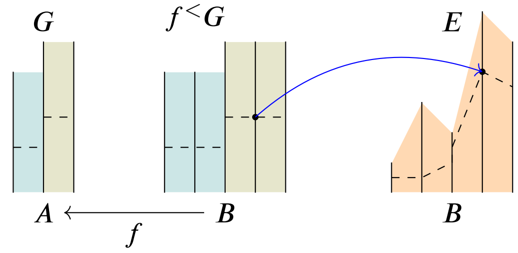
The right-hand side of the adjunction becomes a hom-set in the slice category C/A. G itself gets coarsely fibrated by some $q \colon G \to A$.

Remember that $G \;\; B$ can be understood as a pullback along the mapping $! \colon B \to 1$, or a change of base from 1 to B. If we want to replace 1 with A, we should replace the product $G \;\; B$ with a more general pullback of q. Such a change of base is parameterized by a new morphism $f \colon B \to A$.

$$\begin{tikzcd}
G \;\; B \arrow[r, "\pi_1"] \arrow[d, "\pi_2"] \arrow[dr, phantom, "\lrcorner", very near start] & G \arrow[d, "!"] \\
B \arrow[r, "!"] & 1
\end{tikzcd}
\quad \longrightarrow \quad
\begin{tikzcd}
f^{<}G \arrow[r, "g"] \arrow[d, "f^{<}q"'] \arrow[dr, phantom, "\lrcorner", very near start] & G \arrow[d, "q"] \\
B \arrow[r, "f"] & A
\end{tikzcd}$$

The result is that, instead of a bunch of G fibers over B, we get a pullback $f^{<}G$ that is populated by groups of fibers from the fibration $q \colon G \to A$. This way A serves as an atlas that enumerates all the patches populated by uniform fibers.

Imagine, for instance, that A is a two-element set. The fibration q will split G into two fibers. They will serve as our generic fibers. These fibers are now replanted over the two patches in B to form $f^{<}G$. The replanting is guided by f^{*1}.



The adjunction that defines the dependent function type is therefore:

$$(C/B)\left(f^{<}\left\langle \begin{matrix} G \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right) \text{ ö } (C/A)\left(\left\langle \begin{matrix} G \\ q \end{matrix} \right\rangle, \Pi_f \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right)$$

This is a generalization of an adjunction that we used to define the object of sections $S(E)$. This one defines a new object $\Pi_f E$ that is a rearrangement of the object of sections.

The adjunction is a mapping between morphisms in their respective slice categories:

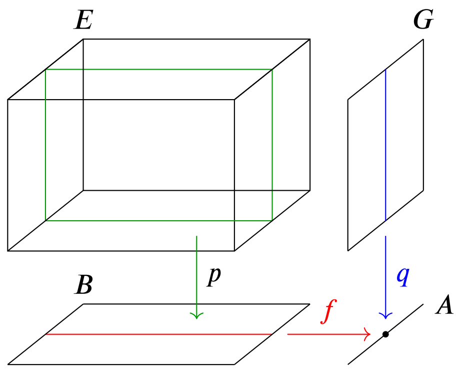
$$\begin{tikzcd}
f^{<}G \arrow[rr, "\phi"] \arrow[dr, "f^{<}q"'] & & E \arrow[dl, "p"] & G \arrow[rr, "\phi^T"] \arrow[dr, "q"'] & & \Pi_f E \arrow[dl, "\Pi_f p"] \\
& B & & & A &
\end{tikzcd}$$

To gain some intuition into this adjunction, let's consider how it works on sets.

- The right hand side operates in a coarsely grained fibration over the atlas A. It is a family of functions, one function per patch. For every patch we get a function from the "thick fiber" of G (drawn in blue below) to the "thick fiber" of $\Pi_f E$ (not shown).

- The left hand side operates in a more finely grained fibration over B. These fibers are grouped into small bundles over patches. Once we pick a patch (drawn in red below), we get a family of functions from that patch to the corresponding patch in E (drawn in green)—a section of a small bundle in E. So, patch-by-patch, we get small sections of E.

The adjunction tells us that the elements of the "thick fiber" of $\Pi_f E$ correspond to small sections of E over the same patch.



In category theory, the dependent product, which is the right adjoint to the base change functor $f^{<}$, is denoted by $f_{<}$. For a given $f : b \to a$, it's a functor:

$$f_{<} : C/b \to C/a$$

The following exercises shed some light on the role played by f. It can be seen as localizing the sections of E by restricting them to "neighborhoods" defined by f^{*1}.

Exercise 11.4.1. *Consider what happens when A is a two-element set* ˆ$0, 1$‘ *and f maps the whole of B to one element, say 1. How would you define the function on the right-hand side of the adjunction? What should it do to the fiber over 0?*

Exercise 11.4.2. *Let's pick G to be a singleton set 1, and let $x : 1 \to A$ be a fibration that selects an element in A. Using the adjunction, show that:*

- *$f^{<}1$ has two types of fibers: singletons over the elements of $f^{*1}(x)$ and empty sets otherwise.*
- *A mapping $\phi : f^{<}1 \to E$ is equivalent to a selection of elements, one from each fiber of E over the elements of $f^{*1}(x)$. In other words, it's a partial section of E over the subset $f^{*1}(x)$ of B.*
- *A fiber of $\Pi_f E$ over a given x is such a partial section.*
- *What happens when A is also a singleton set?*

Universal quantification

The logical interpretation of the dependent product $\Pi_{x:B} T(x)$ is a universally quantified proposition. An element of $\Pi_{x:B} T(x)$ is a section—the proof that it's possible to select an element from each member of the family $T(x)$. It means that none of them is empty. In other words, it's a proof of the proposition:

$$\text{Å}_{x:B} T(x)$$

11.5 Equality

Our first experience in mathematics involves equality. We learn that

$$1 + 1 = 2$$

and we don't think much of it afterwards.

But what does it mean that $1 + 1$ is equal to 2? Two is a number, but one plus one is an expression, so they are not the same thing. There is some mental processing that we have to perform before we pronounce these two things equal.

Contrast this with the statement $0 = 0$, in which both sides of equality are *the same thing*.

It makes sense that, if we are to define equality, we'll have to at least make sure that everything is equal to itself. We call this property *reflexivity*.

Recall our definition of natural numbers:

This is how we can define equality for natural numbers:

We are recursively stripping S's in each number until one of them reaches Z. If the other reaches Z at the same time, we pronounce the numbers we started with to be equal, otherwise they are not.

Equational reasoning

Notice that, when defining equality in Haskell, we were already using the equal sign. For instance, the equal sign in:

tells us that wherever we see the expression we can replace it with and vice versa.

This is the principle of substituting equals for equals, which is the basis for *equational reasoning* in Haskell. We can't encode proofs of equality directly in Haskell, but we can use equational reasoning to reason about Haskell programs. This is one of the main advantages of pure functional programming. You can't perform such substitutions in imperative languages, because of side effects.

If we want to prove that $1+1$ is 2, we have to first define addition. The definition can either be recursive in the first or in the second argument. This one recurses in the second argument:

We encode $1 + 1$ as:

We can now use the definition of to simplify this expression. We try to match the first clause, and we fail, because is not the same as . But the second clause matches. In it, is an arbitrary number, so we can substitute for it, and get:

In this expression we can perform another substitution of equals using the first clause of the definition of (again, with replaced by):

We arrive at:

We can clearly see that the right-hand side is the encoding of 2. But we haven't shown that our definition of equality is reflexive so, in principle, we don't know if

yields . We have to use step-by-step equational reasoning again:

We can use this kind of reasoning to prove statements about concrete numbers, but we run into problems when reasoning about generic numbers—for instance, showing that something is true for all . Using our definition of addition, we can easily show that is the same as . But we can't prove that is the same as . The latter proof requires the use of induction.

We end up distinguishing between two kinds of equality. One is proven using substitutions, or rewriting rules, and is called *definitional equality*. You can think of it as macro expansion or inline expansion in programming languages. It also involves β-reductions: performing function application by replacing formal parameters by actual arguments, as in:

The second more interesting kind of equality is called *propositional equality* and it may require actual proofs.

Equality vs isomorphism

We said that category theorists prefer isomorphism over equality—at least when it comes to objects. It is true that, within the confines of a category, there is no way to differentiate between isomorphic objects. In general, though, equality is stronger than isomorphism. This is a problem, because it's very convenient to be able to substitute equals for equals, but it's not always clear that one can substitute isomorphic for isomorphic.

Mathematicians have been struggling with this problem, mostly trying to modify the definition of isomorphism—but a real breakthrough came when they decided to simultaneously weaken the definition of equality. This led to the development of *homotopy type theory*, or HoTT for short.

Roughly speaking, in type theory, specifically in Martin-Löf theory of dependent types, equality is represented as a type, and in order to prove equality one has to construct an element of that type—in the spirit of the Curry-Howard interpretation.

Furthermore, in HoTT, the proofs themselves can be compared for equality, and so on ad infinitum. You can picture this by considering proofs of equality not as points but as some abstract paths that can be morphed into each other; hence the language of homotopies.

In this setting, instead of isomorphism, which involves strict equalities of arrows:

$$f \circ g = id$$

$$g \circ f = id$$

one defines an *equivalence*, in which these equalities are treated as types.

The main idea of HoTT is that one can impose the *univalence axiom* which, roughly speaking, states that equalities are equivalent to equivalences, or symbolically:

$$(\boldsymbol{A} = \boldsymbol{B}) \text{ ö } (\boldsymbol{A} \text{ ö } \boldsymbol{B})$$

Notice that this is an axiom, not a theorem. We can either take it or leave it and the theory is still valid (at least we think so).

Equality types

Suppose that you want to compare two terms for equality. The first requirement is that both terms be of the same type. You can't compare apples with oranges. Don't get confused by some programming languages allowing comparisons of unlike terms: in every such case there is an implicit conversion involved, and the final equality is always between same-type values.

For every pair of values there is, in principle, a separate type of proofs of equality. There is a type for $0 = 0$, there is a type for $1 = 1$, and there is a type for $1 = 0$; the latter hopefully uninhabited.

Equality type, a.k.a., identity type, is therefore a dependent type: it depends on the two values that we are comparing. It's usually written as Id_A, where A is the type of both values, or using an infix notation as $x =_A y$ (equal sign with the subscript A).

For instance, the type of equality of two zeros is written as $\mathrm{Id}_{\mathbb{N}}(0, 0)$ or:

$$0 =_{\mathbb{N}} 0$$

Notice: this is not a statement or a term. It's a *type*, like or . You can define a value of this type if you have an introduction rule for it.

Introduction rule

The introduction rule for the equality type is the dependent function:

$$\mathrm{refl}_A \colon \Pi_{x:A} \mathrm{Id}_A(x, x)$$

which can be interpreted in the spirit of propositions as types as the proof of the statement:

$$\mathring{A}_{x:A}\; x = x$$

This is the familiar reflexivity: it shows that, for all x of type A, x is equal to itself. You can apply this function to some concrete value x of type A, and it will produce a new value of type $\mathrm{Id}_A(x, x)$.

We can now prove that $0 = 0$. We can execute $\mathrm{refl}_{\mathbb{N}}(0)$ to get a value of the type $0 =_{\mathbb{N}} 0$. This value is the proof that the type is inhabited, and therefore corresponds to a true proposition.

This is the only introduction rule for equality, so you might think that all proofs of equality boil down to "they are equal because they are the same." Surprisingly, this is not the case.

β-reduction and η-conversion

In type theory we have this interplay of introduction and elimination rules that essentially makes them the inverse of each other.

Consider the definition of a product. We introduce it by providing two values, $x \colon A$ and $y \colon B$ and we get a value $p \colon A \quad B$. We can then eliminate it by extracting two values using two projections. But how do we know if these are the same values that we used to construct it? This is something that we have to postulate. We call it the computation rule or the β-reduction rule.

Conversely, if we are given a value $p \colon A \quad B$, we can extract the two components using projections, and then use the introduction rule to recompose it. But how do we know that we'll get the same p? This too has to be postulated. This is sometimes called the uniqueness condition, or the η-conversion rule.

In the categorical model of type theory these two rules follow from the universal construction.

The equality type also has the elimination rule, which we'll discuss shortly, but we don't impose the uniqueness condition. It means that it's possible that there are some equality proofs that were not obtained using refl.

This is exactly the weakening of the notion of equality that makes HoTT interesting to mathematicians.

Induction principle for natural numbers

Before formulating the elimination rule for equality, it's instructive to first discuss a simpler elimination rule for natural numbers. We've already seen such rule describing primitive recursion. It allowed us to define recursive functions by specifying a value init and a function step.

Using dependent types, we can generalize this rule to define the *dependent elimination rule* that is equivalent to the principle of mathematical induction.

The principle of induction can be described as a device to prove, in one fell swoop, whole families of propositions indexed by natural numbers. For instance, the statement that is equal to is really an infinite number of propositions, one per each value of .

We could, in principle, write a program that would meticulously verify this statement for a very large number of cases, but we'd never be sure if it holds in general. There are some conjectures about natural numbers that have been tested this way using computers but, obviously, they can never exhaust an infinite set of cases.

Roughly speaking, we can divide all mathematical theorems into two groups: the ones that can be easily formulated and the ones whose formulation is complex. They can be further subdivided into the ones whose proofs are simple, and the ones that are hard or impossible to prove. For instance, the famous Fermat's Last Theorem was extremely easy to formulate, but its proof required some massively complex mathematical machinery.

Here, we are interested in theorems about natural numbers that are both easy to formulate and easy to prove. We'll assume that we know how to generate a family of propositions or, equivalently, a dependent type $T(n)$, where n is a natural number.

We'll also assume that we have a value:

$$\text{init}\colon T(Z)$$

or, equivalently, the proof of the zeroth proposition; and a dependent function:

$$\text{step}\colon \Pi_{n:\mathbb{N}}\ (T(n) \to T(Sn))$$

This function is interpreted as generating a proof of the $(n+1)$st proposition from the proof of the nth proposition.

The *dependent elimination rule* for natural numbers postulates that, given such init and step, there exists a dependent function:

$$f\colon \Pi_{n:\mathbb{N}}\, T(n)$$

This function is interpreted as providing the proof that $T(n)$ is true for all n.

Moreover, this function, when applied to zero reproduces init:

$$f(Z) = \mathrm{init}$$

and, when applied to the successor of n, is consistent with taking a step:

$$f(Sn) = (\mathrm{step}(n))(f(n))$$

(Here, $\mathrm{step}(n)$ produces a function, which is then applied to the value $f(n)$.) These are the two *computation rules* for natural numbers.

Notice that the induction principle is not a theorem about natural numbers. It's part of the *definition* of the type of natural numbers.

Not all dependent mappings out of natural numbers can be decomposed into init and step, just as not all theorems about natural numbers can be proven inductively. There is no η-conversion rule for natural numbers.

Equality elimination rule

The elimination rule for equality type is somewhat analogous to the induction principle for natural numbers. There we used init to ground ourselves at the start of the journey, and step to make progress. The elimination rule for equality requires a more powerful grounding, but it doesn't have a step. There really is no good analogy for how it works, other than through a leap of faith.

The idea is that we want to construct a mapping *out* of the equality type. But since equality type is itself a two-parameter family of types, the mapping out should be a dependent function. The target of this function is another family of types:

$$T(x, y, p)$$

that depends on the pair of values that are being compared $x, y \colon A$, and the proof of equality $p \colon \mathrm{Id}(x, y)$.

The function we are trying to construct is:

$$f : \Pi_{x,y:A}\Pi_{p:\mathrm{Id}(x,y)}\, T(x, y, p)$$

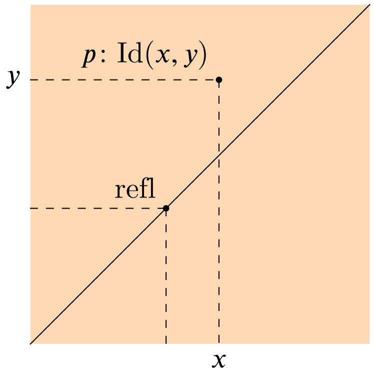
It's convenient to think of it as generating a proof that for all points x and y, and for every proof that the two are equal, the proposition $T(x, y, p)$ is true. Notice that, potentially, we have a different proposition for *every proof* that the two points are equal.

The least that we can demand from $T(x, y, p)$ is that it should be true when x and y are literally the same, and the equality proof is the obvious refl. This requirement can be expressed as a dependent function:

$$t \colon \Pi_{x:A}\, T\,(x, x, \mathrm{refl}(x))$$

Notice that we are not even considering proofs of $x = x$, other than those given by reflexivity. Do such proofs exist? We don't know and we don't care.

So this is our grounding, the starting point of a journey that should lead us to defining our f for all pairs of points and all proofs of equality. The intuition is that we are defining f as a function on a plane (x, y), with a third dimension given by p. To do that, we're given something that's defined on the diagonal (x, x), with p restricted to refl.



You would think that we need something more, some kind of a step that would move us from one point to another. But, unlike with natural numbers, there is no *next* point or *next* equality proof to jump to. All we have at our disposal is the function t and nothing else.

Therefore we postulate that, given a type family $T(x, y, p)$ and a function:

$$t\colon \Pi_{x:A}\, T\ (x, x, \mathrm{refl}(x))$$

there exists a function:

$$f\colon \Pi_{x,y:A}\Pi_{p:\mathrm{Id}(x,y)}\, T(x, y, p)$$

such that (computation rule):

$$f(x, x, \mathrm{refl}(x)) = t(x)$$

Notice that the equality in the computation rule is *definitional equality*, not a type.

Equality elimination tells us that it's always possible to extend the function t, which is defined on the diagonal, to the whole 3-d space.

This is a very strong postulate. One way to understand it is to argue that, within the framework of type theory—which is formulated using the language of introduction and elimination rules, and the rules for manipulating those—it's *impossible* to define a type family $T(x, y, p)$ that would *not* satisfy the equality elimination rule.

The closest analogy that we've seen so far is the result of parametricity, which states that, in Haskell, all polymorphic functions between endofunctors are automatically natural transformations. Another example, this time from calculus, is that any analytic function defined on the real axis has a unique extension to the whole complex plane.

The use of dependent types blurs the boundary between programming and mathematics. There is a whole spectrum of languages, starting with Haskell barely dipping its toes in dependent types while still firmly established in commercial usage, all the way to theorem provers, which are helping mathematicians formalize mathematical proofs.

Chapter 12

Algebras

The essence of algebra is the formal manipulation of expressions. But what are expressions, and how do we manipulate them?

The first things to observe about algebraic expressions like $2(x + y)$ or $ax^2 + bx + c$ is that there are infinitely many of them. There is a finite number of rules for making them, but these rules can be used in infinitely many combinations. This suggests that the rules are used *recursively*.

In programming, expressions are virtually synonymous to (parsing) trees. Consider this simple example of an arithmetic expression:

It's a recipe for building trees. We start with little trees using the constructor. We then plant these seedlings into nodes, and so on.

Such recursive definitions work perfectly well in a programming language. The problem is that every new recursive data structure would require its own library of functions that operate on it.

From type-theory point of view, we've been able to define recursive types, such as natural numbers or lists, by providing, in each case, specific introduction and elimination rules. What we need is something more general, a procedure for generating arbitrary recursive types from simpler pluggable components.

There are two orthogonal concerns when it comes to recursive data structures. One is the machinery of recursion. The other is the pluggable components.

We know how to work with recursion: We assume that we know how to construct small trees. We then use the recursive step to plant those trees into nodes to make bigger trees.

Category theory tells us how to formalize this imprecise description.

12.1 Algebras from Endofunctors

The idea of planting smaller trees into nodes requires that we formalize what it means to have a data structure with holes—a "container for stuff." This is exactly what functors are for. Because we want to use these functors recursively, they have to be *endo*-functors.

For instance, the endofunctor from our earlier example would be defined by the following data structure, where 's mark the spots:

Information about all possible shapes of expressions is abstracted into one single functor.

The other important piece of information when defining an algebra is the recipe for evaluating expressions. This, too, can be encoded using the same endofunctor.

Thinking recursively, let's assume that we know how to evaluate all subtrees of a larger expression. Then the remaining step is to plug these results into the top level node and evaluate it.

For instance, suppose that the 's in the functor were replaced by integers—the results of evaluation of the subtrees. It's pretty obvious what we should do in the last step. If the top of our tree is just a leaf (which means there were no subtrees to evaluate) we'll just return the integer stored in it. If it's a node, we'll add the two integers in it. This recipe can be encoded as:

We have made some seemingly obvious assumptions based on common sense. For instance, since the node was called we assumed that we should add the two numbers. But multiplication or subtraction would work equally well.

Since the leaf contained an integer, we assumed that the expression should evaluate to an integer. But there is an equally plausible evaluator that pretty-prints the expression by converting it to a string. This evaluator uses concatenation instead of addition:

In fact there are infinitely many evaluators, some sensible, others less so, but we shouldn't be judgmental. Any choice of the target type and any choice of the evaluator should be equally valid. This leads to the following definition:

An *algebra* for an endofunctor F is a pair (c, α). The object c is called the *carrier* of the algebra, and the evaluator $\alpha \colon Fc \to c$ is called the *structure map*.

In Haskell, given the functor we define:

Notice that the evaluator is *not* a polymorphic function. It's a specific choice of a function for a specific type . There may be many choices of the carrier types and there be many different evaluators for a given type. They all define separate algebras.

We have previously defined two algebras for . This one has as a carrier:

and this one has as a carrier:

12.2 Category of Algebras

Algebras for a given endofunctor F form a category. An arrow in that category is an algebra morphism, which is a structure-preserving arrow between their carrier objects.

Preserving structure in this case means that the arrow must commute with the two structure maps. This is where functoriality comes into play. To switch from one structure map to another, we have to be able to lift an arrow that goes between their carriers.

Given an endofunctor F, an *algebra morphism* between two algebras (a, α) and (b, β) is an arrow $f \colon a \to b$ that makes this diagram commute:

$$\begin{tikzcd}
Fa \arrow[r, "Ff"] \arrow[d, "\alpha"] & Fb \arrow[d, "\beta"] \\
a \arrow[r, "f"] & b
\end{tikzcd}$$

In other words, the following equation must hold:

$$f \circ \alpha = \beta \circ F f$$

The composition of two algebra morphisms is again an algebra morphism, which can be seen by pasting together two such diagrams (a functor maps composition to composition). The identity arrow is also an algebra morphism, because

$$id_a \circ \alpha = \alpha \circ F(id_a)$$

(a functor maps identity to identity).

The commuting condition in the definition of an algebra morphism is very restrictive. Consider for instance a function that maps an integer to a string. In Haskell there is a function (actually, a method of the class) that does it. It is *not* an algebra morphism from to .

Exercise 12.2.1. *Show that is not an algebra morphism. Hint: Consider what happens to the node.*

Initial algebra

The initial object in the category of algebras for a given functor F is called the *initial algebra* and, as we'll see, it plays a very important role.

By definition, the initial algebra (i, ι) has a unique algebra morphism f from it to any other algebra (a, α). Diagrammatically:

$$\begin{tikzcd}
Fi \arrow[r, "Ff"] \arrow[d, "\iota"] & Fa \arrow[d, "\alpha"] \\
i \arrow[r, dashed, "f"] & a
\end{tikzcd}$$

This unique morphism is called a *catamorphism* for the algebra (a, α). It is sometimes written using "banana brackets" as $(\!|\alpha|\!)$.

Exercise 12.2.2. *Let's define two algebras for the following functor:*

The first algebra:

The second algebra:

Make a convincing argument that (logarithm) is an algebra morphism between these two. (is a built-in floating-point number type.)

12.3 Lambek's Lemma and Fixed Points

Lambek's lemma says that the structure map ι of the initial algebra is an isomorphism.

The reason for it is the self-similarity of algebras. You can lift any algebra (a, α) using F, and the result $(Fa, F\alpha)$ is also an algebra with the structure map $F\alpha \colon F(Fa) \to Fa$.

In particular, if you lift the initial algebra (i, ι), you get a new algebra with the carrier Fi and the structure map $F\iota \colon F(Fi) \to Fi$. It follows then that there must be a unique algebra morphism from the initial algebra to it:

$$\begin{tikzcd}
Fi \arrow[r, "Fh"] \arrow[d, "\iota"] & F(Fi) \arrow[d, "F\iota"] \\
i \arrow[r, dashed, "h"] & Fi
\end{tikzcd}$$

This h is the inverse of ι. To see that, let's consider the composition $\iota \circ h$. It is the arrow at the bottom of the following diagram

$$\begin{tikzcd}
Fi \arrow[r, "Fh"] \arrow[d, "\iota"] & F(Fi) \arrow[r, "F\iota"] \arrow[d, "F\iota"] & Fi \arrow[d, "\iota"] \\
i \arrow[r, dashed, "h"] & Fi \arrow[r, "\iota"] & i
\end{tikzcd}$$

This is a pasting of the original diagram with a trivially commuting diagram. Therefore the whole rectangle commutes. We can interpret this as $\iota \circ h$ being an algebra morphism from (i, ι)

to itself. But there already is such an algebra morphism—the identity. So, by uniqueness of the mapping out from the initial algebra, these two must be equal:

$$\iota \circ h = id_i$$

Knowing that, we can now go back to the previous diagram, which states that:

$$h \circ \iota = F\iota \circ Fh$$

Since F is a functor, it maps composition to composition and identity to identity. Therefore the right hand side is equal to:

$$F(\iota \circ h) = F(id_i) = id_{Fi}$$

We have thus shown that h is the inverse of ι, which means that ι is an isomorphism. In other words:

$$Fi \text{ ö } i$$

We interpret this identity as stating that i is a fixed point of F (up to isomorphism). The action of F on i "doesn't change it."

There may be many fixed points, but this one is the *least fixed point* because there is an algebra morphism from it to any other fixed point. The least fixed point of an endofunctor F is denoted μF, so we write:

$$i = \mu F$$

Fixed point in Haskell

Let's consider how the definition of the fixed point works with our original example given by the endofunctor:

Its fixed point is a data structure defined by the property that acting on it reproduces it. If we call this fixed point , the fixed point equation becomes (in pseudo-Haskell):

Expanding we get:

Compare this with the recursive definition (actual Haskell):

We get a recursive data structure as a solution to the fixed-point equation.

In Haskell, we can define a fixed point data structure for any functor (or even just a type constructor). As we'll see later, this doesn't always give us the carrier of the initial algebra. It only works for those functors that have the "leaf" component.

Let's call the fixed point of a functor . Symbolically, the fixed-point equation can be written as:

$$f(\text{Fix}f) \text{ ö } \text{Fix}f$$

or, in code,

The data constructor is exactly the structure map of the initial algebra whose carrier is . Its inverse is:

The Haskell standard library contains a more idiomatic definition:

To create terms of the type we often use "smart constructors." For instance, with the functor, we would define:

and use it to generate expression trees like this one:

12.4 Catamorphisms

Our goal, as programmers, is to be able to perform a computation over a recursive data structure—to "fold" it. We now have all the ingredients.

The data structure is defined as a fixed point of a functor. An algebra for this functor defines the operation we want to perform. We've seen the fixed point and the algebra combined in the following diagram:

$$\begin{tikzcd}
Fi \arrow[r, "Ff"] \arrow[d, "\iota"] & Fa \arrow[d, "\alpha"] \\
i \arrow[r, dashed, "f"] & a
\end{tikzcd}$$

that defines the catamorphism f for the algebra (a, α).

The final piece of information is the Lambek's lemma, which tells us that ι could be inverted because it's an isomorphism. It means that we can read this diagram as:

$$f = \alpha \circ Ff \circ \iota^{*1}$$

and interpret it as a recursive definition of f.

Let's redraw this diagram using Haskell notation. The catamorphism depends on the algebra so, for the algebra with the carrier and the evaluator , we'll have the catamorphism .

$$\begin{tikzcd}
\texttt{f (Fix f)} \arrow[r, "\texttt{fmap (cata alg)}"] & \texttt{f a} \arrow[d, "\texttt{alg}"] \\
\texttt{Fix f} \arrow[u, "\texttt{out}"] \arrow[r, dashed, "\texttt{cata alg}"] & \texttt{a}
\end{tikzcd}$$

By simply following the arrows, we get this recursive definition:

Here's what's happening: We apply this definition to some . Every is obtained by applying to a functorful of :

The function "strips" the data constructor :

We can now evaluate the functorful of by 'ing over it. This is a recursive application. The idea is that the trees inside the functor are smaller than the original tree, so the recursion eventually terminates. It terminates when it hits the leaves.

After this step, we are left with a functorful of values, and we apply the evaluator to it, to get the final result.

The power of this approach is that all the recursion is encapsulated in one data type and one library function: We have the definition of and the catamorphism . The client of the library provides only the *non-recursive* pieces: the functor and the algebra. These are much easier to deal with. We have decomposed a complex problem into simpler components.

Examples

We can immediately apply this construction to our earlier examples. You can check that:

evaluates to 9 and

evaluates to the string .

Sometimes we want to display the tree on multiple lines with indentation. This requires passing a depth counter to recursive calls. There is a clever trick that uses a function type as a carrier:

The auxiliary function replicates the space character:

The result of:

when printed, looks like this:

Let's try defining algebras for other familiar functors. The fixed point of the functor:

after some renaming, is equivalent to the type of natural numbers

An algebra for this functor consists of a choice of the carrier and an evaluator:

The mapping out of is determined by two things: the value corresponding to and a function corresponding to . In our discussion of the type of natural numbers we called these and . We can now see that the elimination rule for is the catamorphism for this algebra.

Lists as initial algebras

The list type that we've seen previously is equivalent to a fixed point of the following functor, which is parameterized by the type of the list contents :

An algebra for this functor is a mapping out:

which is determined by the value and the function :

A catamorphism for such an algebra is the list recursor:

where can be identified with the fixed point .

We've seen before a recursive function that reverses a list. It was implemented by appending elements to the end of a list, which is very inefficient. It's easy to rewrite this function using a catamorphism, but the problem remains.

Prepending elements, on the other hand, is cheap. A better algorithm would traverse the list, accumulating elements in a first-in-first-out queue, and then pop them one-by-one and prepend them to a new list.

The queue regimen can be implemented using composition of closures: each closure is a function that remembers its environment. Here's the algebra whose carrier is a function type:

At each step, this algebra creates a new function. This function, when executed, will apply the previous function to a list, which is the result of prepending the current element to the function's argument . The resulting closure remembers the current element and the previous function .

The catamorphism for this algebra accumulates a queue of such closures. To reverse a list, we apply the result of the catamorphism for this algebra to the empty list:

This trick is at the core of the fold-left function, . Care should be taken when using it, because of the danger of stack overflow.

Lists are so common that their eliminators (called "folds") are included in the standard library. But there are infinitely many possible recursive data structures, each generated by its own functor, and we can use the same catamorphism on all of them.

It's worth mentioning that the list construction works in any monoidal category with coproducts. We can replace the list functor with the more general:

$$Fx = I + a \otimes x$$

where I is the unit object and $\otimes$ is the tensor product. The solution to the fixed point equation:

$$L_a \text{ ö } I + a \otimes L_a$$

can be formally written as a series:

$$L_a = I + a + a \otimes a + a \otimes a \otimes a + ...$$

We interpret this as a definition of a list, which can be empty I, a singleton a, a two-element list $a \otimes a$ and so on.

Incidentally, if you squint hard enough, this solution can be obtained by following a sequence of formal transformations:

$$\begin{aligned}
L_a &\text{ ö } I + a \otimes L_a \\
L_a * a \otimes L_a &\text{ ö } I \\
(I * a) \otimes L_a &\text{ ö } I \\
L_a &\text{ ö } I/(I * a) \\
L_a &\text{ ö } I + a + a \otimes a + a \otimes a \otimes a + ...
\end{aligned}$$

where the last step uses the formula for the sum of the geometric series. Admittedly, the intermediate steps make no sense, since there is no subtraction or division defined on objects, yet the final result make sense and, as we'll see later, it may be made rigorous by considering a colimit of a chain of objects.

12.5 Initial Algebra from Universality

Another way of looking at the initial algebra, at least in $\mathbf{Set}$, is to view it as a collection of catamorphisms that, as a whole, hint at the existence of an underlying object. Instead of seeing μF as a set of trees, we can look at it as a set of functions from algebras to their carriers.

In a way, this is just another manifestation of the Yoneda lemma: every data structure can be described either by mappings in or mappings out. The mappings in, in this case, are the constructors of the recursive data structure. The mappings out are all the catamorphisms that can be applied to it.

First, let's make the polymorphism in the definition of explicit:

and then flip the arguments. We get:

The function reverses the order of arguments to a function:

This gives us a mapping from to a set of polymorphic functions.

Conversely, given a polymorphic function of the type:

we can reconstruct :

In fact, these two functions, and , are the inverse of each other, establishing the isomorphism between and the type of polymorphic functions:

We can now substitute everywhere we used .

Folding over is easy, since carries in itself its own set of catamorphisms:

You might be wondering how one can construct terms of the type for, let's say lists. It can be done using recursion:

To compile this code you have to use the language pragma:

which puts the type variable in the scope of the clause.

Exercise 12.5.1. *Write a test that takes a list of integers, converts it to the form, and calculates the sum using* .

12.6 Initial Algebra as a Colimit

In general, there is no guarantee that the initial object in the category of algebras exists. But if it exists, Lambek's lemma tells us that it's a fixed point of the endofunctor for those algebras. The construction of this fixed point is a little mysterious, since it involves tying the recursive knot.

Loosely speaking, the fixed point is reached after we apply the functor infinitely many times. Then, applying it once more won't change anything. Infinity plus one is still infinity. This idea can be made precise if we take it one step at a time. For simplicity, let's consider algebras in the category of sets, which has all the nice properties.

We've seen, in our examples, that building instances of recursive data structures always starts with the leaves. The leaves are the parts in the definition of the functor that don't depend on the type parameter: the of the list, the of the tree, the of the , etc.

We can tease them out if we apply our functor F to the initial object—the empty set 0. Since the empty set has no elements, the instances of the type $F0$ are leaves only.

Indeed, the only inhabitant of the type is constructed using . The only inhabitants of the type are , where is an .

In other words, $F0$ is the "type of leaves" for the functor F. Leaves are trees of depth one. For the functor there's only one. The type of leaves for this functor is a singleton:

In the second iteration, we apply F to the leaves from the previous step and get trees of depth at most two. Their type is $F(F0)$.

For instance, these are all the terms of the type :

We can continue this process, adding deeper and deeper trees at each step. In the n-th iteration, the type F^n0 (n-fold application of F to the initial object) describes all trees of depth up to n. However, for every n, there are still infinitely many trees of depth greater than n that are not covered.

If we knew how to define $F^{\varnothing}0$, we would have covered all possible trees. The next best thing that we could try is to add up all those partial trees and construct an infinite sum type. Just like we have defined sums of two types, we can define sums of many types, including infinitely many.

An infinite sum (a coproduct):

$$\sum_{n=0}^{\varnothing} F^n 0$$

is just like a finite sum, except that it has infinitely many constructors i_n:

$$\begin{tikzcd}
0 \arrow[drr, "i_0"'] & F0 \arrow[dr, "i_1"] & F(F0) \arrow[d, "i_2"] & \ldots & F^n0 \arrow[dll, red, "i_n"] & \ldots \\
 & & \sum_{n=0}^{\varnothing} F^n 0 & & &
\end{tikzcd}$$

It has the universal mapping-out property, just like the sum of two types, only with infinitely many cases. (Obviously, we can't express it in Haskell.)

To construct a tree of depth n, we would first select it from $F^n 0$ and use the n-th constructor i_n to inject it into the sum.

There is just one problem: the same tree shape can also be constructed using any of the $F^m 0$, for $m > n$.

Indeed, we've seen the leaf appear in and . In fact it shows up in any nonzero power of acting on .

Similarly, shows up in all powers starting with two.

 shows up in all powers starting with three, and so on...

But there is a way to get rid of all these duplicates. The trick is to replace the sum by a colimit. Instead of a diagram consisting of discrete objects, we can construct a chain (such chains are called ω-chains). Let's call this chain Γ, and its colimit i:

$$i = \operatorname{Colim}\Gamma$$

$$\begin{tikzcd}
0 \arrow[r, "\text{¡}"] \arrow[drr, "i_0"'] & F0 \arrow[r, "F\text{¡}"] \arrow[dr, "i_1"] & F(F0) \arrow[r, "F(F\text{¡})"] \arrow[d, "i_2"] & \ldots \arrow[r] & F^n 0 \arrow[r] \arrow[dll, "i_n"] & \ldots \\
& & i & & &
\end{tikzcd}$$

It's almost the same as the sum, but with additional arrows at the base of the cocone. These arrows are the cumulative liftings of the unique arrow ¡ that goes from the initial object to $F0$ (we called it in Haskell). The effect of these arrows is to collapse the set of infinitely many copies of the same tree down to just one representative.

To see that, consider for instance a tree of depth 3. It can be first found as an element of $F^3 0$, that is to say, as an arrow $t \colon 1 \to F^3 0$. It is injected into the colimit i as the composite $i_3 \circ t$.

$$\begin{tikzcd}
\ldots & 1 \arrow[d, "t"] \arrow[dr, "t''"] & \\
\ldots \arrow[r] & F^3 0 \arrow[r, "F^3(\text{¡})"] \arrow[d, "i_3"] & F^4 0 \arrow[dl, "i_4"] \\
& i &
\end{tikzcd}$$

The same shape of a tree is also found in $F^4 0$, as the composite $t'' = F^3(\text{¡}) \circ t$. It is injected into the colimit as the composite $i_4 \circ t'' = i_4 \circ F^3(\text{¡}) \circ t$.

This time, however, we have the commuting triangle— the face of the cocone:

$$i_4 \circ F^3(\text{¡}) = i_3$$

which means that:

$$i_4 \circ t^{\ddot{\ }} = i_4 \circ F^3(\text{¡}) \circ t = i_3 \circ t$$

The two copies of the tree have been identified in the colimit. You can convince yourself that this procedure removes all duplicates.

The proof

We can prove directly that $i = \text{Colim}\,\Gamma$ is the initial algebra. There is however one assumption that we have to make: the functor F must preserve the colimits of ω-chains. The colimit of $F\Gamma$ must be equal to Fi.

$$\text{Colim}(F\Gamma) \text{ ö } Fi$$

Fortunately, this assumption holds in $\mathbf{Set}$[1].

Here's the sketch of the proof: To show the isomorphism, we'll first construct an arrow $i \to Fi$ and then an arrow $\iota \colon Fi \to i$ in the opposite direction. We'll skip the proof that they are the inverse of each other. Then we'll show the universality of (i, ι) by constructing a catamorphism to an arbitrary algebra.

All subsequent proofs follow a simple pattern. We start with a universal cocone that defines a colimit. Then we construct another cocone based on the same chain. From universality, there must be a unique arrow from the colimit to the apex of this new cocone.

We use this trick to construct the mapping $i \to Fi$. If we can construct a cocone from the chain Γ to $\text{Colim}(F\Gamma)$ then, by universality, there must be an arrow from i to $\text{Colim}(F\Gamma)$. The latter, by our assumption that F preserves colimits, is isomorphic to Fi. So we'll have a mapping $i \to Fi$.

To construct this cocone, first notice that $\text{Colim}(F\Gamma)$ is, by definition, the apex of a cocone $F\Gamma$.

$$\begin{tikzcd}
F0 \arrow[r, "F\text{¡}"] \arrow[drr, "j_1"'] & F(F0) \arrow[r, "F(F\text{¡})"] \arrow[dr, "j_2"] & F^3 0 \arrow[r, "F^3\text{¡}"] \arrow[d, "j_3"] & \dots \arrow[r] & F^n 0 \arrow[r] \arrow[dll, "j_n"] & \dots \\
& & \text{Colim}(F\Gamma) & & &
\end{tikzcd}$$

The diagram $F\Gamma$ is the same as Γ, except that it's missing the naked initial object at the start of the chain.

[1] This is the consequence of the fact that colimits in $\mathbf{Set}$ are built from disjoint unions of sets.

The spokes of the cocone we are looking for, from Γ to $\text{Colim}(F\Gamma)$, are marked in red in the diagram below:

$$\begin{tikzcd}
0 \arrow[r, "¡"] \arrow[d, red, "¡"] & F0 \arrow[r, "F¡"] \arrow[d, red, "F¡"] & F(F0) \arrow[r, "F(F¡)"] \arrow[d, red, "F(F¡)"] & \ldots \arrow[r] & F^{n+1}0 \arrow[r] \arrow[d, red] & \ldots \\
F0 \arrow[r, blue, "F¡"] \arrow[drr, red, "j_1"'] & F(F0) \arrow[r, blue, "F(F¡)"] \arrow[dr, red, "j_2"] & F^3 0 \arrow[r, blue, "F^3¡"] \arrow[d, red, "j_3"] & \ldots \arrow[r, blue] & F^n 0 \arrow[r, blue] \arrow[dll, red, "j_n"] & \ldots \\
& & \text{Colim}(F\Gamma) & & &
\end{tikzcd}$$

Since $i = \text{Colim}\,\Gamma$ is the apex of the universal cocone based on Γ, there must be a unique mapping out of it to $\text{Colim}(F\Gamma)$ which, as we said, was equal to Fi. This is the mapping we were looking for:

$$i \to Fi$$

Next, notice that the chain $F\Gamma$ is a sub-chain of Γ, so it can be embedded in it. It means that we can construct a cocone from $F\Gamma$ to the apex i by going through (a sub-chain of) Γ (the red arrows below).

$$\begin{tikzcd}
& F0 \arrow[r, blue, "F¡"] \arrow[d, red] & F(F0) \arrow[r, blue, "F(F¡)"] \arrow[d, red] & \ldots \arrow[r, blue] & F^n 0 \arrow[r, blue] \arrow[d, red] & \ldots \\
0 \arrow[r, "¡"] \arrow[drr, "i_0"'] & F0 \arrow[r, "F¡"] \arrow[dr, red, "i_1"] & F^2 0 \arrow[r, "F^2¡"] \arrow[d, red, "i_2"] & \ldots \arrow[r] & F^n 0 \arrow[r] \arrow[dll, red, "i_n"] & \ldots \\
& & i & & &
\end{tikzcd}$$

From the universality of the $\text{Colim}(F\Gamma)$ it follows that there is a mapping out

$$\text{Colim}(F\Gamma) \to i$$

and thus we have the mapping in the other direction:

$$\iota \colon Fi \to i$$

This shows that i is a carrier of an algebra. In fact, it can be shown that the two mappings are the inverse of each other, as we would expect from the Lambek's lemma.

To show that (i, ι) is indeed the initial algebra, we have to construct a mapping out of it to an arbitrary algebra $(a, \alpha \colon Fa \to a)$. Again, we can use universality, as long as we can construct a cocone from Γ to a.

$$\begin{tikzcd}
0 \arrow[r, "¡"] \arrow[drr, "f_0"'] & F0 \arrow[r, "F¡"] \arrow[dr, "f_1"] & F(F0) \arrow[r, "F(F¡)"] \arrow[d, "f_2"] & ... \arrow[r] & F^n0 \arrow[r] \arrow[dll, "f_n"] & ... \\
& & a & & &
\end{tikzcd}$$

The zeroth spoke of this cocone goes from 0 to a, so it's just $f_0 = ¡$.

The first spoke, $F0 \to a$, is $f_1 = \alpha \circ F f_0$, because $F f_0 \colon F0 \to Fa$ and $\alpha \colon Fa \to a$.

The third spoke, $F(F0) \to a$ is $f_2 = \alpha \circ F f_1$. And so on...

The unique mapping from i to a is then our catamorphism. With some more diagram chasing, it can be shown that it's indeed an algebra morphism.

Notice that this construction only works if we can "prime" the process by creating the leaves of the functor. If, on the other hand, $F0$ ö 0, then there are no leaves, and all further iterations will keep reproducing 0.

Chapter 13

Coalgebras

Coalgebras are just algebras in the opposite category. End of chapter!

Well, maybe not... As we've seen before, the category in which we're working is not symmetric with respect to duality. In particular, if we compare the terminal and the initial objects, their properties are not symmetric. Our initial object has no incoming arrows, whereas the terminal one, besides having unique incoming arrows, has lots of outgoing arrows.

Since initial algebras were constructed starting from the initial object, we might expect terminal coalgebras—which are their duals, therefore generated from the terminal object—not to be just their mirror images, but to add their own interesting twists.

We've seen that the main application of algebras was in processing recursive data structures: in folding them. Dually, the main application of coalgebras is in generating, or unfolding, the recursive, tree-like, data structures. The unfolding is done using an anamorphism.

We use catamorphisms to chop trees, we use anamorphisms to grow them.

We cannot produce information from nothing so, in general, both a catamorphism and an anamorphism tend to reduce the amount of information that's contained in their input.

After you sum a list of integers, it's impossible to recover the original list.

By the same token, if you grow a recursive data structure using an anamorphism, the seed must contain all the information that ends up in the tree. You don't gain new information, but the advantage is that the information you have is now stored in a form that's more convenient for further processing.

13.1 Coalgebras from Endofunctors

A coalgebra for an endofunctor F is a pair consisting of a carrier a and a structure map: an arrow $a \to Fa$.

In Haskell, we define:

We often think of the carrier as the type of a seed from which we grow the data structure, be it a list or a tree.

For instance, here's a functor that can be used to create a binary tree, with integers stored at the nodes:

We don't even have to define the instance of for it—the clause tells the compiler to generate the canonical one for us (together with the instance to allow conversion to , if we want to display it).

A coalgebra is a function that takes a seed of the carrier type and produces a functorful of new seeds. These new seeds can then be used to generate the subtrees, recursively.

Here's a coalgebra for the functor that takes a list of integers as a seed:

If the seed is empty, it generates a leaf; otherwise it creates a new node. This node stores the head of the list and fills the node with two new seeds. The library function splits a list using a user-defined predicate, here , less-than-or-equal to . The result is a pair of lists: the first one satisfying the predicate; and the second, not.

You can convince yourself that a recursive application of this coalgebra creates a binary sorted tree. We'll use this coalgebra later to implement a sort.

13.2 Category of Coalgebras

By analogy with algebra morphisms, we can define coalgebra morphisms as the arrows between carriers that satisfy a commuting condition.

Given two coalgebras (a, α) and (b, β), the arrow $f : a \to b$ is a coalgebra morphism if the following diagram commutes:

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[d, "\alpha"] & b \arrow[d, "\beta"] \\
Fa \arrow[r, "Ff"] & Fb
\end{tikzcd}$$

The interpretation is that it doesn't matter if we first map the carriers and then apply the coalgebra β, or first apply the coalgebra α and then apply the arrow to its contents, using the lifting Ff.

Coalgebra morphisms can be composed, and the identity arrow is automaticaly a coalgebra morphism. It's easy to see that coalgebras, just like algebras, form a category.

This time, however, we are interested in the terminal object in this category—a *terminal coalgebra*. If a terminal coalgebra (t, τ) exists, it satisfies the dual of the Lambek's lemma.

Exercise 13.2.1. *Lambek's lemma: Show that the structure map τ of the terminal coalgebra (t, τ) is an isomorphism. Hint: The proof is dual to the one for the initial algebra.*

As a consequence of the Lambek's lemma, the carrier of the terminal algebra is a fixed point of the endofunctor in question.

$$Ft \text{ ö } t$$

with τ and τ^{*1} serving as the witnesses of this isomorphism.

It also follows that (t, τ^{*1}) is an algebra; just as (i, ι^{*1}) is a coalgebra, assuming that (i, ι) is the initial algebra.

We've seen before that the carrier of the initial algebra is a fixed point. In principle, there may be many fixed points for the same endofunctor. The initial algebra is the least fixed point and the terminal coalgebra the greatest fixed point.

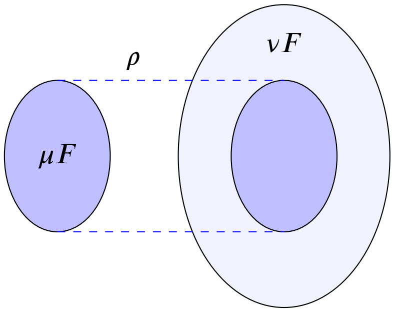
The greatest fixed point of an endofunctor F is denoted by νF, so we have:

$$t = \nu F$$

We can also see that there must be a unique algebra morphism (a catamorphism) from the initial algebra to the terminal coalgebra. That's because the terminal coalgebra is also an algebra.

Similarly, there is a unique coalgebra morphism from the initial algebra (which is also a coalgebra) to the terminal coalgebra. In fact, it can be shown that it's the same underlying morphism $\rho \colon \mu F \to \nu F$ in both cases.

In the category of sets, the carrier set of the initial algebra is a subset of the carrier set of the terminal coalgebra, with the function ρ embedding the former in the latter.



We'll see later that in Haskell the situation is more subtle, because of lazy evaluation. But, at least for functors that have the leaf component—that is, their action on the initial object is non-trivial—Haskell's fixed point type works as a carrier for both the initial algebra and the terminal coalgebra.

Exercise 13.2.2. *Show that, for the identity functor in* Set*, every object is a fixed point, the empty set is the least fixed point, and the singleton set is the greatest fixed point. Hint: The least fixed point must have arrows going to all other fixed points, and the greatest fixed point must have arrows coming from all other fixed points.*

Exercise 13.2.3. *Show that the empty set is the carrier of the initial algebra for the identity functor in* Set*. Dually, show that the singleton set is this functor's terminal coalgebra. Hint: Show that the unique arrows are indeed (co-) algebra morphisms.*

13.3 Anamorphisms

The terminal coalgebra (t, τ) is defined by its universal property: there is a unique coalgebra morphism h from any coalgebra (a, α) to (t, τ). This morphism is called the *anamorphism*. Being a coalgebra morphism, it makes the following diagram commute:

$$\begin{tikzcd}
a \arrow[r, dashed, "h"] \arrow[d, "\alpha"] & t \arrow[d, "\tau"] \\
Fa \arrow[r, "Fh"] & Ft
\end{tikzcd}$$

Just like with algebras, we can use the Lambek's lemma to "solve" for :

$$h = \tau^{*1} \circ F h \circ \alpha$$

The solution is called an anamorphism and is sometimes written using "lens brackets" as $[\!(\alpha)\!]$.

Since the terminal coalgebra (just like the initial algebra) is a fixed point of a functor, the above recursive formula can be translated directly to Haskell as:

Here's the interpretation of this formula: Given a seed of type , we first act on it with the coalgebra . This gives us a functorful of seeds. We expand these seeds by recursively applying the anamorphism using . We then apply the constructor to get the final result.

As an example, we can apply the anamorphism to the coalgebra we defined earlier: takes a list of integers and creates a sorted tree.

We can then use a catamorphsims to fold this tree into a sorted list. We define the following algebra:

It concatenates the left list with the singleton pivot and the right list. To sort a list we combine the anamorphism with the catamorphism:

This gives us a (very inefficient) implementation of quicksort. We'll come back to it in the next section.

Infinite data structures

When studying algebras we relied on data structures that had a leaf component—that is endofunctors that, when acting on the initial object, would produce a result different from the initial object. When constructing recursive data structures we had to start somewhere, and that meant constructing the leaves first.

With coalgebras, we are free to drop this requirement. We no longer have to construct recursive data structures "by hand"—we have anamorphisms to do that for us. An endofunctor that has no leaves is perfectly acceptable: its coalgebras are going to generate infinite data structures.

Infinite data structures are representable in Haskell because of its laziness. Things are evaluated on the need-to-know basis. Only those parts of an infinite data structure that are explicitly demanded are calculated; the evaluation of the rest is kept in suspended animation.

To implement infinite data structures in strict languages, one must resort to representing values as functions—something Haskell does behind the scenes (these functions are called *thunks*).

Let's look at a simple example: an infinite stream of values. To generate it, we first define a functor that looks very much like the one we used to generate lists, except that it lacks the leaf component (the empty-list constructor). You may recognize it as a product functor, with the first component fixed to be the stream's payload:

An infinite stream is the fixed point of this functor.

Here's a simple coalgebra that uses a single integer as a seed:

It stores the current seed as a payload, and seeds the next budding stream with .

The anamorphism for this coalgebra, when seeded with zero, generates the stream of all natural numbers.

In a non-lazy language this anamorphism would run forever, but in Haskell it's instantaneous. The incremental price is paid only when we want to retrieve some of the data, for instance, using these accessors:

13.4 Hylomorphisms

The type of the output of an anamorphism is a fixed point of a functor, which is the same type as the input to a catamorphism. In Haskell, they are both described by the same data type, . Therefore it's possible to compose them together, as we've done when implementing quicksort. In fact, we can combine a coalgebra with an algebra in one recursive function called a *hylomorphism*:

We can rewrite quicksort as a hylomorphism:

Notice that there is no trace of the fixed point in the definition of the hylomorphism. Conceptually, the coalgebra is used to build (unfold) the recursive data structure from the seed, and the algebra is used to fold it into a value of type . But because of Haskell's laziness, the intermediate data structure doesn't have to be materialized in full in memory. This is particularly important when dealing with very large intermediate trees. Only the branches that are currently being traversed are evaluated and, as soon as they have been processed, they are passed to the garbage collector.

Hylomorphisms in Haskell are a convenient replacement for recursive backtracking algorithms, which are very hard to implement correctly in imperative languages. We take advantage of the fact that designing a data structure is easier than following complicated flow of control and keeping track of our place in a recursive algorithm.

This way, data structures can be used to visualize complex flows of control.

The impedance mismatch

We've seen that, in the category of sets, the initial algebras don't necessarily coincide with terminal coalgebras. The identity functor, for instance, has the empty set as the carrier of the initial algebra and the singleton set as the carrier of its terminal coalgebra.

We have other functors that have no leaf components, such as the stream functor. The initial algebra for such a functor is the empty set as well.

In $\mathbf{Set}$, the initial algebra is the subset of the terminal coalgebra, and hylomorphisms can only be defined for this subset. It means that we can use a hylomorphism only if the anamorphism for a particular coalgebra lands us in this subset. In that case, because the embedding of initial algebras in terminal coalgebras is injective, we can find the corresponding element in the initial algebra and apply the catamorphism to it.

In Haskell, however, we have one type, , combining both, the initial algebra and the terminal coalgebra. This is where the simplistic interpretation of Haskell types as sets of values breaks down.

Let's consider this simple stream algebra:

Nothing prevents us from using a hylomorphism to calculate the sum of all natural numbers:

It's a perfectly well-formed Haskell program that passes the type checker. So what value does it produce when we run it? (Hint: It's not $*1/12$.) The answer is: we don't know, because this program never terminates. It runs into infinite recursion and eventually exhausts the computer's resources.

This is the aspect of real-life computations that mere functions between sets cannot model. Some computer function may never terminate.

Recursive functions are formally described by *domain theory* as limits of partially defined functions. If a function is not defined for a particular value of the argument, it is said to return a bottom value $\bot$. If we include bottoms as special elements of every type (these are then called *lifted* types), we can say that our function returns a bottom of the type . In general, catamorphisms for infinite types don't terminate, so we can treat them as returning bottoms.

It should be noted, however, that the inclusion of bottoms complicates the categorical interpretation of Haskell. In particular, many of the universal constructions that rely on uniqueness of mappings no longer work as advertised.

The "bottom" line is that Haskell code should be treated as an illustration of categorical concepts rather than a source of rigorous proofs.

13.5 Terminal Coalgebra from Universality

The definition of an anamorphism can be seen as an expression of the universal property of the terminal coalgebra. Here it is, with the universal quantification made explicit:

What it tells us is that, given any coalgebra, there is a mapping from its carrier to the carrier of the terminal coalgebra, . We know, from the Lambek's lemma, that this mapping is in fact a coalgebra morphism.

Let's uncurry this definition:

We can use this formula as the alternative definition of the carrier for the terminal coalgebra. We can replace with the type we are defining—let's call it . The type signature:

tells us that we can construct an element of from a pair . It looks just like a data constructor, except that it's polymorphic in .

Data types with a polymorphic constructor are called *existential* types. In pseudo-code (not actual Haskell) we would define as:

Compare this with the definition of the least fixed point of an algebra:

To construct an element of an existential type, we have the option of picking the most convenient type—the type for which we have the data required by the constructor.

For instance, we can construct a term of the type by picking as the convenient type, and providing the pair:

The clients of an existential data type have no idea what type was used in its construction. All they know is that such a type *exists*—hence the name. If they want to use an existential type, they have to do it in a way that is not sensitive to the choice that was made in its construction. In practice, it means that an existential type must carry with itself both the producer and the consumer of the hidden value.

This is indeed the case in our example: the producer is just the value of type , and the consumer is the function .

Naively, all that the clients could do with this pair, without any knowledge of what the type was, is to apply the function to the value. But if is a functor, they can do much more. They can repeat the process by applying the lifted function to the contents of , and so on. They end up with all the information that's contained in the infinite stream.

There are several ways of defining existential data types in Haskell. We can use the uncurried version of the anamorphism directly as the data constructor:

Notice that, in Haskell, if we explicitly quantify one type, all other type variables must also be quantified: here, it's the type constructor (however, is not existential in , since it's an explicit parameter).

We can also omit the quantification altogether:

This is because type variables that are not arguments to the type constructor are automatically treated as existentials.

We can also use the more traditional form:

(This one requires the quantification of .)

At the time of this writing there is a proposal to introduce the keyword to Haskell that would make this definition work:

(Later we'll see that existential data types correspond to coends in category theory.)

The constructor of is literally the (uncurried) anamorphism:

If we are given a stream in the form of , we can access its element using accessors functions. This one extracts the first element:

and this one advances the stream:

You can test them on an infinite stream of integers:

13.6 Terminal Coalgebra as a Limit

In category theory we are not afraid of infinities—we make sense of them.

At face value, the idea that we could construct a terminal coalgebra by applying the functor F infinitely many times to some object, let's say the terminal object 1, makes no sense. But the idea is very convincing: Applying F one more time is like adding one to infinity—it's still infinity. So, naively, this is a fixed point of F:

$$F(F^{\text{Ø}}1)\ \text{ö}\ F^{\text{Ø}+1}1\ \text{ö}\ F^{\text{Ø}}1$$

To turn this loose reasoning into a rigorous proof, we have to tame the infinity, which means we have to define some kind of limiting procedure.

As an example, let's consider the product functor:

$$F_a x = a \quad x$$

Its terminal coalgebra is an infinite stream. We'll approximate it by starting with the terminal object 1. The next step is:

$$F_a 1 = a \quad 1\ \text{ö}\ a$$

which we could imagine is a stream of length one. We can continue with:

$$F_a(F_a 1) = a \quad (a \quad 1)\ \text{ö}\ a \quad a$$

a stream of length two, and so on.

This looks promising, but what we need is one object that would combine all these approximations. We need a way to glue the next approximation to the previous one.

Recall, from an earlier exercise, the limit of the "walking arrow" diagram. This limit has the same elements as the starting object in the diagram. In particular, consider the limit of the single-arrow diagram D_1:

$$\begin{tikzcd}
& LimD_1 \arrow[dl, "\pi_0"'] \arrow[d, "\pi_1"] \\
1 & F1 \arrow[l, "!"]
\end{tikzcd}$$

(! is the unique morphism targeting the terminal object 1). This limit has the same elements as $F1$. Similarly, the limit of a two-arrow diagram D_2:

$$\begin{tikzcd}
& & LimD_2 \arrow[dll, "\pi_0"'] \arrow[dl, "\pi_1"] \arrow[d, "\pi_2"] \\
1 & F1 \arrow[l, "!"] & F(F1) \arrow[l, "F!"]
\end{tikzcd}$$

has the same elements as $F(F1)$.

We can continue extending this diagram to infinity. It turns out that the limit of this infinite chain is our fixed point carrier of the terminal coalgebra.

$$\begin{tikzcd}
& & t \arrow[dll, "\pi_0"'] \arrow[dl, "\pi_1"] \arrow[d, "\pi_2"] \arrow[drr, "\pi_n"] & & & \\
1 & F1 \arrow[l, "!"] & F(F1) \arrow[l, "F!"] & ... \arrow[l, "F(F!)"] & F^n1 \arrow[l] & ... \arrow[l, "F^n!"]
\end{tikzcd}$$

The proof of this fact can be obtained from the analogous proof for initial algebras by reversing the arrows.

Chapter 14

Monads

What do a wheel, a clay pot, and a wooden house have in common? They are all useful because of the emptiness in their center.

Lao Tzu says: “The value comes from what is there, but the use comes from what is not there.”

What does the functor, the list functor, and the reader functor have in common? They all have emptiness in their center.

When monads are explained in the context of programming, it’s hard to see the common pattern when you focus on the functors. To understand monads you have to look inside functors and read between the lines of code.

14.1 Programming with Side Effects

So far we’ve been talking about programming in terms of computations that were modeled mainly on functions between sets (with the exception of non-termination). In programming, such functions are called *total* and *pure*.

A total function is defined for all values of its arguments.

A pure function is implemented purely in terms of its arguments and, in case of closures, the captured values—it has no access to, much less the ability to modify the outside world.

Most real-world programs, though, have to interact with the external world: they read and write files, process network packets, prompt users for data, etc. Most programming languages solve this problem by allowing side effect. A side effect is anything that breaks the totality or the purity of a function.

Unfortunately, this shotgun approach adopted by imperative languages makes reasoning about programs extremely hard. When composing effectful computations one has to carefully reason about the composition of effects on a case-by-case basis. To make things even harder, most effects are hidden not only inside the implementation (as opposed to the interface) of a particular function, but also in the implementation of all the functions that it's calling, recursively.

The solution adopted by purely functional languages, like Haskell, is to encode side effects in the return types of pure functions. Amazingly, this is possible for all relevant effects.

The idea is that, instead of a computation of the type with side effects, we use a function , where the type constructor encodes the appropriate effect. At this point there are no conditions imposed on . It doesn't even have to be a , much less a monad. This will come later, when we talk about effect composition.

Below is the list of common effects and their pure-function versions.

Partiality

In imperative languages, partiality is often encoded using exceptions. When a function is called with the "wrong" value for its argument, it throws an exception. In some languages, the type of exception is encoded in the signature of the function using special syntax.

In Haskell, a partial computation can be implemented by a function returning the result inside the functor. Such a function, when called with the "wrong" argument, returns , otherwise is wraps the result in the constructor.

If we want to encode more information about the type of the failure, we can use the functor, with the traditionally passing the error data (often a simple); and encapsulating the real return, if available.

The caller of a -valued function cannot easily ignore the exceptional condition. In order to extract the value, they have to pattern-match the result and decide how to deal with . This is in contrast to the "poor-man's " of some imperative languages where the error condition is encoded using a null pointer.

Logging

Sometimes a computation has to log some values in some external data structure. Logging or auditing is a side effect that's particularly dangerous in concurrent programs, where multiple threads might try to access the same log simultaneously.

The simple solution is for a function to return the computed value paired with the item to be logged. In other words, a logging computation of the type can be replaced by a pure function:

where the functor is a thin encapsulation of the product:

with being the type of the log.

The caller of this function is then responsible for extracting the value to be logged. This is a common trick: make the function provide all the data, and let the caller deal with the effects.

Environment

Some computations need read-only access to some external data stored in the environment. The read-only environment, instead of being secretly accessed by a computation, can be simply passed to a function as an additional argument. If we have a computation that needs access to some environment , we replace it with a function . At first, this doesn't seem to fit the pattern of encoding side effects in the return type. However, such a function can always be curried to the form:

The return type of this function can be encoded in the reader functor, itself parameterized by the environment type :

This is an example of a delayed side effect. The function:

doesn't want to deal with effects so it delegates the responsibility to the caller. You may think of it as producing a script to be executed at a later time. The function plays the role of a very simple interpreter of this script:

State

The most common side effect is related to accessing and potentially modifying some shared state. Unfortunately, shared state is the notorious source of concurrency errors. This is a serious problem in object-oriented languages where stateful objects can be transparently shared between many clients. In Java, such objects may be provided with individual mutexes at the cost of impaired performance and the risk of deadlocks.

In functional programming we make state manipulations explicit: we pass the state as an additional argument and return the modified state paired with the return value. We replace a stateful computation with

where is the type of state. As before, we can curry such a function to get it to the form:

This return type can be encoded in the following functor:

The caller of such a function is supposed to retrieve the result and the modified state by providing the initial state and calling the helper function, the interpreter, :

Notice that, modulo constructor unpacking, is bona fide function application.

Nondeterminism

Imagine performing a quantum experiment that measures the spin of an electron. Half of the time the spin will be up, and half of the time it will be down. The result is non-deterministic. One way to describe it is to use the many-worlds interpretation: when we perform the experiment, the Universe splits into two universes, one for each result.

What does it mean for a function to be non-deterministic? It means that it will return different results every time it's called. We can model this behavior using the many-worlds interpretation: we let the function return *all possible results* at once. In practice, we'll settle for a (possibly infinite) list of results:

We replace a non-deterministic computation with a pure function returning a functorful of results—this time it's the list functor:

Again, it's up to the caller to decide what to do with these results.

Input/Output

This is the trickiest side effect because it involves interacting with the external world. Obviously, we cannot model the whole world inside a computer program. So, in order to keep the program pure, the interaction has to happen outside of it. The trick is to let the program generate a script. This script is then passed to the runtime to be executed. The runtime is the effectful virtual machine that runs the program.

This script itself sits inside the opaque, predefined functor. The values hidden in this functor are not accessible to the program: there is no function. Instead, the value produced by the program is executed, at least conceptually, *after* the program is finished.

In reality, because of Haskell's laziness, the execution of I/O is interleaved with the rest of the program. Pure functions that comprise the bulk of your program are evaluated on demand—the demand being driven by the execution of the script. If it weren't for I/O, nothing would ever be evaluated.

The object that is produced by a Haskell program is called and its type signature is:

It's the functor containing the unit—meaning: there is no useful value other than the input/output script.

We'll talk about how actions are created soon.

Continuation

We've seen that, as a consequence of the Yoneda lemma, we can replace a value of type with a function that takes a handler for that value. This handler is called a continuation. Calling a handler is considered a side effect of a computation. In terms of pure functions, we encode it as:

where is the following functor:

It's the responsibility of the caller of this function to provide the continuation, a function , and retrieve the result:

This is the instance for :

Notice that this is a covariant functor because the type is in a doubly negative position.

In a cartesian closed category, continuations are generated by the endofunctor:

$$K_r a = r^{r^a}$$

14.2 Composing Effects

Now that we know how to make one giant leap using a function that produces both a value and a side effect, the next problem is to figure out how to decompose this leap into smaller human-sized steps. Or, conversely, how to combine such smaller steps into one larger step.

The way effectful computations are composed in imperative languages is to use regular function composition for the values and let the side effects combine themselves willy-nilly.

When we represent effectful computations as pure functions, we are faced with the problem of composing two functions of the form

In all cases of interest the type constructor happens to be a , so we'll assume that in what follows.

The naive approach would be to unpack the result of the first function, pass the value to the next function, then compose the effects of both functions on the side, and combine them with the result of the second function. This is not always possible, even for cases that we have studied so far, much less for an arbitrary type constructor.

For the sake of the argument, it's instructive to see how we could do it for the functor. If the first function returns , we pattern match it to extract the contents and call the next function with it.

But if the first function returns , we have no value with which to call the second function. We have to short-circuit it, and return directly. So composition is possible, but it means modifying flow of control by skipping the second call based on the side effect of the first call.

For some functors the composition of side effects is possible, for others it's not. How can we characterize those "good" functors?

For a functor to encode composable side effects we must at least be able to implement the following polymorphic higher-order function:

This is very similar to regular function composition:

so it's natural to ask if there is a category in which the former defines a composition of arrows. Let's see what more is needed to construct such a category.

Objects in this new category are the same Haskell types as before. But an arrow $a \twoheadrightarrow b$, is implemented as a Haskell function:

Our can then be used to implement the composition of such arrows.

To have a category, we require that this composition be associative. We also need an identity arrow for every object . This is an arrow $a \twoheadrightarrow a$, so it corresponds to a Haskell function:

It must behave like identity with respect to .

Another way of looking at this arrow is that it lets you add a trivial effect to any value of type . It's the effect that combined with any other effect does nothing to it.

We have just defined a monad! After some renaming and rearranging, we can write it as a typeclass:

The infix operator replaces the function . The function is the identity arrow in our new category. (This is not the definition of the monad you'll find in the Haskell's but, as we'll see soon, it's equivalent to it.)

As an exercise, let's define the instance for . The "fish" operator composes two functions:

into one function of the type . The unit of this composition, , encloses a value in the constructor.

You can easily convince yourself that category laws are satisfied. In particular is the same as and is the same as . The proof of associativity is also pretty straightforward: If any of the functions returns , the result is ; otherwise it's just a straightforward function composition, which is associative.

The category that we have just defined is called the *Kleisli category* for the monad . The functions are called the *Kleisli arrows*. They compose using and the identity arrow is called .

All functors from the previous section are instances. If you look at them as type constructors, or even functors, it's hard to see any similarities between them. The thing they have in common is that they can be used to implement *composable* Kleisli arrows.

As Lao Tze would say: Composition is something that happens *between* things. While focusing our attention on things, we often lose sight of what's in the gaps.

14.3 Alternative Definitions

The definition of a monad using Kleisli arrows has the advantage that the monad laws are simply the associativity and the unit laws of a category. There are two other equivalent definitions of a monad, one preferred by mathematicians, and one by programmers.

First, let's notice that, when implementing the fish operator, we are given two functions as arguments. The only thing a function is useful for is to be applied to an argument. When we apply the first function we get a value of the type . At this point we would be stuck, if it weren't for the fact that is a functor. Functoriality lets us apply the second function to . Indeed the lifting of by is of the type:

This is almost the result we are looking for, if we could only flatten to . This flattening is called . In other words, if we are given:

we can implement :

or, using point free notation:

Conversely, can be implemented in terms of :

The latter may not be immediately obvious, until you realize that the rightmost is applided to , and the leftmost is applied to . We interpret a Haskell function:

as an arrow in the Kleisli category $m(ma) \twoheadrightarrow ma$. Similarly, the function:

implements a Kleisli arrow $ma \twoheadrightarrow a$. Their Kleisli composition produces a Kleisli arrow $m(ma) \twoheadrightarrow a$ or a Haskell function:

This leads us to the equivalent definition of a monad in terms of and :

This is still not the definition you will find in the standard Haskell . Since the fish operator is a generalization of the dot operator, using it is equivalent to point-free programming. It lets us compose arrows without naming intermediate values. Although some consider point-free programs more elegant, most programmers find them difficult to follow.

But function composition is really done in two steps: We apply the first function, then apply the second function to the result. Explicitly naming the intermediate result is often helpful in understanding what's going on.

To do the same with Kleisli arrows, we have to know how to apply the second Kleisli arrow to a named monadic value—the result of the first Kleisli arrow. The function that does that is called *bind* and is written as an infix operator:

Obviously, we can implement Kleisli composition in terms of bind:

Conversely, bind can be implemented in terms of the Kleisli arrow:

This leads us to the following definition:

This is almost the definition you'll find in the , except for the additional constraint. This constraint states the fact that every instance of is also an instance of . We will postpone the discussion of applicatives to the section on monoidal functors.

We can also implement using bind:

The Haskell function goes from to or, as a Kleisli arrow, $ma \twoheadrightarrow a$.

Interestingly, a defined using bind is automatically a functor. The lifting function for it is called

14.4 Monad Instances

We are now ready to define monad instances for the functors we used for side effects. This will allow us to compose side effects.

Partiality

We've already seen the version of the monad implemented using Kleisli composition. Here's the more familiar implementation using bind:

Adding a trivial effect to any value means enclosing it in .

Logging

In order to compose functions that produce logs, we need a way to combine individual log entries. This is why the writer monad:

requires the type of the log to be an instance of . This allows us to append logs, and also to create a trivial effect: an empty log.

The clause is used for introducing local bindings. Here, the result of applying is pattern matched, and the local variables and are initialized. The / construct is an expression whose value is given by the content of the clause.

Environment

The reader monad is a thin encapsulation of a function from the environment to the return type:

Here's the instance:

The implementation of bind for the reader monad creates a function that takes the environment as its argument. This environment is used twice, first to run to get the value of , and then to evaluate the value produced by .

The implementation of ignores the environment.

Exercise 14.4.1. *Define the and the instance for the following data type:*

Hint: You may use this handy function:

State

Like reader, the state monad is a function type:

Its bind is similar, except that the result of acting on has to be run with the modified state .

Applying bind to identity gives us the definition of :

Notice that we are essentially passing the result of the first to the second , except that we have to uncurry the second one so it can accept a pair:

In this form, it's easy to convert it to point-free notation:

There are two basic Kleisli arrows (the first one, conceptually, coming from the terminal object) with which we can construct an arbitrary stateful computation. The first one retrieves the current state:

and the second one modifies it:

A lot of monads come with their own libraries of predefined basic Kleisli arrows.

Nondeterminism

For the list monad, let's consider how we would implement . It must turn a list of lists into a single list. This can be done by concatenating all the inner lists using the library function . From there, we can derive the implementation of bind.

Here, constructs a singleton list. Thus a trivial version of nondeterminism is determinism.

What imperative languages do using nested loops we can do in Haskell using the list monad. Think of in bind as aggregating the result of running the inner loop and as the code that runs in the outer loop.

In many ways, Haskell's list behaves more like what is called an *iterator* or a *generator* in imperative languages. Because of laziness, the elements of the list are rarely stored in memory all at once, so you may conceptualize a Haskell list as a pointer to the head and a recipe for advancing it forward towards the tail. Or you may think of a list as a coroutine that produces, on demand, elements of a sequence.

Continuation

The implementation of bind for the continuation monad:

requires some backward thinking, because of the inherent inversion of control—the "don't call us, we'll call you" principle.

The result of bind is of the type . To construct it, we need a function that takes, as an argument :

We have two ingredients at our disposal:

We'd like to run , and for that we need a continuation that would accept an .

Once we have an , we can execute our . The result is of the type , so we can run it with our continuation .

Taken together, this convoluted process produces the following implementation:

As I mentioned earlier, composing continuations is not for the faint of heart. However, it has to be implemented only once—in the definition of the continuation monad. From there on, the notation will make the rest relatively easy.

Input/Output

The monad's implementation is baked into the language. The basic I/O primitives are available through the library. They are either in the form of Kleisli arrows, or objects (conceptually, Kleisli arrows from the terminal object).

For instance, the following object contains a command to read a line from the standard input:

There is no way to extract the string from it, since it's not there yet; but the program can process it through a further series of Kleisli arrows.

The monad is the ultimate procrastinator: the composition of its Kleisli arrows piles up task after task to be executed later by the Haskell runtime.

To output a string followed by a newline, you can use this Kleisli arrow:

Combining the two, you may construct a simple object:

which echoes a string you type.

14.5 Do Notation

It's worth repeating that the sole purpose of monads in programming is to let us decompose one big Kleisli arrow into multiple smaller ones.

This can be either done directly, in a point-free style, using Kleisli composition ; or by naming intermediate values and binding them to Kleisli arrows using .

Some Kleisli arrows are defined in libraries, others are reusable enough to warrant out-of-line implementation but, in practice, the majority are implemented as single-shot inline lambdas.

Here's a simple example:

which uses an ad-hoc Kleisli arrow of the type defined by the lambda expression:

The body of this lambda is further decomposed using another ad-hoc Kleisli arrow:

Such constructs are so common that there is special syntax called the notation that cuts through a lot of boilerplate. The above code, for instance, can be written as:

The compiler will automatically convert it to a series of nested lambdas. The line is usually read as: " *gets* the result of ."

Here's another example: a function that uses the list monad to generate all possible pairs of elements taken from two lists.

Notice that the last line in a block must produce a monadic value—here this is accomplished using .

Most imperative languages lack the abstraction power to generically define a monad, and instead they attempt to hard-code some of the more common monads. For instance, they implement exceptions as an alternative to the monad, or concurrent tasks as an alternative to the continuation monad. Some, like C++, introduce coroutines that mimic Haskell's notation.

Exercise 14.5.1. *Implement the following function that works for any monad:*

Hint: Use notation to extract the function and the argument. Use to return the result.

Exercise 14.5.2. *Rewrite the function using the bind operators and lambdas.*

14.6 Continuation Passing Style

I mentioned before that the notation provides the syntactic sugar that makes working with continuations more natural. One of the most important applications of continuations is in transforming programs to use CPS (continuation passing style). The CPS transformation is common in compiler construction. Another very important application of CPS is in converting recursion to iteration.

The common problem with deeply recursive programs is that they may blow the runtime stack. A function call usually starts by pushing function arguments, local variables, and the return address on the stack. Thus deeply nested recursive calls may quickly exhaust the (usually fixed-size) runtime stack resulting in a runtime error. This is the main reason why imperative languages prefer looping to recursion, and why most programmers learn about loops before they study recursion. However, even in imperative languages, when it comes to traversing recursive data structures, such as linked lists or trees, recursive algorithms are more natural.

The problem with using loops, though, is that they require mutation. There is usually some kind of a counter or a pointer that is advanced and checked with each turn of the loop. This is why purely functional languages that shun mutation must use recursion in place of loops. But since looping is more efficient and it doesn't consume the runtime stack, the compiler tries to covert recursive calls to loops. In Haskell all tail-recursive functions are turned into loops.

Tail recursion and CPS

Tail recursion means that the recursive call happens at the very end of the function. The function doesn't perform any additional operations on the result of the tail call. For instance this program is not tail recursive, because it has to add to the result of the recursive call:

In contrast, the following implementation is tail recursive because the result of the recursive call to is returned without further modification:

The compiler can easily turn the latter into a loop. Instead of making the recursive call, it will overwrite the value of the first argument with , overwrite the pointer to the head of the list with the pointer to its tail, and then jump to the beginning of the function.

Note however that it doesn't mean that the Haskell compiler won't be able to cleverly optimize the first implementation. It just means that the second implementation, which is tail recursive, is *guaranteed* to be turned into a loop.

In fact, it's always possible to turn recursion into tail recursion by performing the CPS transformation. This is because a continuation encapsulates *the rest of the computation*, so it's always the last call in a function.

To see how it works in practice, consider a simple tree traversal. Let's define a tree that stores strings in both nodes and leaves:

To concatenate these strings we use the traversal that first recurses into the left subtree, and then into the right subtree:

This is definitely not a tail recursive function, and it's not obvious how to turn it into one. However, we can almost mechanically rewrite it using the continuation monad:

We can run the result with the trivial continuation :

This implementation is automatically tail recursive. We can see it clearly by desugaring the do notation:

Let's analyze this code. The function calls itself, passing the left subtree and the following continuation:

This lambda in turn calls with the right subtree and another continuation:

This innermost lambda that has access to all three strings: left, middle, and right. It concatenates them and calls the outermost continuation with the result.

In each case, the recursive call to is the last call, and its result is immediately returned. Moreover, the type of the result is the generic type , which in itself guarantees that we can't perform any operations on it. When we finally run the result of , we pass it the identity (instantiated for the type):

Using named functions

But suppose that our programming language doesn't support anonymous functions. Is it possible to replace the lambdas with named functions? We've done this before when we discussed the adjoint functor theorem. We notice that the lambdas generated by the continuation monad are closures—they capture some values from their environment. If we want to replace them with named functions, we'll have to pass the environment explicitly.

We replace the first lambda with the call to the function named , and pass it the necessary environment in the form or a tuple of three values :

The three values are the string from the current node of the tree, the right subtree, and the outer continuation.

The function makes the recursive call to passing to it the right subtree and a continuation named :

Again, explicitly captures the environment containing two strings and the outer continuation. It performs the concatenation and calls the outer continuation with the result:

Finally, we define the trivial continuation:

that we use to extract the final result:

Defunctionalization

Continuation passing style requires the use of higher order functions. If this is a problem, e.g., when implementing distributed systems, we can always use the adjoint functor theorem to defunctionalize our program.

The first step is to create the sum of all relevant environments, including the empty one we used in :

Notice that this data structure can be reinterpreted as a list or a stack. It can be seen as a list of elements of the following sum type:

This list is our version of the runtime stack necessary to implement a recursive algorithm.

Since we are only interested in producing a string as the final result, we're going to approximate the function type. This is the approximate counit of the adjunction that defines it (see the adjoint functor theorem):

The function can be now implemented without recourse to higher order functions:

To extract the result, we call it with :

14.7 Monads Categorically

In category theory monads first arose in the study of algebras. In particular, the bind operator can be used to implement the very important operation of substitution.

Substitution

Consider this simple expression type. It's parameterized by the type that we can use for naming our variables:

We can, for instance, construct an expression $(2 + a) + b$:

We can implement the instance for :

Now suppose that you want to make a substitution by replacing the variable a with $x_1 + 2$ and b with x_2 (for simplicity, let's not worry about other letters of the alphabet). This substitution is represented by the Kleisli arrow :

As you can see, we were even able to change the type used for naming variables from to .

When we bind this Kleisli arrow to :

we get, as expected, a tree corresponding to $(2 + (x_1 + 2)) + x_2$.

Monad as a monoid

Let's analyze the definition of a monad that uses :

We have an endofunctor and two polymorphic functions.

In category theory, the functor that defines the monad is traditionally denoted by T (probably because monads were initially called "triples"). The two polymorphic functions become natural transformations. The first one, corresponding to , maps the "square" of T (a composition of T with itself) to T:

$$\mu\colon T \circ T \to T$$

(Of course, only *endo*-functors can be squared this way.)

The second, corresponding to , maps the identity functor to T:

$$\eta\colon \text{Id} \to T$$

Compare this with our earlier definition of a monoid in a monoidal category:

$$\mu \colon m \otimes m \to m$$
$$\eta \colon I \to m$$

The similarity is striking. This is why we often call the natural transformation μ the *monadic multiplication*. But in what category can the composition of functors be considered a tensor product?

Enter the category of endofunctors. Objects in this category are endofunctors and arrows are natural transformations.

But there's more structure to that category. We know that any two endofunctors can be composed. How can we interpret this composition if we want to treat endofunctors as objects? An operation that takes two objects and produces a third object looks like a tensor product. The only condition imposed on a tensor product is that it's functorial in both arguments. That is, given a pair of arrows:

$$\alpha \colon T \to T'$$
$$\beta \colon S \to S'$$

we can lift it to the mapping of the tensor product:

$$\alpha \otimes \beta \colon T \otimes S \to T' \otimes S'$$

In the category of endofunctors, the arrows are natural transformations so, if we replace $\otimes$ with $\circ$, the lifting is the mapping:

$$\alpha \circ \beta \colon T \circ T' \to S \circ S'$$

But this is just horizontal composition of natural transformations (now you understand why it's denoted by a circle).

The unit object in this monoidal category is the identity endofunctor, and unit laws are satisfied "on the nose," meaning

$$\mathrm{Id} \circ T = T = T \circ \mathrm{Id}$$

We don't need any unitors. We don't need any associators either, since functor composition is automatically associative.

A monoidal category in which unitors and associators are identity morphisms is called a *strict* monoidal category.

Notice, however, that composition is not symmetric, so this is not a symmetric monoidal category.

So, all said, a monad is a monoid in the monoidal category of endofunctors.

A monad (T, η, μ) consists of an object in the category of endofunctors—meaning an endofunctor T; and two arrows—meaning natural transformations:

$$\eta\colon \text{Id} \to T$$
$$\mu\colon T \circ T \to T$$

For this to be a monoid, these arrows must satisfy monoidal laws. Here are the unit laws (with unitors replaced by strict equalities):

$$\begin{tikzcd}
\text{Id}\circ T \arrow[r, "\eta\circ T"] \arrow[dr, "="'] & T\circ T \arrow[d, "\mu"] & T\circ\text{Id} \arrow[l, "T\circ\eta"'] \arrow[dl, "="] \\
& T &
\end{tikzcd}$$

and this is the associativity law:

$$\begin{tikzcd}
(T\circ T)\circ T \arrow[rr, "="] \arrow[d, "\mu\circ T"] & & T\circ(T\circ T) \arrow[d, "T\circ\mu"] \\
T\circ T \arrow[dr, "\mu"] & & T\circ T \arrow[dl, "\mu"'] \\
& T &
\end{tikzcd}$$

We used the whiskering notation for horizontal composition of $\mu\circ T$ and $T\circ\mu$.

These are the monad laws in terms of μ and η. They can be directly translated to the laws for and . They are also equivalent to the laws of the Kleisli category built from arrows $a \to Tb$.

14.8 Free Monads

A monad lets us specify a sequence of actions that may produce side effects. Such a sequence tells the computer both what to do and how to do it. But sometimes more flexibility is required: We'd like to separate the "what" from the "how." A free monad lets us produce the sequence without committing to a particular monad for its execution. This is analogous to defining a free monoid (a list), which lets us postpone the choice of the algebra to apply to it; or to creating an AST (abstract syntax tree) before compiling it to executable code.

Free constructions are defined as left adjoints to forgetful functors. So first we have to define what it means to forget to be a monad. Since a monad is an endofunctor equipped with additional structure, we'd like to forget this structure. We take a monad (T, η, μ) and keep only T. But in order to define such a mapping as a functor, we first need to define the category of monads.

Category of monads

The objects in the category of monads $\mathrm{Mon}(\mathcal{C})$ are monads (T, η, μ). We can define an arrow between two monads (T, η, μ) and $(T^{\cdot\cdot}, \eta^{\cdot\cdot}, \mu^{\cdot\cdot})$ as a natural transformation between the two endofunctors:

$$\lambda \colon T \to T^{\cdot\cdot}$$

However, since monads are endofunctors *with structure*, we want these natural transformations to preserve the structure. Preservation of unit means that the following diagram must commute:

$$\begin{tikzcd}
& \mathrm{Id} \arrow[dl, "\eta"'] \arrow[dr, "\eta^{\cdot\cdot}"] & \\
T \arrow[rr, "\lambda"] & & T^{\cdot\cdot}
\end{tikzcd}$$

Preservation of multiplication means that the following diagram must commute:

$$\begin{tikzcd}
T \circ T \arrow[r, "\lambda \circ \lambda"] \arrow[d, "\mu"] & T^{\cdot\cdot} \circ T^{\cdot\cdot} \arrow[d, "\mu^{\cdot\cdot}"] \\
T \arrow[r, "\lambda"] & T^{\cdot\cdot}
\end{tikzcd}$$

Another way of looking at $\mathrm{Mon}(\mathcal{C})$ is that it's a category of monoids in the monoidal category $([\mathcal{C}, \mathcal{C}], \circ, \mathrm{Id})$.

Free monad

Now that we have a category of monads, we can define the forgetful functor:

$$U \colon \mathrm{Mon}(\mathcal{C}) \to [\mathcal{C}, \mathcal{C}]$$

that maps every triple (T, η, μ) to T and every monad morphism to the underlying natural transformation.

We would like a free monad to be generated by a left adjoint to this forgetful functor. The problem is that this left adjoint doesn't always exist. As usual, this is related to size issues: monads tend to blow things up. The bottom line is that free monads exist for some, but not all, endofunctors. Therefore we can't define a free monad through an adjunction. Fortunately, in most cases of interest, a free monad can be defined as a fixed point of an algebra.

The construction is analogous to how we defined a free monoid as an initial algebra for the list functor:

or the more general:

$$F_a x = I + a \otimes x$$

This time, however, the monoidal category in which a monad is defined as a monoid is the category of endofunctors $([\mathcal{C}, \mathcal{C}], \circ, \mathrm{Id})$. A free monoid in this category is the initial algebra for the higher order "list" functor that maps functors to functors:

$$\Phi_F G = \mathrm{Id} + F \circ G$$

Here, the coproduct of two functors is defined point-wise. On objects:

$$(F + G)a = Fa + Ga$$

and on arrows:

$$(F + G)f = Ff + Gf$$

(We form a coproduct of two morphisms using the functoriality of the coproduct. We assume that $\mathcal{C}$ is co-cartesian, that is all coproducts exist.)

The initial algebra is the (least) fixed point of this operator, or the solution to the recursive equation:

$$L_F \text{ ö } \mathrm{Id} + F \circ L_F$$

This formula establishes a natural isomorphism between two functors. Going from right to left, $\mathrm{Id} + F \circ L_F \to L_F$, we have a mapping out of the sum, which is equivalent to a pair of natural transformations:

$$\mathrm{Id} \to L_F$$
$$F \circ L_F \to L_F$$

When translating this to Haskell, the components of these transformations become two constructors. We define the following recursive data type parameterized by a functor :

If we think of the functor as a container of values, the constructor takes a functorful of and stashes it away. An arbitrary value of the type is therefore a tree in which every node is a functorful of branches, and each leaf contains a value of the type .

Because this definition is recursive, the instance for it is also recursive:

Here, the outer uses the instance of , while the inner recurses into the branches.

It's easy to show that is a . The monadic unit is just a thin encapsulation of the identity functor:

Monadic multiplication, or , is defined recursively:

The instance for is therefore:

We can also define bind directly:

A free monad accumulates monadic actions in a tree-like structure without committing to any particular evaluation strategy. This tree can be "interpreted" using an algebra. But this time it's an algebra in the category of endofunctors, so its carrier is an endofunctor G and the structure map α is a natural transformation $\Phi_F G \to G$:

$$\alpha \colon \text{Id} + F \circ G \to G$$

This natural transformation, being a mapping out of a sum, is equivalent to a pair of natural transformations :

$$\lambda \colon \text{Id} \to G$$
$$\rho \colon F \circ G \to G$$

We can translate it to Haskell as a pair of polymorphic functions:

Since the free monad is the initial algebra, there is a unique mapping—the catamorphism—from it to any other algebra. Recall how we defined a catamorphism for a regular algebra:

The part unwraps the contents of the fixed point. Here we can do this by pattern-matching on the two constructors of the free monad. If it's a leaf, we apply our λ to it. If it's a node, we recursively process its contents, and apply our ρ to the result:

Many tree-like monads are in fact free monads for simple functors.

Exercise 14.8.1. *A (non-empty) rose tree is defined as:*

Implement conversions back and forth between and .

Exercise 14.8.2. *Implement conversions between a binary tree and , where:*

Exercise 14.8.3. *Find a functor whose free monad is equivalent to the list monad .*

Stack calculator example

As an example, let's consider a stack calculator implemented as an embedded domain-specific language, EDSL. We'll use the free monad to accumulate simple commands written in this language.

The commands are defined by the functor . Think of the parameter as the continuation.

For instance, is supposed to push an integer on the stack and then call the continuation .

The free monad for this functor can be thought of as a tree, with most branches having just one child, thus forming lists. The exception is the node, which has many children, one per every value of .

Here's the free monad for this functor:

In order to create domain-specific programs we'll define a few helper functions. There is a generic one that lifts a functorful of values to a free monad:

We also need a series of "smart constructors," which are Kleisli arrows for our free monad:

Since a free monad is a monad, we can conveniently combine Kleisli arrows using the notation. For instance, here's a toy program that adds two numbers and returns their sum:

In order to execute this program, we need to define an algebra whose carrier is an endofunctor. Since we want to implement a stack-based calculator, we'll use a version of the state functor. Its state is a stack—a list of integers. The state functor is defined as a function type; here it's a function that takes a list and returns a new list coupled with the type parameter :

To run the action, we apply the function to the stack:

We define the algebra as a pair of polymorphic functions corresponding to the two constructors of the free monad, and :

The first function terminates the execution of the program and returns a value:

The second function pattern matches on the type of the command. Each command carries with it a continuation. This continuation has to be run with a (potentially modified) stack. Each command modifies the stack in a different way:

For instance, discards the top of the stack. takes an integer from top of the stack and uses it to pick the branch to be executed. It does it by applying the function to the integer. adds the two numbers at the top of the stack and pushes the result.

Notice that the algebra we have defined does not involve recursion. Separating recursion from the actions is one of the advantages of the free monad approach. The recursion is instead encoded once and for all in the catamorphism.

Here's the function that can be used to run our toy program:

Obviously, the use of partial functions and makes our interpreter fragile. A badly formed program will cause a runtime error. A more robust implementation would use an algebra that allows for error propagation.

The other advantage of using free monads is that the same program may be interpreted using different algebras.

Exercise 14.8.4. *Implement a "pretty printer" that displays the program constructed using our free monad. Hint: Implement the algebra that uses the functor as the carrier:*

14.9 Monoidal Functors

We've seen several examples of monoidal cateogries. Such categories are equipped with some kind of binary operation, e.g., a cartesian product, a sum, composition (in the category of endofunctors), etc. They also have a special object that serves as the unit with respect to that binary operation. Unit and associativity laws are satisfied either on the nose (in strict monoidal categories) or up to isomorphism.

Every time we have more than one instance of some structure, we may ask ourselves the question: is there a whole category of such things? In this case: do monoidal categories form their own category? For this to work we would have to define arrows between monoidal categories.

A *monoidal functor* F from a monoidal category $(\mathcal{C}, \otimes, i)$ to another monoidal category $(\mathcal{D}, \oplus, j)$ maps tensor product to tensor product and unit to unit—all up to isomorphism:

$$Fa \oplus Fb \text{ ö } F(a \otimes b)$$
$$j \text{ ö } Fi$$

Here, on the left-hand side we have the tensor product and the unit in the target category, and on the right their counterparts in the source category.

If the two monoidal categories in question are not strict, that is the unit and associativity laws are satisfied only up to isomorphism, there are additional coherency conditions that ensure that unitors are mapped to unitors and associators are mapped to associators.

The category of monoidal categories with monoidal functors as arrows is called MonCat. In fact it's a 2-category, since one can define structure-preserving natural transformations between monoidal functors.

Lax monoidal functors

One of the perks of monoidal categories is that they allow us to define monoids. You can easily convince yourself that monoidal functors map monoids to monoids. It turns out that you don't need the full power of monoidal functors to accomplish that. Let's consider what the minimal requirements are for a functor to map monoids to monoids.

Let's start with a monoid (m, μ, η) in the monoidal category $(\mathcal{C}, \otimes, i)$. Consider a functor F that maps m to Fm. We want Fm to be a monoid in the target monoidal category $(\mathcal{D}, \oplus, j)$. For that we need to find two mappings:

$$\eta^{\ddot{}} : j \to Fm$$

$$\mu^{\ddot{}} : Fm \oplus Fm \to Fm$$

satisfying monoidal laws.

Since m is a monoid, we do have at our disposal the liftings of the original mappings:

$$F\eta \colon Fi \to Fm$$

$$F\mu \colon F(m \otimes m) \to Fm$$

What we are missing, in order to implement $\eta^{\ddot{}}$ and $\mu^{\ddot{}}$, are two additional arrows:

$$j \to Fi$$

$$Fm \oplus Fm \to F(m \otimes m)$$

A monoidal functor would provide such arrows. However, for what we're trying it accomplish, we don't need these arrows to be invertible.

A *lax monoidal functor* is a functor equipped with a morphism ϕ_i and a natural transformation ϕ_{ab}:

$$\phi_i \colon j \to Fi$$

$$\phi_{ab} \colon Fa \oplus Fb \to F(a \otimes b)$$

satisfying the appropriate unitality and associativity conditions.

Such a functor maps a monoid (m, μ, η) to a monoid $(Fm, \mu^{\cdot\cdot}, \eta^{\cdot\cdot})$ with:

$$\eta^{\cdot\cdot} = F\eta \circ \phi_i$$
$$\mu^{\cdot\cdot} = F\mu \circ \phi_{ab}$$

The simplest example of a lax monoidal functor is an endofunctor that preserves the usual cartesian product. We can define it in Haskell as a typeclass:

Corresponding to ϕ_{ab} we have an infix operator which, according to Haskell conventions, is written in its curried form.

Exercise 14.9.1. *Implement the instance for the list functor.*

Functorial strength

There is another way a functor may interact with monoidal structure, one that hides in plain sight when we do programming. We take it for granted that functions have access to the environment. Such functions are called closures.

For instance, here's a function that captures a variable from the environment and pairs it with its argument:

This definition makes no sense in isolation, but it does when the environment contains the variable , e.g.,

The function returned by calling "closes over" the 5 from its environment.

Now consider the following modification, which returns a singleton list that contains the closure:

As a programmer you'd be very surprised if this didn't work. But what we do here is highly nontrivial: we are smuggling the environment *under* the list functor. According to our model of lambda calculus, a closure is a morphism from the product of the environment and the function argument. Here the lambda, which is really a function of , is defined inside a list functor, but it captures the value that is defined *outside* the list.

The property that lets us smuggle the environment under a functor is called *functorial strength* or *tensorial strength* and can be implemented in Haskell as:

The notation is called a *tuple section* and is equivalent to the partial application of the pair constructor: .

In category theory, strength for an endofunctor F is defined as a natural transformation that smuggles a tensor product into a functor:

$$\sigma \colon a \otimes F(b) \to F(a \otimes b)$$

There are some additional conditions which ensure that it works nicely with the unitors and the associator of the monoidal category in question.

The fact that we were able to implement for an arbitrary functor means that, in Haskell, every functor is strong. This is the reason why we don't have to worry about accessing the environment from inside a functor.

Even more importantly, every monad in Haskell is strong by virtue of being a functor. This is also why every monad is automatically .

If you desugar this code to use monadic bind and lambdas, you'll notice that the final needs access to both and , which are defined in outer environments. This would be impossible without a monad being strong.

In category theory, though, not every endofunctor in a monoidal category is strong. For now, the magic incantation is that the category we're working with is self-enriched, and every endofunctor defined in Haskell is enriched. We'll come back to it when we talk about enriched categories. In Haskell, strength boils down to the fact that we can always a partially applied pair constructor .

Applicative functors

In programming, the idea of applicative functors arose from the following question: A functor lets us lift a function of one variable. How can we lift a function of two or more variables?

By analogy with , we'd like to have a function:

A function of two arguments—here, in its curried form—is a function of one argument returning a function. So, assuming that is a functor, we can the first argument of , which has the type:

over the second argument to get:

The problem is, we don't know how to apply to the remaining argument .

The class of functors that let us do that is called . It turns out that, once we know how to lift a two-argument function, we can lift functions of any number of arguments, except zero. A zero-argument function is just a value, so lifting it means implementing a function:

Here's the Haskell definition:

The application of a functorful of functions to a functorful of arguments is defined as an infix operator that is customarily called "splat."

There is also an infix version of :

which can be used in this terse implementation of :

Both operators bind to the left, which makes this syntax mimic regular function application.

An applicative functor must also satisfy a set of laws:

Exercise 14.9.2. *Implement , a function that lifts a 3-argument function using an applicative functor.*

Closed functors

If you squint at the definition of the splat operator:

you may see it as mapping function objects to function objects.

This becomes clearer if you consider a functor between two categories, both of them closed. You may start with a function object b^a in the source category and apply the functor F to it:

$$F(b^a)$$

Alternatively, you may map the two objects a and b and construct a function object between them in the target category:

$$(Fb)^{Fa}$$

If we demand that the two ways be isomorphic, we get a definition of a strict *closed functor*. But, as was the case with monoidal functors, we are more interested in the lax version, which is equipped with a one-way natural transformation:

$$F(b^a) \to (Fb)^{Fa}$$

If F is an endofunctor, this translates directly into the definition of the splat operator.

The full definition of a lax closed functor includes the mapping of the monoidal unit and some coherence conditions. All said, an applicative functor is a lax closed functor.

In a closed cartesian category, the exponential is related to the cartesian product through the currying adjunction. It's no surprise then, that in such a category lax monoidal and lax closed (applicative) endofunctors are the same.

We can easily express this in Haskell:

where is a function that ignores its second argument:

and is the uncurried function application:

And the other way around we have:

In the latter, we used the pair constructor as a two-argument function.

Monads and applicatives

Since in a cartesian closed category every monad[1] is lax monoidal, it is automatically applicative. We can show it directly by implementing , which has the same type signature as the splat operator:

This connection is expressed in the Haskell definition of a by having as its superclass:

Notice the default implementation of as .

The converse is not true: not every is a . The standard counterexample is the instance for a list functor that uses zipping:

Of course, the list functor is also a monad, so there is another instance based on that. Its splat operator applies every function to every argument.

In programming, monad is more powerful than applicative. That's because monadic code lets you examine the contents of a monadic value and branch depending on it. This is true even for the monad which otherwise provides no means of extracting the value. In this example we are branching on the contents of an object:

Of course, the inspection of the value is postponed until the runtime interpreter of gets hold of this code.

Applicative composition using the splat operator doesn't allow for one part of the computation to inspect the result of the other. This a limitation that can be turned into an advantage.

[1] Again, the correct incantation is "every enriched monad"

The absence of dependencies makes it possible to run the computations in parallel. Haskell's parallel libraries use applicative programming extensively.

On the other hand, monads let us use the very convenient syntax, which is arguably more readable than the applicative syntax. Fortunately, there is a language extension , which instructs the compiler to selectively use applicative constructs in interpreting blocks, whenever there are no dependencies.

Exercise 14.9.3. *Verify laws for the zip instance of the list functor.*

Chapter 15

Monads and Adjunctions

15.1 String Diagrams

A line partitions a plane. We can think of it as either dividing a plane or as connecting two halves of the plane.

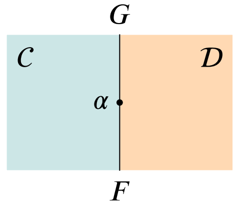
A dot partitions a line. We can think of it as either separating two half-lines or as joining them together.

This is a diagram in which two categories are represented as dots, two functors as arrows, and a natural transformation as a double arrow.

$$\begin{tikzcd} \mathcal{C} \arrow[r, bend left=50, "G"{name=G}] \arrow[r, bend right=50, "F"'{name=F}] & \mathcal{D} \arrow[Rightarrow, from=F, to=G, "\alpha"] \end{tikzcd}$$

But the same idea can be represented by drawing categories as areas of a plane, functors as lines between areas, and natural transformations as dots that join line segments.

The idea is that a functor always goes between a pair of categories, therefore it can be drawn as a boundary between them. A natural transformation always goes between a pair of functors, therefore it can be drawn as a dot joining two segments of a line.



This is an example of a *string diagram*. You read such a digram bottom-up, left-to-right (think of the (x, y) system of coordinates).

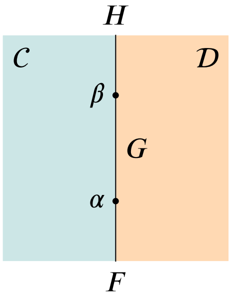
The bottom of this diagram shows the functor F that goes from $\mathcal{C}$ to $\mathcal{D}$. The top of the diagram shows the functor G that goes between the same two categories. The transition happens in the middle, where a natural transformation α maps F to G.

In Haskell, this diagram is interpreted as a polymorphic function between two endofunctors:

So far it doesn't seem like we gain a lot by using this new visual representation. But let's apply it to something more interesting: vertical composition of natural transformations:

$$\begin{tikzcd}[column sep=huge]
\mathcal{C} \arrow[r, bend left=60, "H", ""{name=H, below}] \arrow[r, "G" near end, ""{name=G}] \arrow[r, bend right=60, "F"', ""{name=F, above}] & \mathcal{D}
\arrow[from=F, to=G, Rightarrow, "\alpha"] \arrow[from=G, to=H, Rightarrow, "\beta"]
\end{tikzcd}$$

The corresponding string diagram shows the two categories and three functors between them joined by two natural transformations.



As you can see, you can reconstruct the original diagram from the string diagram by scanning it bottom-to-top.

Again, in Haskell we'll be dealing with three endofunctors, and the vertical composition of after :

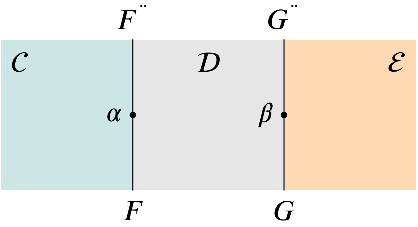
is implemented using regular function composition:

Let's continue with the horizontal composition of natural transformations:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left, "F^{\cdot\cdot}"] \arrow[r, bend right, "F"'] \arrow[r, phantom, "\alpha \Uparrow"] & \mathcal{D} \arrow[r, bend left, "G^{\cdot\cdot}"] \arrow[r, bend right, "G"'] \arrow[r, phantom, "\beta \Uparrow"] & \mathcal{E}
\end{tikzcd}$$

This time we have three categories, so we'll have three areas.

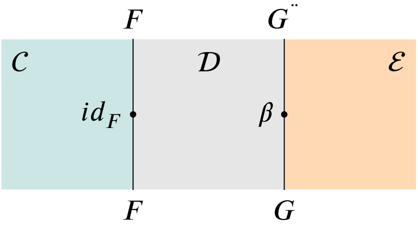
The bottom of the string diagram corresponds to the composition of functors $G \circ F$ (in this order). The top corresponds to $G^{\cdot\cdot} \circ F^{\cdot\cdot}$. One natural transformation, α, connects F to $F^{\cdot\cdot}$; the other, β, connects G to $G^{\cdot\cdot}$.



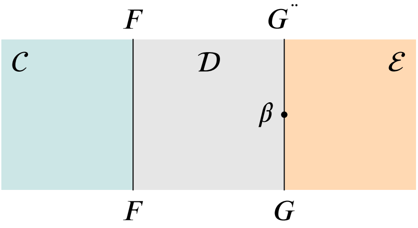
Parallel vertical lines in this new system correspond to functor composition.

You may think of the horizontal composition of natural transformations as happening along the imaginary horizontal line in the middle of the diagram. But what if somebody was sloppy in drawing the diagram, and one of the dots was a little higher than the other? As it turns out, the exact positioning of the dots doesn't matter, due to the interchange law.

But first, let's illustrate whiskering: horizontal composition in which one of the natural transformation is the identity. We can draw it like this:



But, really, the identity can be inserted at any point on a vertical line, so we don't even have to draw it. The following diagram represents the whiskering of $\beta \circ F$.



In Haskell, where is a polymorphic function:

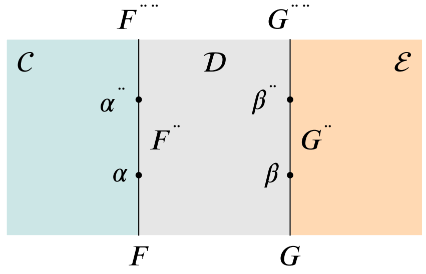
we read this diagram as:

with the understanding that the type checker instantiates the polymorphic function for the correct type.

Similarly, you can easily imagine the diagram for $G\circ\alpha$, and its Haskell realization:

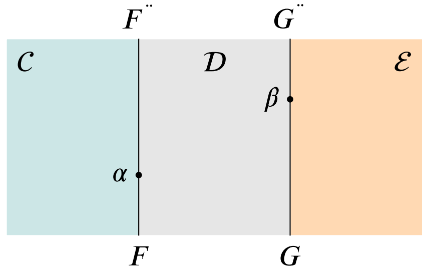
with:

Here's the string diagram that corresponds to the interchange law:



This diagram is purposefully ambiguous. Are we supposed to first do vertical composition of natural transformations and then the horizontal one? Or should we compose $\beta\circ\alpha$ and $\beta'\circ\alpha'$ horizontally, and then compose the results vertically? The interchange law says that it doesn't matter: the result is the same.

Now try to replace a pair of natural transformations in this diagram with identities. If you replace α' and β', you get the horizontal composition of $\beta\circ\alpha$. If you replace α' and β with identity natural transformations, and rename β' to β, you get the diagram in which α is shifted down with respect to β, and so on.



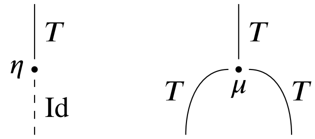
The interchange law tells us that all these diagrams are equal. We are free to slide natural transformations like beads on a string.

String diagrams for the monad

A monad is defined as an endofunctor equipped with two natural transformations, as illustrated by the following diagrams:

$$\begin{tikzcd} C \arrow[r, bend left=50, "T"] \arrow[r, bend right=50, "\text{Id}"'] \arrow[r, phantom, "\eta \Uparrow"] & C \end{tikzcd} \qquad \begin{tikzcd} C \arrow[r, bend left=50, "T"] \arrow[r, bend right=50, "T\circ T"'] \arrow[r, phantom, "\mu \Uparrow"] & C \end{tikzcd}$$

Since we are dealing with just one category, when translating these diagrams to string diagrams, we can dispose of the naming (and the shading) of categories, and just draw the strings alone.



In the first diagram it's customary to skip the dashed line corresponding to the identity functor. The η dot can be used to freely inject a T line into a diagram. Two T lines can be joined by the μ dot.

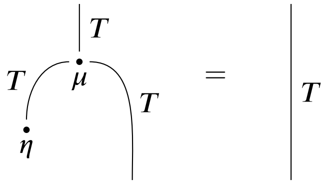
String diagrams are especially useful in expressing monad laws. For instance, we have the left identity law:

$$\mu\circ(\eta\circ T) = id$$

which can be visualized as a commuting diagram:

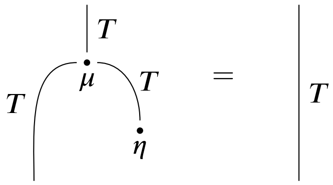
$$\begin{tikzcd} \text{Id}\circ T \arrow[r, "\eta\circ T"] \arrow[dr, "id"'] & T\circ T \arrow[d, "\mu"] \\ & T \end{tikzcd}$$

The corresponding string diagrams represents the equality of the two paths through this diagram:

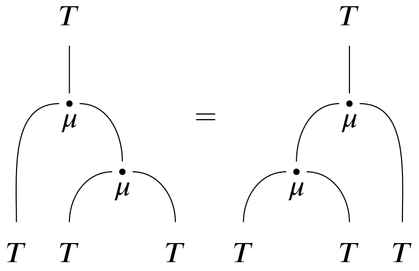


You may think of this equality as the result of yanking the top and bottom strings resulting in the η appendage being retracted into the straight line.

There is a symmetric right identity law:



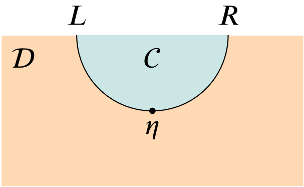
Finally, this is the associativity law in terms of string diagrams:



String diagrams for the adjunction

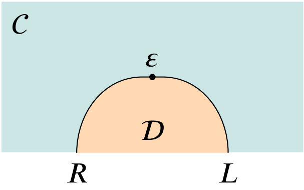
As we discussed before, an adjunction is a relation between a pair of functors, $L\colon \mathcal{D} \to \mathcal{C}$ and $R\colon \mathcal{C} \to \mathcal{D}$. It can be defined by a pair of natural transformations, the unit η and the counit ε, satisfying triangular identities.

The unit of the adjunction can be illustrated by a "cup"-shaped diagram:



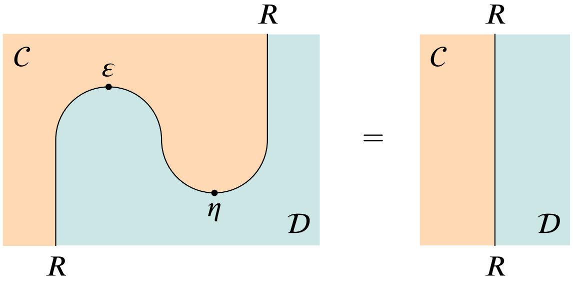
The identity functor at the bottom of the diagram is omitted from the picture. The η dot turns the identity functor below it to the composition $R \circ L$ above it.

Similarly, the counit can be visualized as a "cap"-shaped string diagram with the implicit identity functor at the top:



Triangle identities can be easily expressed using string diagrams. They also make intuitive sense, as you can imagine pulling on the string from both sides to straighten the curve.

For instance, this is the first triangle identity, sometimes called the *zigzag* identity:



Reading the left diagram bottom-to-top produces a series of mappings:

$$Id_{\mathcal{D}}\circ R \xrightarrow{\eta\circ R} R\circ L\circ R \xrightarrow{R\circ\varepsilon} R\circ Id_{\mathcal{C}}$$

This must be equal to the right-hand-side, which may be interpreted as the (invisible) identity natural transformation on R

In the case R is an endofunctor, we can translate the first diagram directly to Haskell. The whiskering of the unit of the adjunction η by R results in the polymorphic function being instantiated at . The whiskering of ε results in the lifting of by the functor R. The vertical composition translates to function composition:

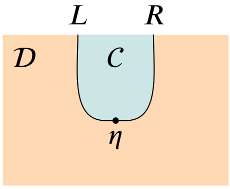
Exercise 15.1.1. *Draw the string diagrams for the second triangle identity and translate them to Haskell.*

15.2 Monads from Adjunctions

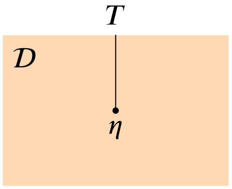
You might have noticed that the same symbol η is used for the unit of the adjunction and for the unit of the monad. This is *not* a coincidence.

At first sight it might seem like we are comparing apples to oranges: an adjunction is defined with two functors between two categories and a monad is defined by one endofunctor operating on a single category. However, the composition of two functors going in opposite directions is an endofunctor, and the unit of the adjunction maps the identity endofunctor to the endofunctor $R\circ L$.

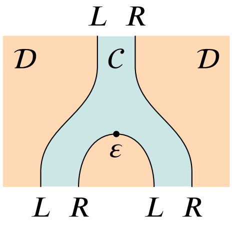
Compare this diagram:



with the one defining the monadic unit:



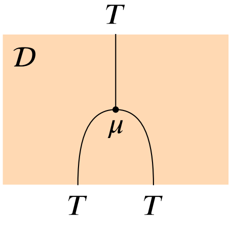
It turns out that, for any adjunction $L \dashv R$, the endofunctor $T = R\circ L$ is a monad, with the multiplication μ defined by the following diagram:



Reading this diagram bottom-to-top, we get the following transformation (imagine slicing it horizontally at the dot):

$$R\circ L\circ R\circ L \xrightarrow{R\circ\varepsilon\circ L} R\circ L$$

Compare this with the definition of the monadic μ:



We get the definition of μ for the monad $R\circ L$ as the double-whiskering of ε:

$$\mu = R\circ\varepsilon\circ L$$

The Haskell translation of the string diagram defining μ in terms of ε is always possible. The monadic multiplication, or , becomes:

where corresponds to the lifting by the endofunctor defined as the composition $R\circ L$. Notice that $\mathcal{D}$ in this case is the Haskell category of types and functions, but $\mathcal{C}$ can be an outside category.

To complete the picture, we can use string diagrams to derive monadic laws using triangle identities. The trick is to replace all strings in monadic laws by pairs of parallel strings and then rearrange them according to the rules.

To summarize, every adjunction $L \dashv R$ with the unit η and counit ε defines a monad $(R \circ L, \eta, R \circ \varepsilon \circ L)$.

We'll see later that, dually, the other composition, $L \circ R$ defines a comonad.

Exercise 15.2.1. *Draw string diagrams to illustrate monadic laws (unit and associativity) for the monad derived from an adjunction.*

15.3 Examples of Monads from Adjunctions

We'll go through several examples of adjunctions that generate some of the monads that we use in programming. We'll expand on these examples later, when we talk about monad transformers.

Most examples involve functors that leave the category of Haskell types and functions, even though the round trip that generates the monad ends up being an endofunctor. This is why it's often impossible to express such adjunctions in Haskell.

To additionally complicate things, there is a lot of bookkeeping related to explicit naming of data constructors, which is necessary for type inference to work. This may sometimes obscure the simplicity on the underlying formulas.

Free monoid and the list monad

The list monad is generated by the free monoid adjunction we've seen before. The unit of this adjunction, $\eta_X : X \to U(FX)$, injects the elements of the set X as the generators of the free monoid FX, after which U extracts the underlying set.

In Haskell, we represent the free monoid as a list type, and its generators are singleton lists. The unit η_X maps elements of X to such singletons:

To implement the counit, $\varepsilon_M : F(UM) \to M$, we take a monoid M, forget its multiplication, and use its set of elements as generators for a new free monoid. A component of the counit at M is then a monoid morphism from the free monoid back to M or, in Haskell, . It turns out that this monoid morphism is a special case of a catamorphism.

First, recall the Haskell implementation of a general list catamorphism:

Here, we interpret as a regular function from to the underlying set of a monoid . The result is interpreted as a *monoid morphism* from the free monoid generated by (that is a list of 's) to . This is just one direction of the adjunction:

$$\mathrm{Set}(a, Um) \text{ ö } \mathrm{Mon}(Fa, m)$$

To get the counit as a monoid morphism we apply to identity. The result is or, in terms of :

It is a monoid morphism since it maps an empty list to the monoidal unit, and concatenation to monoidal product.

Monadic multiplication, or , is given by the whiskering of the counit:

$$\mu = U \circ \varepsilon \circ F$$

You can easily convince yourself that whiskering on the left doesn't do much here, since it's just a lifting of a monoid morphism by the forgetful functor (it keeps the function while forgetting its special property of preserving structure).

The right whiskering by F is more interesting. It means that the component μ_X corresponds to the component of ε at FX, which is the free monoid generated from the set X. This free monoid is defined by:

which gives us the definition of :

As expected, this is the same as : In the list monad, multiplication is concatenation.

The currying adjunction and the state monad

The state monad is generated by the currying adjunction that we used to define the exponential object. The left functor is defined by a product with some fixed object s:

$$L_s a = a \quad s$$

We can, for instance, implement it as a Haskell type:

The right functor is the exponentiation, parameterized by the same object s:

$$R_s c = c^s$$

In Haskell, it's a thinly encapsulated function type:

The monad is given by the composition of these two functors. On objects:

$$(R_s \circ L_s) a = (a \quad s)^s$$

In Haskell we would write it as:

If you expand this definition, it's easy to recognize in it the functor:

The unit of the adjunction $L_s \dashv R_s$ is a mapping:

$$\eta_a \colon a \to (a \quad s)^s$$

which can be implemented in Haskell as:

You may recognized in it a thinly veiled version of for the state monad:

Here's the component of the counit of this adjunction at c:

$$\varepsilon_c \colon c^s \quad s \to c$$

It can be implemented in Haskell as:

which, after stripping data constructors, is equivalent to , or the uncurried version of .

Monad multiplication μ is given by the whiskering of ε from both sides:

$$\mu = R_s \circ \varepsilon \circ L_s$$

Here it is translated to Haskell:

Whiskering on the right doesn't do anything other than select a component of the natural transformation. This is done automatically by Haskell's type inference engine. Whiskering on the left is done by lifting the component of the natural transformation. Again, type inference picks the correct implementation of —here, it's equivalent to precomposition.

Compare this with the implementation of :

Notice the dual use of :

When it's uncurried, its type signature becomes:

which is equivalent to that of .

When partially applied, just strips the data constructor exposing the underlying function type:

M-sets and the writer monad

The writer monad:

is parameterized by a monoid . This monoid is used for accumulating log entries. The adjunction we are going to use involves a category of M-sets for that monoid.

An M-set is a set S on which we define the action of a monoid M. Such an action is a mapping:

$$a\colon M \quad S \to S$$

We often use the curried version of the action, with the monoid element in the subscript position. Thus a_m becomes a function $S \to S$.

This mapping has to satisfy some constraints. The action of the monoidal unit 1 must not change the set, so it has to be the identity function:

$$a_1 = id_S$$

and two consecutive actions must combine to an action of their monoidal product:

$$a_{m_1} \circ a_{m_2} = a_{m_1 \cdot m_2}$$

This choice of the order of multiplication defines what it called the *left action*. (The right action has the two monoidal elements swapped on the right-hand side.)

M-sets form a category MSet. The objects are pairs $(S, a\colon M \quad S \to S)$ and the arrows are *equivariant maps*, that is functions between sets that preserve actions.

A function $f\colon S \to R$ is an *equivariant* mapping from (S, a) to (R, b) if the following diagram commutes, for every m Ë M:

$$\begin{tikzcd}
S \arrow[r, "f"] \arrow[d, "a_m"] & R \arrow[d, "b_m"] \\
S \arrow[r, "f"] & R
\end{tikzcd}$$

In other words, it doesn't matter if we first do the action a_m, and then map the set; or first map the set, and then do the corresponding action b_m.

There is a forgetful functor U from MSet to Set, which assigns the set S to the pair (S, a), thus forgetting the action.

Corresponding to it there is a free functor F. Its action on a set S produces an M-set. It's a set that is a cartesian product of S and M, where M is treated as a set of elements (in other words, the result of the action of a forgetful functor on a monoid). An element of this M-set is a pair $(x \text{ Ë } S, m \text{ Ë } M)$ and the free action is defined by:

$$\phi_n \colon (x, m) \mapsto (x, n \cdot m)$$

leaving the element x unchanged, and only multiplying the m-component.

To show that F is left adjoint to U we have to construct the following natural isomorphism:

$$\mathrm{MSet}(FS, Q) \text{ ö } \mathrm{Set}(S, UQ)$$

for any set S and any M-set Q. If we represent Q as a pair (R, b), the element of the right hand side of the adjunction is a plain function $u \colon S \to R$. We can use this function to construct an equivariant mapping on the left.

The trick here is to notice that such an equivariant mapping $f \colon FS \to Q$ is fully determined by its action on the elements of the form $(x, 1) \text{ Ë } FS$, where 1 is the monoidal unit.

Indeed, from the equivariance condition it follows that:

$$\begin{tikzcd}
(x, 1) \arrow[r, mapsto, "f"] \arrow[d, mapsto, "\phi_m"] & r \arrow[d, mapsto, "b_m"] \\
(x, m \cdot 1) \arrow[r, mapsto, "f"] & r^{\text{¨}}
\end{tikzcd}$$

or:

$$f(\phi_m(x, 1)) = f(x, m) = b_m(f(x, 1))$$

Thus every function $u \colon S \to R$ uniquely defines an equivariant mapping $f \colon FS \to Q$ given by:

$$f(x, m) = b_m(ux)$$

The unit of this adjunction $\eta_S \colon S \to U(FS)$ maps an element x to a pair $(x, 1)$. Compare this with the definition of for the writer monad:

The counit is given by an equivariant map:

$$\varepsilon_Q \colon F(UQ) \to Q$$

The left hand side is the M-set constructed by taking the underlying set of Q and taking its product with the underlying set of M. The original action of Q is forgotten and replaced by the free action. The obvious choice for the counit is:

$$\varepsilon_Q \colon (x, m) \mapsto a_m x$$

where x is an element of (the underlying set of) Q and a is the action defined in Q.

Monad multiplication μ is given by the whiskering of the counit.

$$\mu = U \circ \varepsilon \circ F$$

It means replacing Q in the definition of ε_Q with a free M-set whose action is the free action. In other words, we replace x with (x, m) and a_n with ϕ_n. (Whiskering with U doesn't change anything.)

$$\mu_S \colon ((x, m), n) \mapsto \phi_n(x, m) = (x, n \cdot m)$$

Compare this with the definition of for the writer monad:

Pointed objects and the monad

Pointed objects are objects with a designated element. Since picking an element is done using an arrow from the terminal object, the category of pointed objects is defined using pairs $(a, p \colon 1 \to a)$, where a is an object in C.

The morphisms between these pairs are the arrows in C that preserve the points. Thus a morphism from $(a, p \colon 1 \to a)$ to $(b, q \colon 1 \to b)$ is an arrow $f : a \to b$ such that $q = f \circ p$. This category is also called a *coslice category* and is written as $1/C$.

There is an obvious forgetful functor $U \colon 1/C \to C$ that forgets the point. Its left adjoint is a free functor F that maps an object a to a pair $(1 + a, \text{Left})$. In other words, F freely ads a point to an object using a coporoduct.

The monad is similarly constructed by replacing 1 with a fixed object e.

Exercise 15.3.1. *Show that $U \circ F$ is the monad.*

The continuation monad

The continuation monad is defined in terms of a pair of contravariant functors in the category of sets. We don't have to modify the definition of the adjunction to work with contravariant functors. It's enough to select the opposite category for one of the endpoints.

We'll define the left functor as:

$$L_Z \colon \text{Set}^{op} \to \text{Set}$$

It maps a set X to the hom-set in Set:

$$L_Z X = \text{Set}(X, Z)$$

This functor is parameterized by another set Z. The right functor is defined by essentially the same formula:

$$R_Z \colon \text{Set} \to \text{Set}^{op}$$

$$R_Z X = \mathrm{Set}^{\mathrm{op}}(Z, X) = \mathrm{Set}(X, Z)$$

The composition $R \circ L$ can be written in Haskell as , which is the same as the (covariant) endofunctor that defines the continuation monad.

15.4 Monad Transformers

Suppose that you want to combine multiple effects, say, state with the possibility of failure. One option is to define your own monad from scratch. You define a functor:

together with the function to extract the result (or report failure):

You define the monad instance for it:

and, if you are diligent enough, check that it satisfies the monad laws.

There is no general recipe for combining monads. In that sense, monads are not composable. However, we know that adjunctions are composable. We've also seen how to get monads from adjunctions and, as we'll soon see, every monad can be obtained this way. So, if we can match adjunctions, the monads that they generate will automatically compose.

Consider two composable adjunctions:

$$\begin{tikzcd} \mathcal{C} \arrow[r, bend right, "R^{¨}"'] & \mathcal{D} \arrow[l, bend right, "L^{¨}"'] \arrow[r, bend right, "R"'] & \mathcal{E} \arrow[l, bend right, "L"'] \end{tikzcd}$$

There are three monads in this picture. There is the "inner" monad $R^{¨} \circ L^{¨}$ and the "outer" monad $R \circ L$ as well as the composite $R \circ R^{¨} \circ L^{¨} \circ L$.

If we call the inner monad $T = R^{¨} \circ L^{¨}$, then $R \circ T \circ L$ is the composite monad called the *monad transformer*, because it transforms the monad T into a new monad.

$$\begin{tikzcd} \mathcal{D} \arrow[loop above, "T = R^{¨} \circ L^{¨}"] \arrow[r, bend right, "R"'] & \mathcal{E} \arrow[l, bend right, "L"'] \end{tikzcd}$$

In our example, we can treat as the inner monad:

$$T a = 1 + a$$

It is transformed using the outer adjunction $L_s \dashv R_s$, the one that generates the state monad:

$$L_s a = a \quad s$$

$$R_s c = c^s$$

The result is:

$$(R_s \circ T \circ L_s) a = (1 + a \quad s)^s$$

or, in Haskell:

```
```

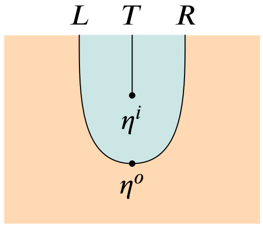
which matches the definition of our monad.

In general, the inner monad T is defined by its unit η^i and multiplication μ^i (the superscript i standing for "inner"). The "outer" adjunction is defined by its unit η^o and counit ε^o.

The unit of the composite monad is the natural transformation:

$$\eta \colon Id \to R \circ T \circ L$$

given by the string diagram:



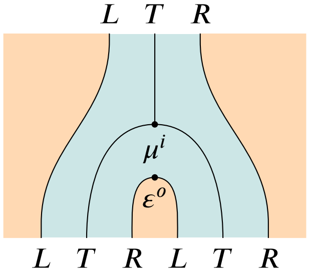
It is the vertical composition of the whiskered inner unit $R \circ \eta^i \circ L$ and the outer unit η^o. In components:

$$\eta_a = R(\eta^i_{La}) \circ \eta^o_a$$

The multiplication of the composite monad is a natural transformation:

$$\mu \colon R \circ T \circ L \circ R \circ T \circ L \to R \circ T \circ L$$

given by the string diagram:



It's the vertical composition of the multiply whiskered outer counit:

$$R \circ T \circ \varepsilon^o \circ T \circ L$$

followed by the whiskered inner multiplication $R \circ \mu^i \circ L$. In components:

$$\mu_c = R(\mu^i_{Lc}) \circ (R \circ T)(\varepsilon^o_{(T \circ L)c})$$

State monad transformer

Let's unpack these equations for the case of the state monad transformer. The state monad is generated by the currying adjunction. The left functor L_s is the product functor , and the right functor R_s is the exponential, a.k.a., the reader functor .

As we've seen before, the outer counit ε^o_a is function application:

and the unit η^o_a is the curried pair constructor:

We'll keep the inner monad (T, η^i, μ^i) arbitrary. In Haskell, we'll call this triple , , and .

The composite monad that we get by applying the state monad transformer to the monad T, is the composition $R \circ T \circ L$ or, in Haskell:

The unit of the monad transformer is the vertical composition of η^o and $R \circ \eta^i \circ L$. In components:

$$\eta_a = R(\eta^i_{La}) \circ \eta^o_a$$

There are a lot of moving parts in this formula, so let's analyze it step-by-step. We start from the right: we have the a-component of the unit of the adjunction, which is an arrow from a to $R(La)$. In Haskell, it's the function .

Let's evaluate this function at some . The result is another function . We pass this function as an argument to $R(\eta^i_{La})$.

η^i_{La} is the component of of the inner monad taken at La. Here, La is the type . So we are instantiating the polymorphic function as a function . (The type inferencer will do this automatically for us.)

Next, we are lifting this component of using R. Here, R is the exponential $(*)^s$, so it lifts a function by post-composition. It will post-compose to whatever function is passed to it. In our case, that's the function that was produced by . Notice that the types match: we are post-composing after .

We can write the result of this composition as:

or, inlining function composition:

We inserted the data constructor to make the type checker happy. This is the of the composite monad in terms of the of the inner monad.

The same reasoning can be applied to the formula for the component of the composite μ at some a:

$$\mu_a = R(\mu^i_{La}) \circ (R \circ T)(\varepsilon^o_{(T\circ L)a})$$

The inner μ^i is the of the monad . Applying R turns it into post-composition.

The outer ε^o is function application taken at $T(La)$ or . It's a function of the type:

which, inserting the appropriate data constructors, can be written as :

The application of $(R\circ T)$ lifts this component of ε using the composition of functors R and T. The former is implemented as post-composition, and the latter is the of the monad .

Putting all this together, we get a point-free formula for of the state monad transformer:

Here, the partially applied strips off the data constructor from the argument :

Our earlier example of can now be rewritten using a monad transformer:

The vanilla monad can be recovered by applying the monad transformer to the identity functor, which has a instance defined in the library (notice that the last type variable is skipped in this definition):

Other monad transformers follow the same pattern. They are defined in the Monad Transformer Library, .

15.5 Monad Algebras

Every adjunction generates a monad, and so far we've been able to define adjunctions for all the monads of interest for us. But is every monad generated by an adjunction? The answer is yes, and there are usually many adjunctions—in fact a whole category of adjunctions—for every monad.

Finding an adjunction for a monad is analogous to factorization. We want to express a functor as a composition of two other functors, $T = R\circ L$. The problem is complicated by the fact that this factorization also requires finding the appropriate intermediate category. We'll find such a category by studying algebras for a monad.

A monad is defined by an endofunctor, and we know that it's possible to define algebras for an endofunctor. Mathematicians often think of monads as tools for generating expressions and algebras as tools for evaluating those expressions. However, expressions generated by monads impose some compatibility conditions on those algebras.

For instance, you may notice that the monadic unit $\eta_a \colon a \to Ta$ has the type signature that looks like the inverse of the structure map of an algebra $\alpha \colon Ta \to a$. Of course, η is a natural transformation that is defined for every type, whereas an algebra has a fixed carrier type. Nevertheless, we might reasonably expect that one might undo the action of the other.

Consider the earlier example of the expression monad . An algebra for this monad is a choice of the carrier type, let's say and an arrow:

Since is a monad, it defines a unit, or , which is a polymorphic function that can be used to generate simple expressions from values. The unit of is:

We can instantiate the unit for an arbitrary type, in particular for the carrier type of our algebra. It makes sense to demand that *evaluating* , where is a character, should give us back the same . In other words, we'd like:

This condition will immediately eliminate a lot of algebras, such as:

The second condition we'd like to impose is that the algebra that's compatible with a monad respects substitution. A monad lets us flatten nested expressions using . An algebra lets us evaluate such expressions.

There are two ways of doing that: we can apply the algebra to a flattened expression, or we can apply it to the inner expression first (using), and then evaluate the resulting expression.

where is of the nested type .

In category theory these two conditions define a monad algebra.

We say that $(a, \alpha \colon Ta \to a)$ is a *monad algebra* for the monad (T, μ, η) if the following diagrams commute:

$$\begin{tikzcd}
a \arrow[r, "\eta_a"] \arrow[dr, "id_a"'] & Ta \arrow[d, "\alpha"] \\
& a
\end{tikzcd}
\qquad
\begin{tikzcd}
T(Ta) \arrow[r, "T\alpha"] \arrow[d, "\mu_a"] & Ta \arrow[d, "\alpha"] \\
Ta \arrow[r, "\alpha"] & a
\end{tikzcd}$$

These laws are sometimes called the unit law and the multiplication law for monad algebras.

Since monad algebras are just special kinds of algebras, they form a sub-category of algebras. Recall that algebra morphisms are arrows that satisfy the following condition:

$$\begin{tikzcd}
Ta \arrow[r, "Tf"] \arrow[d, "\alpha"] & Tb \arrow[d, "\beta"] \\
a \arrow[r, "f"] & b
\end{tikzcd}$$

In light of this definition, we can re-interpret the second monad-algebra diagram as asserting that the structure map α of a monad algebra (the bottom arrow) is also an algebra morphism from (Ta, μ_a) to (a, α). This will come in handy in what follows.

Eilenberg-Moore category

The category of monad algebras for a given monad T on C is called the Eilenberg-Moore category and is denoted by C^T. It turns out that it is a good choice for the intermediate category that lets us factorize the monad T as a composition of a pair of adjoint functors.

The process goes as follows: we define a pair of functors, show that they form an adjunction, and then show that the monad generated by this adjunction is the original monad.

First of all, there is an obvious forgetful functor, which we'll call U^T, from C^T to C. It maps an algebra (a, α) to its carrier a, and treats algebra morphisms as regular morphisms between carriers.

More interestingly, there is a free functor F^T that is the left adjoint to U^T.

$$\begin{tikzcd}
C^T \arrow[r, bend right, "U^T"'] & C \arrow[l, bend right, "F^T"']
\end{tikzcd}$$

On objects, F^T maps an object a of C to a *monad algebra*, an object in C^T. For the carrier of this algebra we pick not a but Ta. For the structure map, which is the mapping $T(Ta) \to Ta$ we pick the component of monad multiplication $\mu_a \colon T(Ta) \to Ta$.

It's easy to check that this algebra (Ta, μ_a) is indeed a monad algebra—the necessary commuting conditions follow from monad laws. Indeed, substituting the algebra (Ta, μ_a) into the monad-algebra diagrams, we get (with the algebra part drawn in red):

$$\begin{tikzcd}
Ta \arrow[r, "\eta_{Ta}"] \arrow[dr, "id_{Ta}"'] & T(Ta) \arrow[d, "\mu_a"] \\
& Ta
\end{tikzcd}
\qquad
\begin{tikzcd}
T(T(Ta)) \arrow[r, "T\mu_a"] \arrow[d, "\mu_{Ta}"] & T(Ta) \arrow[d, "\mu_a"] \\
T(Ta) \arrow[r, "\mu_a"] & Ta
\end{tikzcd}$$

The first diagram is just the left monadic unit law in components. The η_{Ta} arrow corresponds to the whiskering of $\eta \circ T$. The second diagram is the associativity of μ with the two whiskerings $\mu \circ T$ and $T \circ \mu$ expressed in components.

To prove that we have an adjunction, we'll define two natural transformations to serve as the unit and the counit of the adjunction.

For the unit of the adjunction we pick the monadic unit η of T. They both have the same signature—in components, $\eta_a \colon a \to U^T(F^T a)$.

The counit is a natural transformation:

$$\varepsilon \colon F^T \circ U^T \to Id$$

The component of ε at (a, α) is an algebra morphism from the free algebra generated by a, that is (Ta, μ_a), back to (a, α). As we've seen earlier, α itself is such a morphism. We can therefore pick $\varepsilon_{(a,\alpha)} = \alpha$.

Triangular identities for these definitions of η and ε follow from unit laws for the monad and the monad algebra.

As is true for all adjunctions, the composition $U^T \circ F^T$ is a monad. We'll show that this the same monad we started with. Indeed, on objects, the composition $U^T(F^T a)$ first maps a to a free monad algebra (Ta, μ) and then forgets the structure map. The net result is the mapping of a to Ta, which is exactly what the original monad did.

On arrows, it lifts an arrow $f \colon a \to b$ using T. The fact that the arrow Tf is an algebra morphism from (Ta, μ_a) to (Tb, μ_b) follows from naturality of μ:

$$\begin{tikzcd}
T(Ta) \arrow[r, "T(Tf)"] \arrow[d, "\mu_a"] & T(TB) \arrow[d, "\mu_b"] \\
Ta \arrow[r, "Tf"] & TB
\end{tikzcd}$$

Finally, we have to show that the unit and the counit of the monad $U^T \circ F^T$ are the same as the unit and the counit of our original monad.

The units are the same by construction.

The monad multiplication of $U^T \circ F^T$ is given by the whiskering $U^T \circ \varepsilon \circ F^T$ of the unit of the adjunction. In components, this means instantiating ε at (Ta, μ_a), which gives us μ_a (the action of U^T on arrows is trivial). This is indeed the original monad multiplication.

We have thus shown that, for any monad T we can define the Eilenberg-Moore category and a pair of adjoint functors that factorize this monad.

Kleisli category

Inside every Eilenberg-Moore category there is a smaller Kleisli category struggling to get out. This smaller category is the image of the free functor we have constructed in the previous section.

Despite appearances, the image of a functor does not necessarily define a subcategory. Granted, it maps identities to identities and composition to composition. The problem may arise if two arrows that were not composable in the source category become composable in the target category. This may happen if the target of the first arrow is mapped to the same object as the source of the second arrow. In the example below, Ff and Fg are composable, but their composition $Fg \circ Ff$ may be absent from the image of the first category.

$$\begin{tikzcd}
a \arrow[d, "f"] & \\
b & Fa \arrow[d, "Ff"] \arrow[dd, dashed, bend left=60, "Fg \circ Ff"] \\
c \arrow[d, "g"] & Fb = Fc \arrow[d, "Fg"] \\
d & Fd
\end{tikzcd}$$

However, the free functor F^T maps distinct objects into distinct free algebras, so its image is indeed a subcategory of C^T.

We have encountered the Kleisli category before. There are many ways of constructing the same category, and the simplest one is to describe the Kleisli category is in terms of Kleisli arrows.

A Kleisli category for the monad (T, η, μ) is denoted by C_T. Its objects are the same as the objects of C, but an arrow in C_T from a to b is represented by an arrow in C that goes from a to Tb. You may recognize it as the Kleisli arrow we've defined before. Because T is a monad, these Kleisli arrows can be composed using the "fish" operator .

To establish the adjunction:

$$\begin{tikzcd}
C_T \arrow[r, bend right, "R^T"'] & C \arrow[l, bend right, "L_T"']
\end{tikzcd}$$

we define the left functor $L_T \colon C \to C_T$ as identity on objects. We still have to define what it does to arrows. It should map a regular arrow $f \colon a \to b$ to a Kleisli arrow from a to b. This Kleisli arrow $a \twoheadrightarrow b$ is represented by an arrow $a \to Tb$ in C. Such an arrow always exists as the composite $\eta_b \circ f$:

$$L_T f \colon a \xrightarrow{f} b \xrightarrow{\eta_b} Tb$$

The right functor $R_T \colon C_T \to C$ is defined on objects as a mapping that takes an a in the Kleisli category to an object Ta in C. Given a Kleisli arrow $a \twoheadrightarrow b$, which is represented by an arrow $g \colon a \to Tb$, R_T will map it to an arrow $R_T a \to R_T b$, that is an arrow $Ta \to Tb$ in C. We take this arrow to be the composite $\mu_b \circ Tg$:

$$Ta \xrightarrow{Tg} T(Tb) \xrightarrow{\mu_b} Tb$$

To establish the adjunction, we'll show the isomorphism of hom-sets:

$$\mathcal{C}_T(L_T a, b) \text{ ö } \mathcal{C}(a, R_T b)$$

An element of the left hand-side is a Kleisli arrow $a \twoheadrightarrow b$, which is represented by $f : a \to Tb$. We can find the same arrow on the right hand side, since $R_T b$ is Tb. So the isomorphism is between Kleisli arrows in $\mathcal{C}^T$ and the arrows in $\mathcal{C}$ that represent them.

The composite $R_T \circ L_T$ is equal to T and, indeed, it can be shown that this adjunction generates the original monad.

In general, there may be many adjunctions that generate the same monad. Adjunctions themselves form a 2-category, so it's possible to compare adjunctions using adjunction morphisms (1-cells in the 2-category). It turns out that the Kleisli adjunction is the initial object among all adjunctions that generate a given monad. Dually, the Eilenberg-Moore adjunction is terminal.

Chapter 16

Comonads

If it were easily pronounceable, we should probably call side effects “ntext,” because the dual to side effects is “context."

Just like we were using Kleisli arrows to deal with side effects, we use co-Kleisli arrows to deal with contexts.

Let’s start with the familiar example of an environment as a context. We have previously constructed a reader monad from it, by currying the arrow:

This time, however, we’ll treat it as a co-Kleisli arrow, which is an arrow from a “contextualized” argument.

As was the case with monads, we are interested in being able to compose such arrows. This is relatively easy for the environment-carrying arrows:

It’s also straightforward to implement an arrow that serves as an identity with respect to this composition:

This strongly suggests the idea that there is a category in which co-Kleisli arrows serve as morphisms.

Exercise 16.0.1. *Show that the composition of co-Kleisli arrows using is associative.*

16.1 Comonads in Programming

A functor (consider it a stylized upside-down) is a comonad if it supports composition of co-Kleisli arrows:

Here, the composition is written in the form of an infix operator. The unit of composition is called , since it extracts a value from the context.

Let's try it with our example. It is convenient to pass the environment as the first component of the pair. The comonad is then given by the functor that's a partial application of the pair constructor .

As with monads, co-Kleisli composition may be used in point-free style of programming. But we can also use the dual to called :

or the dual to bind called :

Here's how we can implement co-Kleisli composition in terms of and :

Exercise 16.1.1. *Implement in terms of and vice versa.*

The comonad

Interesting examples of comonads deal with larger, sometimes infinite, contexts. Here's an infinite stream:

If we consider such a stream as a value of the type in the context of an infinite tail, we can provide a instance for it:

Here, returns the head of the stream and turns a stream into a stream of streams, in which each consecutive stream is the tail of the previous one.

The intuition is that sets the stage for iteration, but it does it in a very general way. The head of each of the substreams can be interpreted as a future "current position" in the original stream.

It would be easy to perform a computation that goes over the head elements of these streams. But that's not where the power of a comonad lies. It lets us perform computations that require an arbitrary look-ahead. Such a computation requires access not only to heads of consecutive substreams, but to their tails as well.

This is what does: it applies a given co-Kleisli arrow to all the streams generated by :

Here's an example of a co-Kleisli arrow that averages the first five elements of a stream:

It uses a helper function that extracts the first items:

We can run over the whole stream using to smooth local fluctuation. Electrical engineers might recognize this as a simple low-pass filter with implementing the convolution. It produces a running average of the original stream.

Comonads are useful for structuring computations in spatially or temporally extended data structures. Such computations are local enough to define the "current location," but require gathering information from neighboring locations. Signal processing or image processing are good examples. So are simulations, in which differential equations have to be iteratively solved inside volumes: climate simulations, cosmological models, or nuclear reactions, to name a few. Conway's Game of Life is also a good testing ground for comonadic methods.

Sometimes it's convenient to perform calculation on continuous streams of data, postponing the sampling until the very last step. Here's an example of a signal that is a function of time (represented by)

The first component is a continuous stream of 's implemented as a function of time. The second component is the current time.

This is the instance for the continuous stream:

Here, convolves the filter

over the whole stream.

Exercise 16.1.2. *Implement the instance for a bidirectional stream:*

Assume that both list are infinite. Hint: Consider the first list as the past (in reverse order); the head of the second list as the present, and its tail as the future.

Exercise 16.1.3. *Implement a low-pass filter for from the previous exercise. It averages over three values: the current one, the one from the immediate past, and the one from the immediate future. For electrical engineers: implement a Gaussian filter.*

16.2 Comonads Categorically

We can get the definition of a comonad by reversing the arrows in the definition of a monad. Our corresponds to the reversed , and is the reversed .

A comonad is thus an endofunctor W equipped with two natural transformations:

$$\delta \colon W \to W \circ W$$
$$\varepsilon \colon W \to \mathrm{Id}$$

These transformations (corresponding to and , respectively) must satisfy the same identities as the monad, except with the arrows reversed.

These are the counit laws:

$$\begin{tikzcd}
\mathrm{Id} \circ W \arrow[dr, "="'] & W \circ W \arrow[l, "\varepsilon \circ W"'] \arrow[r, "W \circ \varepsilon"] & W \circ \mathrm{Id} \arrow[dl, "="] \\
& W \arrow[u, "\delta"] &
\end{tikzcd}$$

and this is the associativity law:

$$\begin{tikzcd}
(W \circ W) \circ W \arrow[rr, "="] & & W \circ (W \circ W) \\
W \circ W \arrow[u, "\delta \circ W"] & & W \circ W \arrow[u, "W \circ \delta"'] \\
& W \arrow[ul, "\delta"] \arrow[ur, "\delta"'] &
\end{tikzcd}$$

Comonoids

We've seen how monadic laws follow from monoid laws. We can expect that comonad laws should follow from a dual version of a monoid.

Indeed, a *comonoid* is an object w in a monoidal category $(\mathcal{C}, \otimes, I)$ equipped with two morphisms called co-multiplication and a co-unit:

$$\delta \colon w \to w \otimes w$$
$$\varepsilon \colon w \to I$$

We can replace the tensor product with endofunctor composition and the unit object with the identity functor to get the definition of a comonad as a comonoid in the category of endofunctors.

In Haskell we can define a typeclass for the cartesian product:

```
```

Comonoids are less talked about than their siblings, monoids, mainly because they are taken for granted. In a cartesian category, every object can be made into a comonoid just by using the diagonal map $\Delta_a \colon a \to a \quad a$ for co-multiplication, and the unique arrow to the terminal object for counit.

In programming this is something we do without thinking. Co-multiplication means being able to duplicate a value, and counit means being able to abandon a value.

In Haskell, we can easily implement the instance for any type:

In fact, we don't think twice of using the argument of a function twice, or not using it at all. But, if we wanted to be explicit, functions like:

could be written as:

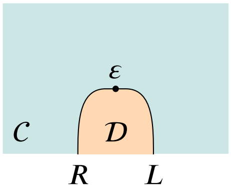
There are some situations, though, when duplicating or discarding a variable is undesirable. This is the case when the argument is an external resource, like a file handle, network port, or a chunk of memory allocated on the heap. Such resources are supposed to have well-defined lifetimes between being allocated and deallocated. Tracking lifetimes of objects that can be easily duplicated or discarded is very difficult and a notorious source of programming errors.

A programming model based on a cartesian category will always have this problem. The solution is to instead use a monoidal (closed) category that doesn't support duplication or destruction of objects. Such a category is a natural setting for *linear types*. Elements of linear types are used in Rust and, at the time of this writing, are being tried in Haskell. In C++ there are constructs that mimic linearity, like and move semantics.

16.3 Comonads from Adjunctions

We've seen that an adjunction $L \dashv R$ between two functors $L\colon \mathcal{D} \to \mathcal{C}$ and $R\colon \mathcal{C} \to \mathcal{D}$ gives rise to a monad $R \circ L\colon \mathcal{D} \to \mathcal{D}$. The other composition, $L \circ R$, which is an endofunctor in $\mathcal{C}$, turns out to be a comonad.

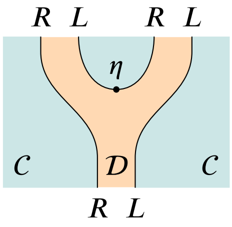
The counit of the adjunction serves as the counit of the comonad. This can be illustrated by the following string diagram:



The comultiplication is given by the whiskering of η:

$$\delta = L \circ \eta \circ R$$

as illustrated by this string diagram:



As before, comonad laws can be derived from triangle identities.

Costate comonad

We've seen that the state monad can be generated from the currying adjunction between the product and the exponential. The left functor was defined as a product with some fixed object s:

$$L_s a = a \quad s$$

and the right functor was the exponentiation, parameterized by the same object s:

$$R_s c = c^s$$

The composition $L_s \circ R_s$ generates a comonad called the *costate comonad* or the *store comonad*.

Translated to Haskell, the right functor assigns a function type to , and the left functor pairs with . The result of the composition is the endofunctor:

or, using GADT notation:

The functor instance post-composes the function to the first component of :

The counit of this adjunction, which becomes the comonadic , is function application:

The unit of this adjunction is a natural transformation $\eta \colon \text{Id} \to R_s \circ L_s$. We've used it as the of the state monad. This is its component at :

To get we need to whisker η it between the two functors:

$$\delta = L_s \circ \eta \circ R_s$$

Whiskering on the right means taking the component of η at the object $R_s c$, and whiskering on the left means lifting this component using L_s. Since Haskell translation of whiskering is a tricky process, let's analyze it step-by-step.

For simplicity, let's fix the type to, say, . We encapsulate the left functor into a :

and keep the right functor a type synonym:

The unit of the adjunction can be written as a natural transformation using explicit :

We can now implement comultiplication as the whiskering of . The whiskering on the right is encoded in the type signature, by using the component of at . The whiskering on the left is done by lifting using the defined for the functor. We use the language pragma to make it explicit which is to be used:

This can be rewritten more explicitly as:

The instance can thus be written as:

The store comonad is a useful programming concept. To understand it, let's consider again the case where is .

We interpret the first component of , the function , to be an accessor to an imaginary infinite stream of values, one for each integer.

The second component can be interpreted as the current index. Indeed, uses this index to retrieve the current value.

With this interpretation, produces an infinite stream of streams, each shifted by a different offset, and performs a convolution on this stream. Of course, laziness saves the day: only the values we explicitly demand will be evaluated.

Notice also that our earlier example of the comonad is reproduced by .

Exercise 16.3.1. *A cellular automaton can be implemented using the store comonad. This is the co-Kleisli arrow describing rule 110:*

A cell can be either live or dead:

Run a few generations of this automaton. Hint: Use the function from the Prelude.

Comonad coalgebras

Dually to monad algebras we have comonad coalgebras. Given a comonad (W, ε, δ), we can construct a coalgebra, which consists of a carrier object a and an arrow $\phi \colon a \to W a$. For this coalgebra to compose nicely with the comonad, we'll require that we be able to extract the value that was injected using ϕ; and that the lifting of ϕ, when acting on the result of ϕ, be equivalent to duplication:

$$\begin{tikzcd}
a & Wa \arrow[l, "\varepsilon_a"'] & & W(Wa) & Wa \arrow[l, "W\phi"'] \\
& a \arrow[ul, "id_a"] \arrow[u, "\phi"'] & & Wa \arrow[u, "\delta_a"'] & a \arrow[l, "\phi"'] \arrow[u, "\phi"']
\end{tikzcd}$$

Just like with monad algebras, comonad coalgebras form a category. Given a comonad (W, ε, δ) in $\mathcal{C}$, its comonad coalgebras form the category called the Eilenberg-Moore category (sometimes prefixed with co-) $\mathcal{C}^W$.

There is a co-Kleisli subcategory of $\mathcal{C}^W$ denoted by $\mathcal{C}_W$

Given a comonad W, we can construct an adjunction using either $\mathcal{C}^W$ or $\mathcal{C}_W$ that reproduces the comonad W. The construction is fully analogous to the one for monads.

Lenses

The coalgebra for the comonad is of particular interest. We'll do some renaming first. Let's call the carrier and the state .

The coalgebra is given by a function:

which is equivalent to a pair of functions:

Such a pair is called a lens: is called the source, and is the focus.

With this interpretation lets us extract the focus, and replaces the focus with a new value to produce a new .

Lenses were first introduced to describe the retrieval and modification of data in database records. Then they found application is working with data structures. A lens objectifies the idea of having read/write access to a part of a larger object. For instance, a lens can focus on one of the components of a pair or a particular component of a record. We'll discuss lenses and optics in the next chapter.

Let's apply the laws of the comonad coalgebra to a lens. For simplicity, let's omit data constructors from the equations. We get the following simplified definitions:

$$\begin{tikzcd}
s & Ws \arrow[l, "\varepsilon_s"'] & & W(Ws) & Ws \arrow[l, "W\phi"'] \\
& a \arrow[ul, "id_s"] \arrow[u, "\phi"'] & & Ws \arrow[u, "\delta_s"'] & s \arrow[l, "\phi"'] \arrow[u, "\phi"']
\end{tikzcd}$$

The first law tells us that applying the result of to the result of results in identity:

This is called the set/get law of the lens. Nothing should change when you replace the focus with the same focus.

The second law requires the application of to the result of :

This should be equal to the application of :

Comparing the two, we get:

Let's apply it to some :

Using the definition of gives us:

We have two equalities. The first components are functions, so we apply them to some and get the set/set lens law:

Setting the focus to and then overwriting it with is the same as setting the focus directly to .

The second components give us the get/set law:

After we set the focus to , the result of is .

Lenses that satisfy these laws are called *lawful lenses*. They are comonad coalgebras for the store comonad.

Chapter 17

Ends and Coends

17.1 Profunctors

In the rarified air of category theory we encounter patterns that are so far removed from their origins that we have problems visualizing them. It doesn't help that the more abstract a pattern gets the more dissimilar the concrete examples of it are.

An arrow from a to b is relatively easy to visualize. We have a very familiar model for it: a function that consumes elements of a and produces elements of b. A hom-set is a collection of such arrows.

A functor is an arrow between categories. It consumes objects and arrows from one category and produces objects and arrows from another. We can think of it a recipe for building such objects (and arrows) from materials provided by the source category. In particular, we often think of an endofunctor as a container of building materials.

A profunctor maps a pair of objects $\langle a, b\rangle$ to a set $P\langle a, b\rangle$ and a pair of arrows:

$$\langle f \colon s \to a, g \colon b \to t\rangle$$

to a function:

$$P\langle f, g\rangle \colon P\langle a, b\rangle \to P\langle s, t\rangle$$

A profunctor is an abstraction that combines elements of many other abstractions. Since it's a functor $C^{op}\ \ C \to \mathrm{Set}$, we can think of it as constructing a set from a pair of objects, and a function from a pair of arrows (one of them going in the opposite direction). This doesn't help our imagination though.

Fortunately, we have a good model for a profunctor: the hom-functor. The set of arrows between two objects behaves like a profunctor when you vary the objects. It also makes sense that there is a difference between varying the source and the target of the hom-set.

We can, therefore, think of an arbitrary profunctor as generalizing the hom-functor. A profunctor provides additional bridges between objects, on top of hom-sets that are already there.

There is, however one big difference between an element of the hom-set $\mathcal{C}(a, b)$ and an element of the set $P\langle a, b\rangle$. Elements of hom-sets are arrows, and arrows can be composed. It's not immediately obvious how to compose profunctors.

Granted, the lifting of arrows by a profunctor can be seen as generalizing composition—just not between profuctors, but between hom-sets and profunctors. For instance, we can "precompose" $P\langle a, b\rangle$ with an arrow $f\colon s \to a$ to obtain $P\langle s, b\rangle$:

$$P\langle f, id_b\rangle\colon P\langle a, b\rangle \to P\langle s, b\rangle$$

Similarly, we can "postcompose" it with $g\colon b \to t$:

$$P\langle id_a, g\rangle\colon P\langle a, b\rangle \to P\langle a, t\rangle$$

This kind of heterogenous composition takes a composable pair consisting of an arrow and an element of a profunctor and produces an element of a profunctor.

A profunctor can be extended this way on both sides by lifting a pair of arrows:

$$\begin{tikzcd}
s \arrow[r, dashed, bend left, "f"] & a \arrow[r, bend right, "P"] & b \arrow[r, dashed, bend left, "g"] & t
\end{tikzcd}$$

Collages

There is no reason to restrict a profunctor to a single category. We can easily define a profunctor between two categories as a functor $P\colon \mathcal{C}^{op}\quad \mathcal{D} \to \mathrm{Set}$. Such a profunctor can be used to glue two categories together by generating the missing hom-sets from the objects in $\mathcal{C}$ to the objects in $\mathcal{D}$.

A collage (or a cograph) of two categories $\mathcal{C}$ and $\mathcal{D}$ is a category whose objects are objects from both categories (a disjoint union). A hom-set between two objects x and y is either a hom-set in $\mathcal{C}$, if both objects are in $\mathcal{C}$; a hom-set in $\mathcal{D}$, if both are in $\mathcal{D}$; or the set $P\langle x, y\rangle$ if x is in $\mathcal{C}$ and y is in $\mathcal{D}$. Otherwise the hom-set is empty.

Composition of morphisms is the usual composition, except if one of the morphisms is an element of $P\langle x, y\rangle$. In that case we lift the morphism we're trying to pre- or post-compose.

It's easy to see that a collage is indeed a category. The new morphisms that go between the two sides of the collage are sometimes called heteromorphisms. They can only go from $\mathcal{C}$ to $\mathcal{D}$, never the other way around.

Seen this way, a profunctor $\mathcal{C}^{op}\quad \mathcal{C} \to \mathrm{Set}$ should really be called an *endo*-profunctor. It defines a collage of $\mathcal{C}$ with itself.

Exercise 17.1.1. *Show that there is a functor from a collage of two categories to a stick-figure "walking arrow" category that has two objects and one arrow between them (and two identity arrows).*

Exercise 17.1.2. *Show that, if there is a functor from $\mathcal{C}$ to the walking arrow category then $\mathcal{C}$ can be split into a collage of two categories.*

Profunctors as relations

Under a microscope, a profunctor looks like a hom-functor, and the elements of the set $P\langle a, b\rangle$ look like individual arrows. But when we zoom out, we can view a profunctor as a relation between objects. These are not the usual relations; they are *proof-relevant* relations.

To understand this concept better, let's consider a regular functor $F \colon \mathcal{C} \to \mathbf{Set}$ (in other words, a co-presheaf). One way to interpret it is to say that it defines a subset of objects of $\mathcal{C}$, namely those objects that are mapped to non-empty sets. Every element of Fa is then treated as a proof that a is a member of this subset. If, on the other hand, Fa is an empty set, then a is not a member of the subset.

We can apply the same interpretation to profunctors. If the set $P\langle a, b\rangle$ is empty, we say that b is not related to a. If it's not empty, we say that each element of the set $P\langle a, b\rangle$ represents a proof that b is related to a. We can then treat a profunctor as a proof-relevant relation.

Notice that we don't assume anything about this relation. It doesn't have to be reflexive, as it's possible for $P\langle a, a\rangle$ to be empty (in fact, $P\langle a, a\rangle$ makes sense only for endo-profunctors). It doesn't have to be symmetric either.

Since the hom-functor is an example of an (endo-) profunctor, this interpretation lets us view the hom-functor in a new light: as a built-in proof-relevant relation between objects in a category. If there's an arrow between two objects, they are related. Notice that this relation is reflexive, since $\mathcal{C}(a, a)$ is never empty: at the very least, it contains the identity morphism.

Moreover, as we've seen before, hom-functors interact with profunctors. If a is related to b through P, and the hom-sets $\mathcal{C}(s, a)$ and $\mathcal{D}(b, t)$ are non-empty, then automatically s is related to t through P. Profunctors are therefore proof-relevant relations that are compatible with the structure of the categories in which they operate.

We know how to compose a profunctor with hom-functors, but how would we compose two profunctors? We can get a clue from the composition of relations.

Suppose that you want to charge your cellphone, but you don't have a charger. In order to connect you to a charger it's enough that you have a friend who owns a charger. Any friend will do. You compose the relation of having a friend with the relation of a person having a charger to get a relation of being able to charge your phone. The proof that you can charge your phone is a pair of proofs, one of friendship and one of the possession of a charger.

In general, we say that two objects are related by the composite relation if there exists an object in the middle that is related to both of them.

Profunctor composition in Haskell

Composition of relations can be translated to profunctor composition in Haskell. Let's first recall the definition of a profunctor:

The key to understanding profunctor composition is that it requires the *existence* of the object in the middle. For object b to be related to object a through the composite $P \diamond Q$ there has to exist an object x that bridges the gap:

$$\begin{tikzcd} a \arrow[r, bend left, "Q", tail] & x \arrow[r, bend left, "P", tail] & b \end{tikzcd}$$

This can be encoded in Haskell using an existential type. Given two profunctors and , their composition is a new profunctor :

We are using a to express the existential nature of the object . The two arguments to the data constructor can be seen as a pair of proofs: one proves that is related to , and the other that is related to . This pair then constitutes the proof that is related to .

An existential type can be seen as a generalization of a sum type. We are summing over all possible types . Just like a finite sum can be constructed by injecting one of the alternatives (think of the two constructors of), the existential type can be constructed by picking one particular type for and injecting it into the definition of .

Just as mapping out from a sum type requires a pair of function, one per each alternative; a mapping out from an existential type requires a family of functions, one per every type. The mapping out from , for instance, is given by a polymorphic function:

The composition of profunctors is again a profunctor, as can be seen from this instance:

This is just saying that you can extend the composite profunctor by extending the first one to the left and the second one to the right.

The fact that this definition of profunctor composition happens to work in Haskell is due to parametricity. The language constraints the types of profunctors in a way that makes this work. In general, though, taking a simple sum over intermediate objects would result in over-counting, so in category theory we have to compensate for that.

17.2 Coends

The over-counting in the naive definition of profunctor composition happens when two candidates for the object in the middle are connected by a morphism:

$$\begin{tikzcd} a \arrow[r, bend left, "Q"] & x \arrow[r, dashed, "f"] & y \arrow[r, bend left, "P"] & b \end{tikzcd}$$

We can either extend Q on the right, by lifting $Q\langle id, f\rangle$, and use y as the middle object; or we can extend P on the left, by lifting $P\langle f, id\rangle$, and use x as the intermediary.

In order to avoid the double-counting, we have to tweak our definition of a sum type when applied to profunctors. The resulting construction is called a coend.

First, let's re-formulate the problem. We are trying to sum over all objects x in the product:

$$P\langle a, x\rangle \quad Q\langle x, b\rangle$$

The double-counting happens because we can open up the gap between the two profunctors, as long as there is a morphism that we can fit between them. So we are really looking at a more general product:

$$P\langle a, x\rangle \quad Q\langle y, b\rangle$$

The important observation is that, if we fix the endpoints a and b, this product is a profunctor in $\langle y, x\rangle$. This is easily seen after a little rearrangement (up to isomorphism):

$$Q\langle y, b\rangle \quad P\langle a, x\rangle$$

We are interested in the sum of the diagonal parts of this profunctor, that is when x is equal to y.

So let's see how we would go about defining the sum of all diagonal entries of a general profunctor P. In fact, this construction works for any functor $P \colon \mathcal{C}^{op} \quad \mathcal{C} \to \mathcal{D}$, not just for $\mathbf{Set}$-valued profunctors.

The sum of the diagonal objects is defined by injections; in this case, one per every object in $\mathcal{C}$. Here we show just two of them and the dashed line representing all the rest:

$$\begin{tikzcd} P\langle y, y\rangle \arrow[rr, dash, dashed] \arrow[dr, "i_y"'] & & P\langle x, x\rangle \arrow[dl, "i_x"] \\ & d & \end{tikzcd}$$

If we were defining a sum, we'd make it a universal object equipped with such injections. But because we are dealing with functors of two variables, we want to identify the injections that are related by "extending" some common ancestor—here, $P\langle y, x\rangle$. We want the following diagram to commute, whenever there is a connecting morphism $f \colon x \to y$:

$$\begin{tikzcd}
& P\langle y, x\rangle \arrow[ld, "P\langle id{,}f\rangle"'] \arrow[rd, "P\langle f{,}id\rangle"] & \\
P\langle y, y\rangle \arrow[rd, "i_y"'] & & P\langle x, x\rangle \arrow[ld, "i_x"] \\
& d &
\end{tikzcd}$$

This diagram is called a co-wedge, and its commuting condition is called the co-wedge condition. For every $f \colon x \to y$, we demand that:

$$i_x \circ P\langle f, id_y\rangle = i_y \circ P\langle id_x, f\rangle$$

The *universal* co-wedge is called a coend.

Since a coend generalizes the sum to a potentially infinite domain, we write it using the integral sign, with the “integration variable” at the top:

$$\int^{x \colon C} P\langle x, x\rangle$$

Universality means that, whenever there is an object d in $\mathcal{D}$ equipped with a family of arrows $g_x \colon P\langle x, x\rangle \to d$ satisfying the co-wedge condition, there is a unique mapping out from the coend:

$$h \colon \int^{x \colon C} P\langle x, x\rangle \to d$$

that factorizes every g_x through the injection i_x:

$$g_x = h \circ i_x$$

Pictorially, we have:

$$\begin{tikzcd}
& P\langle y, x\rangle \arrow[ld, "P\langle id{,}f\rangle"'] \arrow[rd, "P\langle f{,}id\rangle"] & \\
P\langle y, y\rangle \arrow[rd, "i_y"'] \arrow[rdd, bend right, "g_y"'] & & P\langle x, x\rangle \arrow[ld, "i_x"] \arrow[ldd, bend left, "g_x"] \\
& \int^x P\langle x, x\rangle \arrow[d, dashed, "h"] & \\
& d &
\end{tikzcd}$$

Compare this with the definition of a sum of two objects:

$$\begin{tikzcd}
a \arrow[rd, "\text{Left}"] \arrow[rdd, bend right, "f"'] & & b \arrow[ld, "\text{Right}"'] \arrow[ldd, bend left, "g"] \\
& a + b \arrow[d, dashed, "h"] & \\
& d &
\end{tikzcd}$$

Just like the sum was defined as a universal cospan, a coend is defined as a universal co-wedge.

In particular, if you were to construct a coend of a Set-valued profunctor, you would start with a sum (a discriminated union) of all the sets $P\langle x, x\rangle$. Then you would identify all the elements of this sum that satisfy the co-wedge condition. You'd identify the element a Ë $P\langle x, x\rangle$ with the element b Ë $P\langle y, y\rangle$ whenever there is an element c Ë $P\langle y, x\rangle$ and a morphism $f\colon x \to y$, such that:

$$P\langle id, f\rangle(c) = b$$

and

$$P\langle f, id\rangle(c) = a$$

Notice that, in a discrete category (which is just a set of objects with no arrows between them) the co-wedge condition is trivial (there are no f's other than identities), so a coend is just a straightforward sum (coproduct) of the diagonal objects $P\langle x, x\rangle$.

Extranatural transformations

A family of arrows in the target category parameterized by the objects of the source category can often be combined into a single natural transformation between two functors.

The injections in our definition of a cowedge form a family of functions that is parameterized by objects, but they don't neatly fit into a definition of a natural transformation.

$$\begin{tikzcd}
P\langle y, y\rangle \arrow[rr, dash, dashed] \arrow[dr, "i_y"'] & & P\langle x, x\rangle \arrow[dl, "i_x"] \\
& d &
\end{tikzcd}$$

The problem is that the functor $P\colon \mathcal{C}^{op}\ \ \mathcal{C} \to \mathcal{D}$ is contravariant in the first argument and covariant in the second; so its diagonal part, which on objects is defined as $x \mapsto P\langle x, x\rangle$, is neither.

The closest analog of naturality at our disposal is the cowedge condition:

$$\begin{tikzcd}
& P\langle y, x\rangle \arrow[dl, "P\langle id{,}f\rangle"'] \arrow[dr, "P\langle f{,}id\rangle"] & \\
P\langle y, y\rangle \arrow[dr, "i_y"'] & & P\langle x, x\rangle \arrow[dl, "i_x"] \\
& d &
\end{tikzcd}$$

Indeed, as is the case with the naturality square, it involves the interaction between the lifting of a morphism $f\colon x \to y$ (here, in two different ways) and the components of the transformation i.

Granted, the standard naturality condition deals with pairs of functors. Here, the target of the transformation is a fixed object d. But we can always reinterpret it as the output of a constant profunctor $\Delta_d \colon \mathcal{C}^{op}\ \ \mathcal{C} \to \mathcal{D}$. Thus to generalize naturality, we replace Δ_d with an arbitrary profunctor Q.

We reinterpret the cowedge condition as a special case of the more general *extranatural* transformation. An extranatural transformation is a family of arrows:

$$\alpha_{cd} \colon P\langle c, c\rangle \to Q\langle d, d\rangle$$

between two functors of the form:

$$P \colon \mathcal{C}^{op}\ \ \mathcal{C} \to \mathcal{E}$$

$$Q \colon \mathcal{D}^{op}\ \ \mathcal{D} \to \mathcal{E}$$

Extranaturality in c means that the following diagram commutes for any morphism $f \colon c \to c^{\ddot{}}$:

$$\begin{tikzcd}
& P\langle c^{\ddot{}}, c\rangle \arrow[dl, "P\langle id{,}f\rangle"'] \arrow[dr, "P\langle f{,}id\rangle"] & \\
P\langle c^{\ddot{}}, c^{\ddot{}}\rangle \arrow[dr, "\alpha_{c^{\ddot{}}d}"'] & & P\langle c, c\rangle \arrow[dl, "\alpha_{cd}"] \\
& Q\langle d, d\rangle &
\end{tikzcd}$$

Extranaturality in d means that the following diagram commutes for any morphism $g \colon d \to d^{\ddot{}}$:

$$\begin{tikzcd}
& P\langle c, c\rangle \arrow[dl, "\alpha_{cd}"'] \arrow[dr, "\alpha_{cd^{\ddot{}}}"] & \\
Q\langle d, d\rangle \arrow[dr, "Q\langle id{,}g\rangle"'] & & Q\langle d^{\ddot{}}, d^{\ddot{}}\rangle \arrow[dl, "Q\langle g{,}id\rangle"] \\
& Q\langle d, d^{\ddot{}}\rangle &
\end{tikzcd}$$

Given this definition, we get our cowedge condition as the extranaturality of the mapping between the profunctor P and the constant profunctor Δ_d.

We can now reformulate the definition of the coend as a pair (c, i) where c is the object equipped with the extranatural transformation $i \colon P \to \Delta_c$ that is universal among such pairs.

Universality means that for any object d equipped with the extranatural transformation $\alpha \colon P \to \Delta_d$ there is a unique morphism $h \colon c \to d$ that factorizes all the components of α through the components of i:

$$\alpha_x = h \circ i_x$$

We call this object c the coend, and write it as:

$$c = \int^x P\langle x, x\rangle$$

Exercise 17.2.1. *Verify that that, for the extranatural transformation $P \to \Delta_d$, the first extranaturality diamond is equivalent to the cowedge condition, and the second is trivially satisfied.*

Profunctor composition using coends

Equipped with the definition of a coend we can now formally define the composition of two profunctors:

$$(P \diamond Q)\langle a, b\rangle = \int^{x\colon C} Q\langle a, x\rangle \quad P\langle x, b\rangle$$

Compare this with:

The reason why in Haskell we don't have to worry about the co-wedge condition is analogous to the reason why all parametrically polymorphic functions automatically satisfy naturality condition. A coend is defined using a family of injections; in Haskell all these injections are defined by a *single* polymorphic function:

Parametricity then enforces the co-wedge condition.

Coends introduce a new level of abstraction in dealing with profunctors. Calculations using coends usually take advantage of their mapping-out property. To define a mapping out of a coend to some object d:

$$\int^{x} P\langle x, x\rangle \to d$$

it's enough to define a family of functions from the diagonal entries of the functor to d:

$$g_x\colon P\langle x, x\rangle \to d$$

satisfying the cowedge condition. You can get a lot of mileage from this trick, especially when combined with the Yoneda lemma. We'll see examples of this in what follows.

Exercise 17.2.2. *Define a instance for the pair of profunctors:*

Hint: Keep the first four parameters fixed:

Exercise 17.2.3. *Profunctor composition can be expressed using a coend:*

Define a instance for .

Colimits as coends

A function of two variables that ignores one of its arguments is equivalent to a function of one variable. Similarly, a profunctor that ignores one of its arguments is equivalent to a functor.

Conversely, given a functor F, we can construct a trivial profunctor. Its action on objects is given by:

$$P\langle x, y\rangle = F y$$

Its action on a pair of arrows ignores one of the arrows:

$$P\langle f, g\rangle = F g$$

For any $f : x \to y$, our definition of a coend for such a profunctor reduces to the following diagram:

$$\begin{tikzcd}
 & Fx \arrow[dl, "Ff"'] \arrow[dr, dashed, "id_{Fx}"] & \\
Fy \arrow[dr, "i_y"] \arrow[ddr, bend right, "g_y"'] & & Fx \arrow[dl, "i_x"'] \arrow[ddl, bend left, "g_x"] \\
 & \int^x Fx \arrow[d, dashed, "h"] & \\
 & d &
\end{tikzcd}$$

After shrinking the identity arrows, the original co-wedge becomes a co-cone, and the universal condition turns into the definition of a colimit. This justifies the use the coend notation for colimits:

$$\int^x Fx = \text{colim} F$$

The functor F defines a diagram in the target category. The pattern is the whole source category.

We can gain useful intuition if we consider a discrete category, in which a profunctor is a (possibly infinite) matrix and a coend is the sum (coproduct) if its diagonal elements. A profunctor that is constant along one axis corresponds to a matrix whose rows are identical (each given by a "vector" Fx). The sum of the diagonal elements of such a matrix is equal to the sum of all components of the vector Fx.

$$\begin{pmatrix}
Fa & Fb & Fc & ... \\
Fa & Fb & Fc & ... \\
Fa & Fb & Fc & ... \\
... & ... & ... & ...
\end{pmatrix}$$

In a non-discrete category, this sum generalizes to a colimit.

17.3 Ends

Just like a coend generalizes the sum of the diagonal elements of a profunctor—its dual, an end, generalizes the product. A product is defined by its projections, and so is an end.

The generalization of a span that we used in the definition of a product would be an object d with a family of projections, one per every object x:

$$\pi_x \colon d \to P\langle x, x\rangle$$

The dual to a co-wedge is called a wedge:

$$\begin{tikzcd}
& d \arrow[ld, "\pi_x"'] \arrow[rd, "\pi_y"] & \\
P\langle x, x\rangle \arrow[rd, "P\langle id, f\rangle"'] & & P\langle y, y\rangle \arrow[ld, "P\langle f, id\rangle"] \\
& P\langle x, y\rangle &
\end{tikzcd}$$

For every arrow $f \colon x \to y$ we demand that:

$$P\langle f, id_y\rangle \circ \pi_y = P\langle id_x, f\rangle \circ \pi_x$$

The end is a universal wedge. We use the integral sign for it too, this time with the "integration variable" at the bottom.

$$\int_{x \colon C} P\langle x, x\rangle$$

You might be wondering why, in calculus, integrals based on multiplication rather than summation are rarely used. That's because we can always use a logarithm to replace multiplication with addition. We don't have this luxury in category theory, so ends and coends are equally important.

To summarize, an end is an object equipped with a family of morphisms (projections):

$$\pi_a \colon \left(\int_x P\langle x, x\rangle \right) \to P\langle a, a\rangle$$

satisfying the wedge condition.

It is universal among such objects; that is, for any other object d equipped with a family of arrows g_x satisfying the wedge condition, there is a unique morphism h that factorizes the family g_x through the family π_x:

$$g_x = \pi_x \circ h$$

Pictorially, we have:

$$\begin{tikzcd}
& d \arrow[d, dashed, "h"] \arrow[ldd, bend right, "g_x"'] \arrow[rdd, bend left, "g_y"] & \\
& \int_x P\langle x, x\rangle \arrow[ld, "\pi_x"'] \arrow[rd, "\pi_y"] & \\
P\langle x, x\rangle \arrow[rd, "P\langle id, f\rangle"'] & & P\langle y, y\rangle \arrow[ld, "P\langle f, id\rangle"] \\
& P\langle x, y\rangle &
\end{tikzcd}$$

Equivalently, we can say that the end is a pair (e, π) consisting of an object $e = \int_x P\langle x, x\rangle$ and an extranatural transformation $\pi \colon \Delta_d \to e$ that is universal among such pairs. The wedge condition turns out to be a special case of extranaturality condition.

If you were to construct an end of a Set-valued profunctor, you'd start with a giant product of all $P\langle x, x\rangle$ for all objects in the category $\mathcal{C}$, and then prune the tuples that don't satisfy the wedge condition.

In particular, imagine using the singleton set 1 in place of d. The family g_x would select one element from each set $P\langle x, x\rangle$. This would give you a giant tuple. You'd weed out most of these tuples, leaving only the ones that satisfy the wedge condition.

Again, in Haskell, due to parametricity, the wedge condition is automatically satisfied, and the definition of the end for a profunctor simplifies to:

The Haskell implementation of an doesn't showcase the fact that it is dual to the . This is because, at the time of this writing, Haskell doesn't have a built-in syntax for existential types. If it did, the would be implemented as:

The existential/universal duality between a and an means that it's easy to construct a — all you need is to pick one type for which you have a value of the type . On the other end, to construct an you have to provide a whole family of values , one for every type . In other words, you need a polymorphic formula that is parameterized by . A definition of a polymorphic function is a canonical example of such a formula.

Natural transformations as an end

The most interesting application of an end is in concisely defining the set of natural transformations. Consider two functors, F and G, going between two categories $\mathcal{B}$ and $\mathcal{C}$. A natural transformation between them is a family of arrows α_x in $\mathcal{C}$. You may think of it as picking one element α_x from each hom-set $\mathcal{C}(Fx, Gx)$.

$$\begin{tikzcd}
 & Fx \arrow[dd, "\alpha_x \text{Ë} \mathcal{C}(Fx,Gx)"] \\
x \arrow[ur, dashed, "F"] \arrow[dr, dashed, "G"'] & \\
 & Gx
\end{tikzcd}$$

We know that the mapping $\langle a, b\rangle \to \mathcal{C}(a, b)$ defines a profunctor. It turns out that, for any pair of functors, the mapping $\langle a, b\rangle \to \mathcal{C}(Fa, Gb)$ also behaves like a profunctor. The action of this profunctor on a pair of arrows $\langle f \colon s \to a, g \colon b \to t\rangle$ is a function:

$$\mathcal{C}(Fa, Gb) \to \mathcal{C}(Fs, Gt)$$

given by the composition:

$$Fs \xrightarrow{Ff} Fa \xrightarrow{h} Gb \xrightarrow{Gg} Gt$$

where h is an element of $\mathcal{C}(Fa, Gb)$.

The diagonal parts of this profunctor are perfect candidates for the components of a natural transformation. In fact, the end:

$$\int_{x\colon \mathcal{B}} \mathcal{C}(Fx, Gx)$$

defines a set of natural transformations from F to G.

To show this, let's check the wedge condition. Plugging in our profunctor, we get:

$$\begin{tikzcd}
& \int_x \mathcal{C}(Fx, Gx) \arrow[dl, "\pi_a"'] \arrow[dr, "\pi_b"] & \\
\mathcal{C}(Fa, Ga) \arrow[dr, "(Ff \circ *)"'] & & \mathcal{C}\langle Fb, Gb\rangle \arrow[dl, "(* \circ Gf)"] \\
& \mathcal{C}(Fa, Gb) &
\end{tikzcd}$$

We can pick a single element of the set $\int_x \mathcal{C}(Fx, Gx)$ by instantiating the universal condition for the singleton set:

$$\begin{tikzcd}
& 1 \arrow[d, dashed, "\alpha"] \arrow[ddl, bend right, "\alpha_a"'] \arrow[ddr, bend left, "\alpha_b"] & \\
& \int_x \mathcal{C}(Fx, Gx) \arrow[dl, "\pi_a"'] \arrow[dr, "\pi_b"] & \\
\mathcal{C}(Fa, Ga) \arrow[dr, "(Ff \circ *)"'] & & \mathcal{C}\langle Fb, Gb\rangle \arrow[dl, "(* \circ Gf)"] \\
& \mathcal{C}(Fa, Gb) &
\end{tikzcd}$$

We pick the component α_a from the hom-set $\mathcal{C}(Fa, Ga)$ and the component α_b from $\mathcal{C}(Fb, Gb)$. The wedge condition then boils down to:

$$Ff \circ \alpha_a = \alpha_b \circ Gf$$

for any $f\colon a \to b$. This is exactly the naturality condition. So any element α of this end is automatically a natural transformation.

The set of natural transformations, or the hom-set in the functor category, is thus given by the end:

$$[\mathcal{C}, \mathcal{D}](F, G) \text{ ö } \int_{x\colon \mathcal{B}} \mathcal{C}(Fx, Gx)$$

In Haskell, this is consistent with our earlier definition:

As we discussed earlier, to construct an we have to give it a whole family of values parameterized by types. Here, these values are the components of a polymorphic function.

Limits as ends

Just like we were able to express colimits as coends, we can express limits as ends. As before, we define a trivial profunctor that ignores its first argument:

$$P\langle x, y\rangle = Fy$$
$$P\langle f, g\rangle = Fg$$

The universal condition that defines an end becomes the definition of a universal cone:

$$\begin{tikzcd}
 & d \arrow[d, dashed, "h"] \arrow[ddl, bend right, "g_x"'] \arrow[ddr, bend left, "g_y"] & \\
 & \int_x Fx \arrow[dl, "\pi_x"'] \arrow[dr, "\pi_y"] & \\
Fx \arrow[dr, "Ff"'] & & Fy \arrow[dl, dashed, "id_{Fy}"] \\
 & Fy &
\end{tikzcd}$$

We can thus use the end notation for limits:

$$\int_x Fx = \lim F$$

Exercise 17.3.1. *A product is a limit of a functor from a two-object category* 2. *Show that it can be defined as an end. Hint: There are no non-identity morphisms in* 2.

17.4 Continuity of the Hom-Functor

In category theory, a functor F is called *continuous* if it preserves limits (and co-continuous, if it preserves colimits). It means that, if you have a diagram in the source category, it doesn't matter if you first use F to map the diagram and then take the limit in the target category, or take the limit in the source category and then use F to map this limit.

The hom-functor is an example of a functor that is continuous in its second argument. Since a product is the simplest example of a limit, this means, in particular, that:

$$C(x, a \quad b) \text{ ö } C(x, a) \quad C(x, b)$$

The left hand side applies the hom-functor to the product (a limit of a span). The right hand side maps the diagram, here just a pair of objects, and takes the product (limit) in the target category.

The target category for the hom-functor is $\mathbf{Set}$, so this is just a cartesian product. The two sides are isomorphic by the universal property of the product: the mapping into the product is defined by a pair of mappings into the two objects.

Continuity of the hom-functor in the first argument is reversed: it maps colimits to limits. Again, the simplest example of a colimit is the sum, so we have:

$$\mathcal{C}(a+b,x) \text{ ö } \mathcal{C}(a,x) \quad \mathcal{C}(b,x)$$

This follows from the universality of the sum: a mapping out of the sum is defined by a pair of mapping out of the two objects.

It can be shown that an end can be expressed as a limit, and a coend as a colimit. Therefore, by continuity of the hom-functor, we can always pull out the integral sign from inside a hom-set. By analogy with the product, we have the mapping-in formula for an end:

$$\mathcal{D}\left(d, \int_a P\langle a,a\rangle\right) \text{ ö } \int_a \mathcal{D}(d, P\langle a,a\rangle)$$

By analogy with the sum, we have a mapping-out formula for the coend:

$$\mathcal{D}\left(\int^a P\langle a,a\rangle, d\right) \text{ ö } \int_a \mathcal{D}(P\langle a,a\rangle, d)$$

Notice that, in both cases, the right-hand side is an end.

17.5 Fubini Rule

The Fubini rule in calculus states the conditions under which we can switch the order of integration in double integrals. It turns out that we can similarly switch the order of double ends and coends. The Fubini rule for ends works for functors of the form $P\colon \mathcal{C} \quad \mathcal{C}^{op} \quad \mathcal{D} \quad \mathcal{D}^{op} \to \mathcal{E}$. The following expressions, as long as they exist, are isomorphic:

$$\int_{c\colon \mathcal{C}} \int_{d\colon \mathcal{D}} P\langle c,c\rangle\langle d,d\rangle \text{ ö } \int_{d\colon \mathcal{D}} \int_{c\colon \mathcal{C}} P\langle c,c\rangle\langle d,d\rangle \text{ ö } \int_{\langle c,d\rangle\colon \mathcal{C} \ \mathcal{D}} P\langle c,c\rangle\langle d,d\rangle$$

In the last end, the functor P is reinterpreted as $P\colon (\mathcal{C} \quad \mathcal{D})^{op} \quad (\mathcal{C} \quad \mathcal{D}) \to \mathcal{E}$

The analogous rule works for coends as well.

17.6 Ninja Yoneda Lemma

Having expressed the set of natural transformations as an end, we can now rewrite the Yoneda lemma. This is the original formulation:

$$[\mathcal{C}, \mathrm{Set}](\mathcal{C}(a,*), F) \text{ ö } Fa$$

Here, F is a (covariant) functor from $\mathcal{C}$ to Set (a co-presheaf), and so is the hom-functor $\mathcal{C}(a, *)$. Expressing the set of natural transformations as an end we get:

$$\int_{x\colon \mathcal{C}} \mathrm{Set}(\mathcal{C}(a,x), Fx) \text{ ö } Fa$$

Similarly, we have the Yoneda lemma for a contravariant functor (a presheaf) G:

$$\int_{x\colon \mathcal{C}} \mathrm{Set}(\mathcal{C}(x,a), Gx) \text{ ö } Ga$$

These versions of the Yoneda lemma, expressed in terms of ends, are often half-jokingly called ninja-Yoneda lemmas. The fact that the "integration variable" is explicit makes them somewhat easier to use in complex formulas.

There is also a dual set of ninja co-Yoneda lemmas that use coends instead. For a covariant functor, we have:

$$\int^{x\colon \mathcal{C}} \mathcal{C}(x,a)\quad Fx \text{ ö } Fa$$

and for the contravariant one we have:

$$\int^{x\colon \mathcal{C}} \mathcal{C}(a,x)\quad Gx \text{ ö } Ga$$

Physicists might notice the similarity of these formulas to integrals involving the Dirac delta function—strictly speaking, a distribution. This is why profunctors are sometimes called distributors, following the adage that "distributors are to functors as distributions are to functions." Engineers might notice the similarity of the hom-functor to the impulse function.

This intuition is often expressed by saying that we can perform the "integration over x" in this formula, resulting in replacing x with a in the integrand Gx.

If $\mathcal{C}$ is a discrete category, the coend reduces to the sum (coproduct), and the hom-functor reduces to the unit matrix (the Kronecker delta). The co-Yoneda lemma then becomes:

$$\sum_j \delta_i^j v_j = v_i$$

In fact, a lot of linear algebra translates directly to the theory of Set-valued functors. You may often view such functors as vectors in a vector space, in which hom-functors form a basis. Profunctors become matrices and coends are used to multiply such matrices, calculate their traces, or multiply vectors by matrices.

Yet another name for profunctors, especially in Australia, is "bimodules." This is because the lifting of morphisms by a profunctor is somewhat similar to the left and right actions on sets.

We'll now proceed with the proof of the co-Yoneda lemma, which is quite instructive, as it uses a few common tricks. Most importantly, we rely on the corollary of the Yoneda lemma, which says that, if all the mappings out from two objects to an arbitrary object are isomorphic, then the two objects are themselves isomorphic. In our case, we'll consider the mappings-out to an arbitrary set S and show that:

$$\mathrm{Set}\left(\int^{x\colon C} C(x,a) \quad Fx, S\right) \text{ ö } \mathrm{Set}(Fa, S)$$

Using the co-continuity of the hom-functor, we can pull out the integral sign, replacing the coend with an end:

$$\int_{x\colon C} \mathrm{Set}\,(C(x,a) \quad Fx, S)$$

Since the category of sets is cartesian closed, we can curry the product:

$$\int_{x\colon C} \mathrm{Set}\,\big(C(x,a), S^{Fx}\big)$$

We can now use the Yoneda lemma to "integrate over x." The result is S^{Fa}. Finally, in Set, the exponential object is isomorphic to the hom-set:

$$S^{Fa} \text{ ö } \mathrm{Set}(Fa, S)$$

Since S was arbitrary, we conclude that:

$$\int^{x\colon C} C(x,a) \quad Fx \text{ ö } Fa$$

Exercise 17.6.1. *Prove the contravariant version of the co-Yoneda lemma.*

Yoneda lemma in Haskell

We've already seen the Yoneda lemma implemented in Haskell. We can now rewrite it in terms of an end. We start by defining a profunctor that will go under the end. Its type constructor takes a functor and a type and generates a profunctor that's contravariant in and covariant in :

The Yoneda lemma establishes the isomorphism between the end over this profunctor and the type obtained by acting with the functor on . This isomorphism is witnessed by a pair of functions:

Similarly, the co-Yoneda lemma uses a coend over the following profunctor:

The isomorphism is witnessed by a pair of functions. The first one says that if you have a function and a functorful of then you can make a functorful of using the :

You can do it without knowing anything about the existential type .

The second says that if you have a functorful of , you can create a coend by injecting it (together with the identity function) into the existential type:

17.7 Day Convolution

Electrical engineers are familiar with the idea of convolution. We can convolve two streams by shifting one of them and summing its product with the other one:

$$(f \star g)(x) = \int_{*\text{Ø}}^{\text{Ø}} f(y)g(x * y)dy$$

This formula can be translated almost verbatim to category theory. We can start by replacing the integral with a coend. The problem is, we don't know how to subtract objects. We do however know how to add them, in a co-cartesian category.

Notice that the sum of the arguments to the two functions is equal to x. We could enforce this condition by introducing the Dirac delta function or the "impulse function," $\delta(a + b * x)$. In category theory we use the hom-functor $\mathcal{C}(a + b, x)$ to do the same. Thus we can define a convolution of two Set-valued functors:

$$(F \star G)x = \int^{a,b} \mathcal{C}(a + b, x) \quad Fa \quad Gb$$

Informally, if we could define subtraction as the right adjoint to coproduct, we'd write:

$$\int^{a,b} \mathcal{C}(a + b, x) \quad Fa \quad Gb \, \text{ö} \int^{a,b} \mathcal{C}(a, b * x) \quad Fa \quad Gb \, \text{ö} \int^{b} F(b * x) \quad Gb$$

There is nothing special about coproduct so, in general, Day convolution is defined in any monoidal category with a tensor product:

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \otimes b, x) \quad Fa \quad Gb$$

In fact, Day convolution for a monoidal category $(\mathcal{C}, \otimes, I)$ endows the category of co-presheaves $[\mathcal{C}, \mathrm{Set}]$ with a monoidal structure. Simply said, if you can multiply objects in $\mathcal{C}$, you can multiply set-valued functors on $\mathcal{C}$.

It's easy to check that Day convolution is associative (up to isomorphism) and that $\mathcal{C}(I, *)$ serves as the unit object. For instance, we have:

$$(\mathcal{C}(I, *) \star G)x = \int^{a,b} \mathcal{C}(a \otimes b, x) \quad \mathcal{C}(I, a) \quad Gb \, \text{ö} \int^{b} \mathcal{C}(I \otimes b, x) \quad Gb \, \text{ö} \, Gx$$

So the unit of Day convolution is the Yoneda functor taken at monoidal unit, which lends itself to the anagrammatic slogan, "ONE of DAY is the YONEDA of ONE."

If the tensor product is symmetric, then the corresponding Day convolution is also symmetric (up to isomorphism).

In the special case of a cartesian closed category, we can use the currying adjunction to simplify the formula:

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \quad b, x) \quad Fa \quad Gb \text{ ö} \int^{a,b} \mathcal{C}(a, x^b) \quad Fa \quad Gb \text{ ö} \int^{b} F(x^b) \quad Gb$$

In Haskell, the product-based Day convolution can be defined using an existential type:

If we think of functors as containers of values, Day convolution tells us how to combine two different containers into one—given a function that combines two different values into one.

Exercise 17.7.1. *Define the instance for .*

Exercise 17.7.2. *Implement the associator for .*

Applicative functors as monoids

We've seen before the definition of applicative functors as lax monoidal functors. It turns out that, just like monads, applicative functors can also be defined as monoids.

Recall that a monoid is an object in a monoidal category. The category we're interested in is the co-presheaf category $[\mathcal{C}, \mathbf{Set}]$. If $\mathcal{C}$ is cartesian, then the co-presheaf category is monoidal with respect to Day convolution, with the unit object $\mathcal{C}(I, *)$. A monoid in this category is a functor F equipped with two natural transformations that serve as unit and multiplication:

$$\eta \colon \mathcal{C}(I, *) \to F$$

$$\mu \colon F \star F \to F$$

In particular, in a cartesian closed category where the unit is the terminal object, $\mathcal{C}(1, a)$ is isomorphic to a, and the component of the unit at a is:

$$\eta_a \colon a \to Fa$$

You may recognize this function as in the definition of .

Let's consider the set of natural transformations from which μ is taken. We'll write it as an end:

$$\mu \text{ Ë} \int_x \mathbf{Set}\Big((F \star F)x, Fx\Big)$$

Pluggin in the definition of Day convolution, we get:

$$\int_x \mathbf{Set}\Big(\int^{a,b} \mathcal{C}(a \quad b, x) \quad Fa \quad Fb, Fx\Big)$$

We can pull out the coend using co-continuity of the hom-functor:

$$\int_{x,a,b} \text{Set}\Big(\mathcal{C}(a \quad b, x) \quad Fa \quad Fb, Fx\Big)$$

We can then use the currying adjunction in Set to obtain:

$$\int_{x,a,b} \text{Set}\Big(\mathcal{C}(a \quad b, x), \text{Set}(Fa \quad Fb, Fx)\Big)$$

Finally, we apply the Yoneda lemma to perform the integration over x:

$$\int_{a,b} \text{Set}\Big(Fa \quad Fb, F(a \quad b)\Big)$$

The result is the set of natural transformations from which to select the second part of the lax monoidal functor:

Free Applicatives

We have just learned that applicative functors are monoids in the monoidal category:

$$([\mathcal{C}, \text{Set}], \mathcal{C}(I, *), \star)$$

It's only natural to ask what a free monoid in that category is.

Just like we did with free monads, we'll construct a free applicative as the initial algebra, or the least fixed point of the list functor. Recall that the list functor was defined as:

$$\Phi_a x = 1 + a \otimes x$$

In our case it becomes:

$$\Phi_F G = \mathcal{C}(I, *) + F \star G$$

Its fixed point is given by the recursive formula:

$$A_F \text{ ö } \mathcal{C}(I, *) + F \star A_F$$

When translating this to Haskell, we observe that functions from the unit are isomorphic to elements of .

Corresponding to the two addends in the definition of A_F, we get two constructors:

I have inlined the definition of Day convolution:

The easiest way to show that is an applicative functor is to go through :

Since is a generalization of a list, the instance for free applicative generalizes the idea of list concatenation. We do the pattern matching on the first list, resulting in two cases.

In the first case, instead of an empty list we have . Prepending it to the second argument doesn't change the length of the list, but it modifies the type of the values stored in it. It pairs each of them with :

The second case is a "list" whose head is a functorful of 's, and the tail is of the type . The two are glued using a function .

To produce the result, we concatenate the two tails using the recursive call to and prepend to it. To glue this head to the new tail we have to provide a function that re-associates the pairs:

The complete instance is thus:

Once we have the instance, it's straightforward to produce the instance:

Exercise 17.7.3. *Define the instance for the free applicative.*

17.8 The Bicategory of Profunctors

Since we know how to compose profunctors using coends, the question arises: is there a category in which they serve as morphisms? The answer is yes, as long as we relax the rules a bit. The problem is that the categorical laws for profunctor composition are nor satisfied "on the nose," but only up to isomorphism.

For instance, we can try to show associativity of profunctor composition. We start with:

$$((P \diamond Q) \diamond R)\langle s,t\rangle = \int^b \left(\int^a P\langle s,a\rangle \;\; Q\langle a,b\rangle\right) \;\; R\langle b,t\rangle$$

and, after a few transformations, arrive at:

$$(P \diamond (Q \diamond R))\langle s,t\rangle = \int^a P\langle s,a\rangle \;\; \left(\int^b Q\langle a,b\rangle \;\; R\langle b,t\rangle\right)$$

We use the associativity of the product and the fact that we can switch the order of coends using the Fubini theorem. Both are true only up to isomorphism. We don't get associativity "on the nose."

The identity profunctor turns out to be the hom-functor, which can be written symbolically as $\mathcal{C}(*,=)$, with placeholders for both arguments. For instance:

$$(\mathcal{C}(*,=) \diamond P)\langle s,t\rangle = \int^a \mathcal{C}(s,a) \quad P\langle a,t\rangle \text{ ö } P\langle s,t\rangle$$

This is the consequence of the (contravariant) ninja co-Yoneda lemma, which is also an isomorphism, not an equality.

A category in which categorical laws are satisfied up to isomorphism is called a bicategory. Notice that such a category must be equipped with 2-cells—morphisms between morphisms, which we've already seen in the definition of a 2-category. We need those in order to be able to define isomorphisms between 1-cells.

A bicategory $\mathbf{Prof}$ has (small) categories as objects, profunctors as 1-cells, and natural transformations as 2-cells.

Since profunctors are functors $\mathcal{C}^{op} \quad \mathcal{D} \to \mathbf{Set}$, the standard definition of natural transformations between them applies. It's a family of functions parameterized by objects of $\mathcal{C}^{op} \quad \mathcal{D}$, which are themselves pairs of objects.

The naturality condition for a transformation $\alpha_{\langle a,b\rangle}$ between two profunctors P and Q takes the form:

$$\begin{tikzcd}
& P\langle a,b\rangle \arrow[dl, "\alpha_{\langle a,b\rangle}"'] \arrow[dr, "P\langle f,g\rangle"] & \\
Q\langle a,b\rangle \arrow[dr, "Q\langle f,g\rangle"'] & & P\langle s,t\rangle \arrow[dl, "\alpha_{\langle s,t\rangle}"] \\
& Q\langle s,t\rangle &
\end{tikzcd}$$

for every pair of arrows:

$$\langle f\colon s \to a, g\colon b \to t\rangle$$

Monads in a bicategory

We've seen before that categories, functors, and natural transformations form a 2-category $\mathbf{Cat}$. Let's focus on one object, a category $\mathcal{C}$, that is a 0-cell in $\mathbf{Cat}$. The 1-cells that start and end at this object form a regular category, in this case it's the functor category $[\mathcal{C},\mathcal{C}]$. The objects in this category are endo-1-cells of the outer 2-category $\mathbf{Cat}$. The arrows between them are the 2-cells of the outer 2-category.

This endo-one-cell category is automatically equipped with monoidal structure. We simply define the tensor product as the composition of 1-cells—all 1-cells with the same source and target compose. The monoidal unit object is the identity 1-cell, I. In $[\mathcal{C},\mathcal{C}]$ this product is the composition of endofunctors and the unit is the identity functor.

If we now focus our attention on just one endo-1-cell F, we can "square" it, that is use the monoidal product to multiply it by itself. In other words, use the 1-cell composition to create $F \circ F$. We say that F is a monad if we can find 2-cells:

$$\mu \colon F \circ F \to F$$

$$\eta \colon I \to F$$

that behave like multiplication and unit, that is they make the associativity and unit diagrams commute.

$$\begin{tikzcd}
\mathcal{C} \arrow[loop, out=150, in=30, distance=8em, "F \circ F"'] \arrow[loop, out=140, in=40, distance=5em, "F"'] \arrow[loop, out=130, in=50, distance=2.5em, "I"']
\end{tikzcd}$$

In fact a monad can be defined in an arbitrary bicategory, not just the 2-category $\mathbf{Cat}$.

Prearrows as monads in $\mathbf{Prof}$

Since $\mathbf{Prof}$ is a bicategory, we can define a monad in it. It is an endo-profunctor (a 1-cell):

$$P \colon \mathcal{C}^{op} \quad \mathcal{C} \to \mathbf{Set}$$

equipped with two natural transformations (2-cells):

$$\mu \colon P \diamond P \to P$$

$$\eta \colon \mathcal{C}(*, =) \to P$$

that satisfy the associativity and unit conditions.

Let's look at these natural transformations as elements of ends. For instance:

$$\mu \text{ Ë } \int_{\langle a,b \rangle} \mathbf{Set}\Big(\int^x P\langle a,x \rangle \quad P\langle x,b \rangle, P\langle a,b \rangle \Big)$$

By co-continuity, this is equivalent to:

$$\int_{\langle a,b \rangle, x} \mathbf{Set}\Big(P\langle a,x \rangle \quad P\langle x,b \rangle, P\langle a,b \rangle \Big)$$

Similarly, the unit natural transformation is:

$$\eta \text{ Ë } \int_{\langle a,b \rangle} \mathbf{Set}(\mathcal{C}(a,b), P\langle a,b \rangle)$$

In Haskell, such profunctor monads are called pre-arrows:

An is a that is also a Tambara module. We'll talk about Tambara modules in the next chapter.

17.9 Existential Lens

The first rule of category-theory club is that you don't talk about the internals of objects.

The second rule of category-theory club is that, if you have to talk about the internals of objects, use arrows only.

Existential lens in Haskell

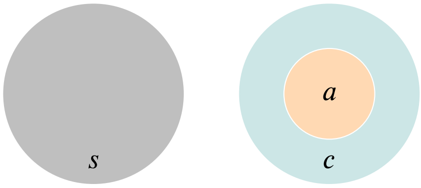
What does it mean for an object to be a composite—to have parts? At the very minimum, you should be able to retrieve a part of such an object. Even better if you can replace that part with a new one. This pretty much defines a lens:

Here, extracts the part from the whole , and replaces that part with a new . Lens laws help to reinforce this picture. And it's all done in terms of arrows.

Another way of describing a composite object is to say that it can be split into a focus and a residue. The trick is that, although we want to know what type the focus is, we don't care about the type of the residue. All we need to know about the residue is that it can be combined with the focus to recreate the whole object.

In Haskell, we would express this idea using an existential type:

This tells us that there exists some unspecified type , such that can be split into, and reconstructed from, a product .



The / version of the lens can be derived from this existential form.

Notice that we don't need to know anything about the type of the residue. We take advantage of the fact that the existential lens contains both the producer and the consumer of and we're just mediating between the two.

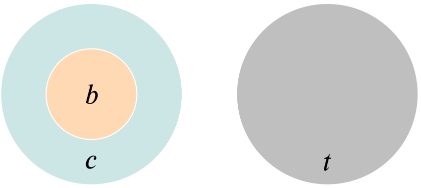
It's impossible to extract a "naked" residue, as witnessed by the fact that the following code doesn't compile:

Existential lens in category theory

We can easily translate the new definition of the lens to category theory by expressing the existential type as a coend:

$$\int^c \mathcal{C}(s, c \quad a) \quad \mathcal{C}(c \quad a, s)$$

In fact, we can generalize it to a type-changing lens, in which the focus a can be replaced with a new focus of a different type b. Replacing a with b will produce a new composite object t:



The lens is now parameterized by two pairs of objects: $\langle s, t\rangle$ for the outer ones, and $\langle a, b\rangle$ for the inner ones. The existential residue c remains hidden:

$$\mathcal{L}\langle s, t\rangle\langle a, b\rangle = \int^c \mathcal{C}(s, c \quad a) \quad \mathcal{C}(c \quad b, t)$$

The product under the coend is the diagonal part of the profunctor that is covariant in y and contravariant in x:

$$\mathcal{C}(s, y \quad a) \quad \mathcal{C}(x \quad b, t)$$

Exercise 17.9.1. *Show that:*

$$\mathcal{C}(s, y \quad a) \quad \mathcal{C}(x \quad b, t)$$

is a profunctor in $\langle x, y\rangle$.

Type-changing lens in Haskell

In Haskell, we can define a type-changing lens as the following existential type:

As before, we can use the existential lens to get and set the focus:

The two functions, and are often called the *forward* and the *backward* pass. The forward pass can be used to extract the focus . The backward pass provides the answer to the question: If we wanted the result of the forward pass to be some other , what should we pass to it?

And sometimes we're just asking: What change should we make to the input if we wanted to change the focus by . The latter point of view is especially useful when using lenses to describe neural networks.

The simplest example of a lens acts on a product. It can extract or replace one component of the product, treating the other as the residue. In Haskell, we'd implement it as:

Here, the type of the whole is the product . When we replace with we end up with the target type . Since the source and the target are already products, the two functions in the definition of the existential lens are just identities.

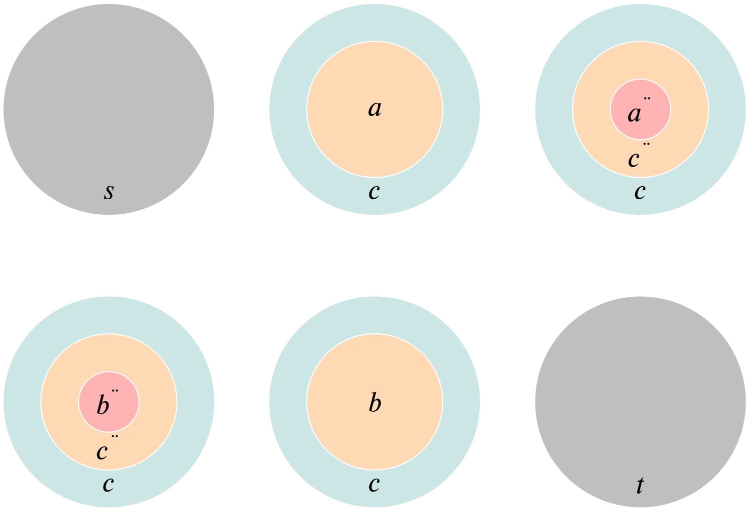
Lens composition

The main advantage of using lenses is that they compose. A composition of two lenses lets us zoom in on a subcomponent of a component.

Suppose that we start with a lens that lets us access the focus and change it to . This focus is part of a whole described by the source and the target .

We also have the inner lens that can access the focus of inside the whole of , and replace it with to give us a new .

We can now construct a composite lens that can access and inside of and . The trick is to realize that we can take, as the new residue, a product of the two residues:



The left mapping in the new lens is given by the following composite:

$$s \xrightarrow{l_1} (c, a) \xrightarrow{(id, l_2)} (c, (c'', a'')) \xrightarrow{assoc''} ((c, c''), a'')$$

and the right mapping is given by:

$$((c, c''), b'') \xrightarrow{assoc} (c, (c'', b'')) \xrightarrow{(id, r_2)} (c, b) \xrightarrow{r_1} t$$

We have used the associativity and functoriality of the product:

As an example, let's compose two product lenses:

and apply it to a nested product:

Our composite lens lets us not only retrieve the innermost component:

but also replace it with a value of a different type (here,):

Category of lenses

Since lenses can be composed, you might be wondering if there is a category in which lenses are hom-sets.

Indeed, there is a category Lens whose objects are pairs of objects in $\mathcal{C}$, and arrows from $\langle s, t\rangle$ to $\langle a, b\rangle$ are elements of $\mathcal{L}\langle s, t\rangle\langle a, b\rangle$.

The formula for the composition of existential lenses is too complicated to be useful in practice. In the next chapter we'll see an alternative representation of lenses using Tambara modules, in which composition is just a composition of functions.

17.10 Lenses and Fibrations

There is an alternative view of lenses using the language of fiber bundles. A projection p that defines a fibration can be seen as "decomposing" the bundle E into fibers.

In this view, p plays the role of :

$$p \colon E \to B$$

The base B represents the type of the focus and E represents the type of the composite from which that focus can be extracted.

The other part of the lens, , is a mapping:

$$q \colon E \quad B \to E$$

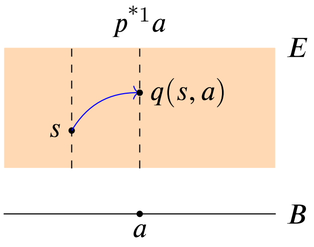
Let's see how we can interpret it using fibrations.

Transport law

We interpret q as "transporting" an element of the bundle E to a new fiber. The new fiber is specified by an element of B.

This property of the transport is expressed by the get/set lens law, or the *transport* law, that says that "you get what you set":

We say that $q(s, a)$ transports s to a new fiber over a:



We can rewrite this law in terms of p and q:

$$p{\circ}q = \pi_2$$

where π_2 is the second projection from the product.

Equivalently, we can represent it as a commuting diagram:

$$\begin{tikzcd}
E \quad B \arrow[rd, "q"] \arrow[dd, "\varepsilon\ id"'] & \\
& E \arrow[ld, "p"] \\
B &
\end{tikzcd}$$

Here, instead of using the projection π_2, I used a comonoidal counit ε:

$$\varepsilon\colon\ E \to 1$$

followed by the unit law for the product. Using a comonoid makes it easier to generalize this construction to a tensor product in a monoidal category.

Identity law

Here's the set/get law or the *identity* law. It says that "nothing changes if you set what you get":

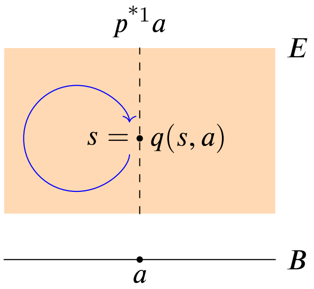
We can write it in terms of a comonoidal comultiplication:

$$\delta\colon\ E \to E \quad E$$

The set/get law requires the following composite to be an identity:

$$E \xrightarrow{\delta} E \quad E \xrightarrow{id\ \ p} E \quad B \xrightarrow{q} E$$

Here's the illustration of this law in a bundle:



Composition law

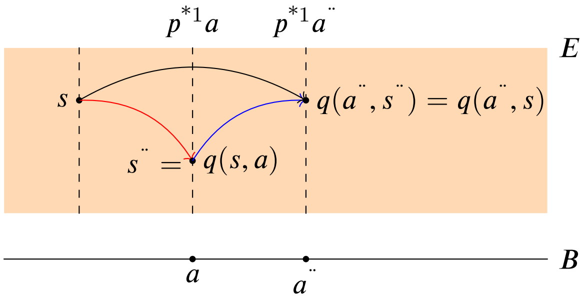
Finally, here's the set/set law, or the *composition* law. It says that "the last set wins":

and the corresponding commuting diagram:

$$\begin{tikzcd}
E\ \ B\ \ B \arrow[d, "id\ \varepsilon\ id"'] \arrow[drr, "q\ id"] & & \\
E\ \ B \arrow[d, "q"'] & & E\ \ B \arrow[dll, "q"] \\
E & &
\end{tikzcd}$$

Again, to get rid of the middle B, I used the counit rather than a projection from the product.

This is what the set/set law looks like in a bundle:

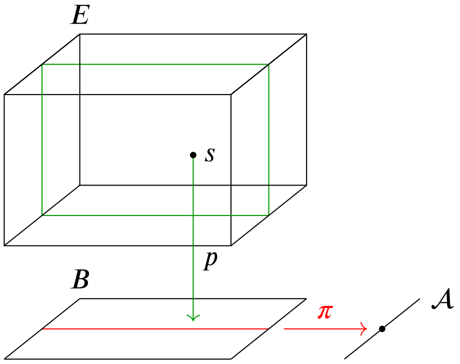


Type-changing lens

A type-changing lens generalizes transport to act between bundles. We have to define a whole family of bundles. We start with a category $\mathcal{A}$ whose objects define the types that we will use for the foci of our lens.

We construct the set B as the combined set of all elements of all focus types. B is fibrated over $\mathcal{A}$—the projection π sending an element of B to its corresponding type. You may think of B as the set of objects of the coslice category $1/\mathcal{A}$.

The bundle of bundles E is a set that's fibered over B with the projection p. Since B itself is fibered over $\mathcal{A}$, E is transitively fibered over $\mathcal{A}$, with the composite projection $\pi \circ p$. It's this coarser fibration that splits E into a family of bundles. Each of these bundles corresponds to a different type of the composite for a given focus type. A type-changing lens will move between these bundles.



The projection p takes an element s Ë E and produces an element b Ë B whose type is given by πb. This is the generalization of .

The transport q, which corresponds to , takes an element s Ë E and an element b Ë B and produces a new element t Ë E. The important observation is that s and t may belong to different sub-bundles of E.

The transport satisfies the following laws:

The get/set law (transport):

$$p(q(b,s)) = b$$

The set/get law (identity):

$$q(p(s),s) = s$$

The set/set law (composition):

$$q(c, q(b,s)) = q(c,s)$$

17.11 Important Formulas

This is a handy (co-)end calculus cheat-sheet.

- Continuity of the hom-functor:

$$\mathcal{D}\left(d, \int_a P\langle a,a\rangle\right) \text{ ö } \int_a \mathcal{D}\left(d, P\langle a,a\rangle\right)$$

- Co-continuity of the hom-functor:

$$\mathcal{D}\left(\int^a P\langle a,a\rangle, d\right) \text{ ö } \int_a \mathcal{D}\left(P\langle a,a\rangle, d\right)$$

- Ninja Yoneda:

$$\int_x \mathrm{Set}(\mathcal{C}(a,x), Fx) \text{ ö } Fa$$

- Ninja co-Yoneda:

$$\int^x \mathcal{C}(x,a) \quad Fx \text{ ö } Fa$$

- Ninja Yoneda for contravariant functors (presheaves):

$$\int_x \mathrm{Set}(\mathcal{C}(x,a), Gx) \text{ ö } Ga$$

- Ninja co-Yoneda for contravariant functors:

$$\int^x \mathcal{C}(a,x) \quad Gx \text{ ö } Ga$$

- Day convolution:

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \otimes b, x) \quad Fa \quad Gb$$

Chapter 18

Tambara Modules

It's not often that an obscure corner of category theory gains sudden prominence in programming. Tambara modules got a new lease on life in their application to profunctor optics. They provide a clever solution to the problem of composing optics. We've seen that, in the case of lenses, the getters compose nicely using function composition, but the composition of setters involves some shenanigans. The existential representation doesn't help much. The profunctor representation, on the other hand, makes composition a snap.

The situation is somewhat analogous to the problem of composing geometric transformations in graphics programming. For instance, if you try to compose two rotations around two different axes, the formula for the new axis and the angle is quite complicated. But if you represent rotations as matrices, you can use matrix multiplication; or, if you represent them as quaternions, you can use quaternion multiplication. Profunctor representation lets you compose optics using straightforward function composition.

18.1 Tannakian Reconstruction

Monoids and their Representations

The theory or representations is a science in itself. Here, we'll approach it from the categorical perspective. Instead of groups, we'll consider monoids. A monoid can be defined as a special object in a monoidal category, but it can also be thought of as a single-object category $\mathcal{M}$. If we call its object $<$, the hom-set $\mathcal{M}(<, <)$ contains all the information we need.

What we called "product" in a monoid is replaced by the composition of morphisms. By the laws of a category, it's associative and unital—the identity morphism serving as the monoidal unit.

In this sense, every single-object category is automatically a monoid and all monoids can be made into single-object categories.

For instance, a monoid of the whole numbers with addition can be thought of as a category with a single abstract object $<$ and a morphism for every number. To compose two such morphisms, you add their numbers, as in the example below:

$$\begin{tikzcd}
< \arrow[r, bend left, "2"] \arrow[rr, bend right, "5"'] & < \arrow[r, bend left, "3"] & <
\end{tikzcd}$$

The morphism corresponding to zero is automatically the identity morphism.

We can *represent* a monoid as transformations of a set. Such a representation is given by a functor $F\colon \mathcal{M} \to \mathrm{Set}$. It maps the single object $<$ to some set S, and it maps the hom-set $\mathcal{M}(<,<)$ to a set of functions $S \to S$. By functor laws, it maps identity to identity and composition to composition, so it preserves the structure of the monoid.

If the functor is fully faithful, its image contains exactly the same information as the monoid and nothing more. But, in general, functors cheat. The hom-set $\mathrm{Set}(S,S)$ may contain some other functions that are not in the image of $\mathcal{M}(<,<)$; and multiple morphisms in $\mathcal{M}$ may be mapped to a single function.

In the extreme, the whole hom-set $\mathcal{M}(<,<)$ may be mapped to the identity morphism id_S. So, just by looking at the set S—the image of $<$ under the functor F—we cannot dream of reconstructing the original monoid.

Not all is lost, though, if we are allowed to look at all the representations of a given monoid simultaneously. Such representations form a category—the functor category $[\mathcal{M}, \mathrm{Set}]$, a.k.a. the co-presheaf category over $\mathcal{M}$. Arrows in this category are natural transformations.

Since the source category $\mathcal{M}$ contains only one object, naturality conditions take a particularly simple form. A natural transformation $\alpha\colon F \to G$ has only one component, a function $\alpha\colon F < \to G <$. Given a morphism $m\colon < \to <$, the naturality square reads:

$$\begin{tikzcd}
F < \arrow[r, "\alpha"] \arrow[d, "Fm"] & G < \arrow[d, "Gm"] \\
F < \arrow[r, "\alpha"] & G <
\end{tikzcd}$$

It's a relationship between three functions acting on two sets:

$$\begin{tikzcd}
F < \arrow[loop above, "Fm"] \arrow[rr, bend right, "\alpha"'] & & G < \arrow[loop above, "Gm"]
\end{tikzcd}$$

The naturality condition tells us that:

$$Gm \circ \alpha = \alpha \circ Fm$$

In other words, if you pick any element x in the set $F\ <$, you can map it to $G <$ using α and then apply the transformation Gm corresponding to m; or you can first apply the transformation Fm and then map the result using α. The result must be the same.

Such functions are called *equivariant*. We often call Fm the *action* of m on the set $F\ <$. An equivariant function connects an action on one set to its corresponding action on another set using either pre-composition or post-composition.

Tannakian reconstruction of a monoid

How much information do we need to reconstruct a monoid from its representations? Just looking at the sets is definitely not enough, since any monoid can be represented on any set. But if we include structure-preserving functions between those sets, we might have a chance.

Given a functor $F\colon \mathcal{M} \to \mathrm{Set}$, consider the set of *all* functions $F\ <\to F\ <$, that is the hom-set $\mathrm{Set}(F\ <, F\ <),$. At least some of these functions are the actions of the monoid. These are the functions of the form Fm, where m is an arrow in $\mathcal{M}$. Keep in mind, though, that there may be many more functions in that hom-set that don't correspond to actions.

Now let's look at another set $G\ <$, which is the image of some other functor G. In its hom-set $\mathrm{Set}(G\ <, G\ <)$ we'll find, among others, the corresponding actions of the form Gm. An equivariant function, that is a natural transformation in $[\mathcal{M}, \mathrm{Set}]$, will connect any two related action Fm to Gm.

Now imagine creating a gigantic tuple by taking one function from each of the sets $\mathrm{Set}(F\ <, F\ <)$, for all functors $F\colon \mathcal{M} \to \mathrm{Set}$. We are interested only in tuples whose elements are connected. Here's what I mean: If we pick g Ë $\mathrm{Set}(G\ <, G\ <)$ and h Ë $\mathrm{Set}(H\ <, H\ <)$ and there is a natural transformation (equivariant function) α between the two functors G and H, we want the two functions to be related:

$$\alpha \circ g = h \circ \alpha$$

or, pictorially:

$$\begin{tikzcd}
G < \arrow[loop above, "g"] \arrow[r, bend right, "\alpha"'] & H < \arrow[loop above, "h"]
\end{tikzcd}$$

Notice that this correlation also works on pairs g and h which are not of the form $g = Gm$ and $h = Hm$.

Such tuples are exactly the elements of the end:

$$\int_F \mathrm{Set}(F\ <, F\ <)$$

whose wedge condition provides the constraints we are looking for.

$$\begin{tikzcd}
& \int_F \mathrm{Set}(F <, F <) \arrow[ld, "\pi_G"'] \arrow[rd, "\pi_H"] & \\
\mathrm{Set}(G <, G <) \arrow[rd, "\alpha\circ *"'] & & \mathrm{Set}(H <, H <) \arrow[ld, "*\circ\alpha"] \\
& \mathrm{Set}(G <, H <) &
\end{tikzcd}$$

Here α is a morphism in the functor category $[\mathcal{M}, \mathrm{Set}]$:

$$\alpha \colon G \to H$$

This natural transformation has only one component, which we'll also call α. It's an equivariant function between the two representations.

Here are some details. The profunctor under the end is given by:

$$P\langle G, H\rangle = \mathrm{Set}(G <, H <)$$

It's a functor of the form:

$$P \colon [\mathcal{M}, \mathrm{Set}]^{op} \quad [\mathcal{M}, \mathrm{Set}] \to \mathrm{Set}$$

Consider its action on pairs of morphisms in $[\mathcal{M}, \mathrm{Set}]$. Given a pair of natural transformations:

$$\alpha \colon G^{\text{¨}} \to G$$
$$\beta \colon H \to H^{\text{¨}}$$

their lifting is a function:

$$P\langle \alpha, \beta\rangle \colon P\langle G, H\rangle \to P\langle G^{\text{¨}}, H^{\text{¨}}\rangle$$

Substituting our definition of P, we have:

$$P\langle \alpha, \beta\rangle \colon \mathrm{Set}(G <, H <) \to \mathrm{Set}(G^{\text{¨}} <, H^{\text{¨}} <)$$

We get this function by pre-composing with α and post-composing with β (these functions are the only components of the two natural transformations α and β):

$$P\langle \alpha, \beta\rangle = \beta\circ * \circ\alpha$$

That is, given a function f Ë $\mathrm{Set}(G <, H <)$, we produce a function $\beta\circ f\circ\alpha$ Ë $\mathrm{Set}(G^{\text{¨}} <, H^{\text{¨}} <)$.

In the wedge condition, if we pick g to be the element of $\mathrm{Set}(G <, G <)$ and h to be the element of $\mathrm{Set}(H <, H <)$, we reproduce our condition:

$$\alpha\circ g = h\circ\alpha$$

The Tannakian reconstruction theorem, in this case, tells us that:

$$\int_F \mathrm{Set}(F <, F <) \text{ ö } \mathcal{M}(<, <)$$

In other words, we can recover the monoid from its representations. We'll see the proof of this theorem in the context of a more general statement.

Cayley's theorem

In group theory, Cayley's theorem states that every group is isomorphic to a (subgroup of the) group of permutations. A *group* is just a monoid in which every element has an inverse. Permutations are bijective functions that map a set to itself. They *permute* the elements of a set.

In category theory, Cayley's theorem is practically built into the definition of a monoid and its representations.

The connection between the single-object interpretation and the more traditional set-of-elements interpretation of a monoid is easy to establish. We do this by constructing the functor $F\colon \mathcal{M} \to \mathrm{Set}$ that maps $<$ to the special set S that is equal to the hom-set: $S = \mathcal{M}(<, <)$. Elements of this set are identified with morphisms in $\mathcal{M}$. We define the action of F on morphisms as post-composition:

$$(Fm)n = m \circ n$$

Here *m* is a morphism in $\mathcal{M}$ and *n* is an element of S, which happens to also be a morphism in $\mathcal{M}$.

We can take this particular representation as an alternative definition of a monoid in the monoidal category Set. All we need is to implement unit and multiplication:

$$\eta\colon 1 \to S$$
$$\mu\colon S \quad S \to S$$

The unit picks the element of S that corresponds to $id_<$ in $\mathcal{M}(<, <)$. Multiplication of two elements *m* and *n* is given by the element that corresponds to $m \circ n$.

At the same time we can look at S as an image of $F\colon \mathcal{M} \to \mathrm{Set}$, in which case it's the functions $S \to S$ that form a representation of the monoid. This is the essence of the Cayley's theorem: Every monoid can be represented by a set of endo-functions.

In programming, the best example of applying the Cayley's theorem is in the efficient implementation of list reversal. Recall the naive recursive implementation of reversal:

It splits the list into the head and the tail, reverses the tail, and appends a singleton made out of the head to the result. The problem is that every append has to traverse the growing list resulting in $O(N^2)$ performance.

Remember, however, that a list is a (free) monoid:

We can use Cayley's theorem to represent this monoid as functions on lists:

To represent a list, we turn it into a function. It's a function (a closure) that prepends this list to its argument :

This representation is called a *difference list*.

To turn a function back to a list, it's enough to apply it to an empty list:

It's easy to check that the representation of an empty list is an identity function, and that the representation of a concatenation of two lists is a composition of representations:

So this is exactly the Cayley representation of the list monoid.

We can now translate the reversal algorithm to produce this new representation:

and turn it back to a list:

At first sight it might seem like we haven't done much except add a layer of conversion on top of our recursive algorithm. Except that the new algorithm has $O(N)$ rather than $O(N^2)$ performance. To see that, consider reversing a simple list . The function turns this list into a composition of functions:

It does it in linear time. The function executes this composite starting with an empty list. But notice that each *prepends* its argument to the cumulative result. In particular, the final executes:

Unlike appending, prepending is a constant-time operation, so the whole algorithm takes $O(N)$ time.

Another way of looking at it is to realize that queues up the actions in the order of the elements of the list, starting at the head. But the queue of functions is executed in the first-in-first-out (FIFO) order.

Because of Haskell's laziness, list reversal using has similar performance:

This is because , before returning the result, traverses the list left-to-right accumulating functions (closures). It then executes them as necessary, in the FIFO order.

Proof of Tannakian reconstruction

Monoid reconstruction is a special case of a more general theorem in which, instead of the single-object category, we use a regular category. As in the monoid case, we'll reconstruct the hom-set, only this time it will be a regular hom-set between two objects. We'll prove the formula:

$$\int_{F \colon [\mathcal{C},\mathrm{Set}]} \mathrm{Set}(Fa, Fb) \text{ ö } \mathcal{C}(a, b)$$

The trick is to use the Yoneda lemma to represent the action of F:

$$Fa \text{ ö } [\mathcal{C}, \mathrm{Set}](\mathcal{C}(a, *), F)$$

and the same for Fb. We get:

$$\int_{F \colon [\mathcal{C},\mathrm{Set}]} \mathrm{Set}([\mathcal{C}, \mathrm{Set}](\mathcal{C}(a, *), F), [\mathcal{C}, \mathrm{Set}](\mathcal{C}(b, *), F))$$

Notice that the two sets of natural transformations here are hom-sets in $[\mathcal{C}, \mathrm{Set}]$.

Recall the corollary to the Yoneda lemma that works for any category $\mathcal{A}$:

$$[\mathcal{A}, \mathrm{Set}](\mathcal{A}(x, *), \mathcal{A}(y, *)) \text{ ö } \mathcal{A}(y, x)$$

We can write it using an end:

$$\int_{z \colon \mathcal{C}} \mathrm{Set}(\mathcal{A}(x, z), \mathcal{A}(y, z)) \text{ ö } \mathcal{A}(y, x)$$

In particular, we can replace $\mathcal{A}$ with the functor category $[\mathcal{C}, \mathrm{Set}]$. We get:

$$\int_{F \colon [\mathcal{C},\mathrm{Set}]} \mathrm{Set}([\mathcal{C}, \mathrm{Set}](\mathcal{C}(a, *), F), [\mathcal{C}, \mathrm{Set}](\mathcal{C}(b, *), F)) \text{ ö } [\mathcal{C}, \mathrm{Set}](\mathcal{C}(b, *), \mathcal{C}(a, *))$$

We can then apply the Yoneda lemma again to the right hand side and get:

$$\mathcal{C}(a, b)$$

which is exactly the sought after result.

It's important to realize how the structure of the functor category enters the end through the wedge condition. It does that through natural transformations. Every time we have a natural transformation between two functors $\alpha \colon G \to H$, the following diagram must commute:

$$\begin{tikzcd}
& \int_F \mathrm{Set}(Fa, Fb) \arrow[dl, "\pi_G"'] \arrow[dr, "\pi_H"] & \\
\mathrm{Set}(Ga, Gb) \arrow[dr, "\mathrm{Set}(id,\alpha)"'] & & \mathrm{Set}(Ha, Hb) \arrow[dl, "\mathrm{Set}(\alpha,id)"] \\
& \mathrm{Set}(Ga, Hb) &
\end{tikzcd}$$

To get some intuition about Tannakian reconstruction, you may recall that Set-valued functors have the interpretation as proof-relevant subsets. A functor $F \colon \mathcal{C} \to \mathrm{Set}$ (a co-presheaf) defines a subset of the objects of (a small category) $\mathcal{C}$. We say that an object a is in that subset only if Fa is non-empty. Each element of Fa can then be interpreted as a proof of that.

But unless the category in question is discrete, not all subsets will correspond to co-presheaves. In particular, whenever there is an arrow $f \colon a \to b$, there also is a function $Ff \colon Fa \to Fb$. According to our interpretation, such function maps every proof that a is in the subset defined by F to a proof that b is in that subset. Co-presheaves thus define proof-relevant subsets that are compatible with the structure of the category.

Let's reinterpret Tannakian reconstruction in the same spirit.

$$\int_{F \colon [\mathcal{C}, \mathrm{Set}]} \mathrm{Set}(Fa, Fb) \text{ ö } \mathcal{C}(a, b)$$

An element of the left-hand side is a proof that for every subset that is compatible with the structure of $\mathcal{C}$, if a belongs to that subset, so does b. That is only possible if there is an arrow from a to b.

Tannakian reconstruction in Haskell

We can immediately translate this result to Haskell. We replace the end by . The left hand side becomes:

and the right hand side is the function type .

We've seen polymorphic functions before: they were functions defined for all types, or sometimes for classes of types. Here we have a function that is defined for all functors. It says: give me a functorful of 's and I'll produce a functorful of 's—no matter what functor you use. The only way this can be implemented (using parametric polymorphism) is if this function has secretly captured a function of the type and is applying it using .

Indeed, one direction of the isomorphism is just that: capturing a function and ping it over the argument:

The other direction uses the Yoneda trick:

where the identity functor is defined as:

This kind of reconstruction might seem trivial and pointless. Why would anyone want to replace function type with a much more complicated type:

It's instructive, though, to think of as the precursor of all optics. It's a lens that focuses on the b part of a. It tells us that a contains enough information, in one form or another, to construct a b. It's a "getter" or an "accessor."

Obviously, functions compose. What's interesting though is that functor representations also compose, and they compose using simple function composition, as seen in this example:

Other optics don't compose so easily, but their functor (and profunctor) representations do.

Tannakian reconstruction with adjunction

The trick in generalizing the Tannakian reconstruction is to define the end over some specialized functor category $\mathcal{T}$ by first applying the forgetful functor to its functors. Let's assume that we have the free/forgetful adjunction $F \dashv U$ between two functor categories $\mathcal{T}$ and $[\mathcal{C}, \mathrm{Set}]$:

$$\mathcal{T}(FQ, P) \text{ ö } [\mathcal{C}, \mathrm{Set}](Q, UP)$$

where Q is a functor in $[\mathcal{C}, \mathrm{Set}]$ and P a functor in $\mathcal{T}$.

Our starting point for Tannakian reconstruction is the following end:

$$\int_{P\colon \mathcal{T}} \mathrm{Set}\Big((UP)a, (UP)s\Big)$$

Incidentally, the mapping $\mathcal{T} \to \mathrm{Set}$ parameterized by the object a, and given by the formula:

$$P \mapsto (UP)a$$

is sometimes called the *fiber functor*, so the end formula can be interpreted as a set of natural transformations between two fiber functors. Conceptually, a fiber functor describes an "infinitesimal neighborhood" of an object. It maps functors to sets but, more importantly, it maps natural transformations to functions. These functions probe the environment in which the object lives. In particular, natural transformations in $\mathcal{T}$ are involved in wedge conditions that define the end. (In calculus, stalks of sheaves play a very similar role.)

As we did before, we first apply the Yoneda lemma to get:

$$\int_{P\colon \mathcal{T}} \mathrm{Set}\Big([\mathcal{C}, \mathrm{Set}]\Big(\mathcal{C}(a, *), UP\Big), [\mathcal{C}, \mathrm{Set}]\Big(\mathcal{C}(s, *), UP\Big)\Big)$$

We can now use the adjunction:

$$\int_{P\colon \mathcal{T}} \mathrm{Set}\Big(\mathcal{T}\Big(F\mathcal{C}(a, *), P\Big), \mathcal{T}\Big(F\mathcal{C}(s, *), P\Big)\Big)$$

We end up with a mapping between two natural transformations in the functor category $\mathcal{T}$. We can perform the integration using the corollary to the Yoneda lemma, giving us:

$$\mathcal{T}\Big(F\mathcal{C}(s,*), F\mathcal{C}(a,*)\Big)$$

We can apply the adjunction once more:

$$\mathrm{Set}\Big(\mathcal{C}(s,*), (U\circ F)\mathcal{C}(a,*)\Big)$$

and the Yoneda lemma again:

$$\Big((U\circ F)\mathcal{C}(a,*)\Big)s$$

The final observation is that the composition $U\circ F$ of adjoint functors is a monad in the functor category. Let's call this monad Φ. The result is the following identity that will serve as the foundation for profunctor optics:

$$\int_{P\colon \mathcal{T}} \mathrm{Set}\Big((UP)a, (UP)s\Big) \text{ ö } \Big(\Phi\mathcal{C}(a,*)\Big)s$$

The right-hand side is the action of the monad $\Phi = U\circ F$ on the representable functor $\mathcal{C}(a,*)$, which is then evaluated at s.

Compare this with the earlier formula for Tannakian reconstruction, especially if we rewrite it in the following form:

$$\int_{F\colon [\mathcal{C},\mathrm{Set}]} \mathrm{Set}(Fa, Fs) \text{ ö } \mathcal{C}(a,*)s$$

Keep in mind that, in the derivation of optics, we'll replace a and s with pairs of objects $\langle a,b\rangle$ and $\langle s,t\rangle$ from $\mathcal{C}^{op}\quad\mathcal{C}$. In that case our functors will become profunctors.

18.2 Profunctor Lenses

Our goal is to find a functor representation for optics. We've seen before that, for instance, type-changing lenses can be seen as hom-sets in the Lens category. The objects in Lens are pairs of objects from some category $\mathcal{C}$, and a hom-set from one such pair $\langle s,t\rangle$ to another $\langle a,b\rangle$ is given by the coend formula:

$$\mathcal{L}\langle s,t\rangle\langle a,b\rangle = \int^{c} \mathcal{C}(s, c\quad a)\quad \mathcal{C}(c\quad b, t)$$

Notice that the pair of hom-sets in this formula can be seen as a single hom-set in the product category $\mathcal{C}^{op}\quad\mathcal{C}$:

$$\mathcal{L}\langle s,t\rangle\langle a,b\rangle = \int^{c} (\mathcal{C}^{op}\quad\mathcal{C})(c \text{ Ö } \langle a,b\rangle, \langle s,t\rangle)$$

where we define the action of c on a pair $\langle a,b\rangle$ as:

$$c \text{ Ö } \langle a,b\rangle = \langle c\quad a, c\quad b\rangle$$

This is a shorthand notation for the diagonal part of a more general action of $\mathcal{C}^{op} \quad \mathcal{C}$ on itself given by:

$$\langle c, c^{\ddot{}} \rangle \text{ Ö } \langle a, b\rangle = \langle c \quad a, c^{\ddot{}} \quad b\rangle$$

This suggests that, to represent such optics, we should be looking at co-presheaves on the category $\mathcal{C}^{op} \quad \mathcal{C}$, that is, we should be considering profunctor representations.

Iso

As a quick check of this idea, let's apply our reconstruction formula to the simple case of $\mathcal{T} = [\mathcal{C}^{op} \quad \mathcal{C}, \mathbf{Set}]$ with no additional structure. In that case we don't need to use the forgetful functors, or the monad Φ, and we just get the straightforward application of Tannakian reconstruction:

$$\mathcal{O}\langle s,t\rangle\langle a,b\rangle = \int_{P\colon \mathcal{T}} \mathbf{Set}\Big(P\langle a,b\rangle, P\langle s,t\rangle\Big) \text{ ö } \Big((\mathcal{C}^{op} \quad \mathcal{C})(\langle a,b\rangle, *)\Big)\langle s,t\rangle$$

The right hand side evaluates to:

$$(\mathcal{C}^{op} \quad \mathcal{C})(\langle a,b\rangle, \langle s,t\rangle) = \mathcal{C}(s,a) \quad \mathcal{C}(b,t)$$

This optic is known in Haskell as (or an adapter):

and it, indeed, has a profunctor representation corresponding to the following end:

Given a pair of functions it's easy to construct this profunctor-polymorphic function:

This is simply saying that any profunctor can be used to lift a pair of functions.

Conversely, we may ask the question: How can a single polymorphic function map the set $P\langle a,b\rangle$ to the set $P\langle s,t\rangle$ for *every* profunctor imaginable? The only thing this function knows about the profunctor is that it can lift a pair of functions. Therefore it must be a closure that either contains or is able to produce a pair of functions .

Exercise 18.2.1. *Implement the function:*

Hint: Use the fact that a pair of identity functions can be used to construct the following profunctor:

Profunctors and lenses

Let's try to apply the same logic to lenses. We have to find a class of profunctors to plug into our profunctor representation. Let's assume that the forgetful functor U only forgets additional structure but doesn't change the sets, so the set $P\langle a,b\rangle$ is the same as the set $(UP)\langle a,b\rangle$.

Let's start with the existential representation. We have at our disposal an object c and a pair of functions:

$$\langle f, g \rangle \colon \mathcal{C}(s, c \quad a) \quad \mathcal{C}(c \quad b, t)$$

We want to build a profunctor representation, so we have to be able to map the set $P\langle a, b\rangle$ to the set $P\langle s, t\rangle$. We could get $P\langle s, t\rangle$ by lifting these two functions, but only if start from $P\langle c \;\; a, c \;\; b\rangle$. Indeed:

$$P\langle f, g\rangle \colon P\langle c \quad a, c \quad b\rangle \to P\langle s, t\rangle$$

What we are missing is the mapping:

$$P\langle a, b\rangle \to P\langle c \quad a, c \quad b\rangle$$

And this is exactly the additional structure we shall demand from our profunctor class.

Tambara module

A profunctor P equipped with the family of transformations:

$$\alpha_{\langle a,b\rangle,c} \colon P\langle a, b\rangle \to P\langle c \quad a, c \quad b\rangle$$

is called a *Tambara module*.

We want this family to be natural in a and b, but what should we demand from c? The problem with c is that it appears twice, once in the contravariant, and once in the covariant position. So, if we want to interact nicely with arrows like $h \colon c \to c'$, we have to modify the naturality condition. We may consider a more general profunctor $P\langle c' \quad a, c \quad b\rangle$ and treat α as producing its diagonal elements, ones in which c' is the same as c.

A transformation α between diagonal parts of two profunctors P and Q is called a *dinatural transformation* (*di*-agonally natural) if the following diagram commutes for any $f \colon c \to c'$:

$$\begin{tikzcd}
 & P\langle c', c\rangle \arrow[dl, "P\langle f{,}c\rangle"'] \arrow[dr, "P\langle c'{,}f\rangle"] & \\
P\langle c, c\rangle \arrow[d, "\alpha_c"'] & & P\langle c', c'\rangle \arrow[d, "\alpha_{c'}"] \\
Q\langle c, c\rangle \arrow[dr, "P\langle c{,}f\rangle"'] & & Q\langle c', c'\rangle \arrow[dl, "P\langle f{,}c\rangle"] \\
 & Q\langle c, c'\rangle &
\end{tikzcd}$$

(I used the common shorthand $P\langle f, c\rangle$, reminiscent of whiskering, for $P\langle f, id_c\rangle$.)

In our case, the dinaturality condition simplifies to:

$$\begin{tikzcd}
& P\langle a,b\rangle \arrow[dl, "\alpha_{\langle a,b\rangle,c}"'] \arrow[dr, "\alpha_{\langle a,b\rangle,c^{\cdot\cdot}}"] & \\
P\langle c\quad a,c\quad b\rangle \arrow[dr, "P\langle c\ a{,}f\ b\rangle"'] & & P\langle c^{\cdot\cdot}\quad a,c^{\cdot\cdot}\quad b\rangle \arrow[dl, "P\langle f\ b{,}c\ b\rangle"] \\
& P\langle c\quad a,c^{\cdot\cdot}\quad b\rangle &
\end{tikzcd}$$

(Here, again $P\langle f\quad b,c\quad b\rangle$ stands for $P\langle f\quad id_b,id_{c\ b}\rangle$.)

There is one more consistency condition on Tambara modules: they must preserve the monoidal structure. The action of multiplying by c makes sense in a cartesian category: we have to have a product for any pair of objects, and we want to have the terminal object to serve as the unit of multiplication. Tambara modules have to respect unit and preserve multiplication. For the unit (terminal object), we impose the following condition:

$$\alpha_{\langle a,b\rangle,1} = id_{P\langle a,b\rangle}$$

For multiplication, we have:

$$\alpha_{\langle a,b\rangle,c^{\cdot\cdot}\ c}\ \text{ö}\ \alpha_{\langle c\ a,c\ b\rangle,c^{\cdot\cdot}}\circ\alpha_{\langle a,b\rangle,c}$$

or, pictorially:

$$\begin{tikzcd}
P\langle a,b\rangle \arrow[rr, "\alpha_{\langle a,b\rangle,c^{\cdot\cdot}\ c}"] \arrow[dr, "\alpha_{\langle a,b\rangle,c}"'] & & P\langle c^{\cdot\cdot}\quad c\quad a,c^{\cdot\cdot}\quad c\quad b\rangle \\
& P\langle c\quad a,c\quad b\rangle \arrow[ur, "\alpha_{\langle c\ a,c\ b\rangle,c^{\cdot\cdot}}"'] &
\end{tikzcd}$$

(Notice that the product is associative up to isomorphism, so there is a hidden associator in this diagram.)

Since we want Tambara modules to form a category, we have to define morphisms between them. These are natural transformations that preserve the additional structure. Let's say we have a natural transformation ρ between two Tambara modules $\rho\colon (P,\alpha)\to(Q,\beta)$. We can either apply α and then ρ, or do ρ first and then β. We want the result to be the same:

$$\begin{tikzcd}
P\langle a,b\rangle \arrow[r, "\alpha_{\langle a,b\rangle,c}"] \arrow[d, "\rho_{\langle a,b\rangle}"'] & P\langle c\quad a,c\quad b\rangle \arrow[d, "\rho_{\langle c\ a,c\ b\rangle}"] \\
Q\langle a,b\rangle \arrow[r, "\beta_{\langle a,b\rangle,c}"] & Q\langle c\quad a,c\quad b\rangle
\end{tikzcd}$$

Keep in mind that the structure of the Tambara category is encoded in these natural transformations. They will determine, through the wedge condition, the shape of the ends that enter the definition of profunctor lenses.

Profunctor lenses

Now that we have some intuition about how Tambara modules are related to lenses, let's go back to our main formula:

$$\mathcal{L}\langle s,t\rangle\langle a,b\rangle = \int_{P\colon\mathcal{T}} \mathrm{Set}\Big((UP)\langle a,b\rangle,(UP)\langle s,t\rangle\Big) \text{ ö } \Big(\Phi(\mathcal{C}^{op}\quad\mathcal{C})(\langle a,b\rangle,*)\Big)\langle s,t\rangle$$

This time we're taking the end over the Tambara category. The only missing part is the monad $\Phi = U\circ F$ or the functor F that freely generates Tambara modules.

It turns out that, instead of guessing the monad, it's easier to guess the comonad. There is a comonad in the category of profunctors that takes a profunctor P and produces another profunctor ΘP. Here's the formula:

$$(\Theta P)\langle a,b\rangle = \int_c P\langle c\quad a,c\quad b\rangle$$

You can check that this is indeed a comonad by implementing ε and δ (and). For instance, ε maps $\Theta P\to P$ using the projection π_1 for the terminal object (the unit of cartesian product).

What's interesting about this comonad is that its coalgebras are Tambara modules. Again, these are coalgebras that map profunctors to profunctors. They are natural transformations $P\to\Theta P$. We can write such a natural transformation as an element of the end:

$$\int_{a,b}\mathrm{Set}\Big(P\langle a,b\rangle,(\Theta P)\langle a,b\rangle\Big) = \int_{a,b}\int_c \mathrm{Set}\Big(P\langle a,b\rangle,P\langle c\quad a,c\quad b\rangle\Big)$$

I used the continuity of the hom-functor to pull out the end over c. The resulting end encodes a set of natural (dinatural in c) transformations that define a Tambara module:

$$\alpha_{\langle a,b\rangle,c}\colon P\langle a,b\rangle\to P\langle c\quad a,c\quad b\rangle$$

In fact, these coalgebras are *comonad* coalgebras, that is they are compatible with the comonad Θ. In other words, Tambara modules form the Eilenberg-Moore category of coalgebras for the comonad Θ.

The left adjoint to Θ is a monad Φ given by the formula:

$$(\Phi P)\langle s,t\rangle = \int^{u,v,c}(\mathcal{C}^{op}\quad\mathcal{C})\Big(c\text{ Ö }\langle u,v\rangle,\langle s,t\rangle\Big)\quad P\langle u,v\rangle$$

where I used the shorthand notation:

$$(\mathcal{C}^{op}\quad\mathcal{C})\Big(c\text{ Ö }\langle u,v\rangle,\langle s,t\rangle\Big) = \mathcal{C}(s,c\quad u)\quad\mathcal{C}(c\quad v,t)$$

This adjunction can be easily verified using some end/coend manipulations: The mapping out of ΦP to some profunctor Q can be written as an end. The coends in Φ can then be taken out using co-continuity of the hom-functor. Finally, applying the ninja-Yoneda lemma produces the mapping into ΘQ. We get:

$$[(\mathcal{C}^{op}\ \ \mathcal{C}, \mathrm{Set}](P\Phi, Q)\ \text{ö}\ [(\mathcal{C}^{op}\ \ \mathcal{C}, \mathrm{Set}](P, \Theta Q)$$

Replacing Q with P we immediately see that the set of algebras for Φ is isomorphic to the set of coalgebras for Θ. In fact they are monad algebras for Φ. This means that the Eilenberg-Moore category for the monad Φ is the same as the Tambara category.

Recall that the Eilenberg-Moore construction factorizes a monad into a free/forgetful adjunction. This is exactly the adjunction we were looking for when deriving the formula for profunctor optics.

What remains is to evaluate the action of Φ on the representable functor:

$$\Big(\Phi(\mathcal{C}^{op}\ \ \mathcal{C})(\langle a,b\rangle,*)\Big)\langle s,t\rangle = \int^{u,v,c} (\mathcal{C}^{op}\ \ \mathcal{C})\Big(c\ \text{Ö}\ \langle u,v\rangle,\langle s,t\rangle\Big)\ \ (\mathcal{C}^{op}\ \ \mathcal{C})\Big(\langle a,b\rangle,\langle u,v\rangle\Big)$$

Applying the co-Yoneda lemma, we get:

$$\int^{c} (\mathcal{C}^{op}\ \ \mathcal{C})\Big(c\ \text{Ö}\ \langle a,b\rangle,\langle s,t\rangle\Big) = \int^{c} \mathcal{C}(s, c\ \ a)\ \ \mathcal{C}(c\ \ b, t)$$

which is exactly the existential representation of the lens.

Profunctor lenses in Haskell

To define profunctor representation in Haskell we introduce a class of profunctors that are Tambara modules with respect to cartesian product (we'll see more general Tambara modules later). In the Haskell library this class is called . It also appears in the literature as :

The polymorphic function has all the relevant naturality properties guaranteed by parametric polymorphism.

The profunctor lens is just a type synonym for a function type that is polymorphic in profunctors:

The easiest way to implement such a function is to start from the existential representation of a lens and apply followed by to the profunctor argument:

Because profunctor lenses are just functions, we can compose them as such:

The converse mapping from a profunctor representation to the set/get representation of the lens is also possible. For that we need to guess the profunctor that we can feed into . It turns out that the get/set representation of the lens is such a profunctor, when we fix the pair of types and . We define:

It's easy to show that it's indeed a profunctor:

Not only that—it is also a profunctor:

We can now initialize with a trivial pair of a getter and a setter and feed it to our profunctor representation:

18.3 General Optics

Tambara modules were originally defined for an arbitrary monoidal category[1] with a tensor product $\otimes$ and a unit object I. Their structure maps have the form:

$$\alpha_{\langle a,b\rangle,c} \colon P\langle a,b\rangle \to P\langle c \otimes a, c \otimes b\rangle$$

You can easily convince yourself that all coherency laws translate directly to this case, and the derivation of profunctor optics works without a change.

Prisms

From the programming point of view there are two obvious monoidal structures to explore: the product and the sum. We've seen that the product gives rise to lenses. The sum, or the coproduct, gives rise to prisms.

We get the existential representation simply by replacing the product by the sum in the definition of a lens:

$$\mathcal{P}\langle s,t\rangle\langle a,b\rangle = \int^c \mathcal{C}(s, c+a) \quad \mathcal{C}(c+b, t)$$

To simplify this, notice that the mapping out of a sum is equivalent to the product of mappings:

$$\int^c \mathcal{C}(s, c+a) \quad \mathcal{C}(c+b, t) \text{ ö } \int^c \mathcal{C}(s, c+a) \quad \mathcal{C}(c, t) \quad \mathcal{C}(b, t)$$

Using the co-Yoneda lemma, we can get rid of the coend to get:

$$\mathcal{C}(s, t+a) \quad \mathcal{C}(b, t)$$

In Haskell, this defines a pair of functions:

[1] In fact, Tambara modules were originally defined for a category enriched over vector spaces

To understand this, let's first translate the existential form of the prism:

Here either contains the focus or the residue . Conversely, can be built either from the new focus , or from the residue .

This logic is reflected in the conversion functions:

The profunctor representation of the prism is almost identical to that of the lens, except for swapping the product for the sum.

The class of Tambara modules for the sum type is called in the Haskell library, or in the literature:

The profunctor representation is a polymorphic function type:

The conversion from the existential prism is virtually identical to that of the lens:

Again, profunctor prisms compose using function composition.

Traversals

A traversal is a type of optic that focuses on multiple foci at once. Imagine, for instance, that you have a tree that can have zero or more leaves of type . A traversal should be able to get you a list of those nodes. It should also let you replace these nodes with a new list. And here's the problem: the length of the list that delivers the replacements must match the number of nodes, otherwise bad things happen.

A type-safe implementation of a traversal would require us to keep track of the sizes of lists. In other words, it would require dependent types.

In Haskell, a (non-type-safe) traversal is often written as:

with the understanding that the sizes of the two lists are determined by and must be the same.

When translating traversals to categorical language, we'll express this condition using a sum over the sizes of the list. A counted list of size n is an n-tuple, or an element of a^n, so we can write:

$$\text{Tr}\langle s, t\rangle\langle a, b\rangle = \text{Set}\Big(s, \sum_n (\text{Set}(b^n, t) \quad a^n)\Big)$$

We interpret a traversal as a function that, given a source s produces an existential type that is hiding an n. It says that there exists an n and a pair consisting of a function $b^n \to t$ and an n-tuple a^n.

The existential form of a traversal must take into account the fact that the residues for different n's will have, in principle, different types. For instance, you can decompose a tree into an n-tuple of leaves a^n and the residue c_n with n holes. So the correct existential representation for a traversal must involve a coend over all sequences c_n that are indexed by natural numbers:

$$\text{Tr}\langle s, t\rangle\langle a, b\rangle = \int^{c_n} \mathcal{C}(s, \sum_m c_m \quad a^m) \ \ \mathcal{C}(\sum_k c_k \quad b^k, t)$$

The sums here are coproducts in $\mathcal{C}$.

One way to look at sequences c_n is to interpret them as fibrations. For instance, in Set we would start with a set C and a projection $p \colon C \to \mathbb{N}$, where $\mathbb{N}$ is a set of natural numbers. Similarly a^n could be interpreted as a fibration of the free monoid on a (the set of lists of a's) with the projection that extracts the length of the list.

Or we can look at c_n's as mappings from the set of natural numbers to $\mathcal{C}$. In fact, we can treat natural numbers as a discrete category $\mathcal{N}$, in which case c_n's are functors $\mathcal{N} \to \mathcal{C}$.

$$\text{Tr}\langle s, t\rangle\langle a, b\rangle = \int^{c \colon [\mathcal{N}, \mathcal{C}]} \mathcal{C}(s, \sum_m c_m \quad a^m) \ \ \mathcal{C}(\sum_k c_k \quad b^k, t)$$

To show the equivalence of the two representations, we first rewrite the mapping out of a sum as a product of mappings:

$$\int^{c \colon [\mathcal{N}, \mathcal{C}]} \mathcal{C}(s, \sum_m c_m \quad a^m) \ \ \prod_k \mathcal{C}(c_k \quad b^k, t)$$

and then use the currying adjunction:

$$\int^{c \colon [\mathcal{N}, \mathcal{C}]} \mathcal{C}(s, \sum_m c_m \quad a^m) \ \ \prod_k \mathcal{C}(c_k, [b^k, t])$$

Here, $[b^k, t]$ is the internal hom, which is an alternative notation for the exponential object t^{b^k}.

The next step is to recognize that a product in this formula represents a set of natural transformations in $[\mathcal{N}, \mathcal{C}]$. Indeed, we could write it as an end:

$$\prod_k \mathcal{C}(c_k, [b^k, t] \text{ ö} \int_{k : \mathcal{N}} \mathcal{C}(c_k, [b^k, t])$$

This is because an end over a discrete category is just a product. Alternatively, we could write it as a hom-set in the functor category:

$$[\mathcal{N}, \mathcal{C}](c_*, [b^*, t])$$

with placeholders replacing the arguments to the two functors in question:

$$k \mapsto c_k$$

$$k \mapsto [b^k, t]$$

We can now use the co-Yoneda lemma in the functor category $[\mathcal{N}, \mathcal{C}]$:

$$\int^{c\colon [\mathcal{N},\mathcal{C}]} \mathcal{C}(s, \sum_m c_m \quad a^m) \quad [\mathcal{N}, \mathcal{C}]\Big(c_*, [b^*, t]\Big) \text{ ö } \mathcal{C}(s, \sum_m [b^m, t] \quad a^m)$$

This result is more general than our original formula, but it turns into it when restricted to the category of sets.

To derive a profunctor representation for traversals, we should look more closely at the kind of transformations that are involved. We define the action of a functor $c\colon [\mathcal{N}, \mathcal{C}]$ on a as:

$$c \text{ Ö } a = \sum_m c_m \quad a^m$$

These actions can be composed by expanding the formula using distributivity laws:

$$c \text{ Ö } (c^{\text{¨}} \text{ Ö } a) = \sum_m c_m \quad (\sum_n c^{\text{¨}}_n \quad a^n)^m$$

If the target category is Set, this is equivalent to the following Day convolution (for non-Set categories, one could use the enriched version of the Day convolution):

$$(c \star c^{\text{¨}})_k = \int^{m,n} \mathcal{N}(m+n, k) \quad c_m \quad c^{\text{¨}}_n$$

This gives monoidal structure to the category $[\mathcal{N}, \mathcal{C}]$.

The existential representation of traversals can be written is terms of the action of this monoidal category on $\mathcal{C}$:

$$\mathrm{Tr}\langle s, t\rangle\langle a, b\rangle = \int^{c\colon [\mathcal{N},\mathcal{C}]} \mathcal{C}(s, c \text{ Ö } a) \quad \mathcal{C}(c \text{ Ö } b, t)$$

To derive the profunctor representation of traversals, we have to generalize Tambara modules to the action of a monoidal category:

$$\alpha_{\langle a,b\rangle, c}\colon P\langle a, b\rangle \to P\langle c \text{ Ö } a, c \text{ Ö } b\rangle$$

It turns out that the original derivation of profunctor optics still works for these generalized Tambara modules, and traversals can be written as polymorphic functions:

$$\mathrm{Tr}\langle s, t\rangle\langle a, b\rangle = \int_{P\colon \mathcal{T}} \mathrm{Set}\Big((UP)\langle a, b\rangle, (UP)\langle s, t\rangle\Big)$$

where the end is taken over a generalized Tambara module.

18.4 Mixed Optics

Whenever we have an action of a monoidal category $\mathcal{M}$ on $\mathcal{C}$ we can define the corresponding optic. A category with such an action is called an *actegory*. We can go even further by considering two separate actions. Suppose that $\mathcal{M}$ can act on both $\mathcal{C}$ and $\mathcal{D}$. We'll use the same notation for both actions:

$$\text{Ö} \colon \mathcal{M} \quad \mathcal{C} \to \mathcal{C}$$

$$\text{Ö} \colon \mathcal{M} \quad \mathcal{D} \to \mathcal{D}$$

We can then define the *mixed optics* as:

$$\mathcal{O}\langle s, t\rangle\langle a, b\rangle = \int^{m \colon \mathcal{M}} \mathcal{C}(s, m \,\text{Ö}\, a) \quad \mathcal{D}(m \,\text{Ö}\, b, t)$$

These mixed optics have profunctor representations in terms of profunctors:

$$P \colon \mathcal{C}^{op} \quad \mathcal{D} \to \mathrm{Set}$$

and the corresponding Tambara modules that use two separate actions:

$$\alpha_{\langle a,b\rangle,m} \colon P\langle a, b\rangle \to P\langle m \,\text{Ö}\, a, m \,\text{Ö}\, b\rangle$$

with a an object of $\mathcal{C}$, b and object of $\mathcal{D}$, and m and object of $\mathcal{M}$.

Exercise 18.4.1. *What are the mixed optics for the action of the cartesian product when one of the categories is the terminal category? What if the first category is $\mathcal{C}^{op}$ $\mathcal{C}$ and the second is terminal?*

Chapter 19

Kan Extensions

If category theory keeps raising levels of abstraction it's because it's all about discovering patterns. Once patterns are discovered, it's time to study patterns formed by these patterns, and so on.

We've seen the same recurring concepts described more and more tersely at higher and higher levels of abstraction.

For instance, we first defined the product using a universal construction. Then we saw that the spans in the definition of the product were natural transformations. That led to the interpretation of the product as a limit. Then we saw that we could define it using adjunctions. We were able to combine it with the coproduct in one terse formula:

$$(+) \dashv \Delta \dashv (\ )$$

Lao Tzu said: "If you want to shrink something, you must first allow it to expand."

Kan extensions raise the level of abstraction even higher. Mac Lane said: "All concepts are Kan extensions."

19.1 Closed Monoidal Categories

We've seen how a function object can be defined as the right adjoint to the categorical product:

$$C(a \quad b, c) \text{ ö } C(a, [b, c])$$

Here I used the alternative notation $[b, c]$ for the internal hom—the exponential c^b.

An adjunction between two functors can be thought of as one being the pseudo-inverse of the other. They don't compose to identity, but their composition *is* related to the identity functor through unit and counit. For instance, if you squint hard enough, the counit of the currying adjunction:

$$\varepsilon_{bc} : [b, c] \quad b \to c$$

suggests that $[b, c]$ embodies, in a sense, the inverse of multiplication. It plays a similar role as c/b in:

$$c/b \quad b = c$$

In a typical categorical manner, we may ask the question: What if we replace the product with something else? The obvious thing, replacing it with a coproduct, doesn't work (thus we have no analog of subtraction). But maybe there are other well-behaved binary operations that have a right adjoint.

A natural setting for generalizing a product is a monoidal category with a tensor product $\otimes$ and a unit object I. If we have an adjunction:

$$\mathcal{C}(a \otimes b, c) \text{ ö } \mathcal{C}(a, [b, c])$$

we'll call the category *closed monoidal*. In a typical categorical abuse of notation, unless it leads to confusion, we'll use the same symbol (a pair of square brackets) for the monoidal internal hom as we did for the cartesian hom. There is an alternative lollipop notation for the right adjoint to the tensor product:

$$\mathcal{C}(a \otimes b, c) \text{ ö } \mathcal{C}(a, b \multimap c)$$

It is often used in the context of linear types.

The definition of the internal hom works well for a symmetric monoidal category. If the tensor product is not symmetric, the adjunction defines a *left closed* monoidal category. The left internal hom is adjoint to the "post-multiplication" functor $(* \otimes b)$. The right-closed structure is defined as the right adjoint to the "pre-multiplication" functor $(b \otimes *)$. If both are defined than the category is called *bi-closed*.

Internal hom for Day convolution

As an example, consider the symmetric monoidal structure in the category of co-presheaves with Day convolution:

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \otimes b, x) \quad Fa \quad Gb$$

We are looking for the adjunction:

$$[\mathcal{C}, \mathrm{Set}](F \star G, H) \text{ ö } [\mathcal{C}, \mathrm{Set}](F, [G, H]_{\mathrm{Day}})$$

The natural transformation on the left-hand side can be written as an end over x:

$$\int_x \mathrm{Set}\Big(\int^{a,b} \mathcal{C}(a \otimes b, x) \quad Fa \quad Gb, Hx \Big)$$

We can use co-continuity to pull out the coends:

$$\int_{x,a,b} \mathrm{Set}\Big(C(a \otimes b, x) \quad Fa \quad Gb, Hx\Big)$$

We can then use the currying adjunction in Set (the square brackets stand for the internal hom in Set):

$$\int_{x,a,b} \mathrm{Set}\Big(Fa, [C(a \otimes b, x) \quad Gb, Hx]\Big)$$

Finally, we use the continuity of the hom-set to move the two ends inside the hom-set:

$$\int_{a} \mathrm{Set}\Big(Fa, \int_{x,b} [C(a \otimes b, x) \quad Gb, Hx]\Big)$$

We discover that the right adjoint to Day convolution is given by:

$$\Big([G, H]_{\mathrm{Day}}\Big)a = \int_{x,b} \Big(\!\!\Big(C(a \otimes b, x), [Gb, Hx]\Big)\!\!\Big) \text{ ö } \int_{b} [Gb, H(a \otimes b)]$$

The last transformation is the application of the Yoneda lemma in Set.

Exercise 19.1.1. *Implement the internal hom for Day convolution in Haskell. Hint: Use a type alias.*

Exercise 19.1.2. *Implement witnesses to the adjunction:*

Powering and co-powering

In the category of sets, the internal hom (the function object, or the exponential) is isomorphic to the external hom (the set of morphisms between two objects):

$$C^B \text{ ö } Set(B, C)$$

We can therefore rewrite the currying adjunction that defines the internal hom in Set as:

$$\mathrm{Set}(A \quad B, C) \text{ ö } \mathrm{Set}\Big(A, \mathrm{Set}(B, C)\Big)$$

We can generalize this adjunction to the case where B and C are not sets but objects in some category C. The external hom in any category is always a set. But the left-hand side is no longer defined by a product. Instead it defines the action of a set A on an object b:

$$C(A \cdot b, c) \text{ ö } \mathrm{Set}\Big(A, C(b, c)\Big)$$

that is called a *co-power*.

You may think of this action as adding together (taking a coproduct of) A copies of b. For instance, if A is a two-element set 2, we get:

$$\mathcal{C}(2 \cdot b, c) \cong \mathrm{Set}\Big(2, \mathcal{C}(b, c)\Big) \cong \mathcal{C}(b, c) \times \mathcal{C}(b, c) \cong \mathcal{C}(b + b, c)$$

In other words,

$$2 \cdot b \cong b + b$$

In this sense a co-power defines multiplication in terms of iterated addition, the way we learned it in school.

If we multiply b by the hom-set $\mathcal{C}(b, c)$ and take the coend over b's, the result is isomorphic to c:

$$\int^b \mathcal{C}(b, c) \cdot b \cong c$$

Indeed, the mappings to an arbitrary x from both sides are isomorphic due to the Yoneda lemma:

$$\mathcal{C}\Big(\int^b \mathcal{C}(b, c) \cdot b, x\Big) \cong \int_b \mathrm{Set}\Big(\mathcal{C}(b, c), \mathcal{C}(b, x)\Big) \cong \mathcal{C}(c, x)$$

As expected, in Set, the co-power decays to the cartesian product.

$$\mathrm{Set}(A \cdot B, C) \cong \mathrm{Set}\Big(A, \mathrm{Set}(B, C)\Big) \cong \mathrm{Set}(A \times B, C)$$

Similarly, we can express powering as iterated multiplication. We use the same right-hand side, but this time we use the mapping-in to define the *power*:

$$\mathcal{C}(b, A \pitchfork c) \cong \mathrm{Set}\Big(A, \mathcal{C}(b, c)\Big)$$

You may think of the power as multiplying together A copies of c. Indeed, replacing A with 2 results in:

$$\mathcal{C}(b, 2 \pitchfork c) \cong \mathrm{Set}\Big(2, \mathcal{C}(b, c)\Big) \cong \mathcal{C}(b, c) \times \mathcal{C}(b, c) \cong \mathcal{C}(b, c \times c)$$

In other words:

$$2 \pitchfork c \cong c \times c$$

which is a fancy way of writing c^2.

If we power c by the hom-set $\mathcal{C}(c', c)$ and take the end over all c's, the result is isomorphic to c':

$$\int_c \mathcal{C}(c', c) \pitchfork c \cong c'$$

This follows from the Yoneda lemma. Indeed the mappings from any x to both sides are isomorphic:

$$\mathcal{C}\Big(x, \int_c \mathcal{C}(c', c) \pitchfork c\Big) \cong \int_c \mathrm{Set}\Big(\mathcal{C}(c', c), \mathcal{C}(x, c)\Big) \cong \mathcal{C}(x, c')$$

In Set, the power decays to the exponential, which is isomorphic to the hom-set:

$$A \pitchfork C \cong C^A \cong \mathrm{Set}(A, C)$$

This is the consequence of the symmetry of the product.

$$\mathrm{Set}(B, A \pitchfork C) \cong \mathrm{Set}(A, \mathrm{Set}(B, C)) \cong \mathrm{Set}(A \times B, C)$$

$$\cong \mathrm{Set}(B \times A, C) \cong \mathrm{Set}(B, \mathrm{Set}(A, C))$$

19.2 Inverting a functor

One aspect of category theory is to discard information by performing lossy transformations; the other is recovering the lost information. We've seen examples of making up for lost data with free functors—the adjoints to forgetful functors. Kan extensions are another example. Both make up for data that is lost by a functor that is not invertible.

There are two reasons why a functor might not be invertible. One is that it may map multiple objects or arrows into a single object or arrow. In other words, it's not injective on objects or arrows. The other reason is that its image may not cover the whole target category. In other words, it's not surjective on objects or arrows.

Consider for instance an adjunction $L \dashv R$. Suppose that R is not injective, and it collapses two object c and c' into a single object d

$$Rc = d$$
$$Rc' = d$$

L has no chance of undoing it. It can't map d to both c and c' at the same time. The best it can do is to map d to a "more general" object Ld that has arrows to both c and c'. These arrows are needed to define the components of the counit of the adjunction:

$$\varepsilon_c \colon Ld \to c$$
$$\varepsilon_{c'} \colon Ld \to c'$$

where Ld is both $L(Rc)$ and $L(Rc')$

$$\begin{tikzcd}
Ld \arrow[d, "\varepsilon_c"] \arrow[dd, bend right, "\varepsilon_{c'}"'] & \\
c \arrow[r, dashed] & d \arrow[lu, dashed, bend right, "L"'] \\
c' \arrow[ru, dashed, bend right, "R"'] &
\end{tikzcd}$$

Moreover, if R is not surjective on objects, the functor L must somehow be defined on those objects of $\mathcal{D}$ that are not in the image of R. Again, naturality of the unit and counit will constrain possible choices, as long as there are arrows connecting these objects to the image of R.

Obviously, all these constraints mean that an adjunction can only be defined in very special cases.

Kan extensions are even weaker than adjunctions.

If adjoint functors work like inverses, Kan extensions work like fractions.

This is best seen if we redraw the diagrams defining the counit and the unit of an adjunction. In the first diagram, L seems to play the role of $1/R$. In the second diagram R pretends to be $1/L$.

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "\mathrm{Id}"] \arrow[d, bend right, "R"'] & \mathcal{C} & \mathcal{D} \arrow[r, "\mathrm{Id}"] \arrow[d, bend right, "L"'] & \mathcal{D} \arrow[dl, Rightarrow, shorten=4mm, "\eta"] \\
\mathcal{D} \arrow[ur, bend right, "L"'] \arrow[ur, Rightarrow, shorten=4mm, "\varepsilon"] & & \mathcal{C} \arrow[ur, bend right, "R"'] &
\end{tikzcd}$$

The right Kan extension $\mathrm{Ran}_P F$ and the left Kan extension $\mathrm{Lan}_P F$ generalize these by replacing the identity functor with some functor $F \colon \mathcal{E} \to \mathcal{C}$. The Kan extensions then play the role of fractions F/P. Conceptually, they undo the action of P and follow it with the action of F.

$$\begin{tikzcd}
\mathcal{E} \arrow[r, "F"] \arrow[d, bend right, "P"'] & \mathcal{C} & \mathcal{E} \arrow[r, "F"] \arrow[d, bend right, "P"'] & \mathcal{C} \arrow[dl, Rightarrow, shorten=4mm, "\eta"] \\
\mathcal{B} \arrow[ur, bend right, "\mathrm{Ran}_P F"'] \arrow[ur, Rightarrow, shorten=4mm, "\varepsilon"] & & \mathcal{B} \arrow[ur, bend right, "\mathrm{Lan}_P F"'] &
\end{tikzcd}$$

Just like with adjunctions, the "undoing" is not complete. The composition $\mathrm{Ran}_P F \circ P$ doesn't reproduce F; instead it's related to it through the natural transformation ε called the counit. Similarly, the composition $\mathrm{Lan}_P F \circ P$ is related to F through the unit η.

Notice that the more information F discards, the easier it is for Kan extensions to "invert" the functor P. In as sense, it only has to invert P "modulo F".

Here's the intuition behind Kan extensions. We start with a functor F:

$$\mathcal{E} \xrightarrow{F} \mathcal{C}$$

There is a second functor P that squishes $\mathcal{E}$ in another category $\mathcal{B}$. This may be an embedding that is lossy and non-surjective. Our task is to somehow *extend* the definition of F to cover the whole of $\mathcal{B}$.

In the ideal world we would like the following diagram to commute:

$$\begin{tikzcd}
\mathcal{E} \arrow[r, "F"] \arrow[d, "P"'] & \mathcal{C} \\
\mathcal{B} \arrow[ur, "Kan_P F"'] &
\end{tikzcd}$$

But that would involve equality of functors, which is something we try to avoid at all cost.

The next best thing would be to ask for a natural isomorphism between the two paths through this diagram. But even that seems like asking too much. So we finally settle down on demanding that one path be deformable into another, meaning there is a one-way natural transformation between them. The direction of this transformation distinguishes between right and left Kan extensions.

19.3 Right Kan extension

The right Kan extension is a functor $\text{Ran}_P F$ equipped with a natural transformation ε, called the counit of the Kan extension, defined as:

$$\varepsilon\colon (\text{Ran}_P F)\circ P \to F$$

$$\begin{tikzcd}
\mathcal{E} \arrow[r, "F"] \arrow[d, bend right, "P"'] & \mathcal{C} \\
\mathcal{B} \arrow[ur, bend right, "\text{Ran}_P F"'] \arrow[ur, Rightarrow, shorten=10pt, "\varepsilon"] &
\end{tikzcd}$$

The pair $(\text{Ran}_P F, \varepsilon)$ is universal among such pairs (G, α), where G is a functor $G\colon \mathcal{B} \to \mathcal{C}$ and α is a natural transformation:

$$\alpha\colon G\circ P \to F$$

$$\begin{tikzcd}
\mathcal{E} \arrow[r, "F"] \arrow[d, bend right, "P"'] & \mathcal{C} \\
\mathcal{B} \arrow[ur, bend right, "G"'] \arrow[ur, Rightarrow, shorten=10pt, "\alpha"] &
\end{tikzcd}$$

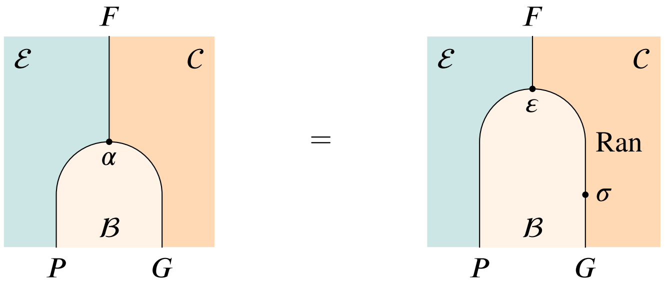
Universality means that for any such (G, α) there is a unique natural transformation $\sigma\colon G \to \text{Ran}_P F$

$$\begin{tikzcd}
\mathcal{E} \arrow[r, "F"] \arrow[d, bend right=40, "P"'] & \mathcal{C} \\
\mathcal{B} \arrow[ur, bend left=40, "G", ""{name=G, below}] \arrow[ur, bend right=40, "\text{Ran}_P F"', ""{name=R, above}] \arrow[from=G, to=R, Rightarrow, "\sigma"] &
\end{tikzcd}$$

which factorizes α, that is:

$$\alpha = \varepsilon \cdot (\sigma\circ P)$$

This is a combination of vertical and horizontal compositions of natural transformations in which $\sigma \circ P$ is the whiskering of σ. Here's the same equation drawn in terms of string diagrams:



If the right Kan extension along P is defined for every functor F, then the universal construction can be generalized to an adjunction—this time it's an adjunction between two functor categories:

$$[\mathcal{E}, \mathcal{C}](G \circ P, F) \text{ ö } [\mathcal{B}, \mathcal{C}](G, \text{Ran}_P F)$$

For every α that is an element of the left-hand side, there is a unique σ that is an element of the right-hand side.

In other words, the right Kan extension, if it exists for every F, is the right adjoint to functor pre-composition:

$$(* \circ P) \dashv \text{Ran}_P$$

The component of the counit of this adjunction at F is ε.

This is somewhat reminiscent of the currying adjunction:

$$\mathcal{E}(a \quad b, c) \text{ ö } \mathcal{E}(a, [b, c])$$

in which the product is replaced by functor composition. (The analogy is not perfect, since composition can be considered a tensor product only in the category of endofunctors.)

Right Kan extension as an end

Recall the ninja Yoneda lemma:

$$Fb \text{ ö } \int_e \text{Set}(\mathcal{B}(b, e), Fe)$$

Here, F is a co-presheaf, that is a functor from $\mathcal{B}$ to Set. The right Kan extensions of F along P generalizes this formula:

$$(\text{Ran}_P F)b \text{ ö } \int_e \text{Set}\Big(\mathcal{B}(b, Pe), Fe\Big)$$

This works for a co-presheaf. In general we are interested in $F \colon \mathcal{E} \to \mathcal{C}$, so we need to replace the hom-set in Set by a power. Thus he right Kan extension is given by the following end (if it exists):

$$(\mathrm{Ran}_P F) b \text{ ö } \int_e \mathcal{B}(b, Pe) \pitchfork Fe$$

The proof essentially writes itself: at every step there is only one thing to do. We start with the adjunction:

$$[\mathcal{E}, \mathcal{C}](G \circ P, F) \text{ ö } [\mathcal{B}, \mathcal{C}](G, \mathrm{Ran}_P F)$$

and rewrite it using ends:

$$\int_e \mathcal{C}\Big(G(Pe), Fe\Big) \text{ ö } \int_b \mathcal{C}\Big(Gb, (\mathrm{Ran}_P F) b\Big)$$

We plug in our formula to get:

$$\text{ö } \int_b \mathcal{C}\Big(Gb, \int_e \mathcal{B}(b, Pe) \pitchfork Fe\Big)$$

We use the continuity of the hom-functor to pull the end to the front:

$$\text{ö } \int_b \int_e \mathcal{C}\Big(Gb, \mathcal{B}(b, Pe) \pitchfork Fe\Big)$$

Then we use the definition of power:

$$\int_b \int_e \mathrm{Set}\Big(\mathcal{B}(b, Pe), \mathcal{C}(Gb, Fe)\Big)$$

and apply the Yoneda lemma:

$$\int_e \mathcal{C}\Big(G(Pe), Fe\Big)$$

This result is indeed the left-hand side of the adjunction.

If F is a co-presheaf, the power in the formula for the right Kan extension decays to the exponential/hom-set:

$$(\mathrm{Ran}_P F) b \text{ ö } \int_e \mathrm{Set}\Big(\mathcal{B}(b, Pe), Fe\Big)$$

Notice also that, if P has a left adjoint, let's call it P^{*1}, that is:

$$\mathcal{B}(b, Pe) \text{ ö } \mathcal{E}(P^{*1} b, e)$$

we could use the ninja Yoneda lemma to evaluate the end in:

$$(\mathrm{Ran}_P F) b \text{ ö } \int_e \mathrm{Set}\Big(\mathcal{B}(b, Pe), Fe\Big) \text{ ö } \int_e \mathrm{Set}(\mathcal{E}(P^{*1} b, e), Fe) \text{ ö } F(P^{*1} b)$$

to get:

$$\mathrm{Ran}_P F \text{ ö } F \circ P^{*1}$$

Since the adjunction is a weakening of the idea of an inverse, this result is in agreement with the intuition that the Kan extension "inverts" P and follows it with F.

Right Kan extension in Haskell

The end formula for the right Kan extension can be immediately translated to Haskell:

The counit ε of the right Kan extension is a natural transformation from the composition of after to :

To implement it, we have to produce the value of the type given a polymorphic function

We do it by instantiating this function at the type and calling it with the identity on :

The computational power of the right Kan extension comes from its universal property. We start with a functor G equipped with a natural transformation:

$$\alpha\colon G \circ P \to F$$

This can be expressed as a Haskell data type:

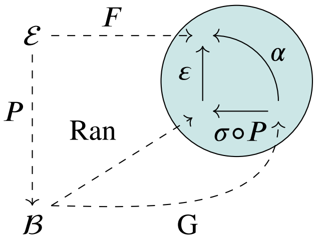
Universality tells us that there is a unique natural transformation σ from this functor to the corresponding right Kan extension:

that factorizes α through the counit ε:

$$\alpha = \varepsilon \cdot (\sigma \circ P)$$

Recall that whiskering means that we instantiate at . It is then followed by . The factorization of α is thus given by:

The components of the three natural transformations are all morphism in the target category $\mathcal{C}$:



Exercise 19.3.1. *Implement the instance for .*

Limits as Kan extensions

We have previously defined limits as universal cones. The definition of a cone involves two categories: the indexing category $\mathcal{J}$ that defines the shape of the diagram, and the target category $\mathcal{C}$. A diagram is a functor $D \colon \mathcal{J} \to \mathcal{C}$ that embeds the shape in the target category.

We can introduce a third category 1: the terminal category that contains a single object and a single identity arrow. We can then use a functor X from that category to pick the apex x of the cone in $\mathcal{E}$. Since 1 is terminal in $\mathbf{Cat}$, we also have the unique functor from $\mathcal{J}$ to it, which we'll call $!$. It maps all objects to the only object of 1, and all arrows to its identity arrow.

It turns out that the limit of D is the right Kan extension of the diagram D along $!$. First, let's observe that the composition $X \circ !$ maps the shape $\mathcal{J}$ to a single object x, so it does the job of the constant functor Δ_x. It thus picks the apex of a cone. A cone with the apex x is a natural transformation γ:

$$\begin{tikzcd}
\mathcal{J} \arrow[r, "D"] \arrow[d, bend right, "!"'] & \mathcal{C} \\
1 \arrow[ur, bend right, "X"'] \arrow[ur, Rightarrow, shorten <=6pt, shorten >=12pt, "\gamma"]
\end{tikzcd}$$

The following diagrams illustrate this. On the left we have two categories: 1 with a single object $<$, and $\mathcal{J}$ with three objects forming the shape for the diagram. On the right we have the image of D and the image of $X \circ !$, which is the apex x. The three components of γ connect the apex x to the diagram. Naturality of γ ensures that the triangles that form the sides of the cone commute.

$$\begin{tikzcd}
& < & & & x \arrow[dl, "\gamma_1"'] \arrow[dr, "\gamma_2"] \arrow[dd, "\gamma_3"] & \\
1 \arrow[rr] \arrow[dr] & & 2 \arrow[dl] & D1 \arrow[rr] \arrow[dr] & & D2 \arrow[dl] \\
& 3 & & & D3 &
\end{tikzcd}$$

The right Kan extension $(\text{Ran}_! D, \varepsilon)$ is the universal such cone. $\text{Ran}_! D$ is a functor from 1 to $\mathcal{C}$, so it selects an object in $\mathcal{C}$. This is indeed the apex, $\text{Lim} D$, of the universal cone.

Universality means that for any pair (X, γ) there is a natural transformation $\sigma \colon X \to \text{Ran}_! D$

$$\begin{tikzcd}
\mathcal{J} \arrow[r, "D"] \arrow[d, "!"'] & \mathcal{C} \\
1 \arrow[ur, bend left, "X"] \arrow[ur, bend right, "\text{Ran}_! D"'] \arrow[ur, phantom, "\Downarrow \sigma"]
\end{tikzcd}$$

which factorizes γ.

The transformation σ has only one component $\sigma_<$, which is an arrow h connecting the apex x to the apex $\mathrm{Lim}D$. The factorization:

$$\gamma = \varepsilon \cdot (\sigma \circ !)$$

reads, in components:

$$\gamma_i = \varepsilon_i \circ h$$

It makes the triangles in the following diagram commute:

$$\begin{tikzcd}
 & x \arrow[d, dashed, "h"] \arrow[ddl, "\gamma_1"'] \arrow[ddrr, "\gamma_2"] \arrow[dddr, "\gamma_3"] & & \\
 & \mathrm{Lim}D \arrow[dl, "\varepsilon_1"] \arrow[drr, "\varepsilon_2"] \arrow[ddr, "\varepsilon_3"] & & \\
D1 \arrow[rrr] \arrow[drr] & & & D2 \arrow[dl] \\
 & & D3 &
\end{tikzcd}$$

This universal condition makes $\mathrm{Lim}D$ the limit of the diagram D.

Left adjoint as a right Kan extension

We started by describing Kan extensions as a generalization of adjunctions. Looking at the pictures, if we have a pair of adjoint functors $L \dashv R$, we expect the left functor to be the right Kan extension of the identity along the right functor.

$$L \text{ ö } \mathrm{Ran}_R \mathrm{Id}$$

Indeed, the counit of the Kan extension is the same as the counit of the adjunction:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "\mathrm{Id}"] \arrow[d, bend right, "R"'] & \mathcal{C} \\
\mathcal{D} \arrow[ur, bend right, "L"'] \arrow[ur, Rightarrow, shorten >=12pt, shorten <=4pt, "\varepsilon"] &
\end{tikzcd}$$

We also have to show universality:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "\mathrm{Id}"] \arrow[d, bend right, "R"'] & \mathcal{C} \\
\mathcal{D} \arrow[ur, bend right, "G"'] \arrow[ur, Rightarrow, "\alpha", shorten=10pt] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{C} \arrow[r, "R"] \arrow[d, "R"'] & \mathcal{D} \\
\mathcal{D} \arrow[ur, bend left=40, "G", ""{name=G, below}] \arrow[ur, bend right=40, "L"', ""{name=L}] &
\arrow[Rightarrow, from=G, to=L, "\sigma"]
\end{tikzcd}$$

To do that, we have at our disposal the unit of the adjunction:

$$\eta\colon \mathrm{Id} \to R\circ L$$

We construct σ as the composite:

$$G \to G\circ\mathrm{Id} \xrightarrow{G\circ\eta} G\circ R\circ L \xrightarrow{\alpha\circ L} \mathrm{Id}\circ L \to L$$

In other words, we define σ as:

$$\sigma = (\alpha\circ L)\cdot(G\circ\eta)$$

We could ask the converse question: if $\mathrm{Ran}_R\mathrm{Id}$ exists, is it automatically the left adjoint to R? It turns out that we need one more condition for that: The Kan extension must be preserved by R, that is:

$$R\circ\mathrm{Ran}_R\mathrm{Id} \text{ ö } \mathrm{Ran}_R R$$

We'll see in the next section that the right-hand side of this condition defines the codensity monad.

Exercise 19.3.2. *Show the factorization condition:*

$$\alpha = \varepsilon\cdot(\sigma\circ R)$$

for the σ that was defined above. Hint: draw the corresponding string diagrams and use the triangle identity for the adjunction.

Codensity monad

We've seen that every adjunction $L \dashv F$ produces a monad $F\circ L$. It turns out that this monad is the right Kan extension of F along F. Interestingly, even if F doesn't have a left adjoint, the Kan extension $\mathrm{Ran}_F F$ is still a monad called the *codensity monad* denoted by T^F:

$$T^F = \mathrm{Ran}_F F$$

If we were serious about the interpretation of Kan extensions as fractions, a codensity monad would correspond to F/F. A functor for which this "fraction" is equal to identity is called codense.

To see that T^F is a monad, we have to define monadic unit and multiplication:

$$\eta \colon \mathrm{Id} \to T^F$$

$$\mu \colon T^F \circ T^F \to T^F$$

Both follow from universality. For every (G, α) we have a σ:

$$\begin{tikzcd}
\mathcal{D} \arrow[r, "F"] \arrow[d, bend right, "F"'] & \mathcal{C} \\
\mathcal{C} \arrow[ur, bend right, "G"'] \arrow[ur, Rightarrow, shorten=8pt, "\alpha"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{D} \arrow[r, "F"] \arrow[d, "F"'] & \mathcal{C} \\
\mathcal{C} \arrow[ur, bend left, "G"] \arrow[ur, bend right, "T^F{=}\mathrm{Ran}_F F"'] \arrow[ur, phantom, "\overset{\sigma}{\Rightarrow}"]
\end{tikzcd}$$

To get the unit, we replace G with the identity functor Id and α with the identity natural transformation.

To get multiplication, we replace G with $T^F \circ T^F$ and note that we have at our disposal the counit of the Kan extension:

$$\varepsilon \colon T^F \circ F \to F$$

We can chose α of the type:

$$\alpha \colon T^F \circ T^F \circ F \to F$$

to be the composite:

$$T^F \circ T^F \circ F \xrightarrow{id \circ \varepsilon} T^F \circ F \xrightarrow{\varepsilon} F$$

or, using the whiskering notation:

$$\alpha = \varepsilon \cdot (T^F \circ \varepsilon)$$

The corresponding σ gives us the monadic multiplication.

Let's now show that, if we start from an adjunction:

$$\mathcal{D}(Lc, d) \text{ ö } \mathcal{C}(c, Fd)$$

the codensity monad is given by $F \circ L$. Let's start with the mapping from an arbitrary functor G to $F \circ L$:

$$[\mathcal{C}, \mathcal{C}](G, F \circ L) \text{ ö } \int_c \mathcal{C}(Gc, F(Lc))$$

We can rewrite it using the Yoneda lemma:

$$\text{ö } \int_c \int_d \mathrm{Set}\Big(\mathcal{D}(Lc, d), \mathcal{C}(Gc, Fd)\Big)$$

Here, taking the end over d has the effect of replacing d with Lc. We can now use the adjunction:

$$\text{ö} \int_c \int_d \mathrm{Set}\Big(\mathcal{C}(c, Fd), \mathcal{C}(Gc, Fd)\Big)$$

and perform the ninja-Yoneda integration over c to get:

$$\text{ö} \int_d \mathcal{C}(G(Fd), Fd)$$

This, in turn, defines a set of natural transformations:

$$\text{ö}\ [\mathcal{D}, \mathcal{C}](G \circ F, F)$$

The pre-composition by F is the left adjoint to the right Kan extension:

$$[\mathcal{D}, \mathcal{C}](G \circ F, F)\ \text{ö}\ [\mathcal{C}, \mathcal{C}](G, \mathrm{Ran}_F F)$$

Since G was arbitrary, we conclude that $F \circ L$ is indeed the codensity monad $\mathrm{Ran}_F F$.

Since every monad can be derived from some adjunction, it follows that *every monad is a codensity monad* for some adjunction.

Codensity monad in Haskell

Translating the codensity monad to Haskell, we get:

together with the extractor:

This looks very similar to a continuation monad. In fact it turns into continuation monad if we choose to be the identity functor. We can think of as taking a callback and calling it when the result of type becomes available.

Here's the monad instance:

Again, this is almost exactly like the continuation monad:

This is why has the performance advantages of the continuation passing style. Since it nests continuations "inside out," it can be used to optimize long chains of binds that are produced by blocks.

This property is especially important when working with free monads, which accumulate binds in tree-like structures. When we finally interpret a free monad, these accumulated binds require traversing the ever-growing tree. For every bind, the traversal starts at the root. Compare this with the earlier example of reversing a list, which was optimized by accumulating functions in a FIFO queue. The codensity monad offers the same kind of performance improvement.

Exercise 19.3.3. *Implement the instance for .*

Exercise 19.3.4. *Implement the instance for .*

19.4 Left Kan extension

Just like the right Kan extension was defined as a *right* adjoint to functor pre-compositon, the left Kan extension is defined as the *left* adjoint to functor pre-composition:

$$[\mathcal{B}, \mathcal{C}](\mathrm{Lan}_P F, G) \text{ ö } [\mathcal{E}, \mathcal{C}](F, G \circ P)$$

(There are also adjoints to *post*-composition: they are called Kan lifts.)

Alternatively, $\mathrm{Lan}_P F$ can be defined as a functor equipped with a natural transformation called the unit:

$$\eta \colon F \to \mathrm{Lan}_P F \circ P$$

$$\begin{tikzcd}
\mathcal{E} \arrow[r, "F"] \arrow[d, bend right, "P"'] & \mathcal{C} \\
\mathcal{B} \arrow[ur, bend right, "\mathrm{Lan}_P F"'] &
\arrow[from=1-2, to=2-1, Rightarrow, shorten=4mm, "\eta"]
\end{tikzcd}$$

Notice that the direction of the unit of the left Kan extension is opposite of that of the counit of the right Kan extension.

The pair $(\mathrm{Lan}_P F, \eta)$ is universal, meaning that, for any other pair (G, α), where

$$\alpha \colon F \to G \circ P$$

$$\begin{tikzcd}
\mathcal{E} \arrow[r, "F"] \arrow[d, bend right, "P"'] & \mathcal{C} \\
\mathcal{B} \arrow[ur, bend right, "G"'] &
\arrow[from=1-2, to=2-1, Rightarrow, shorten=4mm, "\alpha"]
\end{tikzcd}$$

there is a unique mapping $\sigma \colon \mathrm{Lan}_P F \to G$

$$\begin{tikzcd}
\mathcal{E} \arrow[r, "F"] \arrow[d, "P"'] & \mathcal{C} \\
\mathcal{B} \arrow[ur, bend left, "G"] \arrow[ur, bend right, "\mathrm{Lan}_P F"', ""{name=L}] &
\end{tikzcd}$$

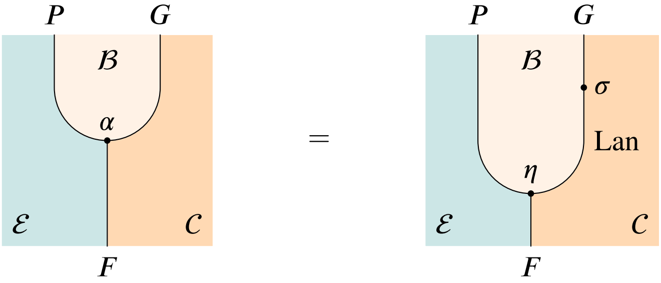
$$\sigma \colon \mathrm{Lan}_P F \Rightarrow G$$

that factorizes α:

$$\alpha = (\sigma \circ P) \cdot \eta$$

Again, the direction of σ is reversed with respect to the right Kan extension.

Using string diagrams, we can picture the universal condition as:



This establishes the one-to-one mapping between two sets of natural transformations. For every α on the left there is a unique σ on the right:

$$[\mathcal{E},\mathcal{C}](F, G\circ P) \text{ ö } [\mathcal{B},\mathcal{C}](\mathrm{Lan}_P F, G)$$

Left Kan extension as a coend

Recall the ninja co-Yoneda lemma. For every co-presheaf F, we have:

$$F b \text{ ö } \int^c \mathcal{B}(c,b) \quad F c$$

The left Kan extension generalizes this formula to:

$$(\mathrm{Lan}_P F)\, b \text{ ö } \int^e \mathcal{B}(Pe, b) \quad F e$$

For a general functor $F : \mathcal{E} \to \mathcal{C}$, we replace the product with the copower:

$$(\mathrm{Lan}_P F)\, b \text{ ö } \int^e \mathcal{B}(Pe, b) \cdot F e$$

As long as the coend in question exists, we can prove this formula by considering a mapping out to some functor G. We represent the set of natural transformations as the end over b:

$$\int_b \mathcal{C}\Big(\int^e \mathcal{B}(Pe,b) \cdot Fe, Gb\Big)$$

Using cocontinuity, we pull out the coend, turning it into an end:

$$\int_b \int_e \mathcal{C}\Big(\mathcal{B}(Pe,b) \cdot Fe, Gb\Big)$$

and we plug in the definition of co-power:

$$\int_b \int_e \mathcal{C}\Big(\mathcal{B}(Pe,b), \mathcal{C}(Fe, Gb)\Big)$$

We can now use the Yoneda lemma to integrate over b, replacing b with Pe:

$$\int_e \mathcal{C}(Fe, G(Pe))\Big) \text{ ö } [\mathcal{E},\mathcal{C}](F, G\circ P)$$

This indeed gives us the left adjoint to functor pre-composition:

$$[\mathcal{B}, \mathcal{C}](\text{Lan}_P F, G) \text{ ö } [\mathcal{E}, \mathcal{C}](F, G \circ P)$$

In $\mathbf{Set}$, the co-power decays to a cartesian product, so we get a simpler formula:

$$(\text{Lan}_P F)\, b \text{ ö } \int^e \mathcal{B}(Pe, b) \quad Fe$$

Notice that, if the functor P has a right adjoint, let's call it P^{*1}:

$$\mathcal{B}(Pe, b) \text{ ö } \mathcal{E}(e, P^{*1} b)$$

we can use the ninja co-Yoneda lemma to get:

$$(\text{Lan}_P F)\, b \text{ ö } (F \circ P^{*1}) b$$

thus reinforcing the intuition that a Kan extension inverts P and follows it with F.

Left Kan extension in Haskell

When translating the formula for the left Kan extension to Haskell, we replace the coend with the existential type. Symbolically:

This is how we would encode the existential using ’s:

The unit of the left Kan extension is a natural transformation from the functor to the composition of after :

To implement the unit, we start with a value of the type . We have to come up with some type , a function , and a value of the type . The obvious choice is to pick and use the identity at :

The computational power of the left Kan extension lies in its universal property. Given a functor and a natural transformation from to the composition of after :

there is a unique natural transformation σ from the corresponding left Kan extension to :

that factorizes α through the unit η:

$$\alpha = (\sigma \circ P) \cdot \eta$$

The whiskering of σ means instantiating it at , so the factorization of α is implemented as:

Exercise 19.4.1. *Implement the instance for .*

Colimits as Kan extensions

Just like limits can be defined as right Kan extensions, colimits can be defined as left Kan extension.

We start with an indexing category $\mathcal{J}$ that defines the shape of the colimit. The functor D selects this shape in the target category $\mathcal{C}$. The apex of the cocone is selected by a functor from the terminal single-object category 1. The natural transformation defines a cocone from D to X:

$$\begin{tikzcd}
\mathcal{J} \arrow[r, "D"] \arrow[d, bend right, "!"'] & \mathcal{C} \arrow[dl, Rightarrow, "\gamma"', shorten >=10pt] \\
1 \arrow[ur, bend right, "X"'] &
\end{tikzcd}$$

Here's an illustrative example of a simple shape consisting of three objects and three morphisms (not counting identities). The object x is the image of the single object $<$ under the functor X:

$$\begin{tikzcd}
1 \arrow[rr] \arrow[dr] & & 2 \arrow[dl] & & D1 \arrow[rr] \arrow[dr] \arrow[ddr, "\gamma_1"'] & & D2 \arrow[dl] \arrow[ddl, "\gamma_2"] \\
& 3 & & & & D3 \arrow[d, "\gamma_3"] & \\
& < & & & & x &
\end{tikzcd}$$

The colimit is the universal cocone, which is given by the left Kan extension of D along the functor $!$:

$$\text{Colim } D = \mathrm{Lan}_! D$$

Right adjoint as a left Kan extension

We've seen that, when we have an adjunction $L \vdash R$, the left adjoint is related to the right Kan extension. Dually, if the right adjoint exists, it can be expressed as the left Kan extension of the identity functor:

$$R \text{ ö } \mathrm{Lan}_L \mathrm{Id}$$

Conversely, if the left Kan extension of identity exists and it preserves the functor L:

$$L \circ \mathrm{Lan}_L \mathrm{Id} \text{ ö } \mathrm{Lan}_L L$$

than $\mathrm{Lan}_L\mathrm{Id}$ is the right adjoint of L. The left Kan extension of L along itself is called the density comonad.

The unit of Kan extension is the same as the unit of the adjunction:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "\mathrm{Id}"] \arrow[d, bend right, "L"'] & \mathcal{C} \\
\mathcal{D} \arrow[ur, bend right, "R"'] \arrow[from=1-2, to=2-1, Rightarrow, shorten=10pt, "\eta"]
\end{tikzcd}$$

The proof of universality is analogous to the one for the right Kan extension.

Exercise 19.4.2. *Implement the instance for the density comonad:*

Day convolution as a Kan extension

We've seen Day convolution defined as a tensor product in the category of co-presheaves over a monoidal category $\mathcal{C}$:

$$(F \star G)c = \int^{a,b} \mathcal{C}(a \otimes b, c) \quad Fa \quad Gb$$

Co-presheaves, that is functors in $[\mathcal{C}, \mathrm{Set}]$, can also be tensored using an *external tensor product*. An external product of two objects, instead of producing an object in the same category, picks an object in a different category. In our case, the product of two functors ends up in the category of co-presheaves on $\mathcal{C} \quad \mathcal{C}$:

$$\bar{\otimes} \colon [\mathcal{C}, \mathrm{Set}] \quad [\mathcal{C}, \mathrm{Set}] \to [\mathcal{C} \quad \mathcal{C}, \mathrm{Set}]$$

The product of two co-presheaves acting on a pair of objects in $\mathcal{C} \quad \mathcal{C}$, is given by the formula:

$$(F \bar{\otimes} G)\langle a, b\rangle = Fa \quad Gb$$

It turns out that Day convolution of two functors can be expressed as a left Kan extension of their external product along the tensor product in $\mathcal{C}$:

$$F \star G \text{ ö } \mathrm{Lan}_{\otimes}(F \bar{\otimes} G)$$

Pictorially:

$$\begin{tikzcd}
\mathcal{C} \quad \mathcal{C} \arrow[r, "F\bar{\otimes}G"] \arrow[d, bend right, "\otimes"'] & \mathrm{Set} \\
\mathcal{C} \arrow[ur, bend right, "\mathrm{Lan}_{\otimes}(F\bar{\otimes}G)"']
\end{tikzcd}$$

Indeed, using the coend formula for the left Kan extension we get:

$$(\mathrm{Lan}_{\otimes}(F\bar{\otimes}G))c \text{ ö } \int^{\langle a,b\rangle} \mathcal{C}(a\otimes b,c)\cdot(F\bar{\otimes}G)\langle a,b\rangle$$

$$\text{ö } \int^{\langle a,b\rangle} \mathcal{C}(a\otimes b,c)\cdot(Fa\quad Gb)$$

Since the two functors are $\mathbf{Set}$-valued, the co-power decays into the cartesian product:

$$\text{ö } \int^{\langle a,b\rangle} \mathcal{C}(a\otimes b,c)\quad Fa\quad Gb$$

and reproduces the formula for Day convolution.

19.5 Useful Formulas

- Co-power:

$$\mathcal{C}(A\cdot b,c) \text{ ö } \mathbf{Set}\Big(A,\mathcal{C}(b,c)\Big)$$

- Power:

$$\mathcal{C}(b,A\pitchfork c) \text{ ö } \mathbf{Set}\Big(A,\mathcal{C}(b,c)\Big)$$

- Right Kan extension:

$$[\mathcal{E},\mathcal{C}](G\circ P,F) \text{ ö } [\mathcal{B},\mathcal{C}](G,\mathrm{Ran}_P F)$$

$$(\mathrm{Ran}_P F)b \text{ ö } \int_e \mathcal{B}(b,Pe)\pitchfork Fe$$

- Left Kan extension:

$$[\mathcal{B},\mathcal{C}](\mathrm{Lan}_P F,G) \text{ ö } [\mathcal{E},\mathcal{C}](F,G\circ P)$$

$$(\mathrm{Lan}_P F)\,b \text{ ö } \int^e \mathcal{B}(Pe,b)\cdot Fe$$

- Right Kan extension in $\mathbf{Set}$:

$$(\mathrm{Ran}_P F)b \text{ ö } \int_e \mathbf{Set}\Big(\mathcal{B}(b,Pe),Fe\Big)$$

- Left Kan extension in $\mathbf{Set}$:

$$(\mathrm{Lan}_P F)b \text{ ö } \int^e \mathcal{B}(Pe,b)\quad Fe$$

Chapter 20

Enrichment

Lao Tzu says: "To know you have enough is to be rich."

20.1 Enriched Categories

This might come as a surprise, but the Haskell definition of a cannot be fully explained without some background in enriched categories. In this chapter I'll try to show that, at least conceptually, enrichment is not a huge step from the ordinary category theory.

Additional motivation for studying enriched categories comes from the fact that a lot of literature, notably the website nLab, contains descriptions of concepts in most general terms, which often means in terms of enriched categories. Most of the usual constructs can be translated just by changing the vocabulary, replacing hom-sets with hom-objects and $\mathbf{Set}$ with a monoidal category $\mathcal{V}$.

Some enriched concepts, like weighted limits and colimits, turn out to be powerful on their own, to the extent that one might be tempted to replace Mac Lane's adage,"All concepts are Kan extensions" with "All concepts are weighted (co-)limits."

Set-theoretical foundations

Category theory is very frugal at its foundations. But it (reluctantly) draws upon set theory. In particular the idea of the hom-set, defined as a set of arrows between two objects, drags in set theory as the prerequisite to category theory. Granted, arrows form a set only in a *locally small* category, but that's a small consolation, considering that dealing with things that are too big to be sets requires even more theory.

It would be nice if category theory were able to bootstrap itself, for instance by replacing hom-sets with more general objects. That's exactly the idea behind enriched categories. These hom-object, though, have to come from some other category that has hom-sets and, at some point we have to fall back on set-theoretical foundations. Nevertheless, having the option of replacing stuctureless hom-sets with something different expands our ability to model more complex systems.

The main property of sets is that, unlike objects, they are not atomic: they have *elements*. In category theory we sometimes talk about *generalized elements*, which are simply arrows pointing at an object; or *global elements*, which are arrows from the terminal object (or, sometimes, from the monoidal unit I). But most importantly, sets define *equality of elements*.

Virtually all that we've learned about categories can be translated into the realm of enriched categories. However, a lot of categorical reasoning involves commuting diagrams, which express the equality of arrows. In the enriched setting we don't have arrows going between objects, so all these constructions will have to be modified.

Hom-Objects

At first sight, replacing hom-sets with objects might seem like a step backward. After all, sets have elements, while objects are formless blobs. However, the richness of hom-objects is encoded in the morphisms of the category they come from. Conceptually, the fact that sets are structure-less means that there are lots of morphisms (functions) between them. Having fewer morphisms often means having more structure.

The guiding principle in defining enriched categories is that we should be able to recover ordinary category theory as a special case. After all hom-sets *are* objects in the category Set. In fact we've worked really hard to express properties of sets in terms of functions rather than elements.

Having said that, the very definition of a category in terms of composition and identity involves morphisms that are *elements* of hom-sets. So let's first re-formulate the primitives of a category without recourse to elements.

Composition of arrows can be defined in bulk as a function between hom-sets:

$$\circ \colon \mathcal{C}(b,c) \quad \mathcal{C}(a,b) \to \mathcal{C}(a,c)$$

Instead of talking about the identity arrow, we can use a function from the singleton set:

$$j_a \colon 1 \to \mathcal{C}(a,a)$$

This shows us that, if we want to replace hom-sets $\mathcal{C}(a,b)$ with objects from some category $\mathcal{V}$, we have to be able to *multiply* these objects to define composition, and we need some kind of *unit object* to define identity. We could ask for $\mathcal{V}$ to be cartesian but, in fact, a monoidal category works just fine. As we'll see, the unit and associativity laws of a monoidal category translate directly to identity and associativity laws for composition.

Enriched Categories

Let $\mathcal{V}$ be a monoidal category with a tensor product $\otimes$, a unit object I, and the associator and two unitors (as well as their inverses):

$$\alpha\colon (a \otimes b) \otimes c \to a \otimes (b \otimes c)$$
$$\lambda\colon I \otimes a \to a$$
$$\rho\colon a \otimes I \to a$$

A category $\mathcal{C}$ enriched over $\mathcal{V}$ has objects and, for any pair of objects a and b, a hom-object $\mathcal{C}(a, b)$. This hom-object is an object in $\mathcal{V}$. Composition is defined using arrows in $\mathcal{V}$:

$$\circ\colon \mathcal{C}(b, c) \otimes \mathcal{C}(a, b) \to \mathcal{C}(a, c)$$

Identity is defined by the arrow:

$$j_a\colon I \to \mathcal{C}(a, a)$$

Associativity is expressed in terms of the associators in $\mathcal{V}$:

$$\begin{tikzcd}
\Big(\mathcal{C}(c,d) \otimes \mathcal{C}(b,c)\Big) \otimes \mathcal{C}(a,b) \arrow[rr, "\alpha"] \arrow[d, "\circ\otimes id"] & & \mathcal{C}(c,d) \otimes \Big(\mathcal{C}(b,c) \otimes \mathcal{C}(a,b)\Big) \arrow[d, "id\otimes\circ"] \\
\mathcal{C}(b,d) \otimes \mathcal{C}(a,b) \arrow[dr, "\circ"] & & \mathcal{C}(c,d) \otimes \mathcal{C}(a,c) \arrow[dl, "\circ"'] \\
& \mathcal{C}(a,d) &
\end{tikzcd}$$

Unit laws are expressed in terms of unitors in $\mathcal{V}$:

$$\begin{tikzcd}
I \otimes \mathcal{C}(a,b) \arrow[r, "\lambda"] \arrow[d, "j_b\otimes id"'] & \mathcal{C}(a,b) \\
\mathcal{C}(b,b) \otimes \mathcal{C}(a,b) \arrow[ur, "\circ"] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{C}(a,b) \otimes I \arrow[r, "\rho"] \arrow[d, "id\otimes j_a"'] & \mathcal{C}(a,b) \\
\mathcal{C}(a,b) \otimes \mathcal{C}(a,a) \arrow[ur, "\circ"] &
\end{tikzcd}$$

Notice that these are all diagrams in $\mathcal{V}$, where we do have arrows forming hom-sets. We still fall back on set theory, but at a different level.

A category enriched over $\mathcal{V}$ is also called a $\mathcal{V}$-category. In what follows we'll assume that the enriching category is *symmetric* monoidal, so we can form opposite and product $\mathcal{V}$-categories.

The category $\mathcal{C}^{op}$ opposite to a $\mathcal{V}$-category $\mathcal{C}$ is obtained by reversing hom-objects, that is:

$$\mathcal{C}^{op}(a, b) = \mathcal{C}(b, a)$$

Composition in the opposite category involves reversing the order of hom-objects, so it only works if the tensor product is symmetric.

We can also define a tensor product of $\mathcal{V}$-categories; again, provided that $\mathcal{V}$ is symmetric. The product of two $\mathcal{V}$-categories $\mathcal{C} \otimes \mathcal{D}$ has, as objects, pairs of objects, one from each category. The hom-objects between such pairs are defined to be tensor products:

$$(\mathcal{C} \otimes \mathcal{D})(\langle c, d\rangle, \langle c', d'\rangle) = \mathcal{C}(c, c') \otimes \mathcal{D}(d, d')$$

We need symmetry of the tensor product in order to define composition. Indeed, we need to swap the two hom-objects in the middle, before we can apply the two available compositions:

$$\circ \colon \Big(\mathcal{C}(c', c'',) \otimes \mathcal{D}(d', d'')\Big) \otimes \Big(\mathcal{C}(c, c') \otimes \mathcal{D}(d, d')\Big) \to \mathcal{C}(c, c'') \otimes \mathcal{D}(d, d'')$$

The identity arrow is the tensor product of two identities:

$$I_{\mathcal{C}} \otimes I_{\mathcal{D}} \xrightarrow{j_c \otimes j_d} \mathcal{C}(c, c) \otimes \mathcal{D}(d, d)$$

Exercise 20.1.1. *Define composition and unit in the $\mathcal{V}$-category $\mathcal{C}^{op}$.*

Exercise 20.1.2. *Show that every $\mathcal{V}$-category $\mathcal{C}$ has an underlying ordinary category $\mathcal{C}_0$ whose objects are the same, but whose hom-sets are given by (monoidal global) elements of the hom-objects, that is elements of $\mathcal{V}(I, \mathcal{C}(a, b))$.*

Examples

Seen from this new perspective, the ordinary categories we've studied so far were trivially enriched over the monoidal category $(\mathbf{Set}, \ , 1)$, with the cartesian product as the tensor product, and the singleton set as the unit.

Interestingly, a 2-category can be seen as enriched over $\mathbf{Cat}$. Indeed, 1-cells in a 2-category are themselves objects in another category. The 2-cells are just arrows in that category. In particular the 2-category $\mathbf{Cat}$ of small categories is enriched in itself. Its hom-objects are functor categories, which are objects in $\mathbf{Cat}$.

Preorders

Enrichment doesn't always mean adding more stuff. Sometimes it looks more like impoverishment, as is the case of enriching over a walking arrow category.

This category has just two objects which, for the purpose of this construction, we'll call False and True. There is a single arrow from False to True (not counting identity arrows), which makes False the initial object and True the terminal one.

$$\begin{tikzcd}
\text{False} \arrow[r, "!"] \arrow[loop above, "id_{\text{False}}"] & \text{True} \arrow[loop above, "id_{\text{True}}"]
\end{tikzcd}$$

To make this into a monoidal category, we define the tensor product, such that:

$$\text{True} \otimes \text{True} = \text{True}$$

and all other combinations produce False. True is the monoidal unit, since:

$$\text{True} \otimes x = x$$

A category enriched over the monoidal walking arrow is called a *preorder*. A hom-object $\mathcal{C}(a, b)$ between any two objects can be either False or True. We interpret True to mean that a precedes b in the preorder, which we write as $a \leq b$. False means that the two objects are unrelated.

The important property of composition, as defined by:

$$\mathcal{C}(b, c) \otimes \mathcal{C}(a, b) \to \mathcal{C}(a, c)$$

is that, if both hom-objects on the left are True, then the right hand side must also be True. (It can't be False, because there is no arrow going from True to False.) In the preorder interpretation, it means that $\leq$ is transitive:

$$b \leq c \text{ á } a \leq b \implies a \leq c$$

By the same reasoning, the existence of the identity arrow:

$$j_a \colon \text{True} \to \mathcal{C}(a, a)$$

means that $\mathcal{C}(a, a)$ is always True. In the preorder interpretation, this means that $\leq$ is reflexive, $a \leq a$.

Notice that a preorder doesn't preclude cycles and, in particular, it's possible to have $a \leq b$ and $b \leq a$ without a being equal to b.

A preorder may also be defined without resorting to enrichment as a *thin category*—a category in which there is at most one arrow between any two objects.

Self-enrichment

Any cartesian closed category $\mathcal{V}$ can be viewed as self-enriched. This is because every external hom-set $\mathcal{C}(a, b)$ can be replaced by the internal hom b^a (the object of arrows).

In fact every *monoidal closed* category $\mathcal{V}$ is self-enriched. Recall that, in a monoidal closed category we have the hom-functor adjunction:

$$\mathcal{V}(a \otimes b, c) \text{ ö } \mathcal{V}(a, [b, c])$$

The counit of this adjunction works as the evaluation morphism:

$$\varepsilon_{bc} \colon [b, c] \otimes b \to c$$

To define composition in this self-enriched category, we need an arrow:

$$\circ \colon [b, c] \otimes [a, b] \to [a, c]$$

The trick is to consider the whole hom-set at once and show that we can always pick a canonical element in it. We start with the set:

$$\mathcal{V}([b, c] \otimes [a, b], [a, c])$$

We can use the adjunction to rewrite it as:

$$\mathcal{V}(([b, c] \otimes [a, b]) \otimes a, c)$$

All we have to do now is to pick an element of this hom-set. We do it by constructing the following composite:

$$([b, c] \otimes [a, b]) \otimes a \xrightarrow{\alpha} [b, c] \otimes ([a, b] \otimes a) \xrightarrow{id \otimes \varepsilon_{ab}} [b, c] \otimes b \xrightarrow{\varepsilon_{bc}} c$$

We used the associator and the counit of the adjunction.

We also need an arrow that defines the identity:

$$j_a \colon I \to [a, a]$$

Again, we can pick it as a member of the hom-set $\mathcal{V}(I, [a, a])$. We use the adjunction:

$$\mathcal{V}(I, [a, a]) \text{ ö } \mathcal{V}(I \otimes a, a)$$

We know that this hom-set contains the left unitor λ, so we can use it to define j_a.

20.2 $\mathcal{V}$-Functors

An ordinary functor maps objects to objects and arrows to arrows. Similarly, an enriched functor F maps object to objects, but instead of acting on individual arrows, it must map hom-objects to hom-objects. This is only possible if the hom-objects in the source category $\mathcal{C}$ belong to the same category as the hom-objects in the target category $\mathcal{D}$. In other words, both categories must be enriched over the same $\mathcal{V}$. The action of F on hom-objects is then defined using arrows in $\mathcal{V}$:

$$F_{ab} \colon \mathcal{C}(a, b) \to \mathcal{D}(Fa, Fb)$$

For clarity we specify the pair of objects in the subscript of F.

A functor must preserve composition and identity. These can be expressed as commuting diagrams in $\mathcal{V}$:

$$\begin{tikzcd}
\mathcal{C}(b,c) \otimes \mathcal{C}(a,b) \arrow[r, "\circ"] \arrow[d, "F_{bc}\otimes F_{ab}"] & \mathcal{C}(a,c) \arrow[d, "F_{ac}"] \\
\mathcal{D}(Fb,Fc) \otimes \mathcal{D}(Fa,Fb) \arrow[r, "\circ"] & \mathcal{D}(Fa,Fb)
\end{tikzcd}
\qquad
\begin{tikzcd}
 & I \arrow[dl, "j_a"'] \arrow[dr, "j_{Fa}"] & \\
\mathcal{C}(a,a) \arrow[rr, "F_{aa}"] & & \mathcal{D}(Fa,Fa)
\end{tikzcd}$$

Notice that I used the same symbol $\circ$ for two different compositions and the same j for two different identity mappings. Their meaning can be derived from the context.

As before, all diagrams are in the category $\mathcal{V}$.

The Hom-functor

The hom-functor in a category that is enriched over a monoidal *closed* category $\mathcal{V}$ is an enriched functor:

$$\text{Hom}_{\mathcal{C}} \colon \mathcal{C}^{op} \otimes \mathcal{C} \to \mathcal{V}$$

Here, in order to define an enriched functor, we have to treat $\mathcal{V}$ as self-enriched.

It's clear how this functor works on (pairs of) objects:

$$\text{Hom}_{\mathcal{C}}\langle a, b\rangle = \mathcal{C}(a,b)$$

To define an enriched functor, we have to define the action of Hom on hom-objects. Here, the source category is $\mathcal{C}^{op} \otimes \mathcal{C}$ and the target category is $\mathcal{V}$, both enriched over $\mathcal{V}$. Let's consider a hom-object from $\langle a, a^{\ddot{}} \rangle$ to $\langle b, b^{\ddot{}} \rangle$. The action of the hom-functor on this hom-object is an arrow in $\mathcal{V}$:

$$\text{Hom}_{\langle a,a^{\ddot{}}\rangle\langle b,b^{\ddot{}}\rangle} \colon (\mathcal{C}^{op} \otimes \mathcal{C})(\langle a, a^{\ddot{}} \rangle, \langle b, b^{\ddot{}} \rangle) \to \mathcal{V}(\text{Hom}\langle a, a^{\ddot{}} \rangle, \text{Hom}\langle b, b^{\ddot{}} \rangle)$$

By definition of the product category, the source is a tensor product of two hom-objects. The target is the internal hom in $\mathcal{V}$. We are thus looking for an arrow:

$$\mathcal{C}(b,a) \otimes \mathcal{C}(a^{\ddot{}}, b^{\ddot{}}) \to [\mathcal{C}(a, a^{\ddot{}}), \mathcal{C}(b, b^{\ddot{}})]$$

We can use the currying hom-functor adjunction to unpack the internal hom:

$$\Big(\mathcal{C}(b,a) \otimes \mathcal{C}(a^{\ddot{}}, b^{\ddot{}})\Big) \otimes \mathcal{C}(a, a^{\ddot{}}) \to \mathcal{C}(b, b^{\ddot{}})$$

We can construct this arrow by rearranging the product and applying the composition twice.

In the enriched setting, the closest we can get to defining an individual morphism from a to b is to use an arrow from the unit object. We define a (monoidal-global) element of a hom-object as a morphism in $\mathcal{V}$:

$$f \colon I \to \mathcal{C}(a, b)$$

We can define what it means to lift such an arrow using the hom-functor. For instance, keeping the first argument constant, we'd define:

$$\mathcal{C}(c, f) \colon \mathcal{C}(c, a) \to \mathcal{C}(c, b)$$

as the composite:

$$\mathcal{C}(c, a) \xrightarrow{\lambda^{*1}} I \otimes \mathcal{C}(c, a) \xrightarrow{f \otimes id} \mathcal{C}(a, b) \otimes \mathcal{C}(c, a) \xrightarrow{\circ} \mathcal{C}(c, b)$$

Similarly, the contravariant lifting of f:

$$\mathcal{C}(f, c) \colon \mathcal{C}(b, c) \to \mathcal{C}(a, c)$$

can be defined as:

$$\mathcal{C}(b, c) \xrightarrow{\rho^{*1}} \mathcal{C}(b, c) \otimes I \xrightarrow{id \otimes f} \mathcal{C}(b, c) \otimes \mathcal{C}(a, b) \xrightarrow{\circ} \mathcal{C}(a, c)$$

A lot of the familiar constructions we've studied in ordinary category theory have their enriched counterparts, with products replaced by tensor products and $\mathbf{Set}$ replaced by $\mathcal{V}$.

Exercise 20.2.1. *What is a functor between two preorders?*

Enriched co-presheaves

Co-presheaves, that is $\mathbf{Set}$-valued functors, play an important role in category theory, so it's natural to ask what their counterparts are in the enriched setting. The generalization of a co-presheaf is a $\mathcal{V}$-functor $\mathcal{C} \to \mathcal{V}$. This is only possible if $\mathcal{V}$ can be made into a $\mathcal{V}$-category, that is when it's monoidal-closed.

An enriched co-presheaf maps object of $\mathcal{C}$ to objects of $\mathcal{V}$ and it maps hom-objects of $\mathcal{C}$ to internal homs of $\mathcal{V}$:

$$F_{ab} \colon \mathcal{C}(a, b) \to [Fa, Fb]$$

In particular, the Hom-functor is an example of a $\mathcal{V}$-valued $\mathcal{V}$-functor:

$$\text{Hom} \colon \mathcal{C}^{op} \otimes \mathcal{C} \to \mathcal{V}$$

The hom-functor is a special case of an enriched profunctor, which is defined as:

$$\mathcal{C}^{op} \otimes \mathcal{D} \to \mathcal{V}$$

Exercise 20.2.2. *The tensor product is a functor in $\mathcal{V}$:*

$$\otimes \colon \mathcal{V} \quad \mathcal{V} \to \mathcal{V}$$

Show that if $\mathcal{V}$ is monoidal closed, the tensor product defines a $\mathcal{V}$-functor. Hint: Define its action on internal homs.

Functorial strength and enrichment

When we were discussing monads, I mentioned an important property that made them work in programming. The endofunctors that define monads must be strong, so that we can access external contexts inside monadic code.

It turns out that the way we have defined endofunctors in Haskell makes them automatically strong. The reason is that strength is related to enrichment and, as we've seen, a cartesian closed category is self-enriched. Let's start with some definitions.

Functorial strength for an endofunctor F in a monoidal category is defined as a natural transformation with components:

$$\sigma_{ab} \colon a \otimes F(b) \to F(a \otimes b)$$

There are some pretty obvious coherence conditions that make strength respect the properties of the tensor product. This is the associativity condition:

$$\begin{tikzcd}
(a \otimes b) \otimes F(c) \arrow[rr, "\sigma_{(a\otimes b)c}"] \arrow[d, "\alpha"] & & F((a \otimes b) \otimes c) \arrow[d, "F(\alpha)"] \\
a \otimes (b \otimes F(c)) \arrow[r, "a\otimes\sigma_{bc}"] & a \otimes F(b \otimes c) \arrow[r, "\sigma_{a(b\otimes c)}"] & F(a \otimes (b \otimes c))
\end{tikzcd}$$

and this is the unit condition:

$$\begin{tikzcd}
I \otimes F(a) \arrow[r, "\sigma_{Ia}"] \arrow[dr, "\lambda"] & F(I \otimes a) \arrow[d, "F(\lambda)"] \\
& F(a)
\end{tikzcd}$$

In a general monoidal category this is called the *left strength*, and there is a corresponding definition of the right strength. In a symmetric monoidal category, the two are equivalent.

An enriched endofunctor maps hom-objects to hom-objects:

$$F_{ab} \colon \mathcal{C}(a, b) \to \mathcal{C}(Fa, Fb)$$

If we treat a monoidal closed category $\mathcal{V}$ as self-enriched, the hom-objects are internal homs, so an enriched endofunctor is equipped with the mapping:

$$F_{ab} \colon [a, b] \to [Fa, Fb]$$

Compare this with our definition of a Haskell :

The function types involved in this definition, and , are the *internal* homs. So a Haskell is indeed an enriched functor.

We don't normally distinguish between external and internal homs in Haskell, since their sets of elements are isomorphic. It's a simple consequence of the currying adjunction:

$$\mathcal{C}(1 \quad b, c) \text{ ö } \mathcal{C}(1, [b, c])$$

and the fact that the terminal object is the unit of the cartesian product.

It turns out that in a self-enriched category $\mathcal{V}$ every strong endofunctor is automatically enriched. Indeed, to show that a functor F is enriched we need to define the mapping between internal homs, that is an element of the hom-set:

$$F_{ab} \text{ Ë } \mathcal{V}([a, b], [Fa, Fb])$$

Using the hom adjunction, this is isomorphic to:

$$\mathcal{V}([a, b] \otimes Fa, Fb)$$

We can construct this mapping by composing the strength and the counit of the adjunction (the evaluation morphism):

$$[a, b] \otimes Fa \xrightarrow{\sigma_{[a,b]a}} F([a, b] \otimes a) \xrightarrow{F\epsilon_{ab}} Fb$$

Conversely, every enriched endofunctor in $\mathcal{V}$ is strong. To show strength, we need to define the mapping σ_{ab}, or equivalently (by hom-adjunction):

$$a \to [Fb, F(a \otimes b)]$$

Recall the definition of the unit of the hom adjunction, the coevaluation morphism:

$$\eta_{ab} \colon a \to [b, a \otimes b]$$

We construct the following composite:

$$a \xrightarrow{\eta_{ab}} [b, a \otimes b] \xrightarrow{F_{b,a\otimes b}} [Fb, F(a \otimes b)]$$

This can be translated directly to Haskell:

with the following definition of :

Since currying and evaluation are built into Haskell, we can further simplify this formula:

20.3 $\mathcal{V}$-Natural Transformations

An ordinary natural transformation between two functors F and G from $\mathcal{C}$ to $\mathcal{D}$ is a selection of arrows from the hom-sets $\mathcal{D}(Fa, Ga)$. In the enriched setting, we don't have arrows, so the next best thing we can do is to use the unit object I to do the selection. We define a component of a $\mathcal{V}$-natural transformation at a as an arrow:

$$\nu_a \colon I \to \mathcal{D}(Fa, Ga)$$

Naturality condition is a little tricky. The standard naturality square involves the lifting of an arbitrary arrow $f \colon a \to b$ and the equality of the following compositions:

$$\nu_b \circ Ff = Gf \circ \nu_a$$

Let's consider the hom-sets that are involved in this equation. We are lifting a morphism f Ë $\mathcal{C}(a, b)$. The composites on both sides of the equation are the elements of $\mathcal{D}(Fa, Gb)$.

On the left, we have the arrow $\nu_b \circ Ff$. The composition itself is a mapping from the product of two hom-sets:

$$\mathcal{D}(Fb, Gb) \quad \mathcal{D}(Fa, Fb) \to \mathcal{D}(Fa, Gb)$$

Similarly, on the right we have $Gf \circ \nu_a$, which a composition:

$$\mathcal{D}(Ga, Gb) \quad \mathcal{D}(Fa, Ga) \to \mathcal{D}(Fa, Gb)$$

In the enriched setting we have to work with hom-objects rather than hom-sets, and the selection of the components of the natural transformation is done using the unit I. We can always produce the unit out of thin air using the inverse of the left or the right unitor.

Altogether, the naturality condition is expressed as the following commuting diagram:

$$\begin{tikzcd}
 & I \otimes \mathcal{C}(a, b) \arrow[r, "\nu_b \otimes F_{ab}"] & \mathcal{D}(Fb, Gb) \otimes \mathcal{D}(Fa, Fb) \arrow[dr, "\circ"] & \\
\mathcal{C}(a, b) \arrow[ur, "\lambda^{*1}"] \arrow[dr, "\rho^{*1}"'] & & & \mathcal{D}(Fa, Gb) \\
 & \mathcal{C}(a, b) \otimes I \arrow[r, "G_{ab} \otimes \nu_a"] & \mathcal{D}(Ga, Gb) \otimes \mathcal{D}(Fa, Ga) \arrow[ur, "\circ"'] &
\end{tikzcd}$$

This also works for an ordinary category, where we can trace two paths through this diagram by first picking an f from $\mathcal{C}(a, b)$. We can then use ν_b and ν_a to pick components of the natural transformation. We also lift f using either F or G. Finally, we use composition to reproduce the naturality equation.

This diagram can be further simplified if we use our earlier definition of the hom-functor's action on global elements of hom-objects. The components of a natural transformation are defined as such global elements:

$$\nu_a \colon I \to \mathcal{D}(Fa, Ga)$$

There are two such liftings at our disposal:

$$\mathcal{D}(d, \nu_b) \colon \mathcal{D}(d, Fb) \to \mathcal{D}(d, Gb)$$

and:

$$\mathcal{D}(\nu_a, d) \colon \mathcal{D}(Ga, d) \to \mathcal{D}(Fa, d)$$

We get something that looks more like the familiar naturality square:

$$\begin{tikzcd}
& \mathcal{D}(Fa, Fb) \arrow[dr, "\mathcal{D}(Fa, \nu_b)"] & \\
\mathcal{C}(a, b) \arrow[ur, "F_{ab}"] \arrow[dr, "G_{ab}"'] & & \mathcal{D}(Fa, Gb) \\
& \mathcal{D}(Ga, Gb) \arrow[ur, "\mathcal{D}(\nu_a, Gb)"'] &
\end{tikzcd}$$

$\mathcal{V}$-natural transformations between two $\mathcal{V}$-functors F and G form a set we call $\mathcal{V}\text{-nat}(F, G)$.

Earlier we have seen that, in ordinary categories, the set of natural transformations can be written as an end:

$$[\mathcal{C}, \mathcal{D}](F, G) \text{ ö } \int_a \mathcal{D}(Fa, Ga)$$

It turns out that ends and coends can be defined for enriched profunctors, so this formula works for enriched natural transformations as well. The difference is that, instead of a *set* of natural transformations $\mathcal{V}\text{-nat}(F, G)$, it defines the *object* of natural transformations $[\mathcal{C}, \mathcal{D}](F, G)$ in $\mathcal{V}$.

The definition of the (co-)end of a $\mathcal{V}$-profunctor $P \colon \mathcal{C} \otimes \mathcal{C}^{op} \to \mathcal{V}$ is analogous to the definition we've seen for ordinary profunctors. For instance, the end is an object e in $\mathcal{V}$ equipped with an extranatural transformation $\pi \colon e \to P$ that is universal among such objects.

20.4 Yoneda Lemma

The ordinary Yoneda lemma involves a Set-valued functor F and a set of natural transformations:

$$[\mathcal{C}, \mathrm{Set}](\mathcal{C}(c, *), F) \text{ ö } Fc$$

To generalize it to the enriched setting, we'll consider a $\mathcal{V}$-valued functor F. As before, we'll use the fact that we can treat $\mathcal{V}$ as self-enriched, as long as it's closed, so we can talk about $\mathcal{V}$-valued $\mathcal{V}$-functors.

The weak version of the Yoneda lemma deals with a *set* of $\mathcal{V}$-natural transformations. Therefore, we have to turn the right hand side into a set as well. This is done by taking the (monoidal-global) elements of Fc. We get:

$$\mathcal{V}\text{-nat}(\mathcal{C}(c, *), F) \text{ ö } \mathcal{V}(I, Fc)$$

The strong version of the Yoneda lemma works with objects of $\mathcal{V}$ and uses the end over the internal hom in $\mathcal{V}$ to represent the object of natural transformations:

$$\int_x [\mathcal{C}(c, x), Fx] \text{ ö } Fc$$

20.5 Weighted Limits

Limits (and colimits) are built around commuting triangles, so they are not immediately translatable to the enriched setting. The problem is that cones are constructed from "wires," that is individual morphisms. You may think of hom-sets as a thick bundle of wires, each wire having zero thickness. When constructing a cone, you are selecting a single wire from a hom-set. We have to replace wires with something thicker.

Consider a diagram, that is a functor D from the indexing category $\mathcal{J}$ to the target category $\mathcal{C}$. The wires for the cone with the apex x are selected from hom-sets $\mathcal{C}(x, Dj)$, where j is an object of $\mathcal{J}$.

$$\begin{tikzcd}
j \arrow[r] \arrow[dr] & k \arrow[dl] \\
& l
\end{tikzcd}
\qquad
\begin{tikzcd}
& x \arrow[dl, bend right, "\mathcal{C}(x,Dj)"'] \arrow[dl] \arrow[dl, bend left] & \\
Dj \arrow[rr] \arrow[dr] & & Dk \arrow[dl] \\
& Dl &
\end{tikzcd}$$

This selection of a j'th wire can be described as a function from the singleton set 1:

$$\gamma_j : 1 \to \mathcal{C}(x, Dj)$$

We can try to gather these functions into a natural transformation:

$$\gamma : \Delta_1 \to \mathcal{C}(x, D*)$$

where Δ_1 is a constant functor mapping all objects of $\mathcal{J}$ to the singleton set. Naturality conditions ensure that the triangles forming the sides of the cone commute.

The set of all cones with the apex x is then given by the set of natural transformations:

$$[\mathcal{J}, \mathrm{Set}](\Delta_1, \mathcal{C}(x, D^*))$$

This reformulation gets us closer to the enriched setting, since it rephrases the problem in terms of hom-sets rather than individual morphisms. We could start by considering both $\mathcal{J}$ and $\mathcal{C}$ to be enriched over $\mathcal{V}$, in which case D would be a $\mathcal{V}$-functor.

There is just one problem: how do we define a constant $\mathcal{V}$-functor $\Delta_x \colon \mathcal{C} \to \mathcal{D}$? Its action on objects is obvious: it maps all objects in $\mathcal{C}$ to one object x in $\mathcal{D}$. But what should it do to hom-objects?

An ordinary constant functor Δ_x maps all morphisms in $\mathcal{C}(a, b)$ to the identity in $\mathcal{D}(x, x)$. In the enriched setting, though, $\mathcal{D}(x, x)$ is an object with no internal structure. Even if it happened to be the unit I, there's no guarantee that for every hom-object $\mathcal{C}(a, b)$ we could find an arrow to I; and even if there was one, it might not be unique. In other words, there is no reason to believe that I is the terminal object.

The solution is to "smear the singularity": instead of using the constant functor to select a single wire, we should use some other "weighting" functor $W \colon \mathcal{J} \to \mathrm{Set}$ to select a thicker "cylinder". Such a weighted cone with the apex x is an element of the set of natural transformations:

$$[\mathcal{J}, \mathrm{Set}]\,(W, \mathcal{C}(x, D^*))$$

A *weighted limit*, also known as an *indexed limit*, $\lim^W D$, is then defined as the universal weighted cone. It means that for any weighted cone with the apex x there is a unique morphism from x to $\lim^W D$ that factorizes it. The factorization is guaranteed by the naturality of the isomorphism that defines the weighted limit:

$$\mathcal{C}(x, \lim^W D) \text{ ö } [\mathcal{J}, \mathrm{Set}](W, \mathcal{C}(x, D^*))$$

The regular, non-weighted limit is often called a *conical* limit, and it corresponds to using the constant functor as the weight.

This definition can be translated almost verbatim to the enriched setting by replacing Set with $\mathcal{V}$:

$$\mathcal{C}(x, \lim^W D) \text{ ö } [\mathcal{J}, \mathcal{V}](W, \mathcal{C}(x, D^*))$$

Of course, the meaning of the symbols in this formula is changed. Both sides are now objects in $\mathcal{V}$. The left-hand side is the hom-object in $\mathcal{C}$, and the right-hand side is the object of natural transformations between two $\mathcal{V}$-functors.

Dually, a weighted colimit is defined by the natural isomorphism:

$$\mathcal{C}(\mathrm{colim}^W D, x) \text{ ö } [\mathcal{J}^{op}, \mathcal{V}](W, \mathcal{C}(D^*, x))$$

Here, the colimit is weighed by a functor $W : \mathcal{J}^{op} \to \mathcal{V}$ from the opposite category.

Weighted (co-)limits, both in ordinary and in enriched categories, play a fundamental role: they can be used to re-formulate a lot of familiar constructions, like (co-)ends, Kan extensions, etc.

20.6 Ends as Weighted Limits

An end has a lot in common with a product or, more generally, with a limit. If you squint hard enough, the projections $\pi_x \colon e \to P\langle a, a\rangle$ form the sides of a cone; except that instead of commuting triangles we have wedges. It turns out that we can express ends as weighted limits. The advantage of this formulation is that it also works in the enriched setting.

We've seen that the end of a $\mathcal{V}$-valued $\mathcal{V}$-profunctor can be defined using the more fundamental notion of an extranatural transformation. This in turn allowed us to define the *object* of natural transformations, which enabled us to define weighted limits. We can now go ahead and extend the definition of the end to work with a more general $\mathcal{V}$-functor of mixed variance with values in a $\mathcal{V}$-category $\mathcal{D}$:

$$P \colon \mathcal{C}^{op} \otimes \mathcal{C} \to \mathcal{D}$$

We'll use this functor as a diagram in $\mathcal{D}$.

At this point mathematicians start worrying about size issues. After all we are embedding a whole category—squared—as a single diagram in $\mathcal{D}$. To avoid the size problems, we'll just assume that $\mathcal{C}$ is small; that is, its objects form a set.

We want to take a weighted limit of the diagram defined by P. The weight must be a $\mathcal{V}$-functor $\mathcal{C}^{op} \otimes \mathcal{C} \to \mathcal{V}$. There is one such functor that's always at our disposal, the hom-functor $\text{Hom}_{\mathcal{C}}$. We will use it to define the end as a weighted limit:

$$\int_c P\langle c, c\rangle = \lim^{\text{Hom}} P$$

First, let's convince ourselves that this formula works in an ordinary (Set-enriched) category. Since ends are defined by their mapping-in property, let's consider a mapping from an arbitrary object d to the weighted limit and use the standard Yoneda trick to show the isomorphism. By definition, we have:

$$\mathcal{D}(d, \lim^{\text{Hom}} P) \text{ ö } [\mathcal{C}^{op} \quad \mathcal{C}, \text{Set}](\mathcal{C}(*, =), \mathcal{D}(d, P(*, =)))$$

We can rewrite the set of natural transformations as an end over pairs of objects $\langle c, c^{\text{¨}} \rangle$:

$$\int_{\langle c, c^{\text{¨}} \rangle} \text{Set}(\mathcal{C}(c, c^{\text{¨}}), \mathcal{D}(d, P\langle c, c^{\text{¨}} \rangle))$$

Using the Fubini theorem, this is equivalent to the iterated end:

$$\int_c \int_{c^{\text{¨}}} \mathrm{Set}(\mathcal{C}(c, c^{\text{¨}}), \mathcal{D}(d, P\langle c, c^{\text{¨}}\rangle))$$

We can now apply the ninja-Yoneda lemma to perform the integration over $c^{\text{¨}}$. The result is:

$$\int_c \mathcal{D}(d, P\langle c, c\rangle) \text{ ö } \mathcal{D}(d, \int_c P\langle c, c\rangle)$$

where we used continuity to push the end under the hom-functor. Since d was arbitrary, we conclude that, for ordinary categories:

$$\lim{}^{\mathrm{Hom}} P \text{ ö } \int_c P\langle c, c\rangle$$

This justifies our use of the weighed limit to define the end in the enriched case.

An analogous formula works for the coend, except that we use the colimit with the hom-functor in the opposite category $\mathrm{Hom}_{\mathcal{C}^{op}}$ as the weight:

$$\int^c P\langle c, c\rangle = \mathrm{colim}^{\mathrm{Hom}_{\mathcal{C}^{op}}} P$$

Exercise 20.6.1. *Show that for ordinary* Set*-enriched categories the weighted colimit definition of a coend reproduces the earlier definition. Hint: use the mapping out property of the coend.*

Exercise 20.6.2. *Show that, as long as both sides exist, the following identities hold in ordinary (*Set*-enriched) categories (they can be generalized to the enriched setting):*

$$lim^W D \text{ ö } \int_{j\,:\,\mathcal{J}} Wj \pitchfork Dj$$

$$colim^W D \text{ ö } \int^{j\,:\,\mathcal{J}} Wj \cdot Dj$$

Hint: Use the mapping in/out with the Yoneda trick and the definition of power and copower.

20.7 Kan Extensions

We've seen how to express limits and colimits as Kan extensions using a functor from the singular category 1. Weighted limits let us dispose of the singularity, and a judicious choice of the weight lets us express Kan extensions in terms of weighted limits.

First, let's work out the formula for ordinary, Set-enriched categories. The right Kan extension is defined as:

$$(\mathrm{Ran}_P F)e \text{ ö } \int_c \mathcal{B}(e, Pc) \pitchfork Fc$$

We'll consider the mapping into it from an arbitrary object d. The derivation follows a number of simple steps, mostly by expanding the definitions.

We start with:

$$\mathcal{D}\Big(d, (\mathrm{Ran}_P F)e\Big)$$

and substitute the definition of the Kan extension:

$$\mathcal{D}\Big(d, \int_c \mathcal{B}(e, Pc) \pitchfork Fc\Big)$$

Using the continuity of the hom-functor, we can pull out the end:

$$\int_c \mathcal{D}\Big(d, \mathcal{B}(e, Pc) \pitchfork Fc\Big)$$

We then use the definition of the pitchfork:

$$\mathcal{D}(d, A \pitchfork d^{\text{¨}}) \text{ ö } \mathrm{Set}\Big(A, \mathcal{D}(d, d^{\text{¨}})\Big)$$

to get:

$$\int_c \mathcal{D}\Big(d, \mathcal{B}(e, Pc) \pitchfork Fc\Big) \text{ ö } \int_c \mathrm{Set}\Big(\mathcal{B}(e, Pc), \mathcal{D}(d, Fc)\Big)$$

This can be written as a set of natural transformation:

$$[\mathcal{C}, \mathrm{Set}]\Big(\mathcal{B}(e, P*), \mathcal{D}(d, F*)\Big)$$

The weighted limit is also defined through the set of natural transformations:

$$\mathcal{D}(d, \lim^W F) \text{ ö } [\mathcal{C}, \mathrm{Set}]\Big(W, \mathcal{D}(d, F*)\Big)$$

leading us to the final result:

$$\mathcal{D}(d, \lim^{\mathcal{B}(e, P*)} F)$$

Since d was arbitrary, we can use the Yoneda trick to conclude that:

$$(\mathrm{Ran}_P F)e = \lim^{\mathcal{B}(e, P*)} F$$

This formula becomes the definition of the right Kan extension in the enriched setting.

Similarly, the left Kan extension can be defined as a weighted colimit:

$$(\mathrm{Lan}_P F)e = \mathrm{colim}^{\mathcal{B}(P*, e)} F$$

Exercise 20.7.1. *Derive the formula for the left Kan extension for ordinary categories.*

20.8 Useful Formulas

- Yoneda lemma:

$$\int_x [\mathcal{C}(c, x), Fx] \text{ ö } Fc$$

- Weighted limit:
$$\mathcal{C}(x, \lim^{W} D) \text{ ö } [\mathcal{J}, \mathcal{V}]\Big(W, \mathcal{C}(x, D*)\Big)$$

- Weighted colimit:
$$\mathcal{C}(\mathrm{colim}^{W} D, x) \text{ ö } [\mathcal{J}^{op}, \mathcal{V}]\Big(W, \mathcal{C}(D*, x)\Big)$$

- Right Kan extension:
$$(\mathrm{Ran}_P F)e = \lim^{\mathcal{B}(e, P*)} F$$

- Left Kan extension:
$$(\mathrm{Lan}_P F)e = \mathrm{colim}^{\mathcal{B}(P*, e)} F$$