The Dao of Functional Programming

Bartosz Milewski

(Last updated: 2024 年 8 月 24 日)

目录

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 $a \in S$ （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 $f : S \to T$ 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 C 或 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：

```
apply :: (a -> b, a) -> b
apply (f, x) = f x
```

Chapter 1

Clean Slate
白板

编程始于类型（Types）和函数（Functions）。你可能对类型和函数是什么有一些先入为主的观念：抛弃它们！它们会混淆你的思维。

不要考虑事物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。我们不应该依附于此。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

引用老子的话[1]：“可描述的类型不是永恒的类型。”换句话说，类型（Type）是一种原始概念。它无法被定义。

我们可以不称它为“类型”，而是称为“对象”（Object）或“命题”（Proposition）。在数学的不同领域（如类型论、范畴论和逻辑）中，这些词汇被用来描述它。

可能有不止一种类型，所以我们需要一种方式来命名它们。我们可以通过指向它们来做到这一点，但为了有效地与他人交流，我们通常为它们命名。因此，我们会谈论类型 a、b、c；或者 `Int`、`Bool`、`Double` 等等。这些只是名称。

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（Arrows）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。目标可以与源相同，在这种情况下，箭头会回环。

类型之间的箭头称为“函数”（Function）。对象之间的箭头称为“态射”（Morphism）。命题之间的箭头称为“蕴涵”（Entailment）。这些只是用来描述不同数学领域中的箭头的

[1]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》（Tao Te Ching 或 Daodejing）的半传奇作者，这是一部关于道教的经典著作。

词汇。你可以互换使用它们。

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。

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。

$$a \xrightarrow{f} b$$

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。

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

1.2 Yin and Yang
阴阳

一个对象由它的连接来定义。箭头（Arrow）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。

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。

一个对每个对象都有唯一出射箭头的对象称为“初始对象”（Initial Object）。

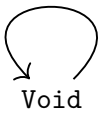
它的对偶是一个从每个对象都有唯一入射箭头的对象。它被称为“终端对象”（Terminal Object）。

在数学中，初始对象通常表示为 0，终端对象表示为 1。

从 0 到任何对象 a 的箭头表示为 $¡_a$，通常缩写为 ¡。

从任何对象 a 到 1 的箭头表示为 $!_a$，通常缩写为 !。

初始对象是一切的源头。作为一种类型，它在 Haskell 中被称为 `Void`。它象征着一切事物源于其中的混沌。由于有一个从 `Void` 到一切事物的箭头，也有一个从 `Void` 到它自身的箭头。



因此，`Void` 产生了 `Void` 和其他一切。

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。

在逻辑中，终端对象象征着终极真理，用 $\boldsymbol{T}$ 或 ⊤ 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，⊤ 都是正确的。

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 ⊥ 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。

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。

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。

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。

在 Haskell 中有无限多的类型，并且从 `Void` 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：`absurd`。

编程	范畴论	逻辑
类型	对象	命题
函数	态射（箭头）	蕴涵
`Void`	初始对象，0	False ⊥
`()`	终端对象，1	True ⊤

1.3 Elements
元素

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。

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 `Void` 没有结构。

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。

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。

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。

在范畴论中，如果 x 是一个箭头，我们说它是 a 的 *全局元素*（Global Element）。

$$1 \xrightarrow{x} a$$

我们经常简单地称之为元素（省略“全局”）。

在类型论中，x: A 意味着 x 是 A 类型的。

在 Haskell 中，我们使用双冒号符号：

```
x :: A
```

（Haskell 对具体类型使用大写名称，对类型变量使用小写名称。）

我们说 `x` 是 `A` 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 `A` 的全局元素。[2]

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。

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

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。

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

任何两个对象之间的箭头形成一个集合[3]。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。

在编程中，我们谈论从 `a` 到 `b` 的函数的*类型*。在 Haskell 中我们写：

```
f :: a -> b
```

意思是 `f` 是“从 `a` 到 `b` 的函数”类型。在这里，`a->b` 只是我们给这个类型起的名字。

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 `a` 到 `b` 的箭头集合。

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 `a` 和 `b` 的关系。我们还没有工具来做到这一点，但我们会到达那里。

目前，让我们记住以下区别：一方面，我们有连接两个对象 `a` 和 `b` 的箭头。这些箭头形成一个集合。另一方面，我们有一个从 `a` 到 `b` 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 `a->b` 的对象的箭头。

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*（Exponential Object）并将其写作 b^a（源对象在指数中）。因此，以下语句：

---

[2] Haskell 类型系统区分 `x :: A` 和 `x :: () -> A`。然而，在范畴语义中，它们表示相同的东西。

[3] 严格来说，这只在*局部小*范畴中成立。

```
f :: a -> b
```

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。

给我看 B^A 的一个元素，我就知道 B 是从 A 推导出来的。

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

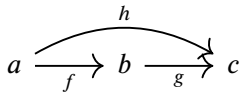
Composition
组合

2.1 Composition
组合

编程是关于组合（Composition）的。借用维特根斯坦（Wittgenstein）的话，可以说：“对于不能分解的东西，不应该谈论。”这不是一种禁止，而是一种事实陈述。研究、理解和描述的过程与分解的过程是一样的，我们的语言也反映了这一点。

我们构建对象（Objects）和箭头（Arrows）词汇的原因正是为了表达组合的思想。

给定一个从 a 到 b 的箭头 f 和一个从 b 到 c 的箭头 g，它们的组合是一个直接从 a 到 c 的箭头。换句话说，如果有两个箭头，其中一个的目标与另一个的源相同，我们总是可以将它们组合起来得到第三个箭头。



在数学中，我们用一个小圆圈来表示组合：

$$h = g \circ f$$

我们读作：“h 是 f 之后的 g。”选择“之后”一词暗示了动作的时间顺序，这在大多数情况下是一种有用的直觉。

组合的顺序可能看起来是反向的，但这是因为我们认为函数是在右边接受参数的。在 Haskell 中，我们用点号代替圆圈：

```
h = g . f
```

这就是每个程序的简要概括。为了完成 `h`，我们将其分解为更简单的问题，即 `f` 和 `g`。这些问题可以进一步分解，依此类推。

现在假设我们能够将 g 自身分解为 $j \circ k$。我们有：

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

我们希望这个分解与以下分解相同：

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

我们希望能够说我们已经将 h 分解为三个更简单的问题：

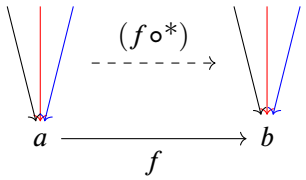
$$h = j \circ k \circ f$$

而不必跟踪哪个分解是先进行的。这就是组合的**结合性**（Associativity），从现在起我们将假设它。

组合是箭头两种映射的来源，分别称为前置组合（Pre-composition）和后置组合（Post-composition）。

当你**后置组合**一个箭头 h 和一个箭头 f 时，它生成箭头 $f \circ h$（箭头 f 是在箭头 h *之后*应用的）。当然，你只能将 h 与源是 h 目标的箭头进行后置组合。通过 f 的后置组合写作 $(f \circ *)$，为 h 留下一个空位。老子会说，“后置组合的有用性来自于不存在的东西。”

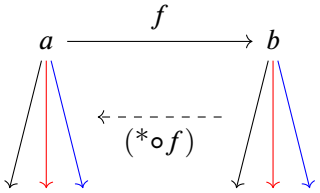
因此，一个箭头 $f : a \to b$ 诱导出箭头的映射 $(f \circ *)$，它将探测 a 的箭头映射到探测 b 的箭头。



由于对象没有**内部**结构，当我们说 f 将 a 转换为 b 时，这正是我们所指的。

后置组合让我们可以将焦点从一个对象转移到另一个对象。

相对地，你可以通过 f 进行**前置组合**，或者应用 $(* \circ f)$ 到源自 b 的箭头，将它们映射到源自 a 的箭头（注意方向的变化）。



前置组合让我们可以将视角从观察者 b 转移到观察者 a。

前置和后置组合是箭头的映射。由于箭头形成集合，这些映射是集合之间的**函数**（Functions）。

另一种看待前置和后置组合的方式是，它们是两个孔的组合运算符 (*∘*) 的部分应用的结果，其中我们可以用一个固定的箭头预填一个孔或另一个孔。

在编程中，出射箭头被解释为从其源提取数据。入射箭头被解释为生成或构造目标。出射箭头定义接口，入射箭头定义构造器。

对于 Haskell 程序员，以下是作为高阶函数的后置组合的实现：

```
postCompWith :: (a -> b) -> (x -> a) -> (x -> b)
postCompWith f = \h -> f . h
```

类似地，前置组合的实现如下：

```
preCompWith :: (a -> b) -> (b -> x) -> (a -> x)
preCompWith f = \h -> h . f
```

做以下练习来证明自己，焦点和视角的转换是可组合的。

Exercise 2.1.1. 假设你有两个箭头，$f\colon a \to b$ 和 $g\colon b \to c$。它们的组合 $g \circ f$ 诱导出箭头的映射 $((g \circ f) \circ *)$。证明如果你先应用 $(f \circ *)$ 然后再应用 $(g \circ *)$，结果是相同的。符号表示为：

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

提示：选择一个任意对象 x 和一个箭头 $h\colon x \to a$，看看是否得到相同的结果。注意，这里的 $\circ$ 是多义词。在右侧，它表示两个后置组合之间的常规函数组合。

Exercise 2.1.2. 确信前一个练习中的组合是结合的。提示：从三个可组合的箭头开始。

Exercise 2.1.3. 证明前置组合 $(* \circ f)$ 是可组合的，但组合的顺序是反向的：

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

2.2 Function application
函数应用

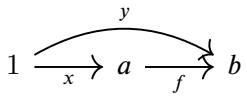
我们准备好写第一个程序了。有句谚语说："千里之行，始于足下。"考虑从 1 到 b 的旅程。我们的一小步可以是从终端对象 1 到某个 a 的箭头。它是 a 的一个元素。我们可以将其写为：

$$1 \xrightarrow{x} a$$

旅程的其余部分是箭头：

$$a \xrightarrow{f} b$$

这两个箭头是可组合的（它们共享对象 a 作为中间对象），它们的组合是从 1 到 b 的箭头 y。换句话说，y 是 b 的一个元素：



我们可以将其写为：

$$y = f \circ x$$

我们使用 f 将 a 的一个元素映射到 b 的一个元素。由于这是我们经常做的事情，我们称之为函数 f 对 x 的应用，并使用简写符号

$$y = f\,x$$

让我们将其翻译为 Haskell。我们从一个类型为 a 的元素 x 开始（简写为 `x :: ()-> a`）

```
x :: a
```

我们将一个函数 f 声明为从 a 到 b 的“箭头对象”的元素

```
f :: a -> b
```

其中的理解（稍后将详细说明）是它对应于从 `a` 到 `b` 的箭头。结果是 b 的一个元素

```
y :: b
```

定义为：

```
y = f x
```

我们称之为将函数应用于参数，但我们能够完全用函数组合来表达它。（注意：在其他编程语言中，函数应用需要使用括号，例如 `y = f(x)`。）

2.3 Identity
恒等

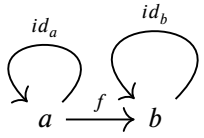
你可以将箭头视为表示变化：对象 a 变为对象 b。一个回环的箭头表示对象自身的变化。但是，变化有它的对偶：没有变化，无为，或者用老子的话来说，无为（Wu Wei）。

每个对象都有一个特殊的箭头，称为恒等（Identity），它使对象保持不变。这意味着，当你将这个箭头与任何其他箭头组合时，无论是入射的还是出射的，你都会得到那个箭头。作为一种动作，恒等箭头什么都不做，也不需要时间。

对象 a 上的恒等箭头称为 id_a。因此，如果我们有一个箭头 $f\colon a \to b$，我们可以在任一侧与恒等箭头组合

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

或用图像表示：



我们可以轻松检查恒等箭头对元素的作用。让我们取一个元素 $x\colon 1 \to a$ 并将其与 id_a 组合。结果是：

$$id_a \circ x = x$$

这意味着恒等箭头使元素保持不变。

在 Haskell 中，我们使用相同的名称 `id` 表示所有的恒等函数（我们不会用其作用的类型下标）。上面的方程式，它指定了 `id` 对元素的作用，直接翻译为：

```
id x = x
```

并且它成为了函数 `id` 的定义。

我们之前已经看到，初始对象和终端对象都有唯一的回环箭头。现在我们说，每个对象都有一个回环的恒等箭头。记住我们之前说过的关于唯一性的内容：如果你能找到两个这样的东西，那么它们一定是相等的。我们必须得出结论，我们之前提到的这些唯一的回环箭头必须是恒等箭头。我们现在可以标注这些图：

在逻辑中，恒等箭头对应于自明的真理。它是一个简单的证明，证明“如果 a 为真，那么 a 为真。”这也被称为恒等规则（Identity Rule）。

如果恒等什么都不做，那为什么我们还要关心它？想象你正在进行一趟旅行，组合了一些箭头，发现自己回到了起点。问题是：你做了什么，还是你浪费了时间？回答这个问题的唯一方法是将你的路径与恒等箭头进行比较。

有些往返旅行会带来变化，而有些则不会。

更重要的是，恒等箭头将允许我们比较对象。它们是同构（Isomorphism）定义的重要组成部分。

Exercise 2.3.1. $(id_a \circ *)$ 对终止于 a 的箭头做了什么？$(* \circ id_a)$ 对源自 a 的箭头做了什么？

2.4 Monomorphisms
单态射

考虑函数 `even`，它测试其输入是否能被 2 整除：

```
even :: Int -> Bool
```

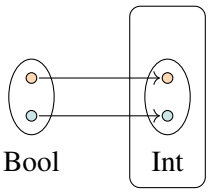
这是一个多对一的函数：所有偶数映射到 `True`，所有奇数映射到 `False`。输入的大部分信息被丢弃了，我们只对其偶性感兴趣，而不是它的实际值。通过丢弃信息，我们得到了抽象[1]。函数（以及后来的函子）体现了抽象。

与此相反的是函数 `injectBool`：

```
injectBool :: Bool -> Int
injectBool b = if b then 1 else 0
```

这个函数不会丢弃信息。你可以从它的结果中恢复它的参数。

不会丢弃信息的函数也很有用：它们可以被认为是将其源注入到其目标中。你可以将源的类型想象为嵌入到目标中的一种形状。在这里，我们将一个包含两个元素的 `Bool` 形状嵌入到整数类型中。

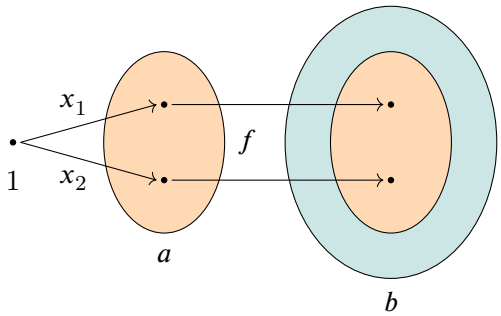


单射（Injective Functions），或单射（Injections），被定义为总是将不同的值分配给不同的参数。换句话说，它们不会将多个元素压缩为一个。

这里有另一种稍微复杂的表达方式：单射仅当两个元素相等时才将它们映射为一个。

我们可以将这个定义翻译为范畴语言，将“元素”替换为从终端对象来的箭头。我们会说，$f : a \to b$ 是单射的，如果对于任何一对全局元素 $x_1 : 1 \to a$ 和 $x_2 : 1 \to a$，有以下蕴涵成立：

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



这个定义的问题是，并非每个范畴都有一个终端对象。一个更好的定义将用任意形状替换全局元素。因此，单射的概念被推广为单态射（Monomorphism）。

[1] 抽象（Abstract）字面意思是“抽离”。

一个箭头 $f\colon a \to b$ 是单态射（Monomorphic）的，如果对于任何选择的对象 c 和一对箭头 $g_1\colon c \to a$ 和 $g_2\colon c \to a$，有以下蕴涵成立：

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

要证明箭头 $f\colon a \to b$ 不是单态射，只需找到一个反例：a 中的两个不同形状，f 将它们映射到 b 中相同的形状。

单态射或简称“单态”（Monos）通常用特殊的箭头表示，如 $a \hookrightarrow b$ 或 $a \rightarrowtail b$。

在范畴论中，对象是不可分的，因此我们只能通过箭头谈论子对象。我们说，单态射 $a \hookrightarrow b$ 选择了 b 中形状为 a 的子对象。

Exercise 2.4.1. 证明从终端对象来的任何箭头都是单态射。

2.5 Epimorphisms
满态射

函数 `injectBool` 是单射（因此是单态射），但它仅覆盖了其目标的一小部分——在无穷多个整数中只有两个。

```
injectBool :: Bool -> Int
injectBool b = if b then 1 else 0
```

相比之下，函数 `even` 覆盖了整个 `Bool`（它可以生成 `True` 和 `False`）。覆盖其整个目标的函数称为满射（Surjection）。

为了推广单射，我们使用了额外的映射输入。为了推广满射，我们将使用映射输出。满射的范畴对应物称为满态射（Epimorphism）。

一个箭头 $f\colon a \to b$ 是满态射（Epimorphism）的，如果对于任何选择的对象 c 和一对箭头 $g_1\colon b \to c$ 和 $g_2\colon b \to c$，有以下蕴涵成立：

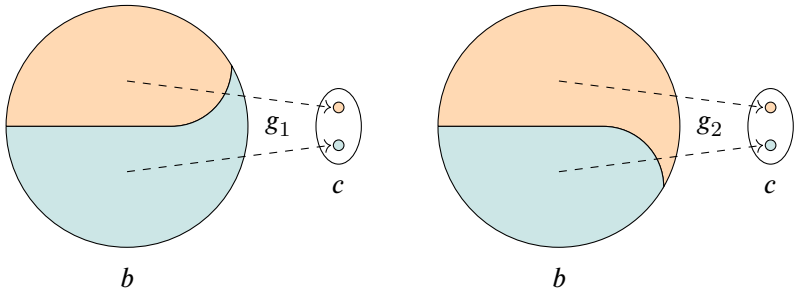
$$g_1 \circ f = g_2 \circ f \implies g_1 = g_2$$

相反，要证明 f 不是满态射，只需选择一个对象 c 和两个不同的箭头 g_1 和 g_2，它们在与 f 预组合时一致。

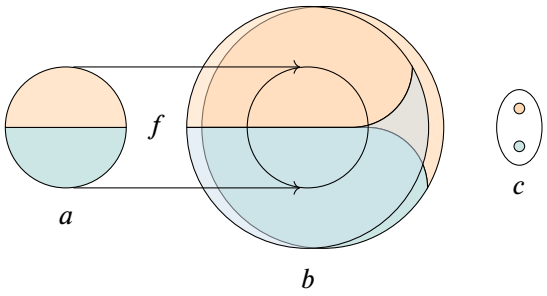
为了深入理解这个定义，我们必须将映射输出可视化。正如映射到对象可以被认为是定义一种形状，映射出对象可以被认为是定义该对象的属性。

当处理集合时，这一点很清楚，尤其是当目标集是有限时。你可以将目标集的一个元素视为定义一种颜色。源中的所有元素映射到该元素时都被“涂”上一种特定的颜色。例如，函数 `even` 将所有偶数整数涂上 `True` 的颜色，将所有奇数整数涂上 `False` 的颜色。

在满态射的定义中，我们有两个这样的映射，g_1 和 g_2。假设它们之间仅有轻微差异。b 的大部分被它们两者相似地涂上了颜色。



如果 f 不是满态射，可能它的像只覆盖了由 g_1 和 g_2 相似地涂上的部分。然后这两个箭头在与 f 预组合时在 a 上的表现一致，即使它们在整体上是不同的。



当然，这只是一个插图。在实际范畴中是不能窥探对象内部的。

满态射或简称“满态”（Epis）通常用特殊箭头表示，如 $a \twoheadrightarrow b$。

在集合中，同时是单射和满射的函数称为双射（Bijection）。它在两个集合的元素之间提供了一个一对一的可逆映射。在范畴论中，这个角色由同构（Isomorphisms）扮演。然而，一般来说，并不是真的同时是单态射和满态射的箭头就是同构的。

Exercise 2.5.1. 证明到终端对象的任何箭头都是满态射。

Chapter 3

Isomorphisms
同构

3.1 Isomorphic Objects
同构对象

当我们说：

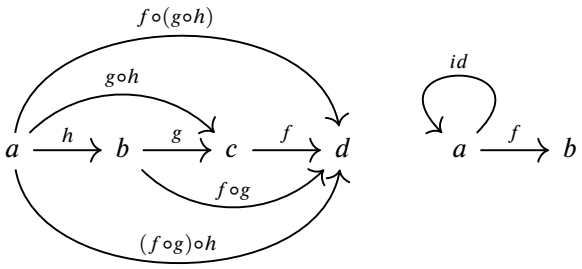
$$f \circ (g \circ h) = (f \circ g) \circ h$$

或者：

$$f = f \circ id$$

我们是在断言箭头的相等性。左边的箭头是一个操作的结果，而右边的箭头是另一个操作的结果。但结果是相等的。

我们通常通过绘制交换图来说明这种相等性，例如：



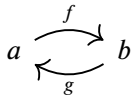
因此我们比较箭头的相等性。

我们不比较对象的相等性[1]。我们将对象视为箭头的汇合点，因此如果我们想比较两个对象，我们会看箭头。

最简单的两个对象之间的关系是箭头。

[1]半开玩笑地说，在范畴论中调用对象的相等性被认为是“邪恶的”。

最简单的往返是两个方向相反的箭头的组合。



可能有两种往返方式。一种是从 a 到 a 的 $g \circ f$。另一种是从 b 到 b 的 $f \circ g$。

如果它们两个都生成身份箭头，那么我们说 g 是 f 的*逆元*：

$$g \circ f = id_a$$

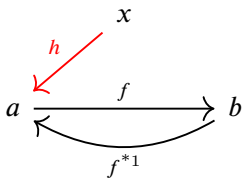
$$f \circ g = id_b$$

我们写作 $g = f^{*1}$（读作 f 的*逆元*）。箭头 f^{*1} 撤销了箭头 f 的作用。

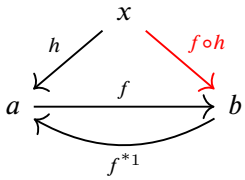
这样的一对箭头被称为*同构*，这两个对象被称为*同构的*。

同构的存在告诉我们它连接的两个对象是什么？

我们说过，对象通过与其他对象的相互作用来描述。因此，让我们考虑从观察者 x 的角度看这两个同构对象的样子。假设有一个箭头 h 从 x 到 a。



有一个对应的箭头从 x 到 b。它只是 $f \circ h$ 的组合，或者 $(f \circ *)$ 对 h 的作用。



类似地，对于任何探测 b 的箭头，有一个对应的探测 a 的箭头。它是 $(f^{*1} \circ *)$ 的作用。

我们可以使用映射 $(f \circ *)$ 和 $(f^{*1} \circ *)$ 在 a 和 b 之间来回移动焦点。

我们可以组合这两个映射（见习题2.1.1）以形成一个往返。结果等同于我们应用复合 $((f^{*1} \circ f) \circ *)$。但这等于 $(id_a \circ *)$，如我们从习题2.3.1中所知，它保持箭头不变。

类似地，由 $f \circ f^{*1}$ 引起的往返保持箭头 $x \to b$ 不变。

这在两个箭头组之间创建了一个“配对系统”。想象每个箭头向它的配对发送消息，这由 f 或 f^{*1} 决定。然后每个箭头会收到一条消息，而且那将是来自它配对的消息。没有箭头会被遗漏，也没有箭头会收到多于一条消息。数学家称这种配对系统为*双射*或一对一的对应。

因此，从 x 的角度来看，箭头与箭头地，两个对象 a 和 b 看起来完全相同。就箭头而言，这两个对象之间没有区别。

两个同构对象具有完全相同的属性。

特别地，如果你将 x 替换为终端对象 1，你会看到这两个对象具有相同的元素。对于每个元素 $x\colon 1 \to a$，有一个对应的元素 $y\colon 1 \to b$，即 $y = f \circ x$，反之亦然。同构对象的元素之间存在双射。

这些不可区分的对象被称为*同构的*，因为它们具有"相同的形状"。你看过一个，就等于看过了所有。

我们将这个同构写作：

$$a \ddot{\circ} b$$

当涉及到对象时，我们用同构代替相等。

在编程中，两个同构的类型具有相同的外部行为。一个类型可以用另一个类型来实现，反之亦然。一个类型可以被另一个类型替换而不改变系统的行为（可能除外性能）。

在经典逻辑中，如果 B 从 A 推导出来，且 A 从 B 推导出来，那么 A 和 B 在逻辑上是等价的。我们经常说"当且仅当"A 为真时，B 为真。然而，与逻辑和类型理论之间的前几次平行不同，如果你认为证明是相关的，这次的等价并不那么直截了当。实际上，它导致了一个新的基础数学分支的发展，称为同伦类型论，简称 HoTT。

Exercise 3.1.1. 提出一个论点，证明从两个同构对象出发的箭头之间存在双射。绘制相应的图。

Exercise 3.1.2. 证明每个对象都与自身同构。

Exercise 3.1.3. 如果有两个终端对象，证明它们是同构的。

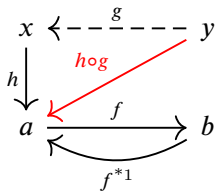
Exercise 3.1.4. 证明前一个习题中的同构是唯一的。

3.2 Naturality
自然性

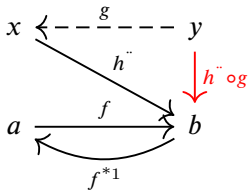
我们已经看到，当两个对象同构时，我们可以使用后组合来在它们之间切换焦点：要么是 $(f \circ *)$，要么是 $(f^{*1} \circ *)$。

相反，要在不同的观察者之间切换，我们会使用前组合。

的确，从 x 到 a 的箭头 h 与从 y 到同一对象的箭头 $h \circ g$ 相关。



类似地，一个从 x 探测 b 的箭头 $h^{¨}$ 对应于一个从 y 探测 b 的箭头 $h^{¨}\circ g$。



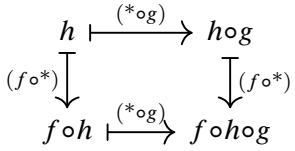
在这两种情况下，我们通过应用前组合 $(*\circ g)$ 来改变视角从 x 到 y。

重要的观察是，视角的改变保持了由同构建立的配对系统。如果从 x 的角度来看两个箭头是配对的，那么从 y 的角度来看它们仍然是配对的。这就像说，先用 g 预组合（切换视角）然后再用 f 后组合（切换焦点），或者先用 f 后组合然后再用 g 预组合是无关紧要的。符号上，我们写作：

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

我们称之为自然性条件。

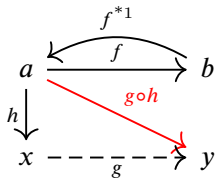
当你将这个等式应用于一个态射 $h\colon x \to a$ 时，它的意义就会揭示出来。两边都简化为 $f\circ h\circ g$。



在这里，自然性条件是由于结合性而自动满足的，但我们很快就会看到它在不太平凡的情况下的推广。

箭头被用来传播同构的信息。自然性告诉我们，所有的对象都获得了它的一致视图，而不管路径如何。

我们也可以逆转观察者和被观察者的角色。例如，使用一个箭头 $h\colon a \to x$，对象 a 可以从 x 探测任意对象。如果有一个箭头 $g\colon x \to y$，它可以将焦点切换到 y。切换视角到 b 是通过 f^{*1} 的前组合完成的。



同样，我们有自然性条件，这次是从同构对的角度来看：

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

在范畴论中，当我们需要从一个地方移动到另一个地方时，通常需要两个步骤。这里，前组合和后组合的操作可以按任意顺序完成——我们说它们交换。但通常情况下，执行步骤的顺序会导致不同的结果。我们经常施加交换条件，并说如果这些条件成立，一个操作与另一个操作是兼容的。

Exercise 3.2.1. 证明 f^{*1} 的自然性条件的两边在作用于 h 时简化为：

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

3.3 Reasoning with Arrows
使用箭头推理

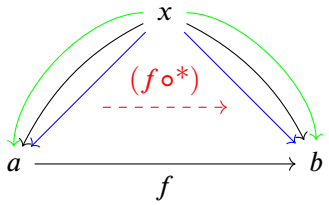
Master Yoneda says: “At the arrows look!”

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.

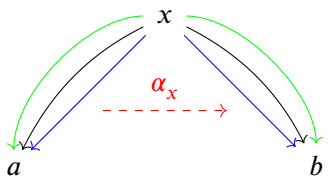
If you want to see if two objects are isomorphic, at the arrows look!

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



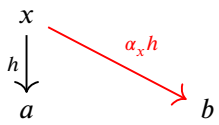
函数 $(f\circ *)$ 将每个箭头 $h\colon x \to a$ 映射到一个箭头 $f\circ h\colon x \to b$。它的逆 $(f^{*1}\circ *)$ 将每个箭头 $h^{¨}\colon x \to b$ 映射到一个箭头 $(f^{*1}\circ h^{¨})$。

假设我们不知道这些对象是否同构，但我们知道在每个对象 x 和箭头 $h\colon x \to a$ 之间存在一个可逆映射 α_x，即在对象 a 和 b 之间的箭头之间存在双射。



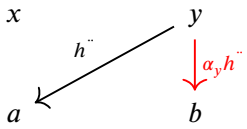
之前，箭头的双射是由同构 f 生成的。现在，箭头的双射是由 α_x 给定的。这是否意味着这两个对象是同构的？我们能从映射 α_x 的族中构造同构 f 吗？答案是肯定的，只要族 α_x 满足自然性条件。

这是 α_x 对特定箭头 h 的作用。



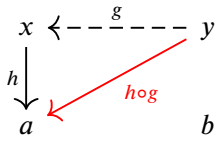
这种映射及其逆 α_x^{*1} 将箭头从 $x \to b$ 映射到箭头 $x \to a$，如果确实存在同构 f，它将扮演 $(f \circ *)$ 和 $(f^{*1} \circ *)$ 的角色。映射族 α 描述了一种“人工”的方式来从 a 切换焦点到 b。

这是从另一个观察者 y 的角度来看相同的情况：



注意，y 使用的是同一族中的另一个映射 α_y。

只要存在一个箭头 $g\colon y \to x$，这两个映射 α_x 和 α_y 就会纠缠在一起。在这种情况下，前组合 $(* \circ g)$ 允许我们切换视角到 y（注意方向）。



我们将焦点切换与视角切换分离。前者由 α 完成，后者由前组合完成。自然性要求这两者之间的兼容性。

确实，从某个 h 开始，我们可以应用 $(* \circ g)$ 切换到 y 的视角，然后应用 α_y 切换焦点到 b：

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

或者我们可以先让 x 使用 α_x 切换焦点到 b，然后使用 $(* \circ g)$ 切换视角：

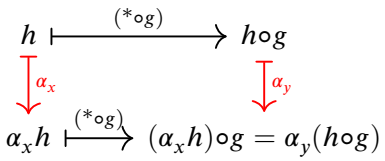
$$(* \circ g) \circ \alpha_x$$

在这两种情况下，我们最终都从 y 的角度看到了 b。我们之前已经做过这个练习，当时我们有一个 a 和 b 之间的同构，我们发现结果是相同的。我们称之为自然性条件。

如果我们希望 α 给我们一个同构，我们必须强加等效的自然性条件：

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

当作用于箭头 $h\colon x \to a$ 时，我们希望这个图交换：

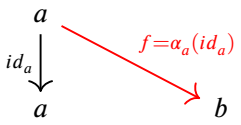


这样我们就知道，用 $(f \circ *)$ 替换所有 α 是可行的。但这样的 f 是否存在？我们能从 α 中重建 f 吗？答案是肯定的，我们将使用 Yoneda 技巧来实现这一点。

由于 α_x 是为每个对象 x 定义的，它也可以为 a 本身定义。根据定义，α_a 将一个从 $a \to a$ 的态射映射到 $a \to b$ 的态射。我们确定至少存在一个从 $a \to a$ 的态射，即身份态射 id_a。事实证明，我们正在寻找的同构 f 由以下公式给出：

$$f = \alpha_a(id_a)$$

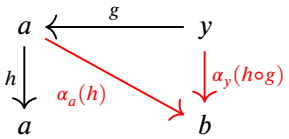
或者形象地表示为：



让我们验证一下。如果 f 确实是我们的同构，那么对于任意 x，α_x 应该等于 $(f \circ *)$。为此，我们将自然性条件替换为 a。我们得到：

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

如以下图所示：



由于 h 的源和目标都是 a，该等式必须在 $h = id_a$ 时也成立：

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

但是 $id_a \circ g$ 等于 g，而 $\alpha_a(id_a)$ 是我们的 f，所以我们得到：

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

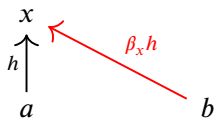
换句话说，对于每个对象 y 和每个态射 $g\colon y \to a$，$\alpha_y = (f \circ *)$。

注意，尽管 α_x 是为每个 x 和每个从 $x \to a$ 的箭头单独定义的，但事实证明它完全由一个单一的身份箭头的值所决定。这就是自然性的力量！

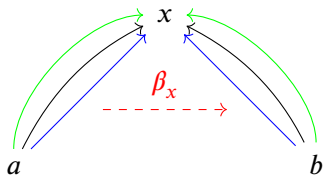
Reversing the Arrows
反转箭头

正如老子所说，观察者与被观察者之间的二元对立是无法完成的，除非允许观察者与被观察者互换角色。

再次，我们想证明两个对象 a 和 b 是同构的，但这次我们想将它们视为观察者。箭头 $h\colon a \to x$ 从 a 的角度探测任意对象 x。之前，当我们知道这两个对象是同构的时，我们可以使用 $(* \circ f^{*1})$ 切换视角到 b。这次我们有一个变换 β_x。它在箭头 $x \to a$ 和 $x \to b$ 之间建立了双射。



如果我们想观察另一个对象 y，我们将使用 β_y 在 a 和 b 之间切换视角，依此类推。



如果两个对象 x 和 y 之间有一个箭头 $g\colon x \to y$，那么我们也可以选择使用 $(g \circ *)$ 切换焦点。如果我们想同时切换视角和切换焦点，有两种方法可以做到。自然性要求结果相同：

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

的确，如果我们用 $(* \circ f^{*1})$ 替换 β，我们将恢复同构的自然性条件。

Exercise 3.3.1. 使用身份态射的技巧，从映射族 β 中恢复 f^{*1}。

Exercise 3.3.2. 使用前一个习题中的 f^{*1}，对于任意对象 y 和任意箭头 $g\colon a \to y$，求出 $\beta_y g$ 的值。

正如老子所说：要展示一个同构，定义一个在一万个箭头之间的自然变换往往比找到两个对象之间的一对箭头更容易。

Chapter 4

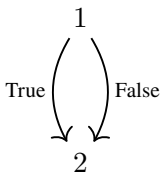
Sum Types
和类型

4.1 Bool
布尔类型

我们已经知道如何组合箭头。那么我们如何组合对象呢?

我们已经定义了 0（初始对象）和 1（终端对象）。那么 2 是什么？它就是 1 加 1。

一个 2 是一个具有两个元素的对象：从 1 发出的两个箭头。我们将其中一个箭头称为`True`，另一个箭头称为`False`。不要将这些名称与初始和终端对象的逻辑解释混淆。这两个只是箭头。

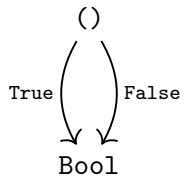


这个简单的概念可以立即用 Haskell 表达[1]，即定义一个类型，传统上称为`Bool`，以纪念其发明者 George Boole（1815-1864）。

```
data Bool where
True  :: () -> Bool
False :: () -> Bool
```

[1]这种定义风格在 Haskell 中称为广义代数数据类型或`GADTs`

它对应于相同的图（只是在 Haskell 中重命名了一些内容）：



正如我们之前所见，元素有一种简写表示法，因此这是一个更紧凑的版本：

```
data Bool where
True  :: Bool
False :: Bool
```

现在我们可以定义Bool类型的一个项，例如

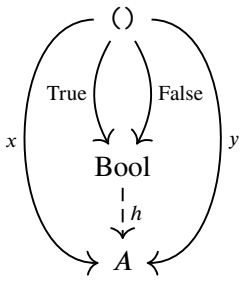
```
x :: Bool
x = True
```

第一行声明x是Bool的一个元素（实际上是一个函数()->Bool），第二行告诉我们它是两个元素中的哪个。

我们在Bool定义中使用的True和False函数被称为*数据构造器*。它们可以用来构造特定的项，就像上面的例子一样。顺便提一下，在 Haskell 中，函数名称通常以小写字母开头，除非它们是数据构造器。

我们对Bool类型的定义仍然不完整。我们知道如何构造一个Bool项，但我们不知道如何使用它。我们必须能够定义从Bool发出的箭头——即从Bool到其他类型的*映射*。

第一个观察结果是，如果我们有一个从Bool到某个具体类型A的箭头h，那么我们自动会得到两个从单位对象到A的箭头，只需通过组合即可。以下两个（扭曲的）三角形是可交换的：



换句话说，每个Bool->A函数都会生成一对A的元素。

给定一个具体类型A：

```
h :: Bool -> A
```

我们有：

```
x = h True
y = h False
```

其中

```
x :: A
y :: A
```

注意，这里使用了应用函数到元素的简写表示法：

```
h True -- 意味着：h . True
```

我们现在准备通过添加一个条件来完成对Bool的定义，即从Bool到A的任何函数不仅生成，而且等价于一对A的元素。换句话说，一对元素唯一地决定了从Bool的函数。

这意味着我们可以以两种方式解释上面的图：给定h，我们可以很容易地得到x和y。但反过来也成立：一对元素x和y唯一地定义了h。

这里有一个双射在起作用。这次是元素对 (x, y) 与箭头 h 之间的一对一映射。

在 Haskell 中，h的定义封装在if, then, else结构中。给定

```
x :: A
y :: A
```

我们定义映射为

```
h :: Bool -> A
h b = if b then x else y
```

在这里，b是Bool类型的一个项。

一般来说，数据类型是通过引入规则创建的，并通过消除规则解构。Bool数据类型有两个引入规则，一个使用True，另一个使用False。if, then, else结构定义了消除规则。

考虑到上面对h的定义，我们可以检索用于定义它的两个项，这被称为计算规则。它告诉我们如何计算h的结果。如果我们用True调用h，结果是x；如果我们用False调用它，结果是y。

我们永远不要忘记编程的目的：将复杂问题分解为一系列更简单的问题。Bool的定义说明了这个想法。每当我们需要构造一个从Bool到某类型的映射时，我们将其分解为构造目标类型的一对元素的两个更小的任务。我们用两个更简单的问题换取了一个更大的问题。

4.2 Enumerations
枚举类型

在 0、1 和 2 之后是什么？一个具有三个数据构造器的对象。例如：

```
data RGB where
Red   :: RGB
Green :: RGB
Blue  :: RGB
```

如果您厌倦了冗长的语法，可以使用此类型定义的简写：

```
data RGB = Red | Green | Blue
```

此引入规则允许我们构造RGB类型的项，例如：

```
c :: RGB
c = Blue
```

要定义从RGB到其他类型的映射，我们需要更通用的消除模式。就像从Bool到某个类型的函数由两个元素决定一样，从RGB到A的函数由A的三个元素x, y, z决定。我们使用模式匹配语法来编写这样的函数：

```
h :: RGB -> A
h Red   = x
h Green = y
h Blue  = z
```

这只是一个函数，它的定义分为三种情况。

我们也可以使用相同的语法用于Bool，代替if, then, else：

```
h :: Bool -> A
h True  = x
h False = y
```

实际上，还有第三种编写相同内容的方法，即case语句：

```
h c = case c of
Red   -> x
Green -> y
Blue  -> z
```

甚至是：

```
h :: Bool -> A
h b = case b of
True  -> x
False -> y
```

编程时，您可以根据需要使用这些方式中的任何一种。

这些模式也适用于具有四个、五个或更多数据构造器的类型。例如，十进制数字是以下之一：

```
data Digit = Zero | One | Two | Three | ... | Nine
```

有一个巨大的 Unicode 字符枚举，称为`Char`。它们的构造器有特殊的名称：你在字符本身前后加上单引号，例如：

```
c :: Char
c = 'a'
```

正如老子所说，万物之模式，需时多年，故人们发明了通配符模式“下划线”，它匹配一切。

由于模式是按顺序匹配的，因此您应该将通配符模式作为系列中的最后一个：

```
yesno :: Char -> Bool
yesno c = case c of
'y' -> True
'Y' -> True
_   -> False
```

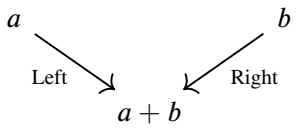
但我们为什么要止步于此呢？类型`Int`可以被看作是在 $*2^{29}$ 到 2^{29} 范围内的整数枚举（或更多，取决于实现）。当然，在这种范围内进行详尽的模式匹配是不可行的，但原理仍然成立。

实际上，类型`Char`用于 Unicode 字符，`Int`用于固定精度整数，`Double`用于双精度浮点数，以及其他几个类型，都内置于语言中。

这些不是无限类型。它们的元素可以被枚举，即使可能需要一万年。类型`Integer`则是无限的。

4.3 Sum Types
和类型

`Bool`类型可以看作是和类型 $2 = 1 + 1$。但没有什么能阻止我们用其他类型替换 1，甚至用不同的类型替换这两个 1。我们可以使用两个箭头定义一个新类型 $a + b$。我们称它们为`Left`和`Right`。定义图就是引入规则：

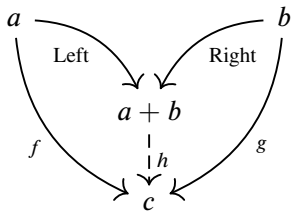


在 Haskell 中，类型 $a + b$ 被称为`Either a b`。类似于`Bool`，我们可以将其定义为：

```
data Either a b where
Left  :: a -> Either a b
Right :: b -> Either a b
```

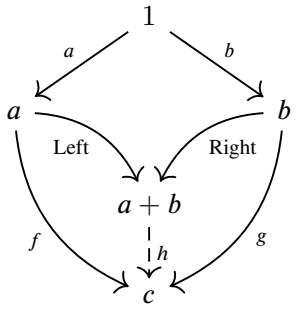
（注意小写字母用于类型变量。）

类似地，从 $a + b$ 到某个类型 c 的映射由以下可交换图决定：



给定一个函数 h，我们只需通过组合它与`Left`和`Right`即可得到一对函数 f 和 g。反过来，这对函数唯一地决定了 h。这是消除规则。

当我们想将此图转换为 Haskell 时，我们需要选择两种类型的元素。我们可以通过定义从终端对象的箭头来完成：



沿着这个图的箭头，我们得到：

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

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

Haskell 语法几乎逐字重复了这些方程，最终得到这个模式匹配语法用于定义`h`：

```
h :: Either a b -> c
h (Left  a) = f a
h (Right b) = g b
```

（再次注意小写字母用于类型变量，并且这些字母与这些类型的项相同。与人类不同，编译器不会因此感到困惑。）

你还可以从右向左阅读这些方程，这样你就会看到和类型的计算规则：定义`h`所使用的两个函数可以通过应用`h`到`(Left a)`和`(Right b)`来恢复。

你也可以使用`case`语法定义`h`：

```
h e = case e of
Left  a -> f a
Right b -> g b
```

那么，数据类型的本质是什么？不过是操纵箭头的一个方法。

4.4 Maybe
可能类型

一个非常有用的数据类型，`Maybe`被定义为 $1 + a$，对于任意 a。这是它在 Haskell 中的定义：

```
data Maybe a where
Nothing :: () -> Maybe a
Just    ::  a -> Maybe a
```

数据构造器`Nothing`是从单位类型的箭头，而`Just`从`a`构造`Maybe a`。`Maybe a`与`Either () a`是同构的。它也可以用简写符号定义：

```
data Maybe a = Nothing | Just a
```

`Maybe`主要用于对部分函数的返回类型进行编码：这些函数在某些参数值上未定义。在这种情况下，这些函数不会失败，而是返回`Nothing`。在其他编程语言中，部分函数通常使用异常（或核心转储）来实现。

4.5 Logic
逻辑

在逻辑中，命题 $A + B$ 被称为*逻辑或*。你可以通过提供 A 的证明或 B 的证明来证明它。任意一个证明都足够了。

如果你想证明 C 从 $A + B$ 中得出，你必须为两种可能性做好准备：一种是有人通过证明 A 来证明 $A + B$（此时 B 可能是假的），另一种是通过证明 B 来证明 $A + B$（此时 A 可能是假的）。在第一种情况下，你必须展示 C 如何从 A 中得出。在第二种情况下，你需要证明 C 从 B 中得出。这些正是 $A + B$ 消除规则中的箭头。

4.6 Cocartesian Categories
余笛卡尔范畴

在 Haskell 中，我们可以使用`Either`定义任意两种类型的和。一个类别中，如果所有和存在，并且初始对象也存在，我们称这个类别为*余笛卡尔*，而和则被称为*余积*。你可能注意到，和类型模仿了数字的加法。事实证明，初始对象扮演了零的角色。

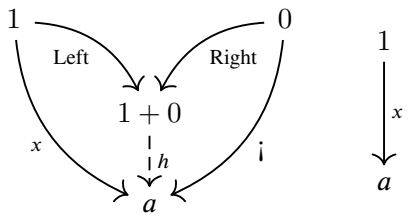
One Plus Zero

一加零

我们首先展示 $1 + 0 \ddot{o} 1$，这意味着终端对象与初始对象的和是同构于终端对象的。此类证明的标准程序是使用 Yoneda 技巧。由于和类型是通过映射出去定义的，我们应该比较它们的箭头。

Yoneda 论证说，如果两个对象有一个自然的双射 β_a，它在从它们到任意对象 a 的箭头集合之间建立，那么这两个对象是同构的。

让我们看看 $1 + 0$ 的定义以及它到任意对象 a 的映射。这个映射由一对 $(x, ¡)$ 定义，其中 x 是 a 的一个元素，$¡$ 是从初始对象到 a 的唯一箭头（在 Haskell 中是`absurd`函数）。



我们想在从 $1 + 0$ 发出的箭头与从 1 发出的箭头之间建立一对一的映射。箭头 h 由一对 $(x, ¡)$ 决定。由于只有一个 $¡$，因此 h 与 x 之间存在一个双射。

我们定义 β_a 将由一对 $(x, ¡)$ 定义的任何 h 映射到 x。反过来，β_a^{*1} 将 x 映射到一对 $(x, ¡)$。但这是一个自然变换吗?

要回答这个问题，我们需要考虑当我们从 a 切换到通过箭头 $g \colon a \to b$ 与 a 连接的某个 b 时会发生什么。我们现在有两个选项：

- 使 h 通过后组合 x 和 $¡$ 与 g 来切换焦点。我们得到一个新对 $(y = g \circ x, ¡)$。然后使用 β_b。
- 使用 β_a 将 $(x, ¡)$ 映射到 x。然后跟随 $(g \circ *)$ 进行后组合。

在这两种情况下，我们都得到相同的箭头 $y = g \circ x$。因此映射 β 是自然的。因此 $1 + 0$ 与 1 是同构的。

在 Haskell 中，我们可以定义构成同构的两个函数，但没有直接表达它们是彼此逆的方式。

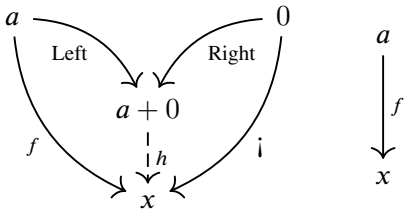
```
f :: Either () Void -> ()
f (Left ()) = ()
f (Right _) = ()

f_1 :: () -> Either () Void
f_1 _ = Left ()
```

函数定义中的下划线通配符表示该参数被忽略。函数`f`的第二个子句是多余的，因为`Void`类型没有项。

Something Plus Zero
某物加零

一个非常相似的论证可以用来证明 $a + 0 \cong a$。以下图解释了这一点。



我们可以通过实现一个（多态）函数`h`，它适用于任何类型`a`，将此论证转化为 Haskell。

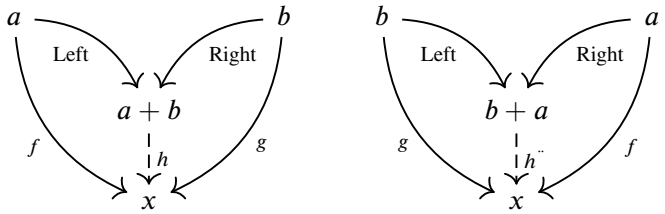
Exercise 4.6.1. *在 Haskell 中实现两个函数，这两个函数形成了`(Either a Void)`与`a`之间的同构。*

我们可以使用类似的论证来证明 $0 + a \cong a$，但和类型的更一般属性使得这点显得多余。

Commutativity
交换性

和类型定义中的图具有漂亮的左右对称性，这表明它满足交换规则，即 $a + b \cong b + a$。

让我们考虑公式两边的映射。你可以很容易地看到，对于左侧由一对 (f, g) 决定的每个 h，右侧有一个对应的 $h^{\sim}$，它由一对 (g, f) 给出。这就建立了箭头之间的双射。



Exercise 4.6.2. *证明上面定义的双射是自然的。提示：f 和 g 通过与 $k\colon x \to y$ 的后组合来改变焦点。*

Exercise 4.6.3. *在 Haskell 中实现见证`(Either a b)`与`(Either b a)`之间同构的函数。注意，这个函数本身是它的逆。*

Associativity
结合性

正如在算术中，我们定义的和是结合的：

$$(a + b) + c \cong a + (b + c)$$

编写左侧的映射很容易：

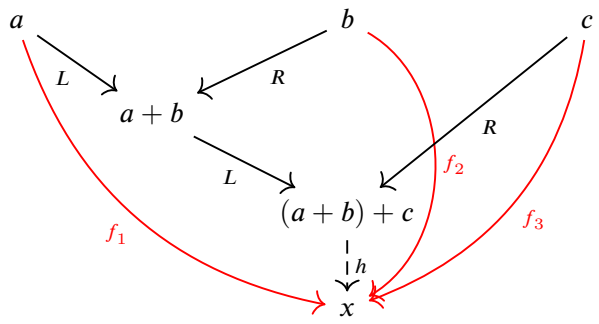
```
h :: Either (Either a b) c -> x
h (Left (Left a))  = f1 a
h (Left (Right b)) = f2 b
h (Right c)        = f3 c
```

注意使用嵌套模式，如(Left (Left a))等。该映射完全由三个函数定义。可以使用相同的函数来定义右侧的映射：

```
h' :: Either a (Either b c) -> x
h' (Left a)          = f1 a
h' (Right (Left b))  = f2 b
h' (Right (Right c)) = f3 c
```

这建立了两个定义映射的函数组之间的一对一映射。这种映射是自然的，因为所有的焦点转换都是通过后组合完成的。因此，两边是同构的。

这个代码也可以用图形化的形式展示。以下是同构左侧的图：



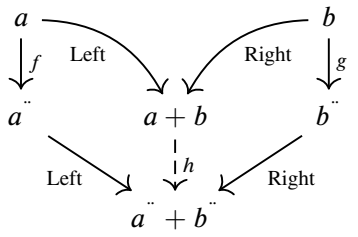
Functoriality
函子性

由于和是由映射出的性质定义的，所以很容易看出当我们改变焦点时会发生什么：它会随着定义积的箭头焦点的变化而“自然地”变化。但当我们移动这些箭头的源时，会发生什么呢？

假设我们有箭头，将 a 和 b 映射到某个 a'' 和 b''：

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

这些箭头与构造器 Left 和 Right 分别组合，定义了和之间的映射：



箭头对 (Left∘f, Right∘g) 唯一地定义了箭头 h: $a+b \to a^{¨} + b^{¨}$。

这种和的性质称为函子性。你可以想象它允许你在和的内部转换两个对象，并获得一个新的和。我们还说函子性允许我们提升一对箭头以操作和。

Exercise 4.6.4. 证明函子性保持组合。提示：取两条可组合的箭头，g: $b \to b^{¨}$ 和 $g^{¨}$: $b^{¨} \to b^{¨¨}$，证明应用 $g^{¨} \circ g$ 给出与首先应用 g 将 $a+b$ 变换为 $a+b^{¨}$，然后应用 $g^{¨}$ 将 $a+b^{¨}$ 变换为 $a+b^{¨¨}$ 相同的结果。

Exercise 4.6.5. 证明函子性保持身份。提示：使用 id_b 并证明它被映射到 id_{a+b}。

Symmetric Monoidal Category
对称幺半范畴

当一个孩子学习加法时，我们称之为算术。当一个成年人学习加法时，我们称之为余笛卡尔范畴。

无论我们是加数字，组合箭头，还是构造对象的和，我们都在重复使用将复杂事物分解为其简单组成部分的相同思想。

正如老子所说，当事物结合形成新事物时，如果操作是结合的，并且有一个中性元素，我们就知道如何处理万事万物。

我们定义的和类型满足这些属性：

$$a + 0 \cong a$$

$$a + b \cong b + a$$

$$(a + b) + c \cong a + (b + c)$$

并且它是函子性的。具有这种操作的范畴称为对称幺半范畴。当操作是和（余积）时，它被称为余笛卡尔。在下一章中，我们将看到另一种被称为笛卡尔的幺半结构，不带“余”字。

Chapter 5

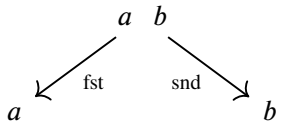
Product Types
积类型

我们可以使用和类型枚举给定类型的可能值，但这种编码可能是低效的。我们需要十个构造器才能编码从零到九的数字。

```
data Digit = Zero | One | Two | Three | ... | Nine
```

但是，如果我们将两个数字组合成一个数据结构，即一个两位数的十进制数，我们就能编码一百个数字。正如老子所说，只需四位数字就可以编码一万个数字。

这种以这种方式组合两种类型的数据类型称为积，或笛卡尔积。它的定义特性是消去规则：有两个箭头从 $a \quad b$ 出发；一个叫“fst”，指向 a，另一个叫“snd”，指向 b。它们被称为（笛卡尔）投影。它们让我们从积 $a \quad b$ 中检索 a 和 b。

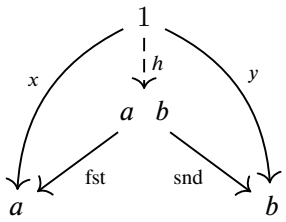


假设有人给了你一个积的元素，即一个从终端对象 1 到 $a \quad b$ 的箭头 h。你可以通过组合轻松检索一对元素：一个给定 a 的元素

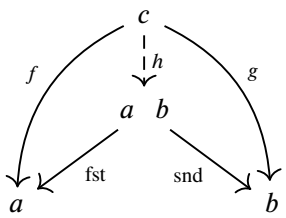
$$x = \text{fst} \circ h$$

和一个给定 b 的元素

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



实际上，给定一个从任意对象 c 到 $a \times b$ 的箭头，我们可以通过组合定义一对箭头 $f : c \to a$ 和 $g : c \to b$。



正如我们之前对和类型所做的那样，我们可以反过来使用这个图来定义积类型：我们规定一个函数对 f 和 g 与从 c 到 $a \times b$ 的映射是一对一对应的关系。这是积的引入规则。

特别地，从终端对象的映射在 Haskell 中用于定义积类型。给定两个元素，`a :: A`和`b :: B`，我们构造积

```
(a, b) :: (A, B)
```

内置的积语法就是这样：一对括号和中间的逗号。它既适用于定义两个类型的积`(A, B)`，也适用于数据构造器`(a, b)`，它接受两个元素并将它们配对在一起。

我们应该始终牢记编程的目的：将复杂问题分解为一系列简单的问题。在积的定义中，我们再次看到这一点。每当我们必须构造一个进入积的映射时，我们将其分解为两个较小的任务，即构造一个函数对，每个函数映射到积的一个组成部分。这就像说，为了实现一个返回一对值的函数，足以实现两个函数，每个函数返回对中一个元素。

Logic
逻辑

在逻辑中，积类型对应于逻辑合取。为了证明 $A \times B$（A 和 B），你需要同时提供 A 和 B 的证明。这些是指向 A 和 B 的箭头。消去规则表明，如果你有 $A \times B$ 的证明，那么你自动获得 A 的证明（通过 fst）和 B 的证明（通过 snd）。

Tuples and Records
元组和记录

正如老子所说，万物的积不过是一个具有万物投影的对象。

我们可以使用元组表示法在 Haskell 中形成任意积。例如，三种类型的积写作`(A, B, C)`。该类型的一个项可以由三个元素构造：`(a, b, c)`。

在数学家所称的“符号滥用”中，零个类型的积写作`()`，一个空元组，它恰好与终端对象或单位类型相同。这是因为积的行为非常像数字的乘法，终端对象扮演了“一”的角色。

在 Haskell 中，与其为所有元组定义单独的投影，我们使用模式匹配语法。例如，要从三元组中提取第三个组件，我们可以这样写

```
thrd :: (a, b, c) -> c
thrd (_, _, c) = c
```

我们对想忽略的组件使用通配符。

老子说，“命名是所有特殊事物的起源。”在编程中，如果不给它们命名，跟踪特定元组中组件的含义是困难的。记录语法允许我们为投影命名。这是以记录方式编写的积的定义：

```
data Product a b = Pair { fst :: a, snd :: b }
```

`Pair`是数据构造器，`fst`和`snd`是投影。

这是如何声明和初始化特定对的：

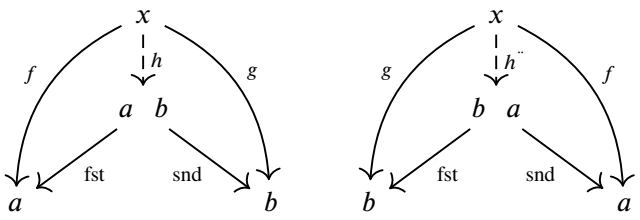
```
ic :: Product Int Char
ic = Pair 10 'A'
```

5.1 Cartesian Category
笛卡尔范畴

在 Haskell 中，我们可以定义任意两种类型的积。具有所有积和终端对象的范畴称为笛卡尔范畴。

Tuple Arithmetic
元组算术

积所满足的恒等式可以使用映射性质导出。例如，要证明 $a \times b \cong b \times a$，考虑以下两个图：



它们显示，对于任何对象 x，到 $a \times b$ 的箭头与到 $b \times a$ 的箭头是一对一对应的。这是因为这些箭头中的每一个都由相同的 f 和 g 对确定。

你可以检查自然性条件是否得到了满足，因为当你使用箭头 $k\colon x' \to x$ 切换焦点时，所有起始于 x 的箭头都通过前组合 $(*\circ k)$ 来转换。

在 Haskell 中，这个同构可以实现为一个函数，它是其自身的逆：

```
swap :: (a, b) -> (b, a)
swap x = (snd x, fst x)
```

这是使用模式匹配编写的相同函数：

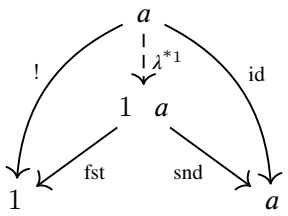
```
swap (x, y) = (y, x)
```

重要的是要记住，积仅在“同构意义上”是对称的。这并不意味着交换对的顺序不会改变程序的行为。对称性意味着交换对的信息内容相同，但访问它需要进行修改。

终端对象是积的单位，$1 \times a \cong a$。见证 $1 \times a$ 和 a 之间同构的箭头称为*左单位元*：

$$\lambda\colon 1 \times a \to a$$

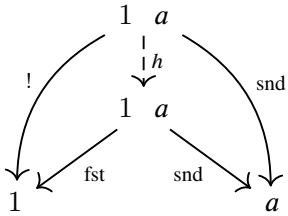
它可以实现为 $\lambda = \mathrm{snd}$。它的逆 λ^{*1} 定义为以下图中的唯一箭头：



从 a 到 1 的箭头称为!（发音为，*bang*）。这确实表明

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

我们仍然需要证明 λ^{*1} 是 snd 的左逆。考虑以下图：



很明显，对于 $h = id$ 来说，这个图是交换的。对于 $h = \lambda^{*1} \circ \mathrm{snd}$，这个图也是交换的，因为我们有：

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

由于 h 应该是唯一的，我们得出结论：

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

这种使用普遍构造的推理是相当标准的。

以下是用 Haskell 编写的其他一些同构（没有证明逆）。这是结合律：

```
assoc :: ((a, b), c) -> (a, (b, c))
assoc ((a, b), c) = (a, (b, c))
```

这是右单位

```
runit :: (a, ()) -> a
runit (a, _) = a
```

这两个函数对应于结合子

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

和右单位元：

$$\rho : a \quad 1 \to a$$

Exercise 5.1.1. 证明左单位证明中的双射是自然的。提示，使用箭头 $g : a \to b$ 来更改焦点。

Exercise 5.1.2. 构造一个箭头

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

这个箭头是唯一的吗？

提示：这是映射到积的映射，所以它由一对箭头给出。这些箭头，反过来，从和类型中映射出来，因此每一个都是由一对箭头给出的。

提示：映射 $b \to 1 + a$ 由 $(Left \circ !)$ 给出

Exercise 5.1.3. 重做上一个练习，这次将 h 视为从和类型中映射出。

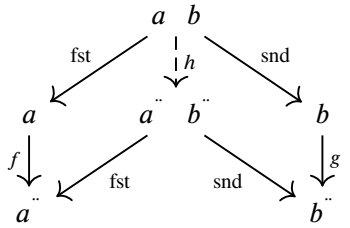
Exercise 5.1.4. 实现一个 *Haskell* 函数`maybeAB :: Either b (a, b) -> (Maybe a, b)`。这个函数是由其类型签名唯一定义的吗？还是有一些余地？

Functoriality
函子性

假设我们有箭头将 a 和 b 映射到某个 $a^{\cdot\cdot}$ 和 $b^{\cdot\cdot}$：

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

这些箭头与投影 fst 和 snd 的组合分别用于定义积之间的映射 h：



这个图的简写符号是：

$$a \quad b \xleftarrow{f\ g} a'' \quad b''$$

这种积的性质称为**函子性**。你可以想象它允许你在积内部转换两个对象以获得新的积。我们还说函子性允许我们**提升**一对箭头以操作积。

5.2 Duality
对偶性

当一个孩子看到一个箭头时，它知道哪一端指向源，哪一端指向目标

$$a \to b$$

但这也许只是一个先入之见。如果我们把 b 称为源，a 称为目标，宇宙会有很大不同吗？

我们仍然能够将这个箭头与这个箭头组合

$$b \to c$$

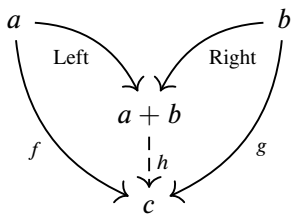
其“目标”b 与 $a \to b$ 的“源”相同，结果仍然是一个箭头

$$a \to c$$

只不过现在我们会说它从 c 到 a。

在这个对偶宇宙中，我们称之为“初始”的对象将被称为“终端”，因为它是从所有对象来的唯一箭头的“目标”。相反，终端对象将被称为初始对象。

现在考虑我们用来定义和对象的这个图：



在新的解释中，箭头 h 将从任意对象 c“到”我们称为 $a + b$ 的对象。这个箭头由一对箭头 (f, g) 唯一定义，其“源”是 c。如果我们将 Left 重命名为 fst，将 Right 重命名为 snd，我们将得到积的定义图。

积是和类型的对偶。

相反，和类型是积的对偶。

范畴论中的每个构造都有其对偶。

如果箭头的方向只是解释问题，那么在编程中，和类型和积类型有什么区别呢？这种区别可以追溯到我们一开始做的一个假设：没有指向初始对象的箭头（除身份箭头外）。这与终端对象有大量指向外部的箭头形成对比，我们用这些箭头来定义（全局）元素。事实上，我们假设每个感兴趣的对象都有元素，而没有元素的对象是与`Void`同构的。

当我们讨论函数类型时，我们将看到更深的差异。

5.3 Monoidal Category
幺半范畴

我们已经看到积满足以下简单规则：

$$1 \quad a \ddot{o} a$$

$$a \quad b \ddot{o} b \quad a$$

$$(a \quad b) \quad c \ddot{o} a \quad (b \quad c)$$

并且是函子性的。

定义了具有这些属性的运算的范畴称为**对称幺半范畴**[1]。当我们处理和类型和初始对象时，我们看到了类似的结构。

一个范畴可以同时有多个幺半结构。当你不想为你的幺半结构命名时，你可以用张量符号代替加号或积号，用字母 I 代替中性元素。对称幺半范畴的规则可以写作：

$$I \otimes a \ddot{o} a$$

$$a \otimes b \ddot{o} b \otimes a$$

$$(a \otimes b) \otimes c \ddot{o} a \otimes (b \otimes c)$$

这些同构通常写作称为结合子的可逆箭头家族和单位子。如果幺半范畴不是对称的，则有单独的左单位和右单位子。

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

$$\lambda \colon I \otimes a \to a$$

$$\rho \colon a \otimes I \to a$$

[1]严格来说，两个对象的积是同构定义的，而幺半范畴中的积必须是确切定义的。但通过选择一个积，我们可以得到一个幺半范畴

对称性由下式见证：

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

函子性让我们提升一对箭头：

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

以操作张量积：

$$a \otimes b \xrightarrow{f \otimes g} a'' \otimes b''$$

如果我们将态射视为动作，那么它们的张量积对应于同时执行两个动作。与态射的序列组合相比，这表明它们的时间顺序。

你可以将张量积想象为积和和的最小公分母。它仍然有一个引入规则，需要两个对象 a 和 b；但它没有消去规则。一旦创建，张量积就“忘记”了它是如何创建的。与笛卡尔积不同，它没有投影。

一些有趣的张量积甚至不是对称的。

Monoids
幺半群

幺半群是非常简单的结构，具有二元运算和一个单位。以加法和零为运算的自然数构成一个幺半群。以乘法和一为运算的自然数也构成一个幺半群。

直觉上，幺半群让你将两件事物结合起来，得到另一件事物。还有一个特殊的事物，将其与任何其他事物结合，都会返回相同的事物。那就是单位元。并且结合必须是结合的。

未被假定的是结合是对称的，或存在逆元素。

定义幺半群的规则让人联想到范畴的规则。不同之处在于，在幺半群中，任何两件事物都是可组合的，而在范畴中通常不是这样：只有当一个箭头的目标是另一个箭头的源时，你才能组合这两个箭头。除非该范畴仅包含一个对象，在这种情况下，所有箭头都是可组合的。

包含单个对象的范畴称为幺半群。组合运算是箭头的组合，单位是身份箭头。

这是一个完全有效的定义。然而，在实践中，我们通常对嵌入更大范畴中的幺半群更感兴趣。特别是在编程中，我们希望能够在类型和函数的范畴内定义幺半群。

但是，在范畴中，我们倾向于整体定义运算，而不是关注个别元素。因此我们从一个对象 m 开始。二元运算是一个有两个参数的函数。由于积的元素是成对的元素，我们可以将二元运算表征为从积 $m \quad m$ 到 m 的箭头：

$$\mu : m \quad m \to m$$

单位元素可以定义为从终端对象 1 的箭头：

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

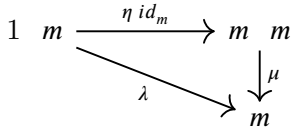
我们可以直接将这种描述翻译为 Haskell，定义一个类型类，它配备了两个方法，传统上称为mappend和mempty：

```
class Monoid m where
mappend :: (m, m) -> m
mempty  :: () -> m
```

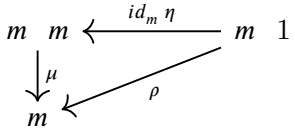
这两个箭头 μ 和 η 必须满足幺半群律，但我们再次需要在整体上表述它们，而不涉及元素。

为了表述左单位律，我们首先创建积 1 m。然后我们使用 η“选择 m 中的单位元素”，或者，用箭头的术语来说，将 1 变成 m。由于我们正在操作积 1 m，我们必须提升对 $\langle \eta, id_m \rangle$，确保我们“不接触”$m$。最后，我们使用 μ 执行“乘法”。

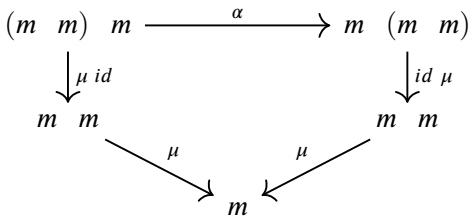
我们希望结果与原始的 m 元素相同，但不提及元素。因此，我们只是使用左单位元 λ 从 1 m 到 m 而不“搅动事物。”



这里是右单位的类似律：



为了表述结合律，我们必须从三重积开始，并在整体上操作。在这里，α 是结合子，它重新排列积而不“搅动事物”。



注意，我们不需要对我们与对象 m 和 1 一起使用的范畴积假设太多，特别是我们从未需要使用投影。这表明上述定义同样适用于任意幺半范畴中的张量积。它甚至不需要是对称的。我们只需假设：存在一个单位对象，积是函子的，并且它满足单位和结合律（最多同构）。

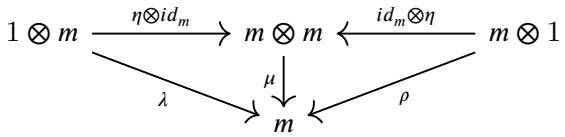
因此，如果我们将 替换为 $\otimes$，并将 1 替换为 I，我们将得到任意幺半范畴中幺半群的定义。

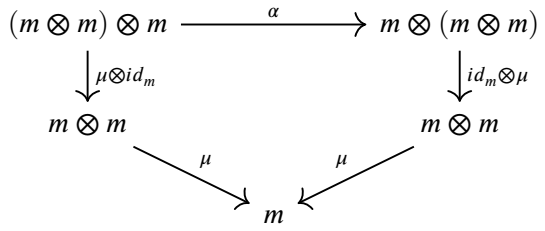
任意幺半范畴中的幺半群是一个配备两个态射的对象 m：

$$\mu : m \otimes m \to m$$

$$\eta : I \to m$$

满足单位和结合律：





我们使用了 $\otimes$ 的函子性来提升箭头对，例如 $\eta \otimes id_m$、$\mu \otimes id_m$ 等。

Chapter 6

Function Types
函数类型

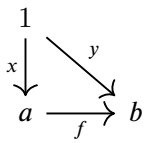
在函数式编程中，还有一种核心的组合方式。当你将一个函数作为参数传递给另一个函数时，外部函数可以将这个参数作为自身机制中的一个可插拔部分。这使得你可以实现一个通用的排序算法，该算法接受任意的比较函数。

如果我们将函数建模为对象之间的箭头，那么拥有一个函数作为参数意味着什么呢?

我们需要一种将函数对象化的方法，以便定义以“箭头对象”作为源或目标的箭头。接受函数作为参数或返回函数的函数被称为 *higher-order function*（高阶函数）。高阶函数是函数式编程的工作马。

Elimination rule
消去规则

函数的定义性特征是它可以应用于一个参数来产生结果。我们已经根据组合定义了函数应用:



这里，f 被表示为从 a 到 b 的箭头，但我们希望能够将 f 替换为箭头对象的一个元素，或者用数学家的话来说，替换为指数对象 b^a；在编程中，我们称之为函数类型`a->b`。

给定 b^a 的一个元素和 a 的一个元素，函数应用应该产生 b 的一个元素。换句话说，

给定一对元素：

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

$$x\colon 1 \to a$$

它应该产生一个元素：

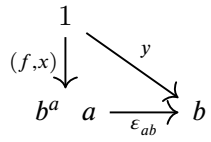
$$y\colon 1 \to b$$

请记住，这里 f 表示 b^a 的一个元素。之前，它是从 a 到 b 的一个箭头。

我们知道，一对元素 (f, x) 等价于积 $b^a \quad a$ 的一个元素。因此，我们可以将函数应用定义为一个单一的箭头：

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

这样，应用的结果 y 由以下交换图定义：



函数应用是函数类型的**消去规则**。

当有人给你一个函数对象的元素时，你唯一能做的就是使用 ε 将它应用于参数类型的一个元素。

Introduction rule
引入规则

为了完成函数对象的定义，我们还需要引入规则。

首先，假设有一种方法可以从其他对象构造函数对象 b^a。这意味着存在一个箭头

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

我们知道我们可以使用 ε_{ab} 消去 h 的结果，但我们必须首先将其乘以 a。因此，首先将 c 乘以 a，然后使用函子性将其映射到 $b^a \quad a$。

函子性让我们可以将一对箭头应用于一个积以得到另一个积。在这里，箭头对是 (h, id_a)（我们希望将 c 转变为 b^a，但我们对修改 a 不感兴趣）

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

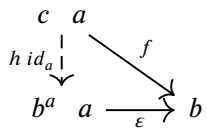
现在我们可以用函数应用继续将其映射到 b:

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

这个复合箭头定义了一个我们称之为 f 的映射：

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

以下是相应的图示：



这个交换图告诉我们，给定一个 h，我们可以构造一个 f；但我们也可以要求反过来：每一个映射 $f : c \times a \to b$ 都应该唯一地定义一个到指数对象的映射 $h : c \to b^a$。

我们可以利用这个属性，即箭头集合之间的一一对应关系来定义指数对象。这是函数对象 b^a 的*引入规则*。

我们已经看到积是通过它的映射进入性质定义的。而函数应用则定义为从一个积中映射出来的*映射出*。

Currying
柯里化

有几种方式可以看待这个定义。一种是将其视为柯里化的一个例子。

到目前为止，我们只考虑了单参数的函数。这并不是一个真正的限制，因为我们总是可以实现一个双参数的函数作为一个从积到函数的（单参数）函数。函数对象定义中的 f 就是这样一个函数：

```
f :: (c, a) -> b
```

而h则是一个返回函数的函数：

```
h :: c -> (a -> b)
```

柯里化就是这两个箭头集合之间的同构。

这种同构可以通过一对（高阶）函数在 Haskell 中表示出来。由于在 Haskell 中，柯里化适用于任何类型，这些函数使用类型变量编写——它们是*多态*的：

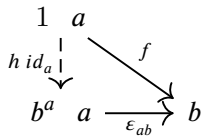
```
curry   :: ((c, a) -> b)   -> (c -> (a -> b))
```

```
uncurry :: (c -> (a -> b)) -> ((c, a) -> b)
```

换句话说，函数对象定义中的 h 可以写成：

$$h = curry\ f$$

当然，以这种方式书写，curry和uncurry的类型对应于函数对象而非箭头。这种区别通常被忽略，因为指数的*元素*和定义它们的*箭头*之间有一一对应的关系。当我们用终端对象替换任意对象 c 时，这一点很容易看出。我们得到：



在这种情况下，h 是对象 b^a 的一个元素，f 是从 1 a 到 b 的一个箭头。但我们知道 1 a 同构于 a，所以实际上，f 是从 a 到 b 的一个箭头。

因此，从现在开始，我们将->箭头称为 → 箭头，而不必过多纠缠于此。这种现象的正确表达方式是说这个范畴是自封闭的。

我们可以将 ϵ_{ab} 写成 Haskell 函数apply：

```
apply :: (a -> b, a) -> b
apply (f, x) = f x
```

但这只是一个语法技巧：函数应用内置于语言中：f x意味着f应用于x。其他编程语言要求函数的参数用括号括起来，而在 Haskell 中则不需要。

尽管将函数应用定义为一个单独的函数似乎是多余的，Haskell 库确实提供了一个用于此目的的中缀操作符$：

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

但这个技巧在于常规的函数应用绑定到左边，例如，f x y与(f x) y相同；但美元符号绑定到右边，因此

```
f  $ g x
```

与f (g x)相同。在第一个例子中，f必须是一个（至少）双参数函数；在第二个例子中，它可以是一个单参数函数。

在 Haskell 中，柯里化无处不在。双参数函数几乎总是被写作返回一个函数的函数。由于函数箭头->绑定到右边，因此不需要为此类类型加括号。例如，配对构造函数具有如下签名：

```
pair :: a -> b -> (a, b)
```

你可以将其视为一个双参数函数，返回一个配对，或者是一个单参数函数，返回一个单参数函数b->(a, b)。因此，可以部分应用此类函数，其结果为另一个函数。例如，我们可以定义：

```
pairWithTen :: a -> (Int, a)
pairWithTen = pair 10 -- partial application of pair
```

Relation to lambda calculus

与 演算的关系

另一种看待函数对象定义的方法是将 c 解释为 f 定义的环境类型。在这种情况下，通常将环境称为 Γ。箭头被解释为使用 Γ 中定义的变量的表达式。

考虑一个简单的例子，表达式：

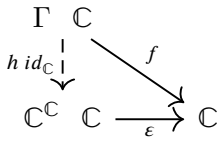
$$ax^2 + bx + c$$

你可以将其看作是由一个实数三元组 (a, b, c) 和一个变量 x（假设为一个复数）参数化的。该三元组是一个积 $\mathbb{R} \quad \mathbb{R} \quad \mathbb{R}$ 的元素。这个积就是我们的表达式的环境 Γ。

变量 x 是 $\mathbb{C}$ 的一个元素。表达式是从积 $\Gamma \quad \mathbb{C}$ 到结果类型（这里也是 $\mathbb{C}$）的箭头：

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

这是从一个积映射出的映射，因此我们可以用它来构造一个函数对象 $\mathbb{C}^{\mathbb{C}}$ 并定义一个映射 $h : \Gamma \to \mathbb{C}^{\mathbb{C}}$



这个新的映射 h 可以被看作是函数对象的构造器。所得的函数对象表示所有从 $\mathbb{C}$ 到 $\mathbb{C}$ 的函数，这些函数可以访问环境 Γ，即参数三元组 (a, b, c)。

对应于我们最初的表达式 $ax^2 + bx + c$，在 $\mathbb{C}^{\mathbb{C}}$ 中有一个特定的函数，我们写作：

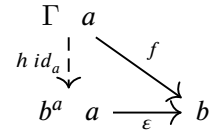
$$\lambda x.\, ax^2 + bx + c$$

或者在 Haskell 中，使用反斜杠代替 λ:

```
\x -> a * x^2 + b * x + c
```

箭头 $h : \Gamma \to \mathbb{C}^{\mathbb{C}}$ 由箭头 f 唯一确定。此映射生成我们称之为 $\lambda x.f$ 的函数。

一般而言，函数对象的定义图变为：



提供表达式 f 的自由参数的环境 Γ 是代表参数类型的多个对象的积（在我们的例子中是 $\mathbb{R} \quad \mathbb{R} \quad \mathbb{R}$）。

空环境由终端对象 1 表示，积的单位。在这种情况下，f 只是一个从 a 到 b 的箭头，而 h 只是从函数对象 b^a 中挑选出与 f 对应的一个元素。

需要牢记的是，通常情况下，函数对象表示依赖于外部参数的函数。这样的函数被称为闭包。闭包是从其环境中捕获值的函数。

以下是我们例子在 Haskell 中的翻译。与 f 对应的是一个表达式：

```
(a :+ 0) * x * x + (b :+ 0) * x + (c :+ 0)
```

如果我们用Double来近似 ℝ，我们的环境是一个积(Double, Double, Double)。类型Complex由另一个类型参数化——这里我们再次使用Double:

```
type C = Complex Double
```

从Double到C的转换是通过将虚部设为零来完成的，例如(a :+ 0)。

相应的箭头 *h* 接受环境并生成一个类型为C -> C的闭包:

```
h :: (Double, Double, Double) -> (C -> C)
h (a, b, c) = \x -> (a :+ 0) * x * x + (b :+ 0) * x + (c :+ 0)
```

Modus ponens
演绎规则

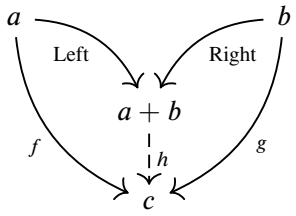
在逻辑中，函数对象对应于一个蕴涵。从终端对象到函数对象的箭头是该蕴涵的证明。函数应用 ε 对应于逻辑学家称之为*演绎规则*的概念：如果你有 $A \Rightarrow B$ 的证明以及 A 的证明，那么这就构成了 B 的证明。

6.1 Sum and Product Revisited
再谈和与积

当函数与其他类型的元素获得相同的地位时，我们拥有了直接将图表转换为代码的工具。

Sum types
和类型

我们从和类型的定义开始。



我们说过，箭头对 (f, g) 唯一地决定了从和类型到其他对象的映射 h。我们可以使用一个高阶函数简洁地将其写成:

```
h = mapOut (f, g)
```

其中:

```
mapOut :: (a -> c, b -> c) -> (Either a b -> c)
mapOut (f, g) = \aorb -> case aorb of
Left  a -> f a
Right b -> g b
```

这个函数以一对函数作为参数，并返回一个函数。

首先，我们通过模式匹配提取`(f, g)`中的`f`和`g`。然后我们使用 lambda 构造一个新函数。这个 lambda 接受一个类型为`Either a b`的参数，我们称之为`aorb`，并对其进行 case 分析。如果它是用`Left`构造的，我们将其内容传递给`f`，否则传递给`g`。

请注意，我们返回的函数是一个闭包。它从环境中捕获`f`和`g`。

我们实现的函数与图示非常相似，但它并不是通常的 Haskell 风格。Haskell 程序员更喜欢将多参数函数柯里化。此外，如果可能的话，他们更喜欢消除 lambdas。

以下是相同函数的 Haskell 标准库中的版本，它以小写字母`either`命名：

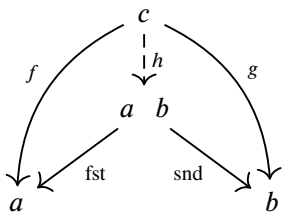
```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x)     =  f x
either _ g (Right y)    =  g y
```

另一方向的双射（从 h 到对 (f, g)）也遵循图的箭头：

```
unEither :: (Either a b -> c) -> (a -> c, b -> c)
unEither h = (h . Left, h . Right)
```

Product types
积类型

积类型通过它们的映射入属性对偶地定义。



这是直接从这个图在 Haskell 中的读取：

```
h :: (c -> a, c -> b) -> (c -> (a, b))
h (f, g) = \c -> (f c, g c)
```

这是用 Haskell 风格编写的版本，作为中缀操作符`&&&`：

```
(&&&) :: (c -> a) -> (c -> b) -> (c -> (a, b))
(f &&& g) c = (f c, g c)
```

双射的另一个方向是：

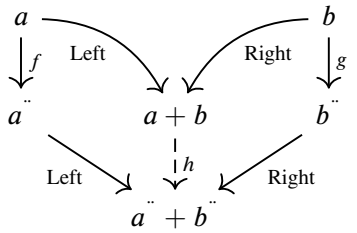
```
fork :: (c -> (a, b)) -> (c -> a, c -> b)
fork h = (fst . h, snd . h)
```

这也与图的读取密切相关。

Functoriality revisited
函子性再探

无论是和类型还是积类型都是函子的，这意味着我们可以将函数应用到它们的内容上。我们准备好将这些图转化为代码。

这是和类型的函子性：



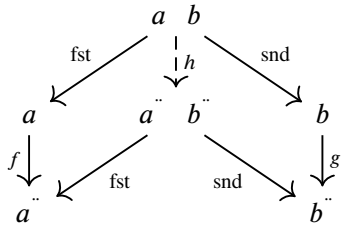
读取这个图，我们可以立即使用`either`编写 h：

```
h f g = either (Left . f) (Right . g)
```

或者我们可以将其扩展并称之为`bimap`：

```
bimap :: (a -> a') -> (b -> b') -> Either a b -> Either a' b'
bimap f g (Left  a) = Left  (f a)
bimap f g (Right b) = Right (g b)
```

对于积类型也类似：



h 可以写成：

```
h f g = (f . fst) &&& (g . snd)
```

或者可以将其扩展为：

```
bimap :: (a -> a') -> (b -> b') -> (a, b) -> (a', b')
bimap f g (a, b) = (f a, g b)
```

在这两种情况下，我们都称这个高阶函数为bimap，因为在 Haskell 中，和类型和积类型都是一个更通用类Bifunctor的实例。

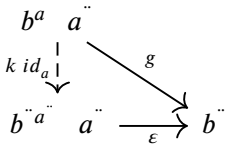
6.2 Functoriality of the Function Type
函数类型的函子性

函数类型或指数类型也是函子的，但有一个不同之处。我们感兴趣的是从 b^a 到 $b^{\prime\prime a^{\prime\prime}}$ 的映射，其中带撇号的对象通过一些箭头与不带撇号的对象相关联——这些箭头有待确定。

指数类型通过其映射入属性定义，因此如果我们在寻找

$$k : b^a \to b^{\prime\prime a^{\prime\prime}}$$

我们应该画出有 k 作为映射入 $b^{\prime\prime a^{\prime\prime}}$ 的图。我们通过将 b^a 替换为 c 并将带撇号的对象替换为不带撇号的对象，从原始定义中得到这个图：



问题是：我们可以找到一个箭头 g 来完成这个图吗?

$$g : b^a \quad a'' \to b''$$

如果我们找到这样一个 g，它将唯一地确定我们的 k。

思考这个问题的方法是考虑如何实现 g。它将 $b^a \quad a''$ 的积作为其参数。可以将其视为一个对：从 a 到 b 的函数对象的一个元素和一个 a'' 的元素。我们唯一能对函数对象做的事情是将其应用于某些东西。但 b^a 需要一个类型为 a 的参数，而我们所拥有的只是 a''。除非有人给我们一个箭头 $a'' \to a$，否则我们无能为力。该箭头应用于 a'' 将生成 b^a 的参数。然而，应用的结果是 b 类型，而 g 应该生成一个 b''。再一次，我们需要一个箭头 $b \to b''$ 来完成我们的赋值。

这可能听起来很复杂，但底线是我们需要两个箭头来连接带撇号和不带撇号的对象。不同之处在于第一个箭头从 a'' 到 a，这在通常的函子性考虑中似乎是逆向的。为了将 b^a 映射到 $b^{\prime\prime a^{\prime\prime}}$，我们需要一对箭头：

$$f : a'' \to a$$

$$g : b \to b''$$

这在 Haskell 中更容易解释。我们的目标是实现一个函数`a' -> b'`，给定一个函数`h :: a -> b`。

这个新函数接受一个类型为`a'`的参数，因此在我们将其传递给`h`之前，我们需要将`a'`转换为`a`。这就是为什么我们需要一个函数`f :: a' -> a`。

由于`h`生成一个`b`，而我们希望返回一个`b'`，我们需要另一个函数`g :: b -> b'`。所有这些都很好地适合于一个高阶函数：

```
dimap :: (a' -> a) -> (b -> b') -> (a -> b) -> (a' -> b')
dimap f g h = g . h . f
```

类似于`bimap`是类型类`Bifunctor`的一个接口，`dimap`是类型类`Profunctor`的成员。

6.3 Bicartesian Closed Categories
双笛卡尔封闭范畴

一个同时为任意对象对定义了积和指数，并且具有终端对象的范畴被称为*笛卡尔封闭*。这个想法是，同态集对于所讨论的范畴来说并不是外来的东西：这个范畴在形成同态集的操作下是“封闭的”。

如果范畴还具有和类型（余积）和初始对象，那么它被称为*双笛卡尔封闭*。

这是建模编程语言的最小结构。

使用这些操作构造的数据类型被称为*代数数据类型*。我们有加法、乘法和指数（但没有减法或除法）类型；以及我们在高中代数中熟悉的所有定律。它们在同构意义上得到满足。还有一个代数定律我们还没有讨论。

Distributivity
分配性

数的乘法对加法是分配的。我们应该期望在双笛卡尔封闭范畴中也是如此吗？

$$b \quad a+c \quad a \ddot{o} (b+c) \quad a$$

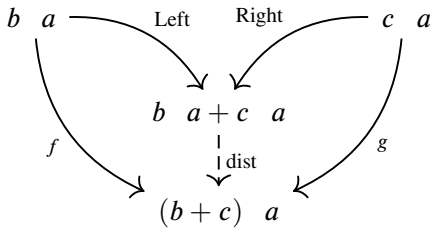
从左到右的映射很容易构造，因为它同时是一个和类型的映射出和积的映射入。我们可以通过逐步将其分解为更简单的映射来构造它。在 Haskell 中，这意味着实现一个函数：

```
dist :: Either (b, a) (c, a) -> (Either b c, a)
```

一个从左边的和类型映射出的映射由一对箭头给出：

$$f : b \quad a \to (b+c) \quad a$$

$$g : c \quad a \to (b+c) \quad a$$



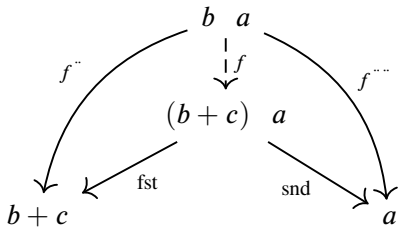
我们将其写成 Haskell 代码：

```
dist = either f g
where
f   :: (b, a) -> (Either b c, a)
g   :: (c, a) -> (Either b c, a)
```

在 `where`子句中定义辅助函数的实现。

现在我们需要实现 f 和 g。它们是积中的映射，因此每一个都对应于一对箭头。例如，第一个由一对箭头给出：

$$f'' : b \quad a \to (b + c)$$
$$f''' : b \quad a \to a$$



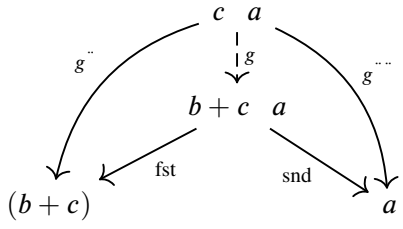
在 Haskell 中：

```
f = f' &&& f''
f'  :: (b, a) -> Either b c
f'' :: (b, a) -> a
```

第一个箭头可以通过投影第一个分量 b 并使用 Left 来构造和类型。第二个只是投影 snd：

$$f'' = \text{Left} \circ \text{fst}$$
$$f''' = \text{snd}$$

类似地，我们将 g 分解为一对 g'' 和 g'''：



将所有这些组合在一起，我们得到：

```
dist = either f g
where
f   = f' &&& f''
f'  = Left . fst
f'' = snd
g   = g' &&& g''
g'  = Right . fst
g'' = snd
```

这些是辅助函数的类型签名：

```
f   :: (b, a) -> (Either b c, a)
g   :: (c, a) -> (Either b c, a)
f'  :: (b, a) -> Either b c
f'' :: (b, a) -> a
g'  :: (c, a) -> Either b c
g'' :: (c, a) -> a
```

它们也可以内联以生成这种简洁形式：

```
dist = either ((Left . fst) &&& snd) ((Right . fst) &&& snd)
```

这种编程风格被称为*无点式*，因为它省略了参数（点）。出于可读性原因，Haskell 程序员更喜欢更明确的风格。上面的函数通常会实现为：

```
dist (Left  (b, a)) = (Left  b, a)
dist (Right (c, a)) = (Right c, a)
```

请注意，我们只使用了和类型和积的定义。双射的另一个方向需要使用指数，因此它仅在双笛卡尔*封闭*范畴中有效。这在 Haskell 的直接实现中并不立即显现：

```
undist :: (Either b c, a) -> Either (b, a) (c, a)
undist (Left b, a)  = Left (b, a)
undist (Right c, a) = Right (c, a)
```

但这是因为在 Haskell 中柯里化是隐式的。

这是这个函数的无点式版本：

```
undist = uncurry (either (curry Left) (curry Right))
```

这可能不是最易读的实现，但它强调了我们需要指数的事实：我们使用了curry和uncurry来实现这个映射。

我们将在后续章节中使用更强大的工具：伴随来回到这个恒等式。

Exercise 6.3.1. 证明:

$$2 \quad a \to a + a$$

其中 2 是布尔类型。首先用图形方式证明，然后实现两个 *Haskell* 函数来见证这个同构。

Chapter 7

Recursion
递归

当你站在两个镜子之间时，你会看到自己的反射，反射的反射，反射的反射的反射，如此反复。每个反射都是基于前一个反射定义的，但它们共同产生了无限。

递归是一种将单一任务分解为许多步骤的模式，这些步骤的数量可能是无限的。

递归是基于对不可能性的暂时搁置。你面临一个可能需要任意多步骤的任务。你暂时假设你知道如何解决它。然后你问自己一个问题："如果我已经解决了除最后一步以外的所有问题，我会如何进行最后一步？"

7.1 Natural Numbers
自然数

自然数 N 的对象并不包含数字。对象没有内部结构。结构由箭头定义。

我们可以使用从终端对象到自然数对象的箭头来定义一个特殊的元素。按照惯例，我们将这个箭头称为 Z，表示"零"。

$$Z : 1 \to N$$

但我们必须能够定义无穷多个箭头，以表示对于每一个自然数，还有另一个比它大一的数字。

我们可以形式化这个声明：假设我们知道如何创建一个自然数 $n : 1 \to N$。我们如何迈出下一步，指向下一个数字——它的继承者？

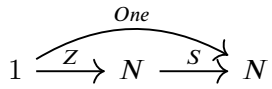
这下一步不必比直接将 n 与一个从 N 到 N 的箭头进行后合成更复杂。这个箭头不应该是恒等，因为我们希望一个数字的继承者与该数字不同。但一个单一的箭头就足够了，我们将其称为 S，表示"继承者"。

与 n 的继承者对应的元素由以下组合给出：

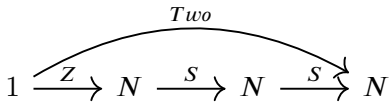
$$1 \xrightarrow{n} N \xrightarrow{S} N$$

（我们有时在同一个图中多次绘制同一个对象，如果我们想要拉直循环箭头。）

特别地，我们可以定义 One 为 Z 的继承者：



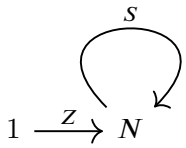
并将 Two 定义为 Z 的继承者的继承者



如此继续下去。

Introduction Rules
引入规则

这两个箭头 Z 和 S 是自然数对象 N 的引入规则。不同的是，其中一个是递归的：S 同时使用 N 作为它的源和目标。



这两个引入规则可以直接翻译为 Haskell 代码：

```
data Nat where
Z :: Nat
S :: Nat -> Nat
```

它们可以用于定义任意自然数；例如：

```
zero, one, two :: Nat
zero = Z
one  = S zero
two  = S one
```

这种自然数类型的定义在实际应用中并不十分有用。然而，它经常被用于定义类型级别的自然数，其中每个数字都有其自己的类型。

你可能会在皮亚诺算术中遇到这种结构。

Elimination Rules
消去规则

引入规则的递归性质使得定义消去规则时稍微有些复杂。我们将按照前几章的模式，首先假设我们给定了一个从 N 映射出的映射：

$$h\colon N \to a$$

然后看看我们能从中推导出什么。

之前，我们能够将这样的 h 分解为更简单的映射（对于和类型和积类型是映射对，对于指数类型是从积映射出）。

N 的引入规则与和类型的引入规则类似（它要么是 Z，要么是继承者），所以我们预计 h 可以分为两个箭头。实际上，我们可以通过组合 $h \circ Z$ 轻松地得到第一个箭头。这个箭头选择 a 的一个元素。我们称它为 init：

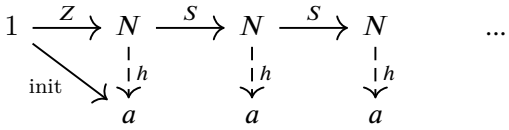
$$\text{init}\colon 1 \to a$$

但没有明显的方法来找到第二个箭头。

为了看清这一点，让我们扩展 N 的定义：

$$1 \xrightarrow{Z} N \xrightarrow{S} N \xrightarrow{S} N \qquad \ldots$$

并将 h 和 init 插入其中：



直观地说，从 N 到 a 的箭头代表了 a 元素的*序列* a_n。第零个元素由以下公式给出：

$$a_0 = \text{init}$$

下一个元素是

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

接下来是

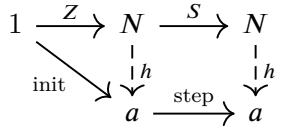
$$a_2 = h \circ S \circ S \circ Z$$

如此继续下去。

因此，我们将一个箭头 h 替换为无穷多个箭头 a_n。当然，新的箭头更简单，因为它们表示 a 的元素，但它们是无穷多个。

问题在于，不管你怎么看，从 N 映射出的任意映射都包含无限量的信息。

我们必须大幅简化问题。由于我们使用了一个单一的箭头 S 来生成所有自然数，我们可以尝试使用一个单一的箭头 $a \to a$ 来生成所有元素 a_n。我们将这个箭头称为 step：



由这样的对 init 和 step 生成的从 N 映射出的映射称为*递归*映射。并非所有从 N 映射出的映射都是递归的。事实上，只有极少数是递归的；但递归映射足以定义自然数对象。

我们使用上面的图作为消去规则。我们规定，从 N 映射出的每个递归映射 h 与一对 init 和 step 一一对应。

这意味着*评估规则*（提取给定 h 的 $(\text{init}, \text{step})$）不能为任意箭头 $h\colon N \to a$ 制定，只能为那些先前使用对 $(\text{init}, \text{step})$ 递归定义的箭头制定。

箭头 init 可以总是通过组合 $h \circ Z$ 恢复。箭头 step 是以下方程的解：

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

如果 h 是使用某个 init 和 step 定义的，那么这个方程显然有解。

重要的是我们要求这个解是*唯一*的。

直观上，这对 init 和 step 生成了元素序列 a_0、a_1、a_2，等等。如果有两个箭头 h 和 $h^{¨}$ 由同一对 $(\text{init}, \text{step})$ 给出，这意味着它们生成的序列是相同的。

所以，如果 h 与 $h^{¨}$ 有某种不同，这将意味着 N 包含的数字不仅仅是由 Z 和 S 生成的数字序列。例如，如果我们在 N 中添加 *1（即使 Z 成为某人的继承者），我们可能会让 h 和 $h^{¨}$ 在 *1 处有所不同，但仍由同一对 init 和 step 生成。唯一性意味着在 Z 和 S 生成的数字之间、之前或之后没有其他自然数。

我们在这里讨论的消去规则对应于*原始递归*。在依赖类型的章节中，我们将看到这种规则的更高级版本，对应于归纳原理。

In Programming
在编程中

消去规则可以在 Haskell 中实现为递归函数：

```
rec :: a -> (a -> a) -> (Nat -> a)
rec init step = \n ->
case n of
Z     -> init
(S m) -> step (rec init step m)
```

这个单一的函数，称为*递归器*，足以实现所有自然数的递归函数。例如，这是我们如何实现加法的：

```
plus :: Nat -> Nat -> Nat
plus n = rec init step
where
init = n
step = S
```

这个函数将n作为参数，并生成一个函数（闭包），该函数接受另一个数字并将n加到它上面。

在实际编程中，程序员更喜欢直接实现递归——这种方法等同于内联递归器rec。以下实现可能更容易理解：

```
plus n m = case m of
Z -> n
(S k) -> S (plus k n)
```

这可以理解为：如果m是零，那么结果是n。否则，如果m是某个k的继承者，那么结果是k + n的继承者。这与说init = n和step = S完全相同。

在命令式语言中，递归通常被迭代所取代。从概念上讲，迭代似乎更容易理解，因为它对应于顺序分解。序列中的步骤通常遵循某种自然顺序。这与递归分解形成对比，在递归分解中，我们假设我们已经完成了第 n 步之前的所有工作，然后将该结果与下一步结合起来。

另一方面，递归在处理递归定义的数据结构（如列表或树）时更加自然。

这两种方法是等效的，编译器通常会将递归函数转换为循环，这被称为*尾递归优化*。

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:*

```
init :: Nat -> Nat
step :: (Nat -> Nat) -> (Nat -> Nat)
```

7.2 Lists
列表

一个事物的列表要么是空的，要么是一个事物后跟着一个事物的列表。

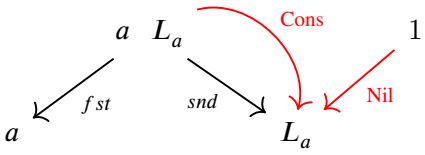
这个递归定义转化为类型 L_a 的两个引入规则，即 a 的列表：

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

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

Nil 元素描述一个空列表，而 Cons 从头部和尾部构造一个列表。

以下图描述了投影和列表构造函数之间的关系。投影提取用 Cons 构造的列表的头部和尾部。



这种描述可以立即翻译为 Haskell 代码：

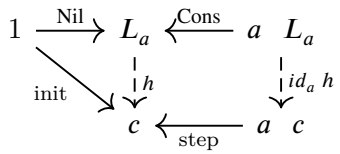
```
data List a where
Nil  :: List a
Cons :: (a, List a) -> List a
```

Elimination Rule
消去规则

假设我们有一个从 a 的列表映射到某个任意类型 c 的映射：

$$h\colon L_a \to c$$

这是我们如何将其插入到列表的定义中：



我们使用了积的函子性来将对 (id_a, h) 应用于积 $a \quad L_a$。

与自然数对象类似，我们可以尝试定义两个箭头，init $= h \circ$Nil 和 step。箭头 step 是以下方程的解：

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

同样，并不是所有的 h 都能简化为这样的一对箭头。

然而，给定 init 和 step，我们可以定义一个 h。这样的函数称为折叠，或列表的降阶同态。

这是 Haskell 中的列表递归器：

```
recList :: c -> ((a, c) -> c) -> (List a -> c)
recList init step = \as ->
case as of
Nil          -> init
Cons (a, as) -> step (a, recList init step as)
```

给定init和step，它生成一个从列表映射出的映射。

列表是一种如此基本的数据类型，以至于Haskell为其提供了内置语法。类型(List a)写作[a]。构造函数Nil是一个空的方括号对[]，而构造函数Cons是一个中缀的冒号(:)。

我们可以对这些构造函数进行模式匹配。一个通用的从列表映射出的映射的形式如下：

```
h :: [a] -> c
h []       = -- empty-list case
h (a : as) = -- case for the head and the tail of a non-empty list
```

对应于列表递归器recList，以下是你可以在标准库中找到的函数foldr（折叠右）的类型签名：

```
foldr :: (a -> c -> c) -> c -> [a] -> c
```

这是一个可能的实现：

```
foldr step init = \as ->
case as of
[] -> init
a : as -> step a (foldr step init as)
```

作为一个例子，我们可以使用foldr来计算自然数列表元素的和：

```
sum :: [Nat] -> Nat
sum = foldr plus Z
```

Exercise 7.2.1. *考虑在列表的定义中用终端对象替换 a 时会发生什么。提示：自然数的一元编码是什么？*

Exercise 7.2.2. *从 L_a 映射到 $1 + a$ 的映射有多少种？我们能用列表递归器得到它们吗？Haskell 函数的签名如何：*

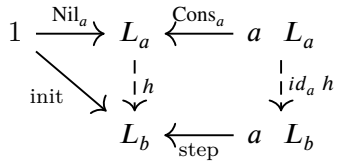
```
h :: [a] -> Maybe a
```

Exercise 7.2.3. *实现一个函数，从列表中提取第三个元素，如果列表足够长。提示：使用Maybe a作为结果类型。*

7.3 Functoriality
函子性

函子性大致意味着转换数据结构“内容”的能力。列表 L_a 的内容类型是 a。给定一个箭头 $f : a \to b$，我们需要定义一个列表映射 $h : L_a \to L_b$。

列表是通过映射出属性定义的，所以我们将消去规则的目标 c 替换为 L_b。我们得到：



由于这里涉及两个不同的列表，我们必须区分它们的构造函数。例如，我们有：

$$\text{Nil}_a : 1 \to L_a$$

$$\text{Nil}_b : 1 \to L_b$$

同样的，对于 Cons 也是如此。

init 的唯一候选是 Nil_b，也就是说，h 作用于一个 a 的空列表会生成一个 b 的空列表：

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

剩下的就是定义箭头：

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

我们可以猜测：

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

这对应于 Haskell 中的函数：

```
mapList :: (a -> b) -> List a -> List b
mapList f = recList init step
where
init = Nil
step (a, bs) = Cons (f a, bs)
```

或者，使用内置的列表语法并内联递归器，

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (a : as) = f a : map f as
```

你可能想知道是什么阻止我们选择 step = snd，结果是：

```
badMap :: (a -> b) -> [a] -> [b]
badMap f [] = []
badMap f (a : as) = badMap f as
```

我们将在下一章中看到，为什么这是一个糟糕的选择。（提示：当我们将badMap应用于 *id* 时会发生什么?）

Chapter 8

Functors
函子

8.1 Categories
范畴

到目前为止，我们只看到了一个范畴——即 *types*（类型）和 *functions*（函数）的范畴。让我们快速总结一下关于范畴的基本信息。

范畴是一个 *objects*（对象）和在它们之间的 *arrows*（箭头）的集合。每一对可组合的箭头都可以进行组合。组合是 *associative*（结合律的），并且每个对象都有一个自身环绕的 *identity*（单位箭头）。

类型和函数形成范畴的事实可以通过在 *Haskell* 中定义组合来表达：

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \x -> g (f x)
```

两个函数 g 和 f 的组合是一个新的函数，它首先将 f 应用于其参数，然后将 g 应用于结果。

identity（单位箭头）是一个多态的“无操作”函数：

```
id :: a -> a
id x = x
```

你可以很容易地验证这种组合是结合律的，并且与 *id* 组合不会改变函数的行为。

基于范畴的定义，我们可以想到各种奇怪的范畴。例如，有一个范畴没有对象也没有箭头。它满足所有范畴的条件，只是内容为空。还有一个范畴只包含一个对象和一个箭头（你能猜出那个箭头是什么吗?）。有一个范畴包含两个不相连的对象，还有一个范畴的两个对象通过一个箭头（加上两个单位箭头）连接，等等。这些是我称之为“stick-figure categories”（简易范畴）的例子——只包含少量对象和箭头的范畴。

Category of Sets
集合的范畴

我们还可以将一个范畴剥离掉所有的箭头（除了单位箭头）。这样的纯对象范畴称为 *discretecategory*（离散范畴）或集合。因为我们将箭头与结构关联，因此集合是一个没有结构的范畴。

集合形成了它们自己的范畴，称为 Set。在这个范畴中的对象是集合，而箭头是集合之间的函数。这些函数被定义为一种特殊的关系，而关系本身被定义为成对的集合。

在最低的近似下，我们可以在集合的范畴中建模编程。我们通常将类型视为值的集合，将函数视为集合论函数。这没有什么问题。事实上，迄今为止我们描述的所有范畴学构造都有其集合论的根源。范畴积是集合笛卡尔积的推广，和则是并集的推广，等等。

范畴论提供了更高的精确性：它在绝对必要的结构和多余细节之间进行了细致的区分。

例如，一个集合论函数不符合我们作为程序员所处理的函数定义。我们的函数必须具有底层算法，因为它们必须能够通过某些物理系统（无论是计算机还是人脑）进行计算。

Opposite Categories
对偶范畴

在编程中，重点是类型和函数的范畴，但我们可以使用这个范畴作为构建其他范畴的起点。

其中一个范畴称为 *oppositecategory*（对偶范畴）。这是一个所有原始箭头都被反转的范畴：在原始范畴中称为箭头源的部分现在称为目标，反之亦然。

范畴 $\mathcal{C}$ 的对偶称为 $\mathcal{C}^{op}$。当我们讨论 *duality*（对偶性）时，我们已经窥见过这个范畴。$\mathcal{C}^{op}$ 的对象与 $\mathcal{C}$ 的对象相同。

每当在 $\mathcal{C}$ 中有一个箭头 $f : a \to b$ 时，在 $\mathcal{C}^{op}$ 中会有一个对应的箭头 $f^{op} : b \to a$。

两个这样的箭头 $f^{op} : a \to b$ 和 $g^{op} : b \to c$ 的组合 $g^{op} \circ f^{op}$ 由箭头 $(f \circ g)^{op}$ 给出（注意顺序的反转）。

$\mathcal{C}$ 中的 *terminalobject*（终端对象）是 $\mathcal{C}^{op}$ 中的 *initialobject*（初始对象），*product*（积）在 $\mathcal{C}$ 中对应于 *sum*（和）在 $\mathcal{C}^{op}$ 中，等等。

Product Categories
积范畴

给定两个范畴 $\mathcal{C}$ 和 $\mathcal{D}$，我们可以构造一个 *productcategory*（积范畴）$\mathcal{C} \quad \mathcal{D}$。在这个范畴中的对象是成对的对象 $\langle c, d \rangle$，箭头是成对的箭头。

如果我们在 $\mathcal{C}$ 中有一个箭头 $f: c \to c'$，并且在 $\mathcal{D}$ 中有一个箭头 $g: d \to d'$，那么在 $\mathcal{C} \times \mathcal{D}$ 中会有一个对应的箭头 $\langle f, g \rangle$。该箭头从 $\langle c, d \rangle$ 指向 $\langle c', d' \rangle$，这两者都是 $\mathcal{C} \times \mathcal{D}$ 中的对象。如果它们的组成部分在 $\mathcal{C}$ 和 $\mathcal{D}$ 中分别是可组合的，则可以组合两个这样的箭头。单位箭头是一对单位箭头。

我们最感兴趣的两个积范畴是 $\mathcal{C} \times \mathcal{C}$ 和 $\mathcal{C}^{op} \times \mathcal{C}$，其中 $\mathcal{C}$ 是我们熟悉的类型和函数的范畴。

在这两个范畴中，对象都是 $\mathcal{C}$ 中的成对对象。在第一个范畴 $\mathcal{C} \times \mathcal{C}$ 中，从 $\langle a, b \rangle$ 到 $\langle a', b' \rangle$ 的 *morphism*（态射）是一个对 $\langle f: a \to a', g: b \to b' \rangle$。在第二个范畴 $\mathcal{C}^{op} \times \mathcal{C}$ 中，态射是一个对 $\langle f: a' \to a, g: b \to b' \rangle$，其中第一个箭头方向相反。

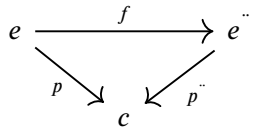
Slice Categories
截面范畴

在一个组织良好的宇宙中，对象总是对象，箭头总是箭头。只是有时箭头的集合可以被视为对象。但 *slicecategories*（截面范畴）打破了这种整洁的分离：它们将单个箭头变成了对象。

Slicecategory $\mathcal{C}/c$ 描述了从范畴 $\mathcal{C}$ 的角度来看特定对象 c 的方式。它是指向 c 的所有箭头的总和。但要指定一个箭头，我们需要指定它的两个端点。由于其中一个端点固定为 c，我们只需指定另一个端点。

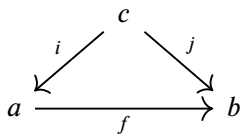
在 $\mathcal{C}/c$ 中的一个对象（也称为 *over * category*）是一个对 $\langle e, p \rangle$，其中 $p: e \to c$。

在两个对象 $\langle e, p \rangle$ 和 $\langle e', p' \rangle$ 之间的箭头是 $\mathcal{C}$ 的一个箭头 $f: e \to e'$，它使得以下三角形交换：



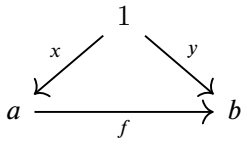
Coslice Categories
余截面范畴

存在一个 *coslicecategory*（余截面范畴）$c/\mathcal{C}$ 的对偶概念，也称为 *under * category*。它是从固定对象 c 发出的箭头的范畴。在这个范畴中的对象是成对的 $\langle a, i: c \to a \rangle$。$c/\mathcal{C}$ 中的态射是使相关三角形交换的箭头。



特别地，如果范畴 $\mathcal{C}$ 有一个 *terminal object*（终端对象）1，那么 *coslice* $1/\mathcal{C}$ 中的对象就是 $\mathcal{C}$ 中所有对象的 *global elements*（全局元素）。

$1/\mathcal{C}$ 的态射对应于箭头 $f : a \to b$，它将 a 的全局元素集合映射到 b 的全局元素集合。



特别地，从类型和函数的范畴构建余截面范畴验证了我们将类型视为值的集合的直觉，其中值由类型的全局元素表示。

8.2 Functors
函子

我们在讨论代数数据类型时已经看到了 *functoriality*（函子性）的例子。其思想是这种数据类型“记住”了其创建的方式，并且我们可以通过将一个箭头应用到其“内容”上来操控这种记忆。

在某些情况下，这种直觉非常有说服力：我们将 *product type*（积类型）视为一个包含其成分的对。毕竟，我们可以通过 *projections*（投影）来获取它们。

在 *function objects*（函数对象）的情况下，这一点不太明显。你可以将函数对象可视化为暗地里存储所有可能的结果，并使用函数参数对它们进行索引。一个从 *Bool* 到其他类型的函数显然等同于一个值对，一个用于 *True*，另一个用于 *False*。将某些函数实现为 *lookup tables*（查找表）是一个众所周知的编程技巧。它称为 *memoization*（备忘录化）。

尽管将以自然数作为参数的函数备忘录化并不现实；我们仍然可以将它们概念化为（无限或甚至不可数的）查找表。

如果你可以将数据类型视为值的容器，那么将函数应用于所有这些值并创建一个变换后的容器是有意义的。当这种可能存在时，我们称该数据类型是 *functorial*（函子性的）。

同样，函数类型需要更多的“怀疑暂停”。你将函数对象视为一个查找表，以某种类型为键。如果你想使用另一种相关类型作为键，你需要一个将新键转换为原始键的函数。这就是为什么函数对象的函子性之一是反向箭头：

```
dimap :: (a' -> a) -> (b -> b') -> (a -> b) -> (a' -> b')
dimap f g h = g . h . f
```

你正在将转换应用于一个具有对类型 a 的值作出响应的“受体”的函数 $h :: a* > b$，并且你希望使用它来处理 $a¨$ 类型的输入。这仅在你有一个从 $a¨$ 到 a 的转换器时可能，即 $f :: a¨* > a$。

一个数据类型“包含”另一个类型的值的概念也可以通过说一个数据类型被另一个数据类型参数化来表达。例如，类型 $List\ a$ 是由类型 a 参数化的。

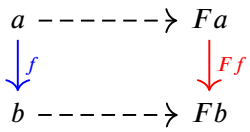
换句话说，$List$ 将类型 a 映射为类型 $List a$。没有参数的 $List$ 本身被称为 $typeconstructor$（类型构造子）。

Functors between Categories
范畴之间的函子

在范畴论中，类型构造子被建模为对象到对象的映射。它是一个作用于对象的函数。这不要与对象之间的箭头混淆，后者是范畴结构的一部分。

事实上，想象一个 $categories$（范畴之间）的映射更容易。源范畴中的每个对象都映射到目标范畴中的一个对象。如果 a 是 $\mathcal{C}$ 中的一个对象，那么在 $\mathcal{D}$ 中有一个对应的对象 Fa。

$Functorialmapping$（函子映射），或 $functor$（函子），不仅映射对象，还映射它们之间的箭头。在第一个范畴中的每个箭头 $f : a \to b$ 在第二个范畴中都有一个对应的箭头：$Ff : Fa \to Fb$。

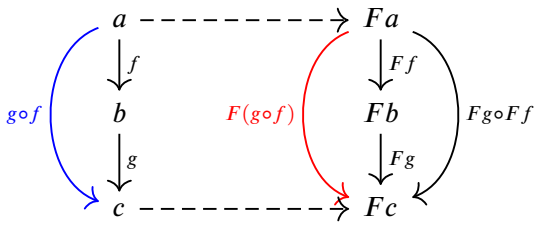


我们使用相同的字母（此处为 F）来命名对象映射和箭头映射。

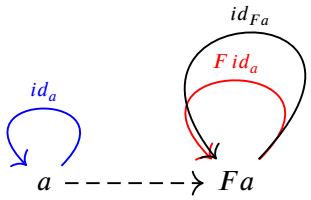
如果范畴提炼出 $structure$（结构）的本质，那么函子就是保留这种结构的映射。在源范畴中相关的对象在目标范畴中也相关。

范畴的结构由箭头及其组合定义。因此，函子必须保留组合。在一个范畴中组合的内容：$h = g \circ f$ 应该在第二个范畴中保持组合：$Fh = F(g \circ f) = Fg \circ Ff$

我们可以在 $\mathcal{C}$ 中组合两个箭头并将复合映射到 $\mathcal{D}$，或者我们可以映射单个箭头，然后在 $\mathcal{D}$ 中组合它们。我们要求结果相同。



最后，函子必须保留 $identityarrows$（单位箭头）：$F\,id_a = id_{Fa}$



这些条件一起定义了什么是保持范畴结构的函子。

同样重要的是要理解什么条件不属于定义的一部分。例如，允许一个函子将多个对象映射为一个对象。它还可以将多个箭头映射为一个箭头，只要它们的端点匹配。

极端情况下，任何范畴都可以映射到一个只有一个对象和一个箭头的单例范畴。

此外，目标范畴中的所有对象或箭头不一定都必须由函子覆盖。极端情况下，我们可以将一个单例范畴的函子映射到任何（非空）范畴。这样的函子选择一个单一对象及其单位箭头。

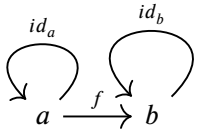
Constant functor（常量函子）Δ_c 是一个函子的例子，它将源范畴中的所有对象映射到目标范畴中的一个对象 c，并将源范畴中的所有箭头映射到该对象的单位箭头 id_c。

在范畴论中，函子常用于在另一个范畴内部创建一个范畴的模型。它们能够将多个对象和箭头合并为一个，这意味着它们生成了源范畴的简化视图。它们"抽象"了源范畴的某些方面。

它们可能只覆盖目标范畴的某些部分，这意味着这些模型嵌入在一个更大的环境中。

从某些极小的"stick-figure categories"简易范畴（例如具有两个对象和一个箭头的范畴）中导出的函子可用于定义更大范畴中的模式。

Exercise 8.2.1. 描述一个源自"*walkingarrow*"范畴的函子。它是一个只有两个对象和一个箭头的简易范畴（再加上必需的单位箭头）。



Exercise 8.2.2. "*walkingiso*"范畴与"*walkingarrow*"范畴相似，但多了一个从 b 回到 a 的箭头。证明从这个范畴出发的函子总是选择目标范畴中的一个同构。

8.3 Functors in Programming
编程中的函子

Endofunctors（自函子）是最容易在编程语言中表达的函子类。这些函子将一个范畴（此处为类型和函数的范畴）映射回自身。

Endofunctors
自函子

自函子的第一部分是将类型映射到类型。这是通过类型构造子完成的，它们是类型级别的函数。

List 类型构造子将一个任意类型 *a* 映射为类型 *Lista*。

Maybe 类型构造子将 *a* 映射为 *Maybea*。

自函子的第二部分是箭头的映射。给定一个函数 $a \to b$，我们希望能够定义一个函数 $Lista \to Listb$ 或 $Maybea \to Maybeb$。这就是我们之前讨论过的数据类型的 *functoriality*（函子性）。函子性使我们能够 *lift*（提升）一个任意的函数到变换后的类型之间的函数。

函子性可以在 *Haskell* 中通过一个 *typeclass*（类型类）来表达。在这种情况下，类型类由类型构造子 f 参数化（在 *Haskell* 中我们使用小写名称作为类型构造子变量）。我们说 f 是一个 *Functor*，如果存在一个相应的函数映射，称为 $fmap$：

```
class Functor f where
fmap :: (a -> b) -> (f a -> f b)
```

编译器知道 f 是一个类型构造子，因为它被应用于类型，例如 fa 和 fb。

要向编译器证明特定类型构造子是一个 *Functor*，我们必须为其提供 $fmap$ 的实现。这通过定义类型类 *Functor* 的一个 *instance*（实例）来实现。例如：

```
instance Functor Maybe where
fmap g Nothing  = Nothing
fmap g (Just a) = Just (g a)
```

一个函子还必须满足一些定律：它必须保持组合性和单位性。这些定律不能在 *Haskell* 中表达，但应该由程序员检查。我们之前已经看到过一个 *badMap* 的定义，它不满足单位定律，但它仍然会被编译器接受。它为列表类型构造子 [] 定义了一个“非法”的 *Functor* 实例。

Exercise 8.3.1. 证明 *WithInt* 是一个函子：

```
data WithInt a = WithInt a Int
```

有一些基本的函子可能看起来很简单，但它们作为其他函子的构建块发挥了作用。

我们有一个 *identityendofunctor*（恒等自函子），它将所有对象映射到自身，并将所有箭头映射到自身。

```
data Id a = Id a
```

Exercise 8.3.2. 证明 *Id* 是一个 *Functor*。提示：为其实现 *Functor* 实例。

我们还有一个常量函子 Δ_c，它将所有对象映射到一个单一对象 c，并将所有箭头映射到该对象的单位箭头 id_c。在 *Haskell* 中，它是一个以目标对象 c 为参数的函子族：

```
data Const c a = Const c
```

这个类型构造子忽略了其第二个参数。

Exercise 8.3.3. *证明 (Constc) 是一个 Functor。提示：该类型构造子接受两个参数，但在 Functor 实例中，它只对第一个参数部分应用。在第二个参数中，它是函子性的。*

Bifunctors
双函子

我们还看到了一些类型构造子，它们接受两个类型作为参数：如积和和类型。它们也是函子性的，但它们提升一对函数而不是一个函数。在范畴论中，我们将这些定义为从积范畴 $\mathcal{C}$ $\mathcal{C}$ 到 $\mathcal{C}$ 的函子。

这样的函子将一对对象映射到一个对象，并将一对箭头映射到一个箭头。

在 *Haskell* 中，我们将这样的函子视为 *Bifunctor* 类型类的成员。

```
class Bifunctor f where
bimap :: (a -> a') -> (b -> b') -> (f a b -> f a' b')
```

编译器通过查看它被应用于两个类型（例如 fab）来推断 f 是一个两个参数的类型构造子。

要向编译器证明特定类型构造子是一个 *Bifunctor*，我们需要定义一个实例。例如，可以这样定义一个对的双函子性：

```
instance Bifunctor (,) where
bimap g h (a, b) = (g a, h b)
```

Exercise 8.3.4. *证明 MoreThanA 是一个双函子：*

```
data MoreThanA a b = More a (Maybe b)
```

Contravariant Functors
逆变函子

从对偶范畴 $\mathcal{C}^{op}$ 出发的函子称为 *contravariant*（逆变）的。它们的特点是提升箭头的方向相反。通常的函子有时称为 *covariant*（协变）的。

在 *Haskell* 中，逆变函子构成了一个类型类 *Contravariant*：

```
class Contravariant f where
contramap :: (b -> a) -> (f a -> f b)
```

将函子分为生产者和消费者有时很方便。在这个类比中，协变函子是一个生产者。你可以通过应用 $fmap$ 一个 $a^* > b$ 的函数，将一个 a 的生产者转变为一个 b 的生产者。相反，要将一个 a 的消费者转变为一个 b 的消费者，你需要一个反向的函数 $b^* > a$。

例子：一个谓词是一个返回 $True$ 或 $False$ 的函数：

```
data Predicate a = Predicate (a -> Bool)
```

很容易看出它是一个逆变函子：

```
instance Contravariant Predicate where
contramap f (Predicate h) = Predicate (h . f)
```

实际上，逆变函子的唯一非平凡例子是函数对象的各种变体。

一种判断给定函数类型在其类型参数上是协变还是逆变的方法是通过分配极性。我们说函数的返回类型处于正位置，因此它是协变的；参数类型处于负位置，因此它是逆变的。但是，如果你将整个函数对象放在另一个函数的负位置上，那么它的极性就会反转。

考虑这个数据类型：

```
data Tester a = Tester ((a -> Bool) -> Bool)
```

它将 a 放在双负位置上，因此是正位置。因此它是一个协变的 $Functor$。它作为 a 的生产者：

```
instance Functor Tester where
fmap f (Tester g) = Tester g'
where g' h = g (h . f)
```

注意这里的括号是重要的。类似的函数 $a^* > Bool^* > Bool$ 将 a 放在负位置上。因为它是一个从 a 返回函数 $(Bool^* > Bool)$ 的函数。等价地，你可以将其变为一个接受对的函数：$(a, Bool)^* > Bool$。无论哪种方式，a 最终都在负位置上。

Profunctors
变截函子

我们之前看到函数类型是函子性的。它一次提升两个函数，就像 $Bifunctor$ 一样，只不过其中一个函数的方向相反。

在范畴论中，这对应于一个从 C^{op} C 到 Set 的函子。这样的函子称为 $Profunctors$（变截函子）。

在 $Haskell$ 中，变截函子形成了一个类型类 $Profunctor$：

```
class Profunctor f where
dimap :: (a' -> a) -> (b -> b') -> (f a b -> f a' b')
```

你可以将一个变截函子视为一个同时是生产者和消费者的类型。它消耗一种类型并产生另一种类型。

函数类型 $a* > b$ 可以写成中缀运算符 * >，它是 *Profunctor* 的一个实例：

```
instance Profunctor (->) where
dimap f g h = g . h . f
```

这与我们的直觉一致，即一个函数 $a* > b$ 消耗类型 a 的参数并产生类型 b 的结果。

在编程中，所有非平凡的变截函子都是函数类型的变体。

8.4 The Hom-Functor
Hom 函子

两个对象之间的箭头形成一个集合。这个集合称为 $hom*set$（同态集），通常写作 $\mathcal{C}(a,b)$。

我们可以将 $hom*set\ \mathcal{C}(a,b)$ 解释为从 a 观察 b 的所有方式。

另一种看待 $hom*sets$ 的方式是将它们视为一个映射，它将一个集合 $\mathcal{C}(a,b)$ 分配给每对对象。集合本身是 Set 范畴中的对象。所以我们有一个范畴之间的映射。

这个映射是函子性的。要看到这一点，让我们考虑当我们转换两个对象 a 和 b 时会发生什么。我们感兴趣的是一个将集合 $\mathcal{C}(a,b)$ 映射到集合 $\mathcal{C}(a^{"},b^{"})$ 的转换。Set 中的箭头是常规函数，因此只需定义它们在集合上对单个元素的操作。

$\mathcal{C}(a,b)$ 的元素是一个箭头 $h\colon a \to b$，$\mathcal{C}(a^{"},b^{"})$ 的元素是一个箭头 $h^{"}\colon a^{"} \to b^{"}$。我们知道如何将一个转换为另一个：我们需要用箭头 $g^{"}\colon a^{"} \to a$ 对 h 进行前组合，并用箭头 $g\colon b \to b^{"}$ 对它进行后组合。

换句话说，将对 $\langle a,b\rangle$ 的一对对象映射为集合 $\mathcal{C}(a,b)$ 的映射是一个 *profunctor*：

$\mathcal{C}^{op}\ \ \mathcal{C} \to \mathrm{Set}$

我们经常感兴趣的是在保持一个对象固定的情况下变换另一个对象。当我们固定源对象并变换目标对象时，结果是一个函子，写作：

$\mathcal{C}(a,*)\colon \mathcal{C} \to \mathrm{Set}$

这个函子在箭头 $g\colon b \to b^{"}$ 上的作用写作：

$\mathcal{C}(a,g)\colon \mathcal{C}(a,b) \to \mathcal{C}(a,b^{"})$

它通过后组合给出：

$\mathcal{C}(a,g) = (g \circ *)$

变换 b 意味着从一个对象切换到另一个对象，因此完整的函子 $\mathcal{C}(a,*)$ 将所有从 a 发出的箭头组合成一个一致的范畴视图。这是“a 的世界观”。

相反，当我们固定目标对象并变换源对象时，我们得到一个逆变函子：

$\mathcal{C}(*,b)\colon \mathcal{C}^{op} \to \mathrm{Set}$

其作用在箭头 $g^{"}\colon a^{"} \to a$ 上写作：

$\mathcal{C}(g^{"},b)\colon \mathcal{C}(a,b) \to \mathcal{C}(a^{"},b)$

它通过前组合给出：

$C(g^{¨}, b) = (*\circ g^{¨})$

函子 $C(*, b)$ 将所有指向 b 的箭头组织成一个一致的视图。它是“b 的世界观”。

我们现在可以重新表述同构章节中的结果。如果两个对象 a 和 b 是同构的，那么它们的 $hom * sets$ 也是同构的。特别是：

$C(a, x)$ ö $C(b, x)$

和

$C(x, a)$ ö $C(x, b)$

我们将在下一章讨论自然性条件。

另一种看待 $hom * functor\ C(a, *)$ 的方法是将其视为一个提供“a 是否与我相连？”问题答案的 *oracle*（神谕）。如果集合 $C(a, x)$ 为空，答案是否定的：“a 与 x 不相连。”否则，集合 $C(a, x)$ 中的每个元素都是存在这种连接的证明。

相反，逆变函子 $C(*, a)$ 回答的问题是：“我与 a 相连吗？”

综上所述，$profunctor\ C(x, y)$ 在对象之间建立了一个 $proof * relevantrelation$（证明相关关系）。集合 $C(x, y)$ 中的每个元素都是 x 与 y 相连的证明。如果集合为空，这两个对象是无关的。

8.5 Functor Composition
函子的组合

就像我们可以组合函数一样，我们也可以组合函子。两个函子是可组合的，如果一个函子的目标范畴与另一个的源范畴相同。

在对象上，*functorcomposition*（函子组合）G 之后的 F 首先将 F 应用于一个对象，然后将 G 应用于结果；在箭头上也是如此。

显然，你只能组合可组合的函子。然而，所有 *endofunctors*（自函子）都是可组合的，因为它们的目标范畴与源范畴相同。

在 *Haskell* 中，函子是一个参数化的数据类型，因此两个函子的组合也是一个参数化的数据类型。在对象上，我们定义：

```
data Compose g f a = Compose (g (f a))
```

编译器推断 f 和 g 必须是类型构造子，因为它们被应用于类型：f 被应用于类型参数 a，g 被应用于结果类型。

或者，你可以告诉编译器前两个参数是类型构造子。你可以通过提供一个 *kindsignature*（种类签名）来实现这一点，这需要一个语言扩展 *KindSignatures*，你可以在源文件的顶部添加：

```
{-# language KindSignatures #-}
```

你还应该导入 *Data.Kind* 库，该库定义了 *Type*：

```
import Data.Kind
```

种类签名就像类型签名一样，只不过它可以用于描述操作类型的函数。

常规类型具有种类 *Type*。类型构造子具有种类 $Type^* > Type$，因为它们将类型映射为类型。

Compose 接受两个类型构造子并生成一个类型构造子，因此它的种类签名是：

```
(Type -> Type) -> (Type -> Type) -> (Type -> Type)
```

完整的定义是：

```
data Compose :: (Type -> Type) -> (Type -> Type) -> (Type -> Type)
where
Compose :: (g (f a)) -> Compose g f a
```

任何两个类型构造子都可以这样组合。目前没有要求它们必须是函子。

但是，如果我们想提升一个函数使用类型构造子的组合 g 之后的 f，那么它们必须是函子。此要求作为约束在实例声明中编码：

```
instance (Functor g, Functor f) => Functor (Compose g f) where
fmap h (Compose gfa) = Compose (fmap (fmap h) gfa)
```

约束 $(Functor g, Functor f)$ 表示条件是两个类型构造子必须是 *Functor* 类的实例。约束后跟随双箭头。

我们正在建立函子性的类型构造子是 $Compose f g$，它是部分应用的 *Compose* 对两个函子。

在 $fmap$ 的实现中，我们对数据构造子 *Compose* 进行模式匹配。它的参数 $g f a$ 是类型 $g(fa)$。我们使用一个 $fmap$ 来“获取 g 下的内容”。然后我们使用 $(fmaph)$ 来获取 f 下的内容。编译器通过分析类型来知道应该使用哪个 $fmap$。

你可以将一个复合函子视为一个容器中的容器。例如，[] 与 *Maybe* 的组合是一个可选值的列表。

Exercise 8.5.1. 定义一个 *Functor* 之后 *Contravariant* 的组合。提示：你可以重用 *Compose*，但你必须提供一个不同的实例声明。

Category of Categories
范畴的范畴

我们可以将函子视为范畴之间的箭头。正如我们刚刚看到的，函子是可组合的，检查这个组合是结合律的很容易。我们还为每个范畴提供了一个恒等（自）函子。因此，范畴

本身似乎形成了一个范畴，我们称之为 Cat。

这是数学家开始担心“大小”问题的地方。这是指代存在悖论的简写。因此，正确的表达是 Cat 是一个小范畴的范畴。但是只要我们不涉及存在性的证明，我们就可以忽略大小问题。

Chapter 9

自然变换 (Natural Transformations)

我们已经看到，当两个对象 a 和 b 是同构的时，它们会在箭头集之间生成双射 (bijections)，我们现在可以将其表达为态射集 (hom-sets) 之间的同构 (isomorphisms)。对于所有 x，我们有：

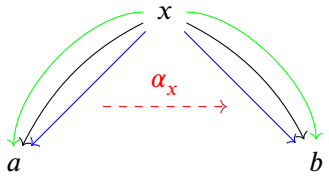
$$C(a,x) \text{ ö } C(b,x)$$
$$C(x,a) \text{ ö } C(x,b)$$

但反过来并不成立。态射集之间的同构并不会导致对象之间的同构，除非满足附加的自然性条件 (naturality conditions)。我们现在将在逐步更一般的设置中重新表述这些自然性条件。

9.1 态射函子之间的自然变换 (Natural Transformations Between Hom-Functors)

一种可以建立两个对象之间同构的方法是直接提供两个箭头，一个是另一个的逆。但通常情况下，更容易间接地完成，通过定义箭头之间的双射，无论是作用在两个对象上的箭头，还是从两个对象发出的箭头。

例如，正如我们之前所看到的，对于每一个 x，我们可能有一个可逆的箭头映射 α_x。



换句话说，对于每一个 x，存在一个态射集的映射：

$$\alpha_x : C(x,a) \to C(x,b)$$

当我们变化 x 时，这两个态射集成为两个反变函子 (contravariant functors)，$\mathcal{C}(*, a)$ 和 $\mathcal{C}(*, b)$，α 可以看作它们之间的一个映射。这样的函子映射，称为变换 (transformation)，实际上是一个个别映射 α_x 的集合，每个 $\mathcal{C}$ 中的对象 x 都有一个映射。

函子 $\mathcal{C}(*, a)$ 描述了世界如何看待 a，而函子 $\mathcal{C}(*, b)$ 描述了世界如何看待 b。

变换 α 在这两个视角之间来回切换。α 的每个分量 α_x 表明，从 x 看 a 的视角与从 x 看 b 的视角是同构的。

我们之前讨论过的自然性条件 (naturality condition) 是：

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

它将取自不同对象的 α 的分量联系起来。换句话说，它将由箭头 $g\colon y \to x$ 连接的两个观察者 x 和 y 的视角联系起来。

这个方程的两边作用在态射集 $\mathcal{C}(x, a)$ 上。结果位于态射集 $\mathcal{C}(y, b)$ 中。我们可以将两边重写为：

$$\begin{aligned} \mathcal{C}(x,a) &\xrightarrow{(*\circ g)} \mathcal{C}(y,a) \xrightarrow{\alpha_y} \mathcal{C}(y,b) \\ \mathcal{C}(x,a) &\xrightarrow{\alpha_x} \mathcal{C}(x,b) \xrightarrow{(*\circ g)} \mathcal{C}(y,b) \end{aligned}$$

与 $g\colon y \to x$ 的预合成 (precomposition) 也是一个态射集的映射。实际上它是通过反变态射函子的提升 (lifting)。我们可以将其分别写为 $\mathcal{C}(g, a)$ 和 $\mathcal{C}(g, b)$。

$$\begin{aligned} \mathcal{C}(x,a) &\xrightarrow{\mathcal{C}(g,a)} \mathcal{C}(y,a) \xrightarrow{\alpha_y} \mathcal{C}(y,b) \\ \mathcal{C}(x,a) &\xrightarrow{\alpha_x} \mathcal{C}(x,b) \xrightarrow{\mathcal{C}(g,b)} \mathcal{C}(y,b) \end{aligned}$$

因此，自然性条件可以重写为：

$$\alpha_y \circ \mathcal{C}(g, a) = \mathcal{C}(g, b) \circ \alpha_x$$

可以通过以下交换图来说明：

$$\begin{array}{ccc} \mathcal{C}(x,a) & \xrightarrow{\mathcal{C}(g,a)} & \mathcal{C}(y,a) \\ \downarrow{\scriptstyle \alpha_x} & & \downarrow{\scriptstyle \alpha_y} \\ \mathcal{C}(x,b) & \xrightarrow{\mathcal{C}(g,b)} & \mathcal{C}(y,b) \end{array}$$

我们现在可以重新表述我们之前的结论：在满足自然性条件的情况下，函子 $\mathcal{C}(*, a)$ 和 $\mathcal{C}(*, b)$ 之间的可逆变换 α 等价于 a 和 b 之间的同构。

对于出射箭头 (outgoing arrows)，我们可以遵循完全相同的推理。这次我们从一个变换 β 开始，其分量为：

$$\beta_x\colon \mathcal{C}(a, x) \to \mathcal{C}(b, x)$$

两个协变函子 (covariant functors) $C(a,*)$ 和 $C(b,*)$ 分别描述了从 a 和 b 的角度看世界。可逆变换 β 告诉我们，这两个视角是等价的，而自然性条件

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

告诉我们，当我们切换焦点时，它们的行为是良好的。

下面的交换图说明了自然性条件：

$$\begin{array}{ccc} C(a,x) & \xrightarrow{C(a,g)} & C(a,y) \\ \downarrow{\scriptstyle \beta_x} & & \downarrow{\scriptstyle \beta_y} \\ C(b,x) & \xrightarrow{C(b,g)} & C(b,y) \end{array}$$

同样，类似的可逆自然变换 β 可以建立 a 和 b 之间的同构。

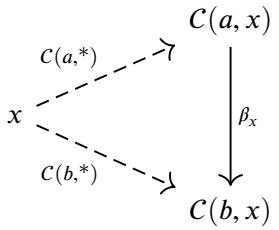
9.2 函子之间的自然变换 (Natural Transformation Between Functors)

上一节中的两个态射函子是

$$Fx = C(a,x)$$
$$Gx = C(b,x)$$

它们都将范畴 C 映射到集合范畴 Set，因为态射集都位于那里。我们可以说它们在 Set 中创建了 C 的两种不同的模型 (models)。

自然变换是两个这种模型之间的结构保持映射。



这个思想自然地扩展到任意一对范畴之间的函子。任何两个函子

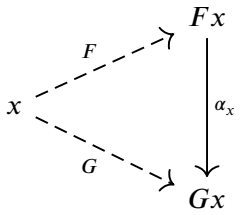
$$F\colon C \to D$$
$$G\colon C \to D$$

可以被看作 D 内 C 的两个不同的模型。

为了将一个模型转换为另一个模型，我们使用 D 中的箭头来连接相应的点。对于 C 中的每个对象 x，我们选择一个箭头从 Fx 到 Gx：

$$\alpha_x\colon Fx \to Gx$$

因此，自然变换将对象映射到箭头。

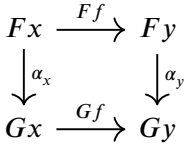


模型的结构既涉及对象，也涉及箭头，所以让我们看看箭头会发生什么。对于 $\mathcal{C}$ 中的每个箭头 $f : x \to y$，在 $\mathcal{D}$ 中有两个相应的箭头：

$$Ff : Fx \to Fy$$
$$Gf : Gx \to Gy$$

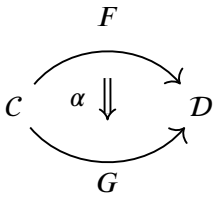
这些是 f 的两个提升 (liftings)。我们可以使用它们在每个模型的边界内移动。然后是 α 的分量，它允许我们在模型之间切换。

自然性条件 (naturality) 表明，无论你先在第一个模型内移动然后跳到第二个模型，还是先跳到第二个模型然后在其中移动，都不应该有区别。这通过交换的自然性方框 (naturality square) 来说明：



这样的满足自然性条件的箭头集合 α_x 称为自然变换 (natural transformation)。

这是一个显示一对范畴、它们之间的两个函子和函子之间的自然变换 α 的图：



因为对于 $\mathcal{C}$ 中的每个箭头都有一个对应的自然性方框，我们可以说自然变换将对象映射到箭头，将箭头映射到交换方框。

如果一个自然变换的每个分量 α_x 都是同构的，则称 α 为自然同构 (natural isomorphism)。

我们现在可以重新陈述关于同构的主要结果：当且仅当在它们的态射函子之间存在自然同构时，两个对象是同构的（无论是协变的，还是反变的态射函子都可以）。

自然变换提供了一种非常方便的高级方式，用于在各种情况下表达交换条件。我们将在此基础上重新表述代数数据类型 (algebraic data types) 的定义。

9.3 编程中的自然变换 (Natural Transformations in Programming)

自然变换是一组由对象参数化的箭头。在编程中，这对应于由类型参数化的函数族，也就是多态函数 (polymorphic function)。

自然变换参数的类型由一个函子描述，返回类型由另一个函子描述。

在 Haskell 中，我们可以定义一个接受两个类型构造器表示两个函子的自然变换数据类型：

```
data Natural :: (Type -> Type) -> (Type -> Type) -> Type where
Natural :: (forall a. f a -> g a) -> Natural f g
```

forall量词告诉编译器，这个函数是多态的——也就是说，它针对每种类型a定义。只要f和g是函子，这个公式就定义了一个自然变换。

但是forall定义的类型非常特殊。它们在*参数多态性*的意义上是多态的。这意味着同一个公式适用于所有类型。我们已经看到了身份函数 (identity function) 的例子，可以写为：

```
id :: forall a. a -> a
id x = x
```

这个函数体非常简单，就是变量x。无论x的类型是什么，公式保持不变。

这与*临时多态性*形成对比。临时多态性函数可以对不同的类型使用不同的实现。一个这样的函数的例子是fmap，它是Functor类型类的成员函数。对于列表有一个fmap的实现，对于Maybe有一个不同的实现，依此类推，每种情况各自实现。

在 Haskell 中，标准的（参数多态）自然变换的定义使用了一个类型同义词 (type synonym)：

```
type Natural f g = forall a. f a -> g a
```

type声明引入了一个别名，作为右侧的简写。

事实证明，将自然变换的类型限制为参数多态性会产生深远的影响。这样的函数自动满足自然性条件。这是一个参数性自动生成所谓*免费定理*的例子。

我们不能在 Haskell 中表达箭头的等式，但我们可以使用自然性来转换程序。特别是，如果alpha是一个自然变换，我们可以将：

```
fmap h . alpha
```

替换为：

```
alpha . fmap h
```

在这里，编译器会自动判断应该使用哪个版本的fmap和alpha的哪个分量。

我们还可以使用更高级的语言选项来显式地做出选择。我们可以使用一对函数来表达自然性：

```
oneWay ::
forall f g a b. (Functor f, Functor g) =>
Natural f g -> (a -> b) -> f a -> g b
oneWay alpha h = fmap @g h . alpha @a
```

```
otherWay ::
forall f g a b. (Functor f, Functor g) =>
Natural f g -> (a -> b) -> f a -> g b
otherWay alpha h = alpha @b . fmap @f h
```

注释@a和@b指定了参数多态函数alpha的分量，注释@f和@g指定了为其实例化的临时多态fmap的函子。

文件顶部需要指定以下 Haskell 扩展：

```
{-# language RankNTypes #-}
{-# language TypeApplications #-}
{-# language ScopedTypeVariables #-}
```

以下是一个有用的函数例子，它是列表函子和Maybe函子之间的自然变换：

```
safeHead :: Natural [] Maybe
safeHead [] = Nothing
safeHead (a : as) = Just a
```

（标准库中的head函数是不安全的，因为它在给定空列表时会出错。）

另一个例子是reverse函数，它反转一个列表。它是列表函子到列表函子的自然变换：

```
reverse :: Natural [] []
reverse [] = []
reverse (a : as) = reverse as ++ [a]
```

顺便说一下，这是一个非常低效的实现。实际的库函数使用了优化的算法。

理解自然变换的一个有用直觉是建立在函子像数据容器 (container) 的想法之上的。对于一个容器，有两种完全正交的操作：你可以转换它包含的数据，而不改变容器的形状。这就是fmap所做的事情。或者你可以在不修改数据的情况下，将数据转移到另一个容器中。这就是自然变换所做的事情：它是将“东西”在容器之间移动，而不知道“东西”是什么的过程。

换句话说，自然变换将一个容器的内容重新打包到另一个容器中。它以与内容类型无关的方式进行，这意味着它不能检查、创建或修改内容。它所能做的只是将其移动到

新位置，或将其丢弃。

自然性条件强制了这两种操作的正交性。无论你是先修改数据然后将其移动到另一个容器中，还是先移动它，然后修改，都没有关系。

这是另一个成功将复杂问题分解为一系列更简单问题的例子。不过请记住，并不是每个包含数据的容器操作都可以这样分解。例如，过滤操作需要既检查数据，又改变容器的大小甚至形状。

另一方面，几乎每个参数多态函数都是自然变换。在某些情况下，你可能需要考虑恒等函子 (identity functor) 或常量函子 (constant functor) 作为源或目标。例如，参数多态的恒等函数可以被视为两个恒等函子之间的自然变换。

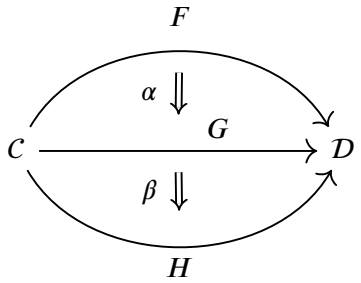
自然变换的垂直组合 (Vertical Composition of Natural Transformations)

自然变换只能在平行函子之间定义，也就是说，这些函子共享相同的源范畴和相同的目标范畴。这种平行函子形成了一个函子范畴 (functor category)。两个范畴 $\mathcal{C}$ 和 $\mathcal{D}$ 之间的函子范畴的标准表示是 $[\mathcal{C}, \mathcal{D}]$。你只需将两个范畴的名称放在方括号之间。

$[\mathcal{C}, \mathcal{D}]$ 中的对象是函子，箭头是自然变换。

为了证明这确实是一个范畴，我们必须定义自然变换的组合。当我们记住自然变换的分量是目标范畴中的常规箭头时，这很容易。这些箭头是可组合的。

确实，假设我们有一个从函子 F 到 G 的自然变换 α。我们想要将其与另一个从 G 到 H 的自然变换 β 组合。



让我们看看这些变换在某个对象 x 处的分量

$$\alpha_x \colon F\, x \to G\, x$$

$$\beta_x \colon G\, x \to H\, x$$

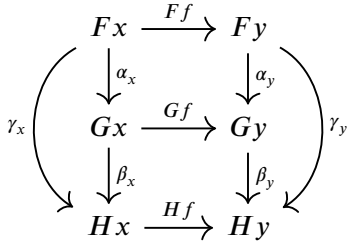
这些仅仅是 $\mathcal{D}$ 中的两个箭头，它们是可组合的。因此，我们可以如下定义一个复合自然变换 γ：

$$\gamma \colon F \to H$$

$$\gamma_x = \beta_x \circ \alpha_x$$

这被称为自然变换的垂直组合 (vertical composition)。你会看到它写作点 $\gamma = \beta \cdot \alpha$ 或简单的并置 $\gamma = \beta\alpha$。

可以通过将两个自然性方框垂直粘合在一起证明 γ 的自然性条件：



在 Haskell 中，自然变换的垂直组合只是应用于多态函数的常规函数组合。使用自然变换在容器之间移动项目的直觉，垂直组合将两个这样的移动组合在一起，一个接一个地进行。

函子范畴 (Functor Categories)

由于自然变换的组合是根据箭头的组合定义的，因此它是自动关联的。

还有一个恒等自然变换 id_F，它定义在每个函子 F 上。它在 x 处的分量是对象 Fx 处的常规恒等箭头：

$$(id_F)_x = id_{Fx}$$

总结一下，对于每一对范畴 $\mathcal{C}$ 和 $\mathcal{D}$，都有一个函子范畴 $[\mathcal{C},\mathcal{D}]$，其箭头为自然变换。

在该范畴中的态射集是两个函子 F 和 G 之间的自然变换集合。按照标准的符号表示方式，我们将其写为：

$$[\mathcal{C},\mathcal{D}](F,G)$$

范畴的名称后面跟着两个对象（在这里是函子）的名称，用括号括起来。

在范畴论中，对象和箭头的表示不同。对象是点，箭头是有尖头的线。

在 Cat，即范畴的范畴中，函子被表示为箭头。但在一个函子范畴 $[\mathcal{C},\mathcal{D}]$ 中，函子是点，自然变换是箭头。

在一个范畴中的箭头在另一个范畴中可能是一个对象。

Exercise 9.3.1. 证明自然变换复合的自然性条件：

$$\gamma_y \circ Ff = Hf \circ \gamma_x$$

提示：使用 γ 的定义以及 α 和 β 的两个自然性条件。

自然变换的水平组合 (Horizontal Composition of Natural Transformations)

自然变换的第二种组合方式是由函子组合引起的。假设我们有一对可组合的函子

$$F\colon \mathcal{C} \to \mathcal{D} \qquad\qquad G\colon \mathcal{D} \to \mathcal{E}$$

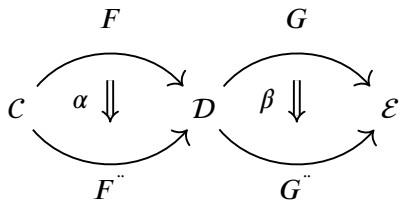
同时，还有另一对可组合的函子：

$$F^{''}: \mathcal{C} \to \mathcal{D} \qquad\qquad G^{''}: \mathcal{D} \to \mathcal{E}$$

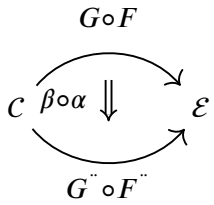
我们还拥有两个自然变换：

$$\alpha: F \to F^{''} \qquad\qquad \beta: G \to G^{''}$$

图示如下：



水平组合 $\beta \circ \alpha$ 将 $G \circ F$ 映射到 $G^{''} \circ F^{''}$。



让我们选择 $\mathcal{C}$ 中的一个对象 x，并尝试定义复合变换 $(\beta \circ \alpha)$ 在 x 处的分量。它应该是 $\mathcal{E}$ 中的一个态射：

$$(\beta \circ \alpha)_x: G(Fx) \to G^{''}(F^{''}x)$$

我们可以使用 α 将 x 映射到一个箭头

$$\alpha_x: Fx \to F^{''}x$$

我们可以用 G 提升这个箭头

$$G(\alpha_x): G(Fx) \to G(F^{''}x)$$

要从这里到达 $G^{''}(F^{''}x)$，我们可以使用 β 的适当分量

$$\beta_{F^{''}x}: G(F^{''}x) \to G^{''}(F^{''}x)$$

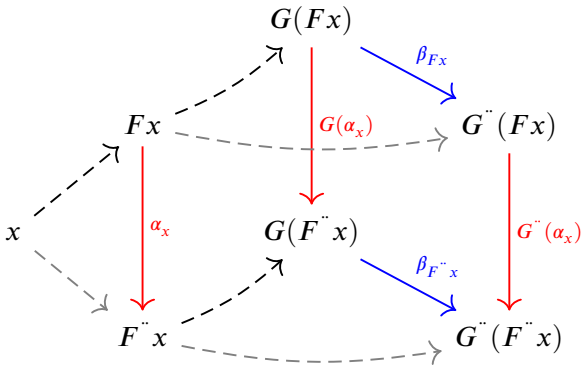
总之，我们有

$$(\beta \circ \alpha)_x = \beta_{F^{''}x} \circ G(\alpha_x)$$

但还有另一个同样合理的候选人：

$$(\beta \circ \alpha)_x = G^{''}(\alpha_x) \circ \beta_{Fx}$$

幸运的是，由于 β 的自然性，它们是相等的。



自然性条件的证明留给有志的读者作为练习。

我们可以直接将其转换为 Haskell。我们从两个自然变换开始：

```
alpha :: forall x. F x -> F' x
beta  :: forall x. G x -> G' x
```

它们的水平组合有以下类型签名：

```
beta_alpha :: forall x. G (F x) -> G' (F' x)
```

它有两个等价的实现。第一个是：

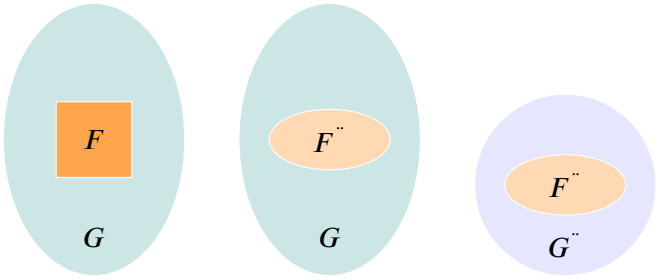
```
beta_alpha = beta . fmap alpha
```

编译器将自动选择正确版本的`fmap`，即适用于函子`G`的版本。第二个实现是：

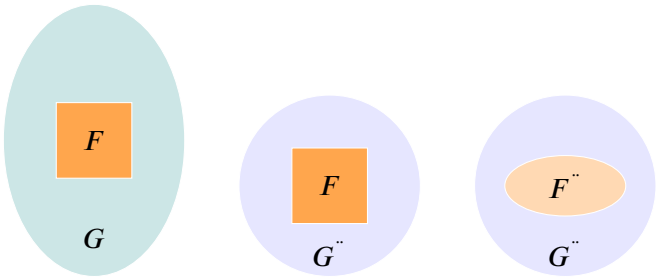
```
beta_alpha = fmap alpha . beta
```

在这里，编译器将选择`fmap`的版本，适用于函子`G'`。

水平组合的直觉是什么？我们之前已经看到，自然变换可以被看作是数据在两个容器 (函子) 之间的重新打包。在这里，我们处理的是嵌套容器。我们从由`G`描述的外部容器开始，该容器中填充了由`F`描述的内部容器。我们有两个自然变换，`alpha`用于将`F`的内容传输到`F'`，`beta`用于将`G`的内容移动到`G'`。有两种方法可以将数据从`G (F x)`移动到`G'(F' x)`。我们可以使用`fmap alpha`来重新打包所有内部容器，然后使用`beta`来重新打包外部容器。



或者我们可以先使用beta来重新打包外部容器，然后应用fmap alpha来重新打包所有内部容器。最终结果是相同的。



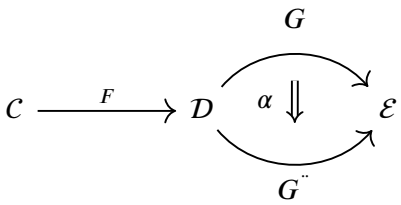
Exercise 9.3.2. 实现*safeHead*和*reverse*的水平组合的两个版本。比较它们在各种参数上的作用。

Exercise 9.3.3. 同样做*reverse*和*safeHead*的水平组合。

Whiskering

水平组合经常在其中一个自然变换是恒等变换时使用。这种组合有一个简写表示法。例如，$\alpha \circ id_F$ 写作 $\alpha \circ F$。

由于图的特征形状，这种组合被称为“Whiskering”。



在分量中，我们有：

$$(\alpha \circ F)_x = \alpha_{Fx}$$

让我们考虑如何将其翻译为 Haskell。自然变换是一个多态函数。由于参数性，它对所有类型都是由同一个公式定义的。因此在右边 Whiskering 不会改变公式，它改变了函数签名。

例如，如果这是alpha的声明：

```
alpha :: forall x. G x -> G' x
```

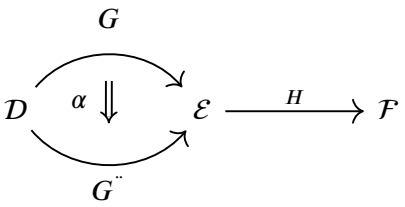
那么它的 Whiskering 版本将是：

```
alpha_f :: forall x. G (F x) -> G' (F x)
alpha_f = alpha
```

由于 Haskell 的类型推导，这种转换是隐式的。当调用一个多态函数时，我们不必指定执行自然变换的哪个分量——类型检查器会通过查看参数的类型来判断。

此时的直觉是我们重新打包外部容器而保留内部容器不变。

类似地，$id_H \circ \alpha$ 写作 $H \circ \alpha$。



在分量中：

$$(H \circ \alpha)_x = H(\alpha_x)$$

在 Haskell 中，通过 H 提升 α_x 是用`fmap`完成的，因此给定：

```
alpha :: forall x. G x -> G' x
```

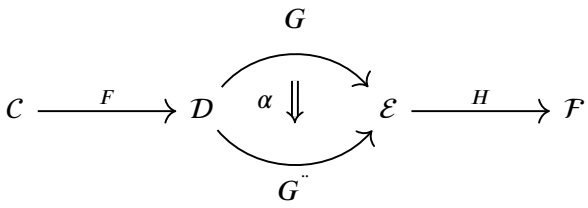
Whiskering 版本将是：

```
h_alpha :: forall x. H (G x) -> H (G' x)
h_alpha = fmap alpha
```

同样，Haskell 的类型推导引擎会判断使用哪个版本的`fmap`（在这里，它是`Functor`实例`G`的那个）。

直觉上，这表明我们重新打包了内部容器的内容，同时保留外部容器不变。

最后，在许多应用中，自然变换在两侧都进行了 Whiskering：



在分量中，我们有：

$$(H \circ \alpha \circ F)x = H(\alpha_{Fx})$$

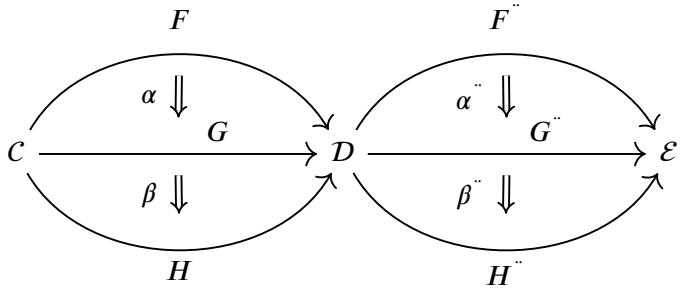
在 Haskell 中：

```
h_alpha_f :: forall x. H (G (F x)) -> H (G' (F x))
h_alpha_f = fmap alpha
```

这里的直觉是我们有一个三层容器；我们正在重新排列中间层，同时保留外层容器和所有内层容器不变。

交换律 (Interchange Law)

我们可以将垂直组合与水平组合结合起来，如下图所示：



交换律 (Interchange law) 规定组合的顺序并不重要：我们可以先做垂直组合，然后是水平组合；或者先做水平组合，然后是垂直组合。

Chapter 10

Adjunctions（伴随）

一位雕塑家通过去除不相关的石头直到雕塑显现出来。一位数学家通过抽象掉不相关的细节直到模式显现出来。

我们能够使用映射入（mapping-in）和映射出（mapping-out）属性来定义许多构造。这些属性可以紧凑地表示为同态集（hom-sets）之间的同构关系。这种自然同构的模式被称为伴随（adjunction），一旦被识别出来，它几乎无处不在。

10.1 The Currying Adjunction（柯里化伴随）

指数对象（exponential）的定义是一个经典的伴随的例子，它关联了映射出和映射入的关系。每一个从积（product）出的映射对应于一个唯一的映射入指数对象：

$$\mathcal{C}(e \quad a, b) \ \ddot{\text{o}} \ \mathcal{C}(e, b^a)$$

在左侧，b 对象充当了焦点；在右侧，e 对象成为了观察者。

我们可以发现有两个函子（functors）在起作用。它们都以 a 为参数。在左侧，我们有积函子 $(* \quad a)$ 应用于 e。在右侧，我们有指数函子 $(*)^a$ 应用于 b。

如果我们将这些函子写作：

$$L_a e = e \quad a$$

$$R_a b = b^a$$

那么同态集之间的自然同构

$$\mathcal{C}(L_a e, b) \ \ddot{\text{o}} \ \mathcal{C}(e, R_a b)$$

就被称为它们之间的伴随。

在具体操作中，这个同构告诉我们，给定一个映射 $\phi \ \ddot{\text{E}} \ \mathcal{C}(L_a e, b)$，就有一个唯一的映射 $\phi^T \ \ddot{\text{E}} \ \mathcal{C}(e, R_a b)$，反之亦然。这些映射有时被称为彼此的*转置*（transpose）——这个术语借用了矩阵代数的名称。

伴随的简写为 $L \dashv R$。将积函子代入 L，将指数函子代入 R，我们可以简洁地将柯里化伴随写为：

$$(* \quad a) \dashv (*)^a$$

指数对象 b^a 有时被称为内部同态，记作 $[a, b]$。这与外部同态相对，外部同态是集合 $\mathcal{C}(a, b)$。外部同态不是 $\mathcal{C}$ 中的对象（除非 $\mathcal{C}$ 本身是 Set）。使用这个记号，柯里化伴随可以写为：

$$\mathcal{C}(e \quad a, b) \cong \mathcal{C}(e, [a, b])$$

在这个伴随成立的范畴被称为笛卡尔封闭范畴（cartesian closed category）。

由于函数在每种编程语言中都起着核心作用，因此笛卡尔封闭范畴构成了所有编程模型的基础。我们将指数 b^a 解释为函数类型 $a \to b$。

这里，e 起着外部环境的作用——在 演算中对应 Γ。$\mathcal{C}(\Gamma \quad a, b)$ 中的态射被解释为在环境 Γ 中扩展了一个类型为 a 的变量的表达式。函数类型 $a \to b$ 因此表示一个闭包（closure），它可能从其环境中捕获一个类型为 e 的值。

顺便说一句，（小）范畴的范畴 Cat 也是笛卡尔封闭的，这在乘积范畴和函子范畴之间的伴随中得到了反映，并使用了相同的内部同态记号：

$$\mathrm{Cat}(\mathcal{A} \quad \mathcal{B}, \mathcal{C}) \cong \mathrm{Cat}(\mathcal{A}, [\mathcal{B}, \mathcal{C}])$$

这里，两个同态集都是自然变换的集合。

10.2 The Sum and the Product Adjunctions（和与积的伴随）

柯里化伴随关联了两个自函子（endofunctors），但伴随可以很容易地推广到在不同范畴之间的函子。我们先来看看一些例子。

The Diagonal Functor（对角函子）

和（sum）与积（product）类型是使用双射（bijections）定义的，其中一侧是一个单一的箭头，另一侧是一对箭头。一对箭头可以看作是积范畴（product category）中的单一箭头。

为了探索这一思想，我们需要定义对角函子（diagonal functor）Δ，这是一个从 $\mathcal{C}$ 到 $\mathcal{C} \quad \mathcal{C}$ 的特殊映射。它将一个对象 x 复制成一对对象 $\langle x, x \rangle$。它也将一个箭头 f 复制为 $\langle f, f \rangle$。

有趣的是，对角函子与我们之前看到的常函子（constant functor）有关。常函子可以被视为一个有两个变量的函子——它只是忽略了第二个变量。我们在 Haskell 定义中见过：

```
data Const c a = Const c
```

为了看到两者之间的联系，让我们将积范畴 $\mathcal{C} \times \mathcal{C}$ 视为函子范畴 $[2, \mathcal{C}]$，换句话说，是 Cat 中的指数对象 $\mathcal{C}^2$。实际上，从 2（带有两个对象的简图范畴）的函子选取了一对对象——这等同于积范畴中的单一对象。

一个从 $\mathcal{C}$ 到 $[2, \mathcal{C}]$ 的函子可以被展开为 $\mathcal{C} \times 2 \to \mathcal{C}$。对角函子忽略了来自 2 的第二个参数：无论第二个参数是 1 还是 2，它做的事情是一样的。这正是常函子所做的。因此，我们为它们都使用了相同的符号 Δ。

顺便提一下，这种论证可以轻松推广到任何索引范畴，而不仅仅是 2。

The Sum Adjunction（和伴随）

回想一下，和是通过其映射出属性定义的。$a + b$ 的所有箭头与分别从 a 和 b 出来的一对箭头一一对应。用同态集的术语，我们可以写成：

$$\mathcal{C}(a + b, x) \cong \mathcal{C}(a, x) \times \mathcal{C}(b, x)$$

这里右边的乘积只是集合的笛卡尔乘积，即对的集合。此外，我们之前已经看到，这种双射在 x 上是自然的。

我们知道一对箭头在积范畴中是一个单一箭头。因此，我们可以将右边的元素看作是在 $\mathcal{C} \times \mathcal{C}$ 中的箭头，从对象 $\langle a, b\rangle$ 到对象 $\langle x, x\rangle$。后者可以通过对 x 作用的对角函子 Δ 获得。我们有：

$$\mathcal{C}(a + b, x) \cong (\mathcal{C} \times \mathcal{C})(\langle a, b\rangle, \Delta x)$$

这是两个不同范畴中同态集之间的双射。它满足自然性条件，因此它是一个自然同构。

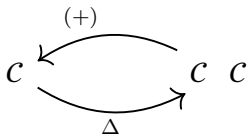
我们还可以在这里发现一对函子。在左边，我们有一个函子，它接受一对对象 $\langle a, b\rangle$ 并生成它们的和 $a + b$：

$$(+) \colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$$

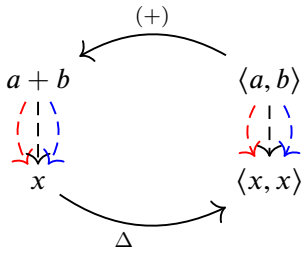
在右边，我们有对角函子 Δ，它朝相反的方向运行：

$$\Delta \colon \mathcal{C} \to \mathcal{C} \times \mathcal{C}$$

总之，我们有两个范畴之间的一对函子：



以及同态集之间的同构：



换句话说，我们有了伴随关系：

$$(+) \dashv \Delta$$

The Product Adjunction（积伴随）

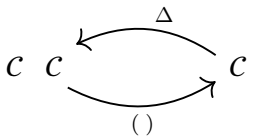
我们可以将相同的推理应用于积的定义。这次我们有一个自然同构，关联的是一对箭头和一个映射入积。

$$\mathcal{C}(x, a) \quad \mathcal{C}(x, b) \ \ddot{\text{o}} \ \mathcal{C}(x, a \quad b)$$

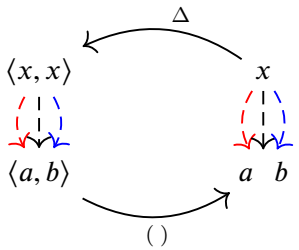
将一对箭头替换为积范畴中的箭头，我们得到：

$$(\mathcal{C} \quad \mathcal{C})(\Delta x, \langle a, b \rangle) \ \ddot{\text{o}} \ \mathcal{C}(x, a \quad b)$$

这就是朝着相反方向运行的两个函子：



以及同态集之间的同构：



换句话说，我们有了伴随关系：

$$\Delta \dashv (\ )$$

Distributivity（分配律）

在双笛卡尔封闭范畴（bicartesian closed category）中，积分配于和。我们已经使用通用构造看到了证明的一种方向。结合 Yoneda 引理，伴随关系给了我们更强大的工具来解决这个问题。

我们希望证明以下自然同构：

$$(b + c) \quad a \ddot{\circ} b \quad a + c \quad a$$

与其直接证明这个等式，我们将展示从两边映射出的所有映射到任意对象 x 是同构的：

$$\mathcal{C}((b + c) \quad a, x) \ddot{\circ} \mathcal{C}(b \quad a + c \quad a, x)$$

左边是从积映射出，因此我们可以将柯里化伴随应用于它：

$$\mathcal{C}((b + c) \quad a, x) \ddot{\circ} \mathcal{C}(b + c, x^a)$$

这给了我们一个从和映射出的映射，根据和伴随，它与两个映射的乘积同构：

$$\mathcal{C}(b + c, x^a) \ddot{\circ} \mathcal{C}(b, x^a) \quad \mathcal{C}(c, x^a)$$

现在我们可以对两个分量应用柯里化伴随的逆映射：

$$\mathcal{C}(b, x^a) \quad \mathcal{C}(c, x^a) \ddot{\circ} \mathcal{C}(b \quad a, x) \quad \mathcal{C}(c \quad a, x)$$

使用和伴随的逆映射，我们得到最终结果：

$$\mathcal{C}(b \quad a, x) \quad \mathcal{C}(c \quad a, x) \ddot{\circ} \mathcal{C}(b \quad a + c \quad a, x)$$

这个证明中的每一步都是自然同构，因此它们的组合也是自然同构。通过 Yoneda 引理，分配律左侧和右侧形成的两个对象因此是同构的。

这一陈述的一个更短的证明来自我们即将讨论的左伴随的性质。

10.3 Adjunction between Functors（函子之间的伴随）

通常，伴随关联了两个在两个范畴之间的相反方向的函子。左函子为

$$L \colon \mathcal{D} \to \mathcal{C}$$

右函子为：

$$R \colon \mathcal{C} \to \mathcal{D}$$

伴随 $L \dashv R$ 定义为两个同态集之间的自然同构。

$$\mathcal{C}(Lx, y) \ddot{\circ} \mathcal{D}(x, Ry)$$

换句话说，我们有一组在 x 和 y 上自然的可逆函数集：

$$\phi_{xy} \colon \mathcal{C}(Lx, y) \to \mathcal{D}(x, Ry)$$

例如，y 上的自然性意味着，对于任意 $f : y \to y^{\cdot\cdot}$，下图是交换的：

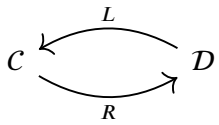
$$\begin{array}{ccc} \mathcal{C}(Lx, y) & \xrightarrow{\mathcal{C}(Lx,f)} & \mathcal{C}(Lx, y^{\cdot\cdot}) \\ \updownarrow{\scriptstyle \phi_{xy}} & & \updownarrow{\scriptstyle \phi_{xy^{\cdot\cdot}}} \\ \mathcal{D}(x, Ry) & \xrightarrow{\mathcal{D}(x,Rf)} & \mathcal{D}(x, Ry^{\cdot\cdot}) \end{array}$$

或者，考虑到同态函子的箭头提升与后合成（post-composition）相同：

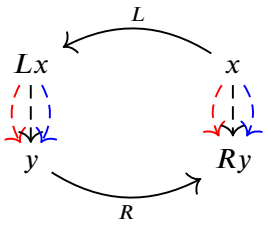
$$\begin{array}{ccc} \mathcal{C}(Lx, y) & \xrightarrow{f\circ *} & \mathcal{C}(Lx, y^{\cdot\cdot}) \\ \updownarrow{\scriptstyle \phi_{xy}} & & \updownarrow{\scriptstyle \phi_{xy^{\cdot\cdot}}} \\ \mathcal{D}(x, Ry) & \xrightarrow{Rf\circ *} & \mathcal{D}(x, Ry^{\cdot\cdot}) \end{array}$$

双头箭头可以沿任意方向（使用 ϕ^{*1}_{xy} 向上）遍历，因为它们是同构的组成部分。

图示上，我们有两个函子：



并且，对于任意一对 x 和 y，两个同构的同态集：



这些同态集来自两个不同的范畴，但集合就是集合。我们说 L 是 R 的左伴随，或者 R 是 L 的右伴随。

在 Haskell 中，这可以简化为一个多参数类型类：

```
class (Functor left, Functor right) => Adjunction left right where
ltor :: (left x -> y) -> (x -> right y)
rtol :: (x -> right y) -> (left x -> y)
```

在文件顶部需要以下指示：

```
{-# language MultiParamTypeClasses #-}
```

因此，在双笛卡尔范畴中，和是对角函子的左伴随；积是其右伴随。我们可以非常简洁地写出这一关系（或者我们可以将其印在粘土中，作为楔形文字的现代版本）：

$$(+) \dashv \Delta \dashv (\,)$$

Exercise 10.3.1. 绘制见证伴随函数 ϕ_{xy} 在 x 上自然性的交换图。

Exercise 10.3.2. 伴随公式左侧的同态集 $\mathcal{C}(Lx, y)$ 暗示 Lx 可以被视为某个函子（共预层）的表示对象。这个函子是什么？提示：它将 y 映射到一个集合。这个集合是什么？

Exercise 10.3.3. 相应地，表示对象 a 对于预层 P 的定义是：

$$Px \ddot{o} \mathcal{D}(x, a)$$

在伴随公式中，Ry 是哪个预层的表示对象。

10.4 Limits and Colimits as Adjunctions（极限与余极限作为伴随）

极限的定义也涉及同态集之间的自然同构：

$$[\mathcal{J}, \mathcal{C}](\Delta_x, D) \ddot{o} \mathcal{C}(x, \mathrm{Lim} D)$$

左侧的同态集在函子范畴中。其元素是锥（cones），即常函子 Δ_x 和图示函子 D 之间的自然变换。右侧的是 $\mathcal{C}$ 中的同态集。

在一个所有极限都存在的范畴中，我们有这两个函子之间的伴随：

$$\Delta_{(*)} : \mathcal{C} \to [\mathcal{J}, \mathcal{C}]$$

和：

$$\mathrm{Lim}(*) : [\mathcal{J}, \mathcal{C}] \to \mathcal{C}$$

相应地，余极限由以下自然同构描述：

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x) \ddot{o} \mathcal{C}(\mathrm{Colim} D, x)$$

我们可以用一个简洁的公式表示这两个伴随：

$$\mathrm{Colim} \dashv \Delta \dashv \mathrm{Lim}$$

特别地，由于积范畴 $\mathcal{C} \quad \mathcal{C}$ 等价于 $\mathcal{C}^2$，或者函子范畴 $[2, \mathcal{C}]$，我们可以将积和余积重新表述为极限和余极限：

$$[2, \mathcal{C}](\Delta_x, \langle a, b \rangle) \ddot{o} \mathcal{C}(x, a \quad b)$$

$$\mathcal{C}(a + b, x) \ddot{o} [2, \mathcal{C}](\langle a, b \rangle, \Delta_x)$$

这里 $\langle a, b \rangle$ 表示一个图示，这是一个函子 $D : 2 \to \mathcal{C}$ 作用于 2 的两个对象。

10.5 Unit and Counit of an Adjunction（伴随的单位元与伴随元）

我们通过比较箭头的相等性来比较它们，但我们更喜欢使用同构来比较对象。

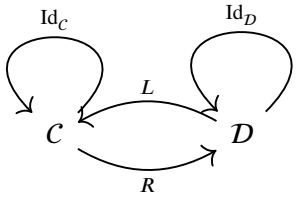
然而，当涉及到函子时，我们却遇到了问题。一方面，它们是函子范畴中的对象，因此同构是比较的方式；但另一方面，它们在 Cat 中是箭头，所以可能直接比较它们的相等性也没什么问题?

为了更清楚地理解这一困境，我们应该问自己为什么我们要使用箭头的相等性。并不是因为我们喜欢相等性，而是因为在一个集合中，除了比较元素是否相等之外，我们无事可做。同态集中的两个元素要么相等要么不相等，仅此而已。

但在 Cat 中情况并非如此，正如我们所知，它是一个 2-范畴。这里，同态集本身具有范畴的结构——即函子范畴。在 2-范畴中，我们有箭头之间的箭头，因此特别地，我们可以定义箭头之间的同构。在 Cat 中，这些同构就是函子之间的自然同构。

然而，尽管我们有选择使用同构代替箭头相等性的选项，但 Cat 中的范畴法则仍然表示为相等性。例如，函子 F 与恒等函子的复合是等于 F 的，同样地，结合律也是如此。如果 2-范畴中的法则严格满足，那么它被称为严格的（strict），而 Cat 就是一个严格 2-范畴的例子。

但就比较范畴而言，我们有更多的选择。范畴是 Cat 中的对象，因此有可能将两个范畴的同构定义为一对函子 L 和 R：



使得：

$$L \circ R = \mathrm{Id}_C$$
$$\mathrm{Id}_D = R \circ L$$

这个定义涉及函子的相等性。然而更糟糕的是，作用在对象上时，它涉及到对象的相等性：

$$L(Rx) = x$$
$$y = R(Ly)$$

这就是为什么更适合讨论一种更弱的范畴等价（equivalence）的概念，其中相等性被替换为同构：

$$L \circ R \ddot{\circ} \mathrm{Id}_C$$
$$\mathrm{Id}_D \ddot{\circ} R \circ L$$

对于对象而言，范畴的等价意味着一个往返生成的对象与原始对象是同构的，而不是相等的。在大多数情况下，这正是我们所需要的。

伴随也定义为一对相反方向的函子，因此有必要问一下往返的结果是什么。

定义伴随的同构适用于任何一对对象 x 和 y

$$\mathcal{C}(Lx, y) \cong \mathcal{D}(x, Ry)$$

因此，特别地，如果我们将 y 替换为 Lx，那么我们得到：

$$\mathcal{C}(Lx, Lx) \cong \mathcal{D}(x, R(Lx))$$

我们现在可以使用 Yoneda 技巧，并在左侧选择恒等箭头 id_{Lx}。该同构将其映射到右侧的一个唯一箭头，我们将其称为 η_x：

$$\eta_x \colon x \to R(Lx)$$

不仅为每个 x 定义了这个映射，而且它在 x 上是自然的。自然变换 η 被称为伴随的单位元（unit）。如果我们注意到左侧的 x 是恒等函子对 x 的作用，那么我们可以写成：

$$\eta \colon \mathrm{Id}_{\mathcal{D}} \to R \circ L$$

作为一个例子，我们来计算余积伴随的单位元：

$$\mathcal{C}(a + b, x) \cong (\mathcal{C} \times \mathcal{C})(\langle a, b\rangle, \Delta x)$$

通过将 x 替换为 $a + b$。我们得到：

$$\eta_{\langle a,b\rangle} \colon \langle a, b\rangle \to \Delta(a + b)$$

这是一对箭头，它们正是两个注入 $\langle \mathrm{Left}, \mathrm{Right}\rangle$。

我们可以通过将 x 替换为 Ry 来做一个类似的操作：

$$\mathcal{C}(L(Ry), y) \cong \mathcal{D}(Ry, Ry)$$

对应于右侧的恒等箭头，我们在左侧得到一个箭头：

$$\varepsilon_y \colon L(Ry) \to y$$

这些箭头形成了另一个自然变换，称为伴随的伴随元（counit）：

$$\varepsilon \colon L \circ R \to \mathrm{Id}_{\mathcal{C}}$$

请注意，如果这两个自然变换是可逆的，它们将证明范畴的等价性。但即使它们不是，这种“半等价”在范畴论的上下文中仍然非常有趣。

作为一个例子，我们来计算积伴随的伴随元：

$$(C \quad C)(\Delta x, \langle a, b\rangle) \ddot{\circ} C(x, a \quad b)$$

通过将 x 替换为 $a \quad b$。我们得到：

$$\varepsilon_{\langle a,b\rangle} \colon \Delta(a \quad b) \to \langle a, b\rangle$$

这是一对箭头，它们正是两个投影 $\langle \text{fst}, \text{snd}\rangle$。

Exercise 10.5.1. 推导出余积伴随的伴随元和积伴随的单位元。

Triangle Identities（三角恒等式）

我们可以使用单位元/伴随元对来形成伴随的一个等价定义。为此，我们首先从一对自然变换开始：

$$\eta \colon \text{Id}_D \to R \circ L$$
$$\varepsilon \colon L \circ R \to \text{Id}_C$$

并施加额外的三角恒等式（triangle identities）。

这些恒等式可以从伴随的标准定义推导出来，通过注意到 η 可以用来用复合 $R \circ L$ 替换恒等函子，实际上让我们可以在任何适用恒等函子的地方插入 $R \circ L$。

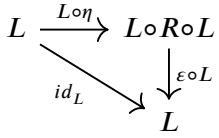
类似地，ε 可以用来消除复合 $L \circ R$（即，将其替换为恒等）。

因此，例如，从 L 开始：

$$L = L \circ \text{Id}_D \xrightarrow{L \circ \eta} L \circ R \circ L \xrightarrow{\varepsilon \circ L} \text{Id}_C \circ L = L$$

在这里，我们使用了自然变换的水平组合，其中之一是恒等变换（也称为 whiskering）。

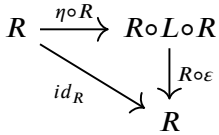
第一个三角恒等式是这种变换链结果为恒等自然变换的条件。图示上：



类似地，我们希望以下自然变换链也组合成恒等式：

$$R = \text{Id}_D \circ R \xrightarrow{\eta \circ R} R \circ L \circ R \xrightarrow{R \circ \varepsilon} R \circ \text{Id}_C = R$$

或者，图示上：



事实证明，伴随还可以用满足三角恒等式的两个自然变换 η 和 ε 来定义：

$$(\varepsilon \circ L) \cdot (L \circ \eta) = id_L$$
$$(R \circ \varepsilon) \cdot (\eta \circ R) = id_R$$

通过这些，可以很容易地恢复同态集的映射。例如，从箭头 $f : x \to Ry$ 开始，它是 $\mathcal{D}(x, Ry)$ 的一个元素。我们可以将其提升为

$$Lf : Lx \to L(Ry)$$

然后我们可以使用 η 来将复合 $L \circ R$ 压缩为恒等。结果是一个箭头 $Lx \to y$，它是 $\mathcal{C}(Lx, y)$ 的一个元素。

使用单位元和伴随元的伴随定义更加通用，因为它可以翻译到任意 2-范畴设置。

Exercise 10.5.2. *给定一个箭头 $g : Lx \to y$，使用 ε 和 R 是一个函子的事实来实现一个箭头 $x \to Ry$。提示：从对象 x 开始，看看你可以如何通过一个中途站到达 Ry。*

The Unit and Counit of the Currying Adjunction（柯里化伴随的单位元与伴随元）

让我们计算柯里化伴随的单位元和伴随元：

$$\mathcal{C}(e \times a, b) \ddot{o} \mathcal{C}(e, b^a)$$

如果我们将 b 替换为 $e \times a$，我们得到

$$\mathcal{C}(e \times a, e \times a) \ddot{o} \mathcal{C}(e, (e \times a)^a)$$

对应于左侧的恒等箭头，我们在右侧得到了伴随的单位元：

$$\eta : e \to (e \times a)^a$$

这是一个积构造器的柯里化版本。在 Haskell 中，我们将其写为：

```
unit :: e -> (a -> (e, a))
unit = curry id
```

伴随元更有趣。将 e 替换为 b^a，我们得到：

$$\mathcal{C}(b^a \times a, b) \ddot{o} \mathcal{C}(b^a, b^a)$$

对应于右侧的恒等箭头，我们得到：

$$\varepsilon : b^a \times a \to b$$

这就是函数应用箭头。

在 Haskell 中：

```
counit :: (a -> b, a) -> b
counit = uncurry id
```

当伴随发生在两个自函子之间时，我们可以使用单位元和伴随元写一个替代的 Haskell 定义：

```
class (Functor left, Functor right) =>
Adjunction left right | left -> right, right -> left where
unit   :: x -> right (left x)
counit :: left (right x) -> x
```

另外的两个子句`left -> right`和`right -> left`告诉编译器，在使用伴随实例时，一个函子可以从另一个推导出来。这个定义需要以下编译扩展：

```
{-# language MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
```

构成柯里化伴随的两个函子可以写成：

```
data L r x = L (x, r)    deriving (Functor, Show)
data R r x = R (r -> x)  deriving Functor
```

以及柯里化的`Adjunction`实例：

```
instance Adjunction (L r) (R r) where
unit x = R (\r -> L (x, r))
counit (L (R f, r)) = f r
```

第一个三角恒等式声明以下多态函数：

```
triangle :: L r x -> L r x
triangle = counit . fmap unit
```

是恒等的，第二个也是：

```
triangle' :: R r x -> R r x
triangle' = fmap counit . unit
```

注意，这两个函数需要使用功能依赖才能正确定义。三角恒等式无法在 Haskell 中表达，因此实现伴随的开发人员必须自己证明它们。

Exercise 10.5.3. 测试柯里化伴随的第一个三角恒等式的一些例子。以下是一个例子：

```
triangle (L (2, 'a'))
```

Exercise 10.5.4. 你会如何测试柯里化伴随的第二个三角恒等式？提示：*`triangle'`*的结果是一个函数，所以你无法直接显示它，但你可以调用它。

10.6 Adjunctions Using Universal Arrows（使用泛箭的伴随）

我们已经看到了使用同态集的同构来定义伴随的方式，以及使用单位元/伴随元对来定义伴随的另一种方式。事实证明，只要满足某种普遍性条件，我们可以只使用该对中的一个元素来定义伴随。为了理解这一点，我们将构造一个新的范畴，其对象是箭头。

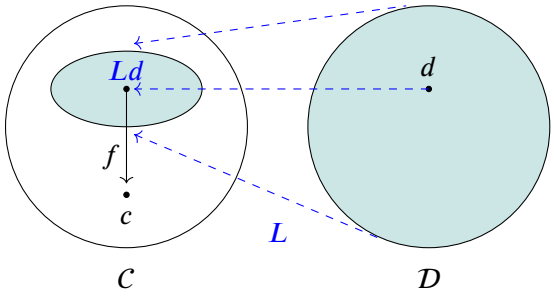
我们之前见过一个这样的范畴的例子——切片范畴（slice category）$\mathcal{C}/c$，它收集了所有汇聚于 c 的箭头。这样的范畴描述了从 $\mathcal{C}$ 中每个可能的角度来看 c 对象的视图。

Comma Category（逗号范畴）

当处理一个伴随关系时：

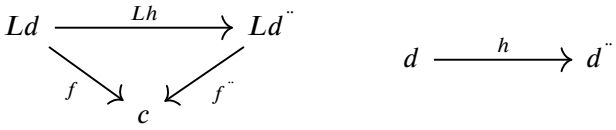
$$\mathcal{C}(Ld, c) \ddot{\text{o}} \mathcal{D}(d, Rc)$$

我们正在从由函子 L 定义的较窄视角观察对象 c。可以将 L 视为定义了一个范畴 $\mathcal{D}$ 在 $\mathcal{C}$ 内部的模型。我们感兴趣的是从这个模型的角度来看 c 的视图。描述这个视图的箭头构成了逗号范畴 L/c。



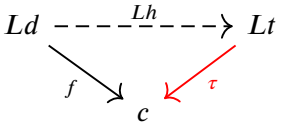
逗号范畴 L/c 中的一个对象是一个对 $\langle d, f\rangle$，其中 d 是 $\mathcal{D}$ 的一个对象，$f\colon Ld \to c$ 是 $\mathcal{C}$ 中的一个箭头。

从 $\langle d, f\rangle$ 到 $\langle d^{¨}, f^{¨}\rangle$ 的态射是一个箭头 $h\colon d \to d^{¨}$，它使得左边的图表交换：



Universal Arrow（泛箭）

从 L 到 c 的泛箭被定义为逗号范畴 L/c 中的终对象。让我们展开这个定义。L/c 中的终对象是一个对 $\langle t, \tau\rangle$，它具有从任意对象 $\langle d, f\rangle$ 的唯一态射。这样的态射是一个箭头 $h\colon d \to t$，它满足交换条件：



换句话说，对于同态集 $\mathcal{C}(Ld,c)$ 中的任意 f，在同态集 $\mathcal{D}(d,t)$ 中有一个唯一的元素 h，使得：

$$f = \tau \circ Lh$$

两个同态集之间的这种一对一映射暗示了潜在的伴随关系。

从伴随推导出的泛箭

让我们首先确认，当函子 L 有一个右伴随 R 时，对于每一个 c，存在从 L 到 c 的一个泛箭。事实上，这个箭头由对 $\langle Rc, \varepsilon_c \rangle$ 给出，其中 ε 是伴随的伴随元。首先，伴随元的分量具有适用于逗号范畴 L/c 中对象的签名：

$$\varepsilon_c \colon L(Rc) \to c$$

我们想要证明 $\langle Rc, \varepsilon_c \rangle$ 是 L/c 中的终对象。也就是说，对于任意对象 $\langle d, f \colon Ld \to c \rangle$，存在一个唯一的 $h \colon d \to Rc$，使得 $f = \varepsilon_c \circ Lh$：

$$\begin{array}{ccc} Ld & \overset{Lh}{\dashrightarrow} & L(Rc) \\ & {\scriptstyle f} \searrow \quad \swarrow {\scriptstyle \varepsilon_c} & \\ & c & \end{array}$$

为了证明这一点，让我们写出 ϕ_{dc} 作为 d 的函数的一个自然性条件：

$$\phi_{dc} \colon \mathcal{C}(Ld,c) \to \mathcal{D}(d,Rc)$$

对于任意箭头 $h \colon d \to d^{¨}$，下图必须交换：

$$\begin{array}{ccc} \mathcal{C}(Ld^{¨},c) & \xrightarrow{*\circ Lh} & \mathcal{C}(Ld,c) \\ \updownarrow {\scriptstyle \phi_{d^{¨},c}} & & \updownarrow {\scriptstyle \phi_{d,c}} \\ \mathcal{D}(d^{¨},Rc) & \xrightarrow{*\circ h} & \mathcal{D}(d,Rc) \end{array}$$

我们可以通过将 $d^{¨}$ 设置为 Rc 来使用 Yoneda 技巧。

$$\begin{array}{ccc} \mathcal{C}(L(Rc),c) & \xrightarrow{*\circ Lh} & \mathcal{C}(Ld,c) \\ \updownarrow {\scriptstyle \phi_{Rc,c}} & & \updownarrow {\scriptstyle \phi_{d,c}} \\ \mathcal{D}(Rc,Rc) & \xrightarrow{*\circ h} & \mathcal{D}(d,Rc) \end{array}$$

我们现在可以选择同态集 $\mathcal{D}(Rc,Rc)$ 的特殊元素，即恒等箭头 id_{Rc}，并将其传播到其余的图表中。左上角变为 ε_c，右下角变为 h，右上角变为与 h 对应的箭头，我们称其为 f：

$$\begin{array}{ccc} \varepsilon_c & \xmapsto{*\circ Lh} & f \\ \updownarrow {\scriptstyle \phi_{Rc,c}} & & \updownarrow {\scriptstyle \phi_{d,c}} \\ id_{Rc} & \xmapsto{*\circ h} & h \end{array}$$

然后，顶部的箭头给出了我们所寻找的等式 $f = (*\circ Lh)\varepsilon_c = \varepsilon_c \circ Lh$。

从泛箭推导出的伴随

反过来说，结果更加有趣。如果对于每个 c，我们都有从 L 到 c 的一个泛箭，即逗号范畴 L/c 中的终对象 $\langle t_c, \varepsilon_c \rangle$，那么我们可以构造一个函子 R，它是 L 的右伴随。该函子对对象的作用由 $Rc = t_c$ 给出，并且 ε_c 在 c 中自动是自然的，并且构成了伴随的伴随元。

还有一个对偶的陈述：可以从泛箭 η_d 的族开始构造一个伴随，这些泛箭构成了逗号范畴 d/R 中的初始对象。

这些结果将帮助我们证明 Freyd 的伴随函子定理。

10.7 伴随的性质

左伴随保持余极限

我们将余极限定义为泛余锥。对于每个余锥——即从图式 $D\colon \mathcal{J} \to \mathcal{C}$ 到常函子 Δ_x 的自然变换——应该有一个从余极限 Colim D 到 x 的唯一分解态射。这个条件可以写为余锥集和特定同态集之间的一对一对应：

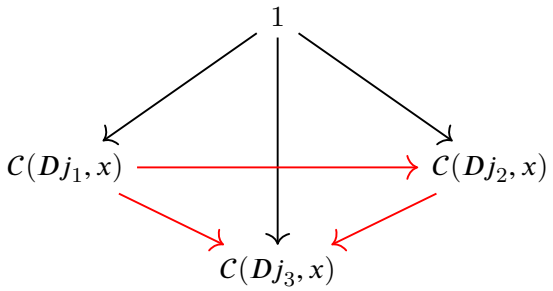
$$[\mathcal{J}, \mathcal{C}](D, \Delta_x) \ddot{\circ} \mathcal{C}(\operatorname{Colim} D, x)$$

分解条件被编码在这个同构的自然性中。

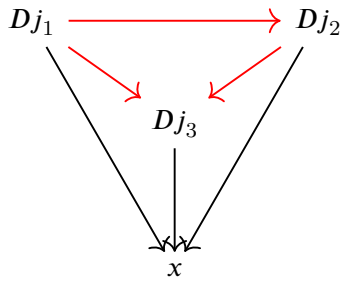
事实证明，余锥集是 Set 中的一个对象，它本身是以下 Set 值函子 $F\colon \mathcal{J} \to \mathrm{Set}$ 的极限：

$$Fj = \mathcal{C}(Dj, x)$$

为了展示这一点，我们将从 F 的极限开始，并最终得到余锥集。您可能记得，一个 Set 值函子的极限等于以单集 1 为顶点的锥的集合。在我们的例子中，每个这样的锥描述了从相应的同态集 $\mathcal{C}(Dj, x)$ 中选择态射：



这些态射的目标都是相同的对象 x，因此它们构成了以 x 为顶点的余锥的边。



以 1 为顶点的锥的交换条件同时也是以 x 为顶点的这个余锥的交换条件。但这些正是集 $[\mathcal{J}, \mathcal{C}](D, \Delta_x)$ 中的余锥。

因此，我们可以用 $\mathcal{C}(D^*, x)$ 的极限替换原始的余锥集，以得到：

$$\mathrm{Lim}\ \mathcal{C}(D^*, x) \ddot{\circ} \mathcal{C}(\mathrm{Colim}\ D, x)$$

同态函子的逆变同态函子有时记为：

$$h_x = \mathcal{C}(*, x)$$

在这个记号中，我们可以写成：

$$Lim\,(h_x \circ D) \ddot{\circ} h_x(Colim\ D)$$

作用于图式 D 的同态函子的极限同构于作用于该图式的余极限的同态函子。这通常被缩写为：同态函子保持余极限。（理解为逆变同态函子将余极限转化为极限。）

保持余极限的函子称为余连续函子。因此，逆变同态函子是余连续的。

现在假设我们有伴随 $L \dashv R$，其中 $L\colon \mathcal{C} \to \mathcal{D}$，$R$ 反方向运行。我们想要证明左函子 L 保持余极限，即：

$$L(\mathrm{Colim}\ D) \ddot{\circ} \mathrm{Colim}(L \circ D)$$

对于任何存在余极限的图式 $D\colon \mathcal{J} \to \mathcal{C}$。

我们将使用 Yoneda 引理来证明两边映射到任意 x 的映射是同构的：

$$\mathcal{D}(L(\mathrm{Colim}\ D), x) \ddot{\circ} \mathcal{D}(\mathrm{Colim}(L \circ D), x)$$

我们将伴随应用于左边，以得到：

$$\mathcal{D}(L(\mathrm{Colim}\ D), x) \ddot{\circ} \mathcal{C}(\mathrm{Colim}\ D, Rx)$$

同态函子保持余极限给我们：

$$\ddot{\circ}\ \mathrm{Lim}\ \mathcal{C}(D^*, Rx)$$

再次使用伴随，我们得到：

$$\ddot{\circ}\ \mathrm{Lim}\ \mathcal{D}((L \circ D)^*, x)$$

同态函子的第二次应用保持余极限给出了我们想要的结果：

$$\ddot{o}\ \mathcal{D}((\text{Colim}\ (L \circ D), x)$$

因为这对任何 x 都是成立的，所以我们得到了我们的结果。

我们可以使用这个结果来重新表述我们在笛卡尔封闭范畴中的分配律的早期证明。我们使用了积是指数的左伴随的事实。左伴随保持余极限。余积是一个余极限，因此：

$$(b + c) \quad a \ddot{o}\ b \quad a + c \quad a$$

这里，左函子是 $Lx = x \quad a$，图式 D 选择一对对象 b 和 c。

右伴随保持极限

使用对偶论证，我们可以证明右伴随保持极限，即：

$$R(\text{Lim}\ D)\ \ddot{o}\ \text{Lim}\ (R \circ D)$$

我们首先证明同态函子保持极限。

$$\text{Lim}\ \mathcal{C}(x, D^*)\ \ddot{o}\ \mathcal{C}(x, \text{Lim}\ D)$$

这可以从一个论证得出，即定义极限的锥集同构于 Set 值函子的极限：

$$Fj = \mathcal{C}(x, Dj)$$

保持极限的函子称为连续函子。

为了证明给定的伴随 $L \dashv R$，右函子 $R\colon \mathcal{D} \to \mathcal{C}$ 保持极限，我们使用 Yoneda 论证：

$$\mathcal{C}(x, R(\text{Lim}\ D))\ \ddot{o}\ \mathcal{C}(x, \text{Lim}\ (R \circ D))$$

实际上，我们有：

$$\mathcal{C}(x, R(\text{Lim}\ D))\ \ddot{o}\ \mathcal{D}(Lx, \text{Lim}\ D)\ \ddot{o}\ \text{Lim}\ \mathcal{D}(Lx, D^*)\ \ddot{o}\ \mathcal{C}(x, \text{Lim}\ (R \circ D))$$

10.8 Freyd 的伴随函子定理

一般来说，函子是有损的——它们不可逆。在某些情况下，我们可以通过用“最佳猜测”替换丢失的信息来弥补这一点。如果我们以有组织的方式进行，我们最终会得到一个伴随。问题是：给定两个范畴之间的一个函子，在什么条件下我们可以构造出它的伴随？

Freyd 的伴随函子定理给出了这个问题的答案。起初，这似乎是一个涉及非常抽象构造的技术性定理，称为解集条件。我们稍后会看到，这个条件直接转化为一种编程技术，称为去函数化。

在接下来的内容中，我们将重点放在构造一个函子 $L\colon \mathcal{D} \to \mathcal{C}$ 的右伴随上。可以使用对偶推理来解决寻找函子 $R\colon \mathcal{C} \to \mathcal{D}$ 的左伴随的相反问题。

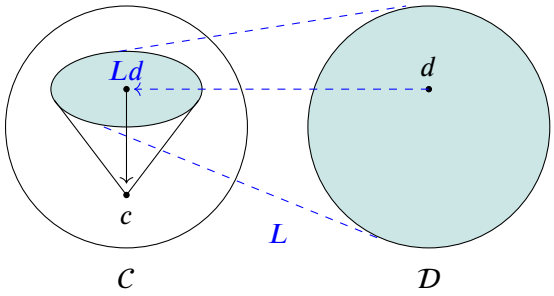
第一个观察是，由于伴随中的左函子保持余极限，我们必须假定我们的函子 L 保持余极限。这给了我们一个提示，即右伴随的构造依赖于在 $\mathcal{D}$ 中构造余极限的能力，并能够通过某种方式将它们带回 $\mathcal{C}$，使用 L。

我们可以要求 $\mathcal{D}$ 中的所有余极限，无论大小，都存在，但这个条件太强了。即使是一个拥有所有余极限的小范畴也会自动成为预序——也就是说，在任何两个对象之间都不能有多个态射。

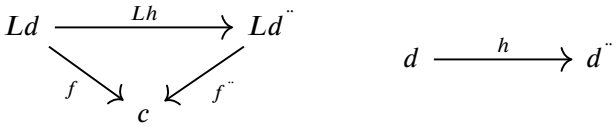
但是让我们暂时忽略大小问题，看看如何为一个保持余极限的函子 L 定义右伴随，其源范畴 $\mathcal{D}$ 是小的并且具有所有余极限，无论大小（因此它是一个预序）。

Freyd 定理在预序中的应用

定义 L 的右伴随的最简单方法是为每个对象 c 构造一个从 L 到 c 的泛箭。这样的箭头是逗号范畴 L/c 中的终对象——这个范畴中的箭头起源于 L 的像，并汇聚于对象 c。

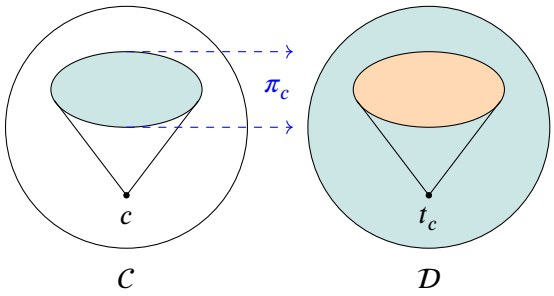


重要的观察是这个逗号范畴描述了 $\mathcal{C}$ 中的一个余锥。这个余锥的基底由那些能够不受阻碍地看到 c 的对象构成。这些基底中的箭头是 L/c 中的态射。正是这些箭头构成了余锥的边。



然后可以将这个余锥的基底投影回 $\mathcal{D}$。存在一个投影 π_c，它将 L/c 中的每一对 (d, f) 映射回 d，从而忽略了箭头 f。它还将 L/c 中的每个态射映射到产生它的 $\mathcal{D}$ 中的箭头。通过这种方式，π_c 定义了一个在 $\mathcal{D}$ 中的图式。这个图式的余极限存在，因为我们假设在 $\mathcal{D}$ 中存在所有余极限。我们称这个余极限为 t_c：

$$t_c = \operatorname{colim} \pi_c$$



让我们看看是否可以使用这个 t_c 来构造逗号范畴 L/c 中的终对象。我们必须找到一个箭头，我们称之为 $\varepsilon_c \colon Lt_c \to c$，使得对 $\langle t_c, \varepsilon_c \rangle$ 在 L/c 中是终的。

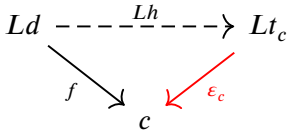
请注意，L 将由 π_c 生成的图式映射回由 L/c 定义的余锥的基底。投影 π_c 只做了忽略这个余锥的边，而保留其基底。

我们现在在 $\mathcal{C}$ 中有两个余锥，它们具有相同的基底：一个顶点为 c 的原始余锥和一个通过对 $\mathcal{D}$ 中的余锥应用 L 得到的新余锥。由于 L 保持余极限，新余锥的余极限是 Lt_c ——即余极限 t_c 的像：

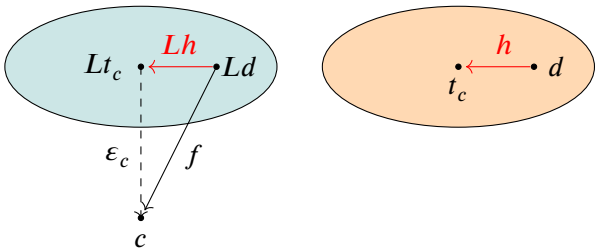
$$\operatorname{colim}\,(L \circ \pi_c) = L(\operatorname{colim}\,\pi_c) = Lt_c$$

通过泛构造，我们推断出必须存在从余极限 Lt_c 到 c 的唯一余锥态射。这个态射，我们称之为 ε_c，使得所有相关的三角形交换。

剩下要证明的是 $\langle t_c, \varepsilon_c \rangle$ 是 L/c 中的终对象，也就是说，对于任意 $\langle d, f \colon Ld \to c \rangle$，存在一个唯一的逗号范畴态射 $h \colon d \to t_c$，使得下图三角形交换：



请注意，任何这样的 d 都自动成为由 π_c 生成的图式的一部分（它是 π_c 作用于 $\langle d, f \rangle$ 的结果）。我们知道 t_c 是 π_c 图式的极限。因此，在极限余锥中，必定有一条从 d 到 t_c 的线。我们选择这条线作为我们的 h。



然后，交换条件从 ε_c 作为两个余锥之间的态射得出。它是唯一的余锥态射，因为 $\mathcal{D}$ 是一个预序。

这证明了对于每一个 c，都有一个从 L 到 c 的泛箭，因此我们有一个定义在对象上的函子 $Rc = t_c$，它是 L 的右伴随。

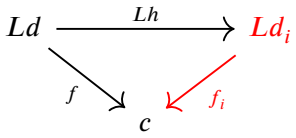
解集条件

前面证明的问题在于，大多数实际情况下的逗号范畴是大的：它们的对象并不构成一个集合。但也许我们可以通过选择较小但有代表性的对象和箭头集来近似逗号范畴?

为了选择对象，我们可以使用从某个索引集 I 的映射。我们定义一组对象 d_i，其中 $i \in I$。由于我们试图近似逗号范畴 L/c，我们选择带有箭头 $f_i : Ld_i \to c$ 的对象。

逗号范畴的相关部分编码在满足交换条件的对象之间的态射中。我们可以尝试将此条件专门化为仅在我们的对象族内应用，但这还不够。我们必须找到一种方法来探测逗号范畴中的所有其他对象。

为此，我们将交换条件重新解释为一个分解任意 $f : Ld \to c$ 通过某个对 $\langle d_i, f_i \rangle$ 的配方：



一个解集是一个由索引集 I 索引的对 $\langle d_i, f_i : Ld_i \to c \rangle$ 的族，可以用来分解任意对 $\langle d, f : Ld \to c \rangle$。这意味着存在一个索引 $i \in I$ 和一个箭头 $h : d \to d_i$，它分解 f：

$$f = f_i \circ Lh$$

表达这个属性的另一种方式是说在逗号范畴 L/c 中存在一个*弱终*的对象集。一个弱终集具有的属性是，对于范畴中的任何对象，都存在一个到该集中的至少一个对象的态射。

我们之前已经看到，对于每个 c，逗号范畴 L/c 中的终对象的存在足以定义伴随。事实证明，我们可以使用解集实现同样的目标。

Freyd 的伴随函子定理的假设规定我们有一个从一个小的完备范畴到另一个范畴的保持余极限的函子 $L : \mathcal{D} \to \mathcal{C}$。这两个条件都与小图式有关。如果我们可以为每个 c 选择一个解集 $\langle d_i, f_i : Ld_i \to c \rangle$，那么右伴随 R 存在。不同的 c 的解集可能不同。

我们之前已经看到，在一个完备范畴中，弱终集的存在足以定义一个终对象。在我们的情况下，这意味着对于任何 c，我们都可以构造从 L 到 c 的泛箭。而这足以定义整个伴随。

伴随函子定理的对偶版本可用于构造左伴随。

去函数化

每种编程语言都允许我们定义函数，但并非所有语言都支持高阶函数（以函数为参数的函数，返回函数的函数，或由函数构造的数据类型）或匿名函数（又称 lambda）。事实

证明，即使在这种语言中，高阶函数也可以通过称为去函数化的过程来实现。这种技术基于伴随函子定理。此外，每当传递函数是不切实际的时候，例如在分布式系统中，都可以使用去函数化。

去函数化背后的思想是函数类型被定义为积的右伴随。

$$C(e \quad a, b) \ddot{o} \, C(e, b^a)$$

伴随函子定理可以用来近似这个伴随。

一般来说，任何有限的程序只能有有限数量的函数定义。这些函数（连同它们捕获的环境）形成了解集，我们可以用它来构造函数类型。在实践中，我们只为少数作为参数的函数或从其他函数返回的函数执行此操作。

使用高阶函数的典型例子是继续传递风格。例如，下面是一个计算列表元素和的函数。但它不是返回和，而是使用结果调用一个继续`k`：

```
sumK :: [Int] -> (Int -> r) -> r
sumK [] k = k 0
sumK (i : is) k =
sumK is (\s -> k (i + s))
```

如果列表为空，则该函数调用继续，和为零。否则，它以两个参数递归调用自己：列表`is`的尾部和一个新的继续：

```
\s -> k (i + s)
```

这个新的继续调用先前的继续`k`，并传递列表头的和及其参数`s`（这是累积的和）。

请注意，这个 lambda 是一个闭包：它是一个变量`s`的函数，但它也可以访问来自其环境的`k`和`i`。

要提取最终的和，我们用平凡的继续调用我们的递归函数，即恒等：

```
sumList :: [Int] -> Int
sumList as = sumK as (\i -> i)
```

匿名函数很方便，但没有什么能阻止我们使用命名函数。然而，如果我们想要将继续提取出来，我们必须明确传递环境。

例如，我们可以替换我们的第一个 lambda：

```
\s -> k (i + s)
```

用函数`more`，但我们必须显式传递对`(i, k)`作为类型`(Int, Int -> r)`的环境：

```
more :: (Int, Int -> r) -> Int -> r
more (i, k) s = k (i + s)
```

另一个 lambda，恒等，使用空环境，因此它变为：

```
done :: Int -> Int
done i = i
```

以下是使用这两个命名函数的算法实现：

```
sumK' :: [Int] -> (Int -> r) -> r
sumK' [] k = k 0
sumK' (i : is) k =
sumK' is (more (i, k))
```

```
sumList :: [Int] -> Int
sumList is = sumK' is done
```

事实上，如果我们只对计算和感兴趣，我们可以将多态类型`r`替换为`Int`，而不做其他更改。

这个实现仍然使用高阶函数。为了消除它们，我们必须分析传递函数作为参数的含义。这样的函数只能以一种方式使用：可以将其应用于其参数。函数类型的这个属性表示为柯里化伴随的伴随元：

$$\varepsilon : b^a \quad a \to b$$

或者，在 Haskell 中，表示为一个高阶函数：

```
apply :: (a -> b, a) -> b
```

这次我们有兴趣从第一个原理构造伴随元。我们已经看到，这可以通过逗号范畴实现。在我们的例子中，积函子 $L_a = (*) \quad a$ 的逗号范畴的一个对象是一个对

$$(e, f : (e \quad a) \to b)$$

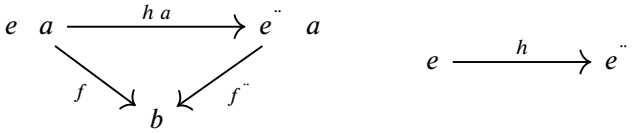
或者，在 Haskell 中：

```
data Comma a b e = Comma e ((e, a) -> b)
```

这个范畴中 (e, f) 和 $(e^{¨}, f^{¨})$ 之间的态射是一个箭头 $h : e \to e^{¨}$，它满足交换条件：

$$f^{¨} \circ h = f$$

我们将这个态射解释为“缩减”环境 e 到 $e^{¨}$。箭头 $f^{¨}$ 能够使用由 $h(e)$ 给出的可能较小的环境产生相同类型为 b 的输出。例如，e 可能包含与从 a 计算 b 无关的变量，而 h 将它们投影出去。



事实上，我们在定义more和done时执行了这种类型的缩减。原则上，我们可以将尾部is传递给这两个函数，因为它在调用点是可访问的。但我们知道它们不需要它。

使用 Freyd 的定理，我们可以将函数对象 $a \to b$ 定义为由逗号范畴定义的图式的余极限。这样的余极限本质上是所有环境的巨大余积，模态射的识别。这种识别将 $a \to b$ 所需的环境减少到最低限度。

在我们的例子中，我们感兴趣的继续是函数`Int -> Int`。事实上，我们并不想生成通用的函数类型`Int -> Int`；只是要容纳我们的两个函数more和done的最小函数。我们可以通过创建一个非常小的解集来实现它。

在我们的例子中，解集由对 $(e_i, f_i : e_i\ a \to b)$ 组成，使得任何对 $(e, f : e\ a \to b)$ 都可以通过其中一个 f_i 的分解。更确切地说，我们感兴趣的唯一两个环境是`(Int, Int ->Int)`用于more，以及空环境`()`用于done。

原则上，我们的解集应该允许分解逗号范畴的每个对象，即类型为：

```
(e, (e, Int) -> Int)
```

但这里我们只对两个特定函数感兴趣。同样，我们并不关心表示的唯一性，因此，我们将使用所有感兴趣环境的余积（如我们在伴随函子定理中所做的），而不是使用余极限。我们最终得到了以下数据类型，它是我们感兴趣的两个环境`()`和`(Int, Int -> Int)`的和。我们最终得到了以下类型：

```
data Kont = Done | More Int Kont
```

请注意，我们递归地将环境的`Int->Int`部分编码为`Kont`。因此，我们还消除了使用函数作为数据构造函数参数的需要。

如果仔细观察这个定义，你会发现它是一个`Int`列表的定义，模一些重命名。每次调用`More`都会在`Kont`堆栈上推送另一个整数。这个解释与我们的直觉一致，即递归算法需要某种运行时堆栈。

我们现在准备实现我们的伴随元的近似。它由两个函数的主体组成，理解是递归调用也通过apply：

```
apply :: (Kont, Int) -> Int
apply (Done, i) = i
apply (More i k, s) = apply (k, i + s)
```

将此与我们之前的比较：

```
done i = i
more (i, k) s = k (i + s)
```

现在可以重写我们的主要算法，而不使用任何高阶函数或 lambdas：

```
sumK'' :: [Int] -> Kont -> Int
sumK'' [] k = apply (k, 0)
```

```
        sumK'' (i : is) k = sumK'' is (More i k)

        sumList'' is = sumK'' is Done
```

去函数化的主要优势在于它可以在分布式环境中使用。远程函数的参数，只要它们是数据结构而不是函数，就可以序列化并通过网络传递。所需的只是接收者可以访问apply。

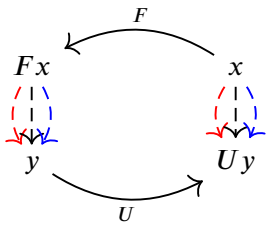
10.9 自由/遗忘伴随（Free/Forgetful Adjunctions）

在伴随关系中，两种函子扮演着不同的角色：伴随关系的图示并不是对称的。没有什么比自由/遗忘伴随关系更能说明这一点了。

遗忘函子（forgetful functor）是指“遗忘”了其源范畴某些结构的函子。这并不是一个严格的定义，但在大多数情况下，遗忘了哪些结构是显而易见的。通常，目标范畴只是集合范畴（Set），这被认为是没有结构的典范。在这种情况下，遗忘函子的结果被称为“底层集合”（underlying set），而该函子本身通常称为 U。

更确切地说，我们说一个函子遗忘了*结构*，如果它的同态集的映射不是满射的，即在目标同态集中有些箭头在源同态集中没有对应的箭头。直观上，这意味着源范畴中的箭头具有某些要保留的结构，所以它们的数量较少；而在目标范畴中这种结构是不存在的。

遗忘函子的左伴随函子被称为*自由函子*（free functor）。

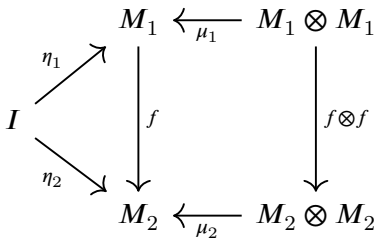


自由/遗忘伴随的一个经典例子是自由幺半群（free monoid）的构造。

幺半群范畴（The Category of Monoids）

在一个幺半范畴 C 中，幺半群构成自己的范畴 $\mathrm{Mon}(C)$。它的对象是幺半群，而它的箭头是保留幺半结构的 C 中的箭头。

下图解释了从一个幺半群 (M_1, η_1, μ_1) 到另一个幺半群 (M_2, η_2, μ_2) 的幺半群态射 f 的含义：



幺半群态射 f 必须将单位映射为单位，即：

$$f \circ \eta_1 = \eta_2$$

并且它必须将乘法映射为乘法：

$$f \circ \mu_1 = \mu_2 \circ (f \otimes f)$$

请记住，张量积 $\otimes$ 是函子性的，因此它可以提升箭头对，如 $f \otimes f$。

特别地，范畴 Set 是单积范畴，具有笛卡尔积和提供单积结构的终对象。

具体来说，Set 中的幺半群是带有附加结构的集合。它们构成自己的范畴 Mon(Set)，并且有一个遗忘函子 U，它简单地将幺半群映射为其元素的集合。当我们说一个幺半群是一个集合时，我们指的是它的底层集合。

自由幺半群（Free Monoid）

我们要构造自由函子

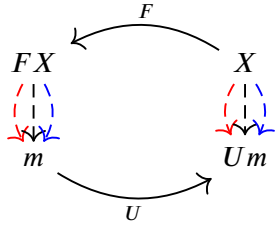
$$F : \mathrm{Set} \to \mathrm{Mon}(\mathrm{Set})$$

它是遗忘函子 U 的伴随函子。

我们从一个任意集合 X 和一个任意幺半群 m 开始。在伴随关系的右侧，我们有从 X 到 Um 的函数集。在左侧，我们有一个从 FX 到 m 的高度受约束的保结构幺半群态射集。如何使这两个同态集同构呢？

在 Mon(Set) 中，幺半群是元素的集合，幺半群态射是这些集合之间的函数，满足附加约束：保留单位和乘法。

另一方面，Set 中的箭头只是没有附加约束的函数。因此，一般来说，幺半群之间的箭头比它们底层集合之间的箭头要少。



这里的想法是：如果我们想要箭头之间有一一对应，我们希望 FX 比 X 大得多。这样一来，从 FX 到 m 的函数会多得多——即使在剔除那些不保留结构的函数后，我们仍然有足够多的函数可以匹配每一个 $f\colon X \to Um$。

我们将从集合 X 开始构造幺半群 FX，并在此过程中添加越来越多的元素。我们将初始集合 X 称为 FX 的**生成元**。我们将从原始函数 f 开始，逐步扩展它，使其作用于越来越多的元素，从而构造幺半群态射 $g\colon FX \to m$。

在生成元 $x \in X$ 上，g 的作用与 f 相同：

$$gx = fx$$

由于 FX 应该是一个幺半群，因此它必须有一个单位。我们不能选择一个生成元作为单位，因为这会对由 f 固定的 g 部分施加约束——它必须将该生成元映射为 m 的单位 e。所以我们只需向 FX 中添加一个额外的元素 e，并称之为单位。我们通过将 g 映射到 m 的单位 e 来定义 g 在它上的作用：

$$ge = e$$

我们还必须定义 FX 中的幺半乘法。让我们从两个生成元 a 和 b 的乘积开始。乘法的结果不能是另一个生成元，因为这同样会对 f 已经固定的 g 部分施加约束——乘积必须映射为乘积。因此，我们必须将生成元的所有乘积都作为 FX 中的新元素。同样，g 在这些乘积上的作用是固定的：

$$g(a \cdot b) = ga \cdot gb$$

继续这种构造，任何新的乘法都会产生 FX 的一个新元素，除非它可以通过应用幺半群定律简化为一个已有的元素。例如，新单位 e 乘以生成元 a 必须等于 a。但我们已经确保 e 映射为 m 的单位，因此乘积 $ge \cdot ga$ 自动等于 ga。

另一种看待此构造的方法是将集合 X 视为字母表。然后，FX 的元素是由此字母表中的字符组成的字符串。生成元是单字母字符串，如“a”、“b”等。单位是一个空字符串“”。乘法是字符串连接，所以“a”乘以“b”是一个新字符串“ab”。连接是自动的结合的和单位的，空字符串作为单位。

自由函子背后的直觉是它们“自由地”生成结构，就像“没有额外约束”一样。它们还以懒惰的方式进行：它们记录操作，而不是执行操作。它们创建了特定领域的通用程序，可以由特定的解释器稍后执行。

自由幺半群“记住了稍后进行乘法”。它将乘法的参数存储在字符串中，但不执行乘法。它只允许根据通用幺半群定律简化其记录。例如，它不必存储乘以单位的指令。它也可以“跳过括号”，因为它是结合的。

Exercise 10.9.1. 自由幺半群伴随 $F \dashv U$ 的单位元（*unit*）和伴随元（*counit*）是什么？

编程中的自由幺半群（Free Monoid in Programming）

在 Haskell 中，幺半群通过以下类型类定义：

```
class Monoid m where
mappend :: m -> m -> m
mempty  :: m
```

这里，mappend是从积映射到映射的柯里化形式：(m, m) -> m。元素mempty对应于从终对象（单积范畴的单位元）的箭头，或简单地说，是m的一个元素。

由某种类型a生成的自由幺半群，即作为生成元的集合，由列表类型[a]表示。空列表作为单位；幺半群乘法实现为列表连接，传统上写成中缀形式：

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys
```

列表是Monoid的一个实例：

```
instance Monoid [a] where
mempty = []
mappend = (++)
```

要证明它是一个自由幺半群，我们必须能够从a的列表构造一个到任意幺半群m的幺半群态射，前提是我们有一个（不受约束的）从a到（m的底层集合）的映射。我们不能在 Haskell 中表达这一切，但我们可以定义如下函数：

```
foldMap :: Monoid m => (a -> m) -> ([a] -> m)
foldMap f = foldr mappend mempty . fmap f
```

此函数使用f将列表元素转换为幺半群值，然后使用mappend折叠它们，从单位mempty开始。

很容易看出，空列表映射为幺半群的单位。也不难看出，两个列表的连接映射为结果的幺半积。因此，确实，foldMap是一个幺半群态射。

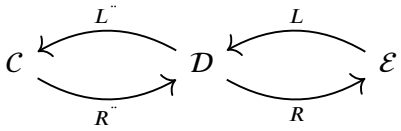
根据自由幺半群作为领域特定乘法程序的直觉，foldMap为该程序提供了一个解释器。它执行所有已延迟的乘法操作。请注意，取决于具体幺半群和函数f的选择，相同的程序可以用许多不同的方式进行解释。

我们将在关于代数的章节中回到作为列表的自由幺半群。

Exercise 10.9.2. *编写一个程序，接受一个整数列表，并以两种不同的方式对其进行解释：一次使用整数的加法幺半群，一次使用整数的乘法幺半群。*

10.10 伴随的范畴（The Category of Adjunctions）

我们可以通过利用定义它们的函子的组合来定义伴随的组合。如果两个伴随 $L \dashv R$ 和 $L'' \dashv R''$，它们共享中间的范畴，则它们是可组合的：



通过组合函子，我们得到一个新的伴随 $(L'' \circ L) \dashv (R \circ R'')$。

实际上，让我们考虑同态集：

$$\mathcal{C}(L''(Le), c)$$

使用 $L'' \dashv R''$ 伴随，我们可以将 L'' 转移到右侧，使其成为 R''：

$$\mathcal{D}(Le, R''c)$$

并且使用 $L \dashv R$ 我们可以类似地将 L 转移：

$$\mathcal{E}(e, R(R''c))$$

结合这两个同构，我们得到复合伴随：

$$\mathcal{C}((L'' \circ L)e, c) \ddot{\text{o}} \mathcal{E}(e, (R \circ R'')c)$$

由于函子组合是结合的，伴随的组合也是结合的。很容易看出，一对恒等函子形成一个平凡的伴随，作为伴随组合的恒等。因此，我们可以定义一个范畴 Adj(Cat)，其中对象是范畴，箭头是伴随（按照惯例，指向左伴随的方向）。

伴随可以纯粹用函子和自然变换定义，也就是 2-范畴 Cat 中的 1-态射和 2-态射。Cat 没有什么特别之处，实际上伴随可以在任何 2-范畴中定义。此外，伴随的范畴本身就是一个 2-范畴。

10.11 抽象的层次（Levels of Abstraction）

范畴论是关于组织我们的知识的。特别是，它可以应用于范畴论知识本身。因此，我们在范畴论中看到很多抽象层次的混合。在一个层次上看到的结构可以组合成更高层次的结构，这些结构展示了更高层次的结构，依此类推。

在编程中，我们习惯于构建抽象层次。值被分组为类型，类型被分组为种类。操作值的函数与操作类型的函数被区别对待。我们通常使用不同的语法来分隔抽象层次。在范畴论中并非如此。

从范畴论的角度来看，一个集合可以被描述为一个离散范畴。集合的元素是该范畴的对象，除了强制性的恒等态射外，它们之间没有箭头。

然后，同一个集合可以在范畴 Set 中被视为一个对象。该范畴中的箭头是集合之间的函数。

范畴 Set，反过来，是范畴 Cat 中的一个对象。Cat 中的箭头是函子。

任意两个范畴 $\mathcal{C}$ 和 $\mathcal{D}$ 之间的函子是函子范畴 $[\mathcal{C}, \mathcal{D}]$ 中的对象。该范畴中的箭头是自然变换。

我们可以定义函子范畴之间的函子、积范畴、对偶范畴，依此类推，无限延伸。

完成这一循环，每个范畴中的同态集都是集合。我们可以定义它们之间的映射和同构，跨越不同的范畴。由于我们可以比较不同范畴中的同态集，因此伴随关系是可能的。

Chapter 11

依赖类型 (Dependent Types)

我们已经见过依赖于其他类型的类型。它们是使用类型构造器和类型参数定义的，例如 `Maybe` 或 `[]`。大多数编程语言都支持泛型数据类型——即参数化的类型。

在范畴论中，这类类型被建模为函子[1]。

一个自然的推广是定义依赖于值的类型。例如，将列表的长度编码到它的类型中通常是有利的。长度为零的列表将具有与长度为一的列表不同的类型，依此类推。

显然，不能改变这种列表的长度，因为这会改变其类型。在函数式编程中，这不是问题，因为所有的数据类型都是不可变的。当向列表前面添加一个元素时，至少在概念上，会创建一个新的列表。对于长度编码的列表，新列表只是一个不同的类型！

这些由值参数化的类型被称为*依赖类型*。像 Idris 或 Agda 这样的语言完全支持依赖类型。在 Haskell 中也可以实现依赖类型，但对它们的支持仍然有限。

在编程中使用依赖类型的原因是为了使程序在形式上是正确的。为了做到这一点，编译器必须能够检查程序员做出的假设。

Haskell 以其强类型系统，能够在编译时揭示很多错误。例如，除非为变量的类型提供 `Monoid` 实例，否则不会允许编写 `a <> b`（`mappend` 的中缀表示法）。

然而，在 Haskell 的类型系统中，没有办法表达或更不用说强制执行幺半群的单位元和结合律。为了实现这一点，`Monoid` 类型类的实例必须携带等式的证明（而不是实际的代码）：

```
assoc :: m <> (n <> p) = (m <> n) <> p
lunit :: mempty <> m = m
runit :: m <> mempty = m
```

依赖类型，特别是等式类型，为实现这一目标铺平了道路。

本章的内容更加高级，并且在本书的其他部分中不会使用，因此您可以在第一次阅

[1]没有 `Functor` 实例的类型构造器可以被认为是来自离散范畴的函子——一个除了恒等箭头外没有其他箭头的范畴。

读时跳过。此外，为避免在纤维和函数之间产生混淆，我决定在本章的部分内容中使用大写字母表示对象。

11.1 依赖向量 (Dependent Vectors)

我们将从一个标准的例子开始，即计数列表或向量：

```
data Vec n a where
VNil  :: Vec Z a
VCons :: a -> Vec n a -> Vec (S n) a
```

如果您包含以下语言声明，编译器将识别此定义为依赖类型：

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
```

类型构造器的第一个参数是自然数 `n`。请注意：这是一个值，而不是类型。类型检查器能够从数据构造器中 `n` 的使用情况推断出来。第一个构造器创建类型为 `Vec Z a` 的向量，第二个构造器创建类型为 `Vec (S n) a` 的向量，其中 `Z` 和 `S` 被定义为自然数的构造器：

```
data Nat = Z | S Nat
```

如果我们使用以下语言声明可以更明确地指定参数：

```
{-# LANGUAGE KindSignatures #-}
```

并导入以下库：

```
import Data.Kind
```

然后我们可以指定 `n` 是一个 `Nat`，而 `a` 是一个 `Type`：

```
data Vec (n :: Nat) (a :: Type) where
VNil  :: Vec Z a
VCons :: a -> Vec n a -> Vec (S n) a
```

使用这些定义之一，我们可以构造一个长度为零的整数向量：

```
emptyV :: Vec Z Int
emptyV = VNil
```

它的类型与长度为一的向量不同：

```
singleV :: Vec (S Z) Int
singleV = VCons 42 VNil
```

等等。

我们现在可以定义一个依赖类型的函数，该函数返回向量的第一个元素：

```
headV :: Vec (S n) a -> a
headV (VCons a _) = a
```

此函数保证仅适用于非零长度的向量。这些是大小符合 (S n) 的向量，这些向量不能是 Z。如果尝试使用 emptyV 调用此函数，编译器将会报错。

另一个例子是将两个向量压缩在一起的函数。其类型签名中编码了两个向量的大小相同 n 的要求（结果也是 n 大小）：

```
zipV :: Vec n a -> Vec n b -> Vec n (a, b)
zipV (VCons a as) (VCons b bs) = VCons (a, b) (zipV as bs)
zipV VNil VNil = VNil
```

依赖类型在编码容器的形状时尤其有用。例如，列表的形状被编码为其长度。一个更高级的例子是在运行时将树的形状编码为值。

Exercise 11.1.1. *实现函数 tailV，该函数返回非零长度向量的尾部。尝试使用 emptyV 调用它。*

11.2 范畴上的依赖类型 (Dependent Types Categorically)

可视化依赖类型的最简单方法是将它们视为由集合的元素索引的类型族。在计数向量的情况下，索引集合是自然数集合 ℕ。

第零个类型是表示空向量的单位类型 ()。与 (S Z) 对应的类型是 a；然后我们有一个 (a, a) 对，接下来是一个三元组 (a, a, a)，依此类推，随着 a 的幂次增加。

如果我们想将整个族视为一个大集合，我们可以取所有这些类型的和。例如，所有 a 的幂次和是熟悉的列表类型，即自由幺半群：

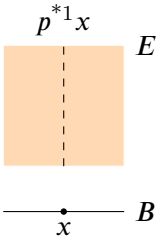
$$\mathrm{List}(a) = 1 + a + a\ \ a + a\ \ a\ \ a + \cdots = \sum_{n:\mathbb{N}} a^n$$

纤维 (Fibrations)

尽管直观上容易理解，但这种观点在推广到范畴论时效果不好，因为在范畴论中我们不喜欢将集合与对象混合。因此，我们将这个图景倒置，而不是谈论将族成员注入和中，我们考虑一个相反方向的映射。

首先，我们可以再次使用集合来可视化这个过程。我们有一个大集合 E 描述整个族，并有一个称为投影函数 p，或*显示映射*，该函数从 E 映射到索引集 B（也称为*基*）。

通常情况下，该函数将多个元素映射到一个元素。然后我们可以讨论某个特定元素 x Ë B 的逆映射作为 p 映射到它的元素的集合。该集合称为*纤维*，记作 $p^{*1}x$（尽管一般来说，p 不是通常意义上的可逆函数）。将 E 视为纤维的集合，E 通常称为*纤维丛*。



现在我们暂时忘记集合的概念。在任意范畴中，纤维化是对象 e 和 b 以及箭头 $p\colon e \to b$ 的一个对。

因此，这实际上只是一个箭头，但上下文至关重要。当箭头被称为纤维化时，我们使用集合的直觉，并将其源 e 想象为一个纤维集合，其中 p 将每个纤维投影到基 b 中的一个点。

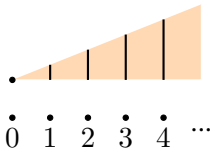
我们甚至可以更进一步：由于（小）范畴形成一个以函子为箭头的范畴 Cat，我们可以定义一个范畴的纤维化，并取另一个范畴作为其基。

作为纤维化的类型族 (Type Families as Fibrations)

因此，我们将类型族建模为纤维化。例如，我们的计数向量族可以表示为一个基为自然数类型的纤维化。整个族是连续幂次的和（上积）的并集：

$$\mathrm{List}(a) = a^0 + a^1 + a^2 + \cdots = \sum_{n\colon \mathbb{N}} a^n$$

其中第零幂次——初始对象——表示大小为零的向量。



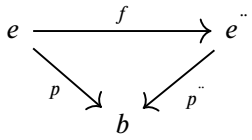
投影 $p\colon \mathrm{List}(a) \to \mathbb{N}$ 是熟悉的 length 函数。

在范畴论中，我们喜欢成批地描述事物——通过保持结构的映射定义事物的内部结构。纤维化就是这样一种情况。如果我们固定基对象 b 并考虑范畴 $\mathcal{C}$ 中所有可能的源对象，以及所有可能的投影到 b 的映射，我们得到一个商范畴 $\mathcal{C}/b$。这个范畴表示我们可以在基 b 上切分范畴 $\mathcal{C}$ 的所有方式。

回想一下，商范畴中的对象是对 $\langle e, p\colon e \to b\rangle$，而两个对象 $\langle e, p\rangle$ 和 $\langle \ddot{e}, \ddot{p}\rangle$ 之间的态射是箭头 $f\colon e \to \ddot{e}$，它与投影相通，即：

$$\ddot{p} \circ f = p$$

最好通过注意到这种态射将 p 的纤维映射到 $\ddot{p}$ 的纤维来可视化这一点。这是一个“保持纤维”的丛之间的映射。

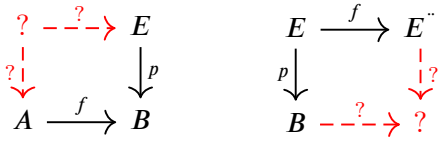


我们的计数向量可以看作是商范畴 $\mathcal{C}/\mathbb{N}$ 中的对象，由对 $\langle \mathrm{List}(a), \mathrm{length} \rangle$ 组成。此范畴中的态射将长度为 n 的向量映射为同样长度 n 的向量。

纤维化的拉回 (Pullbacks)

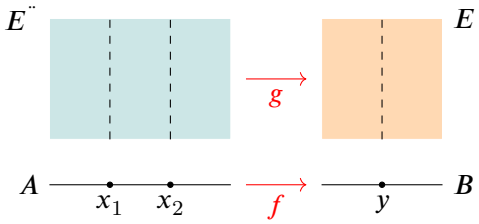
我们见过许多通方格的例子。这样的方格是一个等式的图形表示：两个路径连接方格的对角，每个路径都是两个态射组成的结果是相等的。

如同每个等式一样，我们可能希望用一个或多个未知量替换其组成部分，并尝试解决所得到的等式。例如，我们可以问：是否有一个对象与两个箭头一起能完成一个通方格？如果存在许多这样的对象，是否有一个是通用的？如果缺少拼图的部分是方格的左上角（源），我们称之为拉回（pullback）。如果是右下角（目标），我们称之为推送（pushout）。



让我们从一个特定的纤维化 $p\colon E \to B$ 开始，问问自己：当我们将基从 B 更改为通过映射 $f\colon A \to B$ 相关的某个 A 时，会发生什么？我们能否沿着 f “将纤维拉回”？

再次，我们先考虑集合。想象在 E 中选取一个纤维，在 B 中某个点 y 上方，该点属于 f 的像。若 f 是可逆的，则会有一个元素 $x = f^{*1}y$。我们会将纤维植入它的上方。然而，一般来说，f 不是可逆的。这意味着可能有多个 A 的元素映射到我们的 y。在下面的图中，您会看到两个这样的元素，x_1 和 x_2。我们只需将纤维克隆并植入所有映射到 y 的元素上。这种方式，每个 A 中的点都会有一个纤维从它生长出来。所有这些纤维的总和将形成一个新的纤维丛 $E^{¨}$。



因此我们构造了一个基为 A 的新纤维化。其投影 $p^{¨}\colon E^{¨} \to A$ 将每个点映射到其纤维被植入的点上。还有一个明显的映射 $g\colon E^{¨} \to E$，它将纤维映射到相应的纤维。

通过构造，这个新的纤维化 $\langle E'', p'' \rangle$ 满足条件：

$$p \circ g = f \circ p''$$

它可以表示为一个通方格：

$$\begin{array}{ccc} E'' & \xrightarrow{g} & E \\ {\scriptstyle p''}\downarrow & & \downarrow{\scriptstyle p} \\ A & \xrightarrow{f} & B \end{array}$$

在 Set 中，我们可以将 E'' 显式构造为笛卡尔积 $A \times E$ 的子集，其中 $p'' = \pi_1$，$g = \pi_2$（两个笛卡尔投影）。E'' 的元素是满足条件 $f(a) = p(e)$ 的对 $\langle a, e \rangle$。

这个通方格是范畴论推广的起点。然而，即使在 Set 中，仍然存在许多不同的 A 上方的纤维化，它们使该图表通方。我们必须选择通用的那个。这样的通用构造称为**拉回**，或称为**纤维积**。

在范畴论中，$p\colon e \to b$ 沿着 $f\colon a \to b$ 的拉回是一个对象 e'' 和两个箭头 $p''\colon e'' \to a$ 和 $g\colon e'' \to e$，它们使以下图表通方：

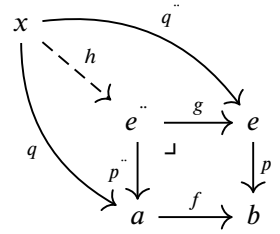
$$\begin{array}{ccc} e'' & \xrightarrow{g} & e \\ {\scriptstyle p''}\downarrow & & \downarrow{\scriptstyle p} \\ a & \xrightarrow{f} & b \end{array}$$

并满足通用条件。

通用条件表明，对于任何其他候选对象 x，它有两个箭头 $q''\colon x \to e$ 和 $q\colon x \to a$，使得 $p \circ q'' = f \circ q$（使大“方格”通方），存在一个唯一的箭头 $h\colon x \to e''$，使得两个三角形通方，即：

$$q = p'' \circ h$$
$$q'' = g \circ h$$

图形表示如下：



方格的上角的角度符号用于标记拉回。

如果我们通过集合和纤维的视角来看拉回，e 是 b 上的纤维丛，而我们正在从 e 中的纤维构造一个新的纤维丛。将这些纤维植入 a 上的位置由 f 的逆像决定。此过程使得 e'' 成为 a 和 b 上的纤维丛，后者的投影为 $p \circ g = f \circ p''$。

这个图中的 x 是某个 a 上的其他纤维丛，其投影为 q。它同时也是一个 b 上的纤维丛，其投影为 $f \circ q = p \circ q^{\cdot\cdot}$。唯一的映射 h 将 q^{*1} 给出的 x 的纤维映射到 $p^{\cdot\cdot *1}$ 给出的 $e^{\cdot\cdot}$ 的纤维。

这个图中的所有映射都作用在纤维上。有些映射重新排列纤维到新的基上——这是拉回的作用。其他映射修改单个纤维——映射 $h\colon x \to e^{\cdot\cdot}$ 这样工作。

如果您将纤维丛视为纤维的容器，纤维的重新排列对应于自然变换，而纤维的修改对应于 `fmap` 的作用。

然后通用条件告诉我们，$q^{\cdot\cdot}$ 可以分解为纤维的修改 h，然后是纤维的重新排列 g。

值得注意的是，选择终对象或单元素集合作为拉回目标会自动给出笛卡尔积的定义：

$$\begin{array}{ccc} b \;\; e & \xrightarrow{\pi_2} & e \\ {\scriptstyle \pi_1}\downarrow & \lrcorner & \downarrow{\scriptstyle !} \\ b & \xrightarrow{!} & 1 \end{array}$$

或者，我们可以将这个图看作是将尽可能多的 e 的副本种植到 b 的每个元素上。当我们谈论依赖和积时，我们将使用这个类比。

还请注意，可以通过将其拉回到终对象来从纤维化中提取单个纤维。在这种情况下，映射 $x\colon 1 \to b$ 选择了基的一个元素，沿着它的拉回提取单个纤维 φ：

$$\begin{array}{ccc} \varphi & \xrightarrow{g} & e \\ {\scriptstyle !}\downarrow & \lrcorner & \downarrow{\scriptstyle p} \\ 1 & \xrightarrow{x} & b \end{array}$$

箭头 g 将该纤维注入回 e。通过更改 x 我们可以选择 e 中的不同纤维。

Exercise 11.2.1. 证明终对象作为目标的拉回就是乘积。

Exercise 11.2.2. 证明可以将拉回定义为棒形范畴的图的极限，该范畴有三个对象：

$$a \to b \leftarrow c$$

Exercise 11.2.3. 证明 b 作为目标的 $\mathcal{C}$ 中的拉回是商范畴 $\mathcal{C}/b$ 中的乘积。提示：定义两个投影作为商范畴中的态射。使用拉回的通用性来证明乘积的通用性。

替换 (Substitution)

我们有两种描述依赖类型的替代方法：一种是作为纤维化，另一种是作为类型族。在后一种框架中，沿着态射 f 的拉回可以解释为替换。当我们有一个由元素 $y\colon \boldsymbol{B}$ 参数化的类型族 $T\,y$ 时，我们总是通过将 $f\,x$ 替换为 y 来定义一个新的类型族。

$$\begin{array}{ccc} T(f\,x) & & T\,y \\ \uparrow & & \uparrow \\ x & \xmapsto{f} & y \end{array}$$

因此，新的类型族由不同的形状参数化。

依赖环境 (Dependent Environments)

在建模 lambda 演算时，我们使用笛卡尔闭范畴的对象同时作为类型和环境。空环境被建模为终对象（单位类型），我们使用乘积构建更复杂的环境。由于乘积在同构（up to isomorphism）下是对称的，乘积类型的顺序无关紧要。

处理依赖类型时，我们必须考虑到添加到环境中的类型可能取决于环境中已经存在的类型的值。如前所述，我们从终对象表示的空环境开始。

弱化 (Weakening)

基变换函子 (Base-Change Functor)

我们使用笛卡尔闭范畴作为编程的模型。要建模依赖类型，我们需要施加一个附加条件：我们要求范畴是*局部笛卡尔闭范畴*。这是一个所有商范畴都是笛卡尔闭的范畴。

特别地，这种范畴具有所有拉回，因此始终可以更改任何纤维化的基。基变换在商范畴之间引入一个函子的映射。

给定两个商范畴 $\mathcal{C}/b$ 和 $\mathcal{C}/a$ 以及一个基之间的箭头 $f : b \to a$，基变换函子 $f^< : \mathcal{C}/a \to \mathcal{C}/b$ 将纤维化 $\langle e, p\rangle$ 映射到纤维化 $f^<\langle e, p\rangle = \langle f^< e, f^< p\rangle$，该纤维化由拉回给出：

$$\begin{array}{ccc} f^< e & \xrightarrow{g} & e \\ {\scriptstyle f^< p}\downarrow & \lrcorner & \downarrow{\scriptstyle p} \\ b & \xrightarrow{f} & a \end{array}$$

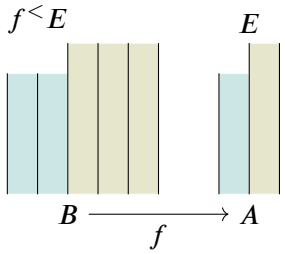
请注意，函子 $f^<$ 的方向与箭头 f 的方向相反。

为了可视化基变换函子，让我们考虑它如何作用于集合。

$$\begin{array}{ccc} f^< \boldsymbol{E} & \xrightarrow{g} & \boldsymbol{E} \\ {\scriptstyle f^< p}\downarrow & \lrcorner & \downarrow{\scriptstyle p} \\ \boldsymbol{B} & \xrightarrow{f} & \boldsymbol{A} \end{array}$$

我们有一个直觉，即纤维化 $\boldsymbol{p}$ 将集合 $\boldsymbol{E}$ 分解为每个 $\boldsymbol{A}$ 点上的纤维。

我们可以将 $\boldsymbol{f}$ 视为另一个纤维化，以类似方式分解 $\boldsymbol{B}$。我们将 $\boldsymbol{B}$ 中的纤维称为“片”。例如，如果 $\boldsymbol{A}$ 是一个两元素集合，那么由 $\boldsymbol{f}$ 给出的纤维化将 $\boldsymbol{B}$ 分成两个片。拉回则将 $\boldsymbol{E}$ 的纤维拉回并将其植入 $\boldsymbol{B}$ 中的每个片中。结果集 $f^<\boldsymbol{E}$ 看起来像一个拼布，其中每个片都种植了 $\boldsymbol{E}$ 的单个纤维的克隆。



由于我们有一个从 B 到 A 的函数，可能会将多个元素映射为一个，因此 B 上的纤维化比 A 上的纤维化更精细。将 E 在 A 上的纤维化转变为在 B 上的纤维化的最简单、最省力的方式是将现有的纤维扩展到（f 的逆像定义的）片。这是拉回的通用构造的本质。

特别地，如果 A 是一个单元素集合（终对象），那么我们只有一个纤维（整个 E），从 $f^< E$ 是一个笛卡尔积 $B \quad E$。这样的丛称为平凡丛。

非平凡丛不是积，但可以局部分解为积。正如 B 是片的和，$f^< E$ 是这些片与 E 的相应纤维的积的和。

您还可以将 A 视为提供了一个枚举基 B 中所有片的地图集。想象 A 是一组国家，而 B 是一组城市。映射 f 为每个城市指定一个国家。

继续这个例子，让 E 是由国家纤维化的语言集。如果我们假设在每个城市中都讲该国的语言，那么基变换函子将国家的语言重新植入其每个城市。

顺便提一下，这种使用局部片和地图集的思想可以追溯到微分几何和广义相对论，我们经常将局部坐标系粘合在一起，以描述拓扑上非平凡的丛，如莫比乌斯带或克莱因瓶。

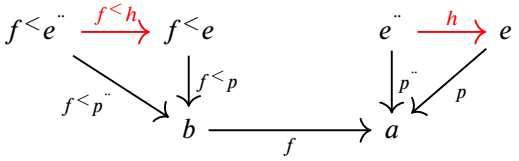
正如我们很快将看到的那样，在局部笛卡尔闭范畴中，基变换函子同时具有左伴随和右伴随。$f^<$ 的左伴随称为 $f_!$（有时发音为“f 下惊叹号”），右伴随称为 $f_<$（“f 下星号”）：

$$f_! \dashv f^< \dashv f_<$$

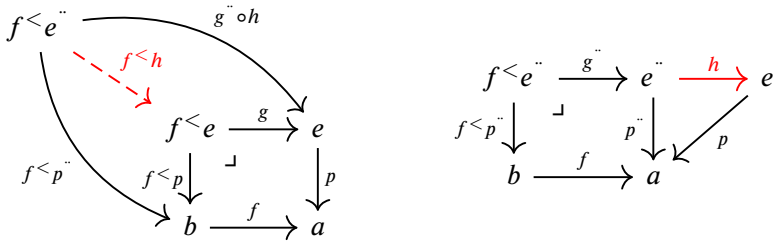
在编程中，左伴随称为依赖和，右伴随称为依赖积或依赖函数：

$$\Sigma_f \dashv f^< \dashv \Pi_f$$

Exercise 11.2.4. 定义基变换函子对 $\mathcal{C}/a$ 中态射的作用，即给定一个态射 h，构造其对映 $f^< h$。



提示：使用拉回的通用性和通方条件：$g'' \circ h \circ p = f^< p'' \circ f$。



11.3 依赖和 (Dependent Sum)

在类型理论中，依赖和，或 sigma 类型 $\Sigma_{x:B}T(x)$，定义为一对元素的类型，其中第二个元素的类型依赖于第一个元素的值。

从概念上讲，和类型通过其映射出性质（mapping-out property）定义。和的映射出是一个对映，如下的伴随关系所示：

$$C(F_1 + F_2, F) \ \ddot{o} \ (C \quad C)(\langle F_1, F_2\rangle, \Delta F)$$

在这里，我们有一对箭头 $(F_1 \to F, F_2 \to F)$，它们定义了和 $S = F_1 + F_2$ 的映射出。在 Set 中，和是带标签的并集。依赖和是由另一个集合的元素标记的和。

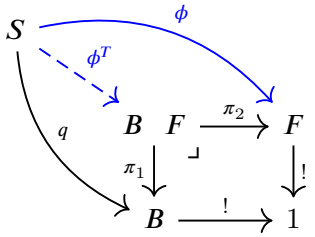
我们的计数向量类型可以被认为是由自然数标记的依赖和。这种类型的元素是一个自然数 `n`（一个值）与一个 n 元组类型 `(a, a, ... a)` 的元素配对。以下是一些用这种表示法编写的整数计数向量：

```
(0, ())
(1, 42)
(2, (64, 7))
(5, (8, 21, 14, -1, 0))
```

更一般地说，依赖和的引入规则假设存在一个由基类型 B 的元素索引的类型族 $T(x)$。然后 $\Sigma_{x:B}T(x)$ 的元素由一对元素 $x\colon B$ 和 $y\colon T(x)$ 构成。

从范畴论的角度看，依赖和被建模为基变换函子的左伴随。

为了理解这一点，我们首先回顾一下积的定义，它是积的元素。我们之前已经注意到，积可以表示为从单元素集合（终对象）拉回的积。以下是积/拉回的通用构造（符号表示了该构造的目标）：



我们还看到积可以通过伴随关系来定义。我们可以在图中发现这种伴随关系：对于每一对箭头 $\langle \phi, q\rangle$，都有一个唯一的箭头 ϕ^T 使得三角形是可交换的。

请注意，如果我们固定 q，我们会在箭头 ϕ 和 ϕ^T 之间得到一一对应的关系。这就是我们感兴趣的伴随关系。

现在，我们可以戴上纤维化的眼镜，注意到 $\langle S, q\rangle$ 和 $\langle B \quad F, \pi_1\rangle$ 是在相同基 B 上的两个纤维化。交换三角形使 ϕ^T 成为商范畴 $\mathcal{C}/B$ 中的一个态射，或者说是纤维间的映射。换句话说，ϕ^T 是同态集的一个成员：

$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} B & F \\ \pi_1 & \end{matrix} \right\rangle\right)$$

由于 ϕ 是同态集 $\mathcal{C}(S, F)$ 的一个成员，我们可以将 ϕ^T 和 ϕ 之间的一一对应关系重写为同态集的同构：

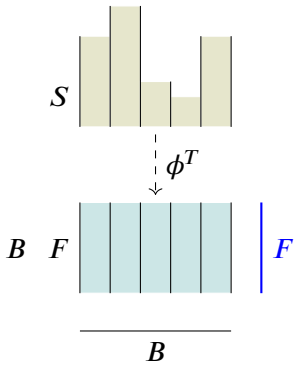
$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} B & F \\ \pi_1 & \end{matrix} \right\rangle\right) \ddot{\circ}\ \mathcal{C}(S, F)$$

实际上，这是一个伴随关系，其中我们有一个遗忘函子 $U : \mathcal{C}/B \to \mathcal{C}$，它将 $\langle S, q\rangle$ 映射到 S，从而遗忘了纤维化。

如果你仔细看这个伴随关系，你可以看到将 S 定义为范畴和（即余积）的轮廓。

首先，在右边你有一个 S 的映射出。将 S 视为由纤维化 $\langle S, q\rangle$ 定义的纤维的和。

其次，回想一下纤维化 $\langle B \quad F, \pi_1\rangle$ 可以被视为在 B 的点上植入许多 F 的副本。这是对对角函子 Δ 的推广，它复制 F，在这里，我们制作了“B 副本”的 F。伴随关系的左侧只是一些箭头，每个箭头将 S 的不同纤维映射到目标纤维 F。

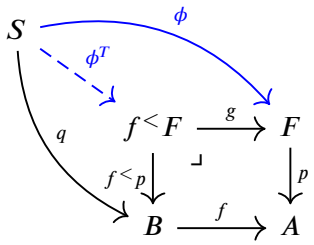


将这一思想应用到我们的计数向量示例中，ϕ^T 代表无限多个函数，每个自然数一个。在实践中，我们使用递归来定义这些函数。例如，这里有一个整数向量的映射：

```
sumV :: Vec n Int -> Int
sumV VNil = 0
sumV (VCons n v) = n + sumV v
```

添加图集 (Adding the atlas)

我们可以通过用任意基 A（图集）替换终对象来推广我们的图表。现在，我们有了一个纤维化 $\langle F, p\rangle$，并且我们使用定义基变换函子 $f^<$ 的拉回方形：



我们可以想象在 B 上的纤维化更细粒度，因为 f 可能将多个点映射到一个点。例如，考虑一个函数 `even :: Nat -> Bool`，它创建了偶数和奇数的两个集合。在此图中，f 定义了对原始 S 的粗略“重采样”。

拉回的通用性导致了以下同态集的同构：

$$(C/B)\left(\begin{pmatrix} S \\ q \end{pmatrix}, f^< \begin{pmatrix} F \\ p \end{pmatrix}\right) \ddot{\circ} (C/A)\left(\begin{pmatrix} S \\ f \circ q \end{pmatrix}, \begin{pmatrix} F \\ p \end{pmatrix}\right)$$

在这里，ϕ^T 是左侧的一个元素，而 ϕ 是对应的右侧元素。

我们将这个同构解释为基变换函子 $f^<$ 和依赖和函子之间的伴随关系。

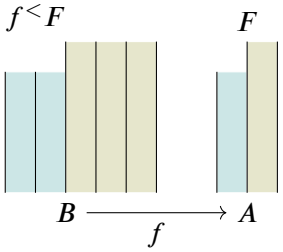
$$(C/B)\left(\begin{pmatrix} S \\ q \end{pmatrix}, f^< \begin{pmatrix} F \\ p \end{pmatrix}\right) \ddot{\circ} (C/A)\left(\Sigma_f \begin{pmatrix} S \\ q \end{pmatrix}, \begin{pmatrix} F \\ p \end{pmatrix}\right)$$

依赖和因此由以下公式给出：

$$\Sigma_f \begin{pmatrix} S \\ q \end{pmatrix} = \begin{pmatrix} S \\ f \circ q \end{pmatrix}$$

这意味着，如果 S 是通过 q 在 B 上纤维化的，并且存在从 B 到 A 的映射 f，那么 S 就会自动（更粗略地）在 A 上纤维化，投影为组合 $f \circ q$。

我们之前已经看到，在 Set 中，f 定义了 B 内的补丁。F 的纤维被重新植入这些补丁中，形成 $f^<F$。局部而言，即在每个补丁内，$f^<F$ 看起来像是一个笛卡尔积。



S 本身以两种方式纤维化：使用 $f \circ q$ 在 A 上粗略地分割，并使用 q 在 B 上精细地切片。

在范畴论中，依赖和是基变换函子 $f^<$ 的左伴随，记作 $f_!$。对于给定的 $f : b \to a$，这是一个函子：

$$f_! : C/b \to C/a$$

它对对象 $(s, q\colon s \to b)$ 的作用由后组合 f 给出：

$$f_!(s, q) = (s, f \circ q)$$

存在量化 (Existential quantification)

在“命题即类型”的解释中，类型族对应于命题族。依赖和类型 $\Sigma_{x:B} T(x)$ 对应于命题：存在一个 x 使得 $T(x)$ 为真：

$$\Sigma_{x:B} T(x)$$

确实，类型 $\Sigma_{x:B} T(x)$ 的一个术语是一个元素对，一个是 $x\colon B$，另一个是 $y\colon T(x)$——这表明 $T(x)$ 对于某个 x 是可居住的。

11.4 依赖积 (Dependent Product)

在类型理论中，依赖积，或依赖函数，或 Π-类型 $\Pi_{x:B} T(x)$，定义为返回类型取决于其参数的值的函数。

它被称为函数，因为你可以对其进行求值。给定一个依赖函数 $f\colon \Pi_{x:B} T(x)$，你可以将其应用于参数 $x\colon B$ 以获得 $f(x)\colon T(x)$ 的值。

Haskell 中的依赖积

一个简单的依赖积示例是一个构造具有给定大小的向量并用给定值填充它的函数：

```
replicateV :: a -> SNat n -> Vec n a
replicateV _ SZ  = VNil
replicateV x (SS n) = VCons x (replicateV x n)
```

在撰写本文时，Haskell 对依赖类型的支持有限，因此依赖函数的实现需要使用单例类型。在这种情况下，作为 `replicateV` 参数的数字作为单例自然数传递：

```
data SNat n where
SZ :: SNat Z
SS :: SNat n -> SNat (S n)
```

（请注意，`replicateV` 是一个双参函数，因此它可以被视为一个对的依赖函数，或者一个返回依赖函数的常规函数。）

集合的依赖积

在描述依赖函数的范畴模型之前，研究它们在集合上的工作方式是很有帮助的。依赖函数从每个集合 $T(x)$ 中选择一个元素。

你可以将这个选择的整体想象为一个巨大的元组——笛卡尔积的一个元素。例如，在 B 是一个两元素集合 ‵1, 2‘ 的简单情况下，依赖函数类型只是一个笛卡尔积 $T(1)$ $T(2)$。通常，你会得到一个以 B 的元素为索引的元组组件。这就是积符号 $\Pi_{x:B}T(x)$ 的意义。

在我们的例子中，`replicateV` 为 `n` 的每个值选择一个特定的计数向量。计数向量等同于元组，因此，对于 `n` 等于零，`replicateV` 返回一个空元组 `()`; 对于 `n = 1`，它返回一个单值 `x`; 对于 `n` 等于二，它复制 `x` 返回 `(x, x)`；以此类推。

函数 `replicateV` 在某个 `x :: a` 上求值时，等效于一个由元组组成的无限元组：

$$((), x, (x, x), (x, x, x), ...)$$

这是类型的一个特定元素：

$$((), a, (a, a), (a, a, a), ...)$$

范畴上的依赖积 (Dependent product categorically)

为了构建依赖函数的范畴模型，我们需要从类型族的角度转换为纤维化的角度。我们从一个纤维化 E/B 开始，该纤维化由投影 $p\colon E \to B$ 进行。依赖函数被称为这个纤维化的*截面*。

如果你将这个纤维化视为从基 B 伸出的许多纤维，截面就像是一个理发：它切断了每个纤维以生成相应的值。在物理学中，这样的截面被称为场——以时空为基。

正如我们谈论函数对象代表函数集一样，我们可以谈论一个对象 $S(E)$，它表示给定纤维化的截面集。

正如我们将函数应用定义为从积中映射出来：

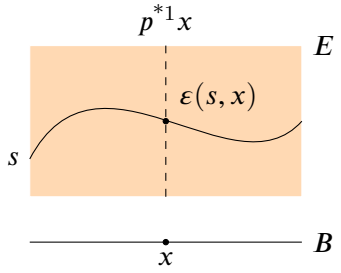
$$\varepsilon_{BC}\colon C^B \quad B \to C$$

我们可以定义依赖函数的应用为一个映射：

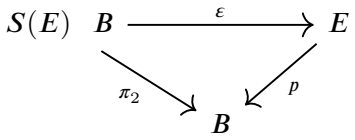
$$\varepsilon\colon S(E) \quad B \to E$$

我们可以将其形象化为从 $S(E)$ 中选择一个截面 s 和基 B 的一个元素 x，并在纤维化 E 中生成一个值。（在物理学中，这对应于在特定时空点测量一个场。）

但这次我们必须坚持这个值在正确的纤维中。如果我们投影 ε 在 (s, x) 上的结果，它应该回到 x。

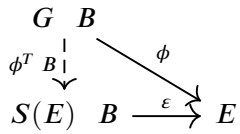


换句话说，该图必须交换：



这使得 ε 成为商范畴 $\mathcal{C}/B$ 中的一个态射。

正如我们定义了函数对象的普遍性一样，截面对象也是普遍的。普遍性条件具有相同的形式：对于任何其他具有箭头 $\phi\colon G \times B \to E$ 的对象 G，存在一个唯一的箭头 $\phi^T\colon G \to S(E)$ 使得以下图表交换：



不同之处在于 ε 和 ϕ 现在都是商范畴 $\mathcal{C}/B$ 中的态射。

ϕ 和 ϕ^T 之间的一一对应关系形成了伴随关系：

$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} G & B \\ & \pi_2 \end{matrix} \right\rangle, \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right) \cong \mathcal{C}\left(G, S(E)\right)$$

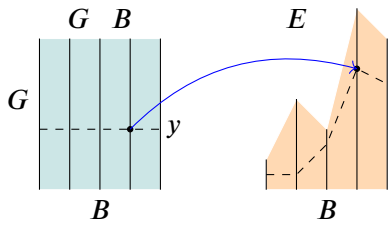
我们可以用它作为截面对象 $S(E)$ 的定义。这个伴随关系的余单元是依赖函数的应用。我们通过用 $S(E)$ 替换 G 并在右侧选择恒等态射来获得它。余单元因此成为同态集的一个成员：

$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} S(E) & B \\ & \pi_2 \end{matrix} \right\rangle, \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right)$$

将上面的伴随关系与定义函数对象 E^B 的柯里化伴随关系进行比较：

$$\mathcal{C}(G \times B, E) \cong \mathcal{C}(G, E^B)$$

现在回想一下，在 Set 中，我们将积 $G \times B$ 解释为在 B 的每个元素上植入 G 的副本。因此，我们的伴随关系的左侧的单个元素是一个函数族，每个纤维一个函数。给定 G 中的任意一个 y，它在 $G \times B$ 中切出一个水平切片。这些是所有 $b \in B$ 的对 (y, b)。我们的函数族将这个切片映射到 E 的对应纤维中，从而创建一个 E 的截面。



伴随关系告诉我们，这些映射族唯一地确定了从 G 到 $S(E)$ 的函数。因此，$S(E)$ 的元素与 E 的截面一一对应。

这些都是集合论的直觉。我们可以通过首先注意到伴随关系的右侧可以很容易地表达为终对象上商范畴 $\mathcal{C}/1$ 中的同态集来推广它们。

确实，$\mathcal{C}$ 中的对象 X 和 $\mathcal{C}/1$ 中的对象 $\langle X, !\rangle$ 之间存在一一对应关系（这里 $!$ 是指向终对象的唯一箭头）。$\mathcal{C}/1$ 中的箭头是 $\mathcal{C}$ 中的箭头，没有额外的约束。因此，我们有：

$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} G & B \\ & \pi_2 \end{matrix} \right\rangle, \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right) \ddot{\circ} (\mathcal{C}/1)\left(\left\langle \begin{matrix} G \\ ! \end{matrix} \right\rangle, \left\langle \begin{matrix} S(E) \\ ! \end{matrix} \right\rangle\right)$$

添加图集 (Adding the atlas)

下一步是通过用更一般的基 A 代替终对象 1 来“模糊焦点”，A 作为图集。

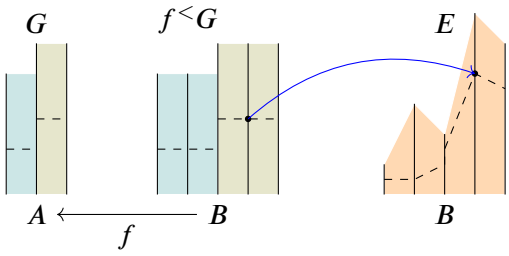
伴随关系的右侧变为商范畴 $\mathcal{C}/A$ 中的同态集。G 本身通过某种 $q\colon G \to A$ 进行粗略纤维化。

请记住，$G \quad B$ 可以理解为沿映射 $!\colon B \to 1$ 的拉回，或从 1 到 B 的基变换。如果我们想用 A 替换 1，我们应该用更一般的 q 的拉回代替积 $G \quad B$。这种基变换由一个新的态射 $f\colon B \to A$ 参数化。

$$\begin{array}{ccc} G \quad B & \xrightarrow{\pi_1} & G \\ \downarrow{\scriptstyle \pi_2} & \lrcorner & \downarrow{\scriptstyle !} \\ B & \xrightarrow{!} & 1 \end{array} \longrightarrow \begin{array}{ccc} f^{<}G & \xrightarrow{g} & G \\ \downarrow{\scriptstyle f^{<}q} & \lrcorner & \downarrow{\scriptstyle q} \\ B & \xrightarrow{f} & A \end{array}$$

结果是，我们得到的不再是 B 上的 G 纤维，而是一个通过 f^{*1} 重新植入 A 中的 G 纤维的拉回 $f^{<}G$。这样 A 就成了一个图集，它枚举了所有由均匀纤维填充的补丁。

例如，假设 A 是一个两元素集合。纤维化 q 将 G 分为两个纤维。它们将作为我们的通用纤维。这些纤维现在根据 f^{*1} 的引导重新植入 B 中的两个补丁中，形成 $f^{<}G$。

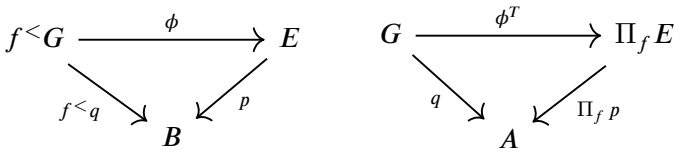


定义依赖函数类型的伴随关系为：

$$(\mathcal{C}/B)\left(f^{<}\left\langle\begin{matrix}G\\q\end{matrix}\right\rangle,\left\langle\begin{matrix}E\\p\end{matrix}\right\rangle\right) \ddot{\circ}\ (\mathcal{C}/A)\left(\left\langle\begin{matrix}G\\q\end{matrix}\right\rangle,\Pi_f\left\langle\begin{matrix}E\\p\end{matrix}\right\rangle\right)$$

这是我们用来定义截面对象 $S(E)$ 的伴随关系的推广。这个定义了一个新对象 $\Pi_f E$，它是截面对象的重新排列。

伴随关系是在它们各自的商范畴中的映射之间的映射：

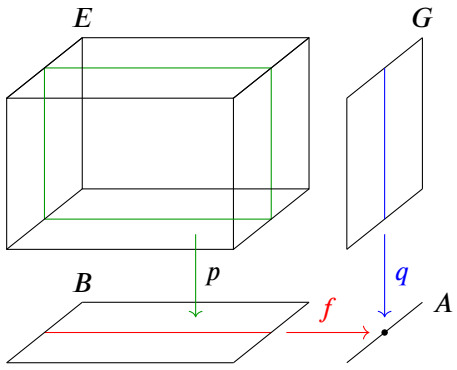


为了更好地理解这个伴随关系，让我们考虑它在集合上是如何工作的。

- 右侧在以图集 A 为基的粗粒纤维化中操作。它是一个函数族，每个补丁一个函数。对于每个补丁，我们从 G 的“厚纤维”（下图中为蓝色）到 $\Pi_f E$ 的“厚纤维”（未显示）得到一个函数。

- 左侧在以 B 为基的更精细的纤维化中操作。这些纤维分组成小束并覆盖补丁。一旦我们选择一个补丁（下图中为红色），我们就从该补丁到 E 中相应补丁（绿色表示）得到一个函数族——E 中小束的截面。因此，逐个补丁，我们得到 E 的小截面。

伴随关系告诉我们，$\Pi_f E$ 的“厚纤维”的元素与相同补丁上的 E 的小截面相对应。



在范畴论中，依赖积是基变换函子 $f^{<}$ 的右伴随，记作 $f_{<}$。对于给定的 $f : b \to a$，它是一个函子：

$$f_{<} : \mathcal{C}/b \to \mathcal{C}/a$$

以下练习揭示了 f 所起的作用。它可以被视为通过限制它们在由 f^{*1} 定义的“邻域”上的截面来定位 E 的截面。

Exercise 11.4.1. 考虑当 A 是一个两元素集合 $\hat{}0,1\text{‘}$ 并且 f 将整个 B 映射到一个元素（比如 1）时会发生什么。你将如何定义伴随关系右侧的函数？它应该对 0 上的纤维做些什么？

Exercise 11.4.2. 我们选择 G 为单元素集合 1，并让 $x\colon 1 \to A$ 作为一个选择 A 中元素的纤维化。使用伴随关系，证明以下结论：

- $f^{<1}$ 有两种类型的纤维：$f^{*1}(x)$ 的元素上的单元集和空集。
- 一个映射 $\phi\colon f^{<1} \to E$ 等同于从 E 的每个 $f^{*1}(x)$ 的纤维中选择一个元素。换句话说，它是 E 在 B 的子集 $f^{*1}(x)$ 上的部分截面。
- $\Pi_f E$ 的一个纤维是这样的部分截面。
- 当 A 也是一个单元素集合时会发生什么？

全称量化 (Universal quantification)

依赖积 $\Pi_{x:B} T(x)$ 的逻辑解释是一个全称命题。类型 $\Pi_{x:B} T(x)$ 的一个元素是一个截面——它证明了可以从 $T(x)$ 的每个成员中选择一个元素。这意味着它们都不是空的。换句话说，它是命题的证明：

$$\text{Å}_{x:B} T(x)$$

11.5 Equality 等式

我们的数学启蒙经验通常涉及到“等式 (Equality)”。我们学到：

$$1 + 1 = 2$$

然后我们就不再多想了。

但 $1 + 1$ 等于 2 到底是什么意思呢？

$$2$$

是一个数字，而

$$1 + 1$$

是一个表达式，所以它们不是同一样东西。在我们将这两者视为相等之前，我们需要进行一些心理处理。

与此相比，考虑 $0 = 0$ 的陈述，其中等号两边确实是“同一件事 (the same thing)”。

显然，如果我们要定义“等式 (Equality)”，至少需要确保一切都等于它自身。我们称这个性质为“自反性 (Reflexivity)”。

回顾我们对“自然数 (Natural Numbers)”的定义：

```
data Nat where
Z :: Nat
S :: Nat -> Nat
```

这是我们如何为“自然数 (Natural Numbers)”定义等式的：

```
equal :: Nat -> Nat -> Bool
equal Z Z = True
equal (S m) (S n) = equal m n
equal _ _ = False
```

我们递归地剥去每个数字中的 S，直到其中一个达到 Z。如果另一个也同时达到 Z，我们认为我们最初的两个数字是相等的，否则它们就不相等。

Equational reasoning 等式推理

注意，当我们在 Haskell 中定义“等式 (Equality)”时，我们已经在使用等号了。例如，在这里：

```
equal Z Z = True
```

这个等号告诉我们，无论何时我们看到表达式 $equal\ Z\ Z$，我们都可以将其替换为 $True$，反之亦然。

这就是用等式代替等式的原则，它是 Haskell 中“等式推理 (Equational Reasoning)”的基础。在 Haskell 中，我们无法直接编码等式的证明，但我们可以使用等式推理来推理 Haskell 程序。这是纯函数式编程的主要优点之一。在命令式语言中，由于副作用的存在，你无法执行这样的替换。

如果我们想证明 $1+1$ 等于 2，我们首先必须定义“加法 (Addition)”。这个定义可以针对第一个或第二个参数进行递归。以下定义在第二个参数上递归：

```
add :: Nat -> Nat -> Nat
add n Z = n
add n (S m) = S (add n m)
```

我们将 $1+1$ 编码为：

```
add (S Z) (S Z)
```

现在，我们可以使用 add 的定义来简化这个表达式。我们尝试匹配第一个子句，但失败了，因为 $S\ Z$ 与 Z 不同。然而，第二个子句匹配。在此子句中，n 是任意数字，因此我们可以用 $S\ Z$ 替换它，得到：

```
add (S Z) (S Z) = S (add (S Z) Z)
```

在这个表达式中，我们可以使用 *add* 定义的第一个子句（再次用 *S Z* 替换 *n*）进行另一个等式替换：

```
add (S Z) Z = (S Z)
```

我们得出：

```
add (S Z) (S Z) = S (S Z)
```

我们可以清楚地看到右边是 2 的编码。但我们还没有证明我们的“等式 (Equality)”定义是自反的，所以原则上我们不知道

```
equal (S (S Z)) (S (S Z))
```

是否产生 *True*。我们必须再次逐步使用等式推理：

```
equal (S (S Z)) (S (S Z)) =
 {- equal 定义的第二个子句 -}
equal (S Z) (S Z) =
 {- equal 定义的第二个子句 -}
equal Z Z =
 {- equal 定义的第一个子句 -}
True
```

我们可以使用这种推理方法来证明具体数字的陈述，但在推理泛化数字时会遇到问题——例如，证明某些东西对所有 *n* 都成立。使用我们的加法定义，我们可以轻松地证明 *add n Z* 与 *n* 相同。但我们无法证明 *add Z n* 与 *n* 相同。后者的证明需要使用“归纳法 (Induction)”。

我们最终区分了两种等式。一种是使用替换或重写规则证明的，称为“定义等式 (Definitional Equality)”。你可以将其视为宏展开或编程语言中的内联展开。它还涉及“β-归约 (β-Reduction)”：通过用实际参数替换形式参数来执行函数应用，如下所示：

```
(\x -> x + x) 2 =
 {- beta 归约 -}
2 + 2
```

第二种更有趣的等式称为“命题等式 (Propositional Equality)”，它可能需要实际的证明。

Equality vs isomorphism 等式与同构

我们说过，范畴论学者更倾向于“同构 (Isomorphism)”而非“等式 (Equality)”——至少在涉及对象时是这样。在一个范畴的范围内，确实没有办法区分同构对象。然而，总的来说，等式比同构更强。这是一个问题，因为能够用等式代替等式非常方便，但用同构代替同构并不总是显而易见。

数学家们一直在努力解决这个问题，主要是尝试修改“同构 (Isomorphism)”的定义——但真正的突破是在他们同时弱化“等式 (Equality)”的定义时出现的。这导致了“同伦类型论 (Homotopy Type Theory, HoTT)”的发展。

粗略地说，在类型论中，特别是在“Martin-Löf 依赖类型理论 (Martin-Löf Theory of Dependent Types)”中，等式被表示为一种类型，为了证明等式，我们必须构造该类型的一个元素——这是“Curry-Howard 解释 (Curry-Howard Interpretation)”的精神。

此外，在 HoTT 中，证明本身可以比较等式，依此类推，可以无限进行。你可以通过将等式的证明视为一些可以彼此变形的抽象路径来描述这一点；因此使用了“同伦 (Homotopies)”的语言。

在这种背景下，取代“同构 (Isomorphism)”的概念是“等价 (Equivalence)”，其中这些等式被视为类型。

HoTT 的主要思想是可以引入“同一性公理 (Univalence Axiom)”，该公理大致表明等式与等价是等价的，符号上表示为：

$$(A = B) \ddot{o} (A \ddot{o} B)$$

请注意，这只是一个公理，而不是一个定理。我们可以接受它，也可以不接受它，理论仍然是有效的（至少我们是这样认为的）。

Equality types 等式类型

假设你想比较两个术语是否相等。首先的要求是这两个术语必须是相同类型的。你不能比较苹果和橘子。不要被某些允许比较不同术语的编程语言所迷惑：在每种情况下，都涉及到隐式转换，最终的等式始终是在相同类型的值之间进行比较。

对于每对值，原则上都有一个单独的等式证明类型。对于 $0 = 0$，有一个类型；对于 $1 = 1$，有一个类型；而对于 $1 = 0$，也有一个类型，后者希望是无人居住的。

“等式类型 (Equality Type)”，又称“恒等类型 (Identity Type)”，因此是一个依赖类型：它取决于我们正在比较的两个值。它通常写作 Id_A，其中 A 是这两个值的类型，或者使用中缀符号表示为 $x =_A y$（下标为 A 的等号）。

例如，两个零的等式类型写作 $\mathrm{Id}_{\mathbb{N}}(0,0)$ 或：

$$0 =_{\mathbb{N}} 0$$

请注意：这不是一个陈述或术语。它是一个“类型 (Type)”，就像 *Int* 或 *Bool* 一样。如果你有它的引入规则，你可以定义这个类型的一个值。

Introduction rule 引入规则

“等式类型 (Equality Type)”的引入规则是依赖函数：

$$\mathrm{refl}_A : \Pi_{x:A} \mathrm{Id}_A(x,x)$$

这可以在“命题即类型 (Propositions as Types)”的精神下解释为以下陈述的证明：

$$\text{Å}_{x:A}\ x = x$$

这就是我们熟悉的“自反性 (Reflexivity)”：它表明，对于所有类型 A 的 x，x 等于它自己。你可以将此函数应用于类型 A 的某个具体值 x，它将生成类型 $\mathrm{Id}_A(x,x)$ 的新值。

现在我们可以证明 $0 = 0$。我们可以执行 $\mathrm{refl}_{\mathbb{N}}(0)$ 以获得类型 $0 =_{\mathbb{N}} 0$ 的值。这个值证明了该类型是可居住的，因此对应于一个真实的命题。

这是等式的唯一引入规则，因此你可能会认为所有的等式证明都归结为“它们相等是因为它们是同一个东西”。令人惊讶的是，事实并非如此。

β-reduction and η-conversion β-归约与 η-转换

在类型论中，我们有引入规则和消除规则的相互作用，这本质上使它们成为彼此的逆。

考虑“积 (Product)”的定义。我们通过提供两个值 $x\colon A$ 和 $y\colon B$ 来引入它，并得到一个值 $p\colon A \quad B$。然后我们可以通过使用两个投影来消除它。但是我们怎么知道这些值是否与我们用来构造它的值相同呢？这就是我们必须假设的。我们称其为“计算规则 (Computation Rule)”或“β-归约规则 (β-Reduction Rule)”。

相反，如果我们给定一个值 $p\colon A \quad B$，我们可以使用投影提取这两个组件，然后使用引入规则重新组合它。但是我们怎么知道我们会得到相同的 p 呢？这也是我们必须假设的。这有时被称为“唯一性条件 (Uniqueness Condition)”或“η-转换规则 (η-Conversion Rule)”。

在类型论的范畴模型中，这两个规则都来自“泛构造 (Universal Construction)”。

“等式类型 (Equality Type)”也有一个消除规则，我们将在稍后讨论，但我们不施加唯一性条件。这意味着可能存在一些不是通过 refl 获得的等式证明。

这正是“同伦类型论 (HoTT)”对数学家有趣的原因所在。

Induction principle for natural numbers 自然数的归纳原理

在为“等式 (Equality)”类型制定消除规则之前，首先讨论一个更简单的“自然数的消除规则 (Elimination Rule for Natural Numbers)”是有益的。我们已经看到这种规则描述了“原始递归 (Primitive Recursion)”，它允许我们通过指定一个值 init 和一个函数 step 来定义递归函数。

使用“依赖类型 (Dependent Types)”，我们可以将此规则推广为“依赖消除规则 (Dependent Elimination Rule)”，这相当于“数学归纳法 (Mathematical Induction)”的原则。

“归纳原理 (Induction Principle)”可以描述为一种工具，用于一举证明由自然数索引的整族命题。例如，$add\ Z\ n$ 等于 n 的陈述实际上是无限多个命题，每个 n 值一个。

原则上，我们可以编写一个程序，仔细验证这一陈述的许多情况，但我们永远无法确定它是否普遍成立。有些关于自然数的猜想已经通过这种方式使用计算机进行测试，但显然它们永远无法穷尽一个无限集合的情况。

粗略地说，我们可以将所有数学定理分为两类：一种是容易表述的，另一种是表述复杂的。它们还可以进一步细分为容易证明的和难以或无法证明的。例如，著名的“费马大定理 (Fermat's Last Theorem)”的表述非常简单，但其证明需要一些非常复杂的数学工具。

在这里，我们对自然数的定理感兴趣，这些定理既易于表述也易于证明。我们将假设我们知道如何生成一族命题，或者等效地，生成一个依赖类型 $\boldsymbol{T}(n)$，其中 n 是一个自然数。

我们还假设我们有一个值：

$$\mathrm{init} : \boldsymbol{T}(\boldsymbol{Z})$$

或者等效地，零命题的证明；以及一个依赖函数：

$$\mathrm{step} : \Pi_{n:\mathbb{N}}\ (\boldsymbol{T}(n) \to \boldsymbol{T}(\boldsymbol{S}\, n))$$

这个函数被解释为从 n 命题的证明生成 $(n+1)$ 命题的证明。

自然数的依赖消除规则 (Dependent Elimination Rule) 规定，给定这样的 init 和 step，存在一个依赖函数：

$$f : \Pi_{n:\mathbb{N}}\, \boldsymbol{T}(n)$$

这个函数被解释为提供 $\boldsymbol{T}(n)$ 对所有 n 都为真的证明。

此外，当这个函数应用于零时，它再现了 init：

$$f(\boldsymbol{Z}) = \mathrm{init}$$

并且，当应用于 n 的继任者时，它与采取 step 一致：

$$f(\boldsymbol{S}\, n) = (\mathrm{step}(n))(f(n))$$

（这里，$\mathrm{step}(n)$ 生成一个函数，然后将其应用于 $f(n)$ 的值。）这些是自然数的两个“计算规则 (Computation Rules)”。

请注意，归纳原理不是关于自然数的定理。它是自然数类型的一部分定义。

并非所有依赖于自然数的映射都可以分解为 init 和 step，就像并非所有关于自然数的定理都可以通过归纳法证明一样。自然数没有 η-转换规则。

Equality elimination rule 等式的消除规则

等式类型的消除规则在某种程度上类似于自然数的归纳原理。在那里我们使用 init 来为旅程的开始做基础，并使用 step 来取得进展。等式的消除规则需要更强大的基础，但它没有 step。没有很好的比喻来解释它的工作原理，除了依赖于一种信念。

我们的想法是，我们想构造一个从等式类型“映射出去 (Mapping Out of)”的函数。但由于等式类型本身是一个双参数类型族，因此映射出去的函数应该是一个依赖函数。这个函数的目标是另一个类型族：

$$T(x, y, p)$$

它依赖于正在比较的值对 $x, y : A$，以及等式证明 $p : \mathrm{Id}(x, y)$。

我们试图构造的函数是：

$$f : \Pi_{x,y:A} \Pi_{p:\mathrm{Id}(x,y)} T(x, y, p)$$

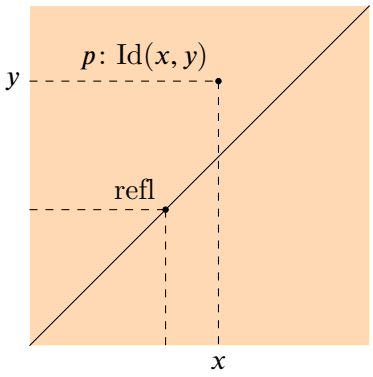
可以方便地将其视为生成一个证明，即对于所有点 x 和 y，并且对于每个证明它们相等的证明，命题 $T(x, y, p)$ 都为真。注意，潜在地，我们对于“每一个证明 (Every Proof)”它们相等都有不同的命题。

我们对 $T(x, y, p)$ 的最低要求是，当 x 和 y 字面相同时，并且等式证明是显然的 refl 时，它应该为真。这个要求可以表示为一个依赖函数：

$$t : \Pi_{x:A} T(x, x, \mathrm{refl}(x))$$

注意，我们甚至没有考虑 $x = x$ 的其他证明，除了那些由自反性给出的。此类证明是否存在？我们不知道，也不在意。

因此，这是我们的基础，应该引导我们为所有点对及所有等式证明定义 f。我们的直觉是，我们将 f 定义为平面 (x, y) 上的一个函数，第三维度由 p 给出。为此，我们给出了在对角线 (x, x) 上定义的东西，p 限制为 refl。



你可能认为我们需要更多的东西，比如某种 step 来将我们从一个点移动到另一个点。但是，与自然数不同，没有“下一个 (Next)”点或“下一个 (Next)”等式证明可以跳转到。我们手头只有函数 t，别无其他。

因此，我们假设，给定一个类型族 $T(x, y, p)$ 和一个函数：

$$t : \Pi_{x:A} T(x, x, \mathrm{refl}(x))$$

存在一个函数：

$$f : \Pi_{x,y:A}\Pi_{p:\mathrm{Id}(x,y)}\, T(x,y,p)$$

使得（计算规则）：

$$f(x,x,\mathrm{refl}(x)) = t(x)$$

注意，计算规则中的等式是“定义等式 (Definitional Equality)”而不是一个类型。

等式的消除规则告诉我们，总是可以将定义在对角线上的函数 t 扩展到整个三维空间。

这是一个非常强的假设。理解它的一种方法是，在类型论的框架内——类型论使用引入规则和消除规则的语言，以及操作这些规则的规则——“不可能 (Impossible)”定义一个不满足等式消除规则的类型族 $T(x,y,p)$。

我们到目前为止看到的最接近的类比是“参数化性 (Parametricity)”的结果，它表明在 Haskell 中，所有多态函数在端函子之间自动是自然变换。另一个例子，这次来自微积分，是任何定义在实数轴上的解析函数在复平面上都有唯一扩展。

依赖类型的使用模糊了编程与数学之间的界限。有一系列语言，从 Haskell 几乎没有涉足依赖类型但仍然坚定地应用于商业用途，到“定理证明器 (Theorem Provers)”，帮助数学家形式化数学证明。

Chapter 12

Algebras 代数

代数的本质是对表达式进行形式化操作。那么，什么是表达式，我们如何操作它们呢?

首先需要注意的是，像 $2(x+y)$ 或 ax^2+bx+c 这样的代数表达式是无穷多的。虽然有有限的规则来生成它们，但这些规则可以以无穷多种组合使用。这表明这些规则是*递归地*使用的。

在编程中，表达式几乎与（解析）树同义。考虑以下一个算术表达式的简单示例:

```
data Expr = Val Int
| Plus Expr Expr
```

这是构建树的一个配方。我们从使用 `Val` 构造函数的小树开始。然后我们将这些幼苗植入节点，依此类推。

```
e2 = Val 2
e3 = Val 3
e5 = Plus e2 e3
e7 = Plus e5 e2
```

这种递归定义在编程语言中运作良好。问题在于每个新的递归数据结构都需要其自己的操作库。

从类型论的角度来看，我们能够定义递归类型，如自然数或列表，这是通过提供特定的引入和消除规则来实现的。我们需要的是更通用的东西，即从更简单的可插入组件生成任意递归类型的过程。

在涉及递归数据结构时，有两个正交的关注点。一个是递归的机制，另一个是可插入的组件。

我们知道如何处理递归：我们假设我们知道如何构建小树。然后我们使用递归步骤将这些小树植入节点中，以构建更大的树。

范畴论告诉我们如何形式化这种不精确的描述。

12.1 Algebras from Endofunctors 从自函子构造代数

将小树植入节点的想法要求我们形式化地定义拥有“空位”的数据结构——一个“容器”。这正是函子的用途。因为我们希望递归地使用这些函子，它们必须是自函子。

例如，我们之前示例中的自函子可以通过以下数据结构定义，其中的 `x` 标记了空位：

```
data ExprF x = ValF Int
| PlusF x x
```

关于所有可能的表达式形状的信息被抽象为一个函子。

定义代数的另一个重要信息是表达式求值的配方。这同样可以使用相同的自函子进行编码。

递归地思考，假设我们知道如何求值一个更大表达式的所有子树。那么剩下的一步就是将这些结果插入顶层节点并对其进行求值。

例如，假设函子中的 `x` 被整数替代——即子树的求值结果。在最后一步该怎么做是很明显的。如果我们的树顶只是一个叶子 `ValF`（这意味着没有子树需要求值），我们只需返回存储在其中的整数。如果它是一个 `PlusF` 节点，我们将把其中的两个整数相加。这个配方可以编码为：

```
eval :: ExprF Int -> Int
eval (ValF n)    = n
eval (PlusF m n) = m + n
```

我们基于常识做了一些看似显而易见的假设。例如，由于节点被称为 `PlusF`，我们假设我们应该将两个数字相加。但乘法或减法同样适用。

由于叶子 `ValF` 包含一个整数，我们假设表达式应当求值为一个整数。但也有同样合理的求值器可以将表达式转换为字符串，这个求值器使用连接而不是相加：

```
pretty :: ExprF String -> String
pretty (ValF n)    = show n
pretty (PlusF s t) = s ++ " + " ++ t
```

事实上，有无限多的求值器，有些合理，有些则不那么合理，但我们不应该做出判断。任何目标类型的选择和任何求值器的选择都应该同样有效。这导致了以下定义：

一个自函子 F 的*代数 (Algebra)* 是一个二元组 (c, α)。对象 c 被称为代数的*载体 (Carrier)*，求值器 $\alpha\colon Fc \to c$ 被称为*结构映射 (Structure Map)*。

在 Haskell 中，给定函子 `f`，我们定义：

```
type Algebra f c = f c -> c
```

请注意，求值器不是多态函数。它是为特定类型 `c` 选择的特定函数。对于给定类型，可能有许多载体类型的选择，也可能有许多不同的求值器。它们都定义了独立的代数。

我们之前定义了两个 `ExprF` 的代数。这个代数的载体是 `Int`：

```
eval :: Algebra ExprF Int
eval (ValF n)    = n
eval (PlusF m n) = m + n
```

而这个代数的载体是 `String`：

```
pretty :: Algebra ExprF String
pretty (ValF n)    = show n
pretty (PlusF s t) = s ++ " + " ++ t
```

12.2 Category of Algebras 代数的范畴

给定自函子 F 的代数构成一个范畴。在该范畴中的箭头是代数态射 (Algebra Morphism)，这是其载体对象之间的结构保持箭头。

在这种情况下，保持结构意味着箭头必须与两个结构映射交换。这就是函子性 (Functoriality) 发挥作用的地方。要从一个结构映射切换到另一个，我们必须能够提升在其载体之间的箭头。

给定一个自函子 F，两个代数 (a, α) 和 (b, β) 之间的一个*代数态射 (Algebra Morphism)* 是一个箭头 $f : a \to b$，使得下图交换：

$$\begin{array}{ccc} Fa & \xrightarrow{Ff} & Fb \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ a & \xrightarrow{f} & b \end{array}$$

换句话说，必须满足以下等式：

$$f \circ \alpha = \beta \circ F f$$

两个代数态射的复合仍然是一个代数态射，这可以通过粘贴两个这样的图来验证（函子将复合映射为复合）。恒等箭头也是一个代数态射，因为

$$id_a \circ \alpha = \alpha \circ F(id_a)$$

（函子将恒等映射为恒等）。

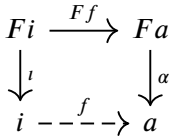
代数态射定义中的交换条件非常严格。例如，考虑一个将整数映射为字符串的函数。在 Haskell 中，有一个 `show` 函数（实际上是 `Show` 类的方法）可以做到这一点。它不是从 `eval` 到 `pretty` 的代数态射。

Exercise 12.2.1. 证明 *`show`* 不是一个代数态射。提示：考虑 *`PlusF`* 节点会发生什么。

Initial algebra 初始代数

给定某个函子的代数范畴中的初始对象称为*初始代数 (Initial Algebra)*，正如我们将看到的，它扮演着非常重要的角色。

根据定义，初始代数 (i, ι) 对于任何其他代数 (a, α) 都有一个唯一的代数态射 f。如图所示：



这个唯一的态射称为该代数 (a, α) 的*卡塔态射 (Catamorphism)*。有时它被写作“香蕉括号” $(\!|\alpha|\!)$。

Exercise 12.2.2. 让我们为以下函子定义两个代数：

```
data FloatF x = Num Float | Op x x
```

第一个代数：

```
addAlg :: Algebra FloatF Float
addAlg (Num x) = log x
addAlg (Op x y) = x + y
```

第二个代数：

```
mulAlg :: Algebra FloatF Float
mulAlg (Num x) = x
mulAlg (Op x y) = x * y
```

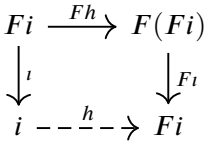
做出令人信服的论证，证明 *log*（对数函数）是这两个代数之间的一个代数态射。（*Float* 是一个内置的浮点数类型。）

12.3 Lambek's Lemma and Fixed Points Lambek 引理与不动点

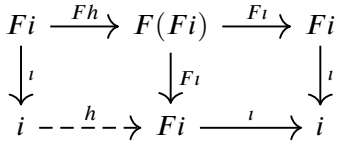
Lambek 引理指出，初始代数的结构映射 ι 是一个同构。

原因在于代数的自相似性。你可以使用 F 提升任何代数 (a, α)，得到的新代数 $(Fa, F\alpha)$ 仍然是一个代数，结构映射是 $F\alpha\colon F(Fa) \to Fa$。

特别地，如果你提升初始代数 (i, ι)，你将得到一个载体为 Fi 的新代数，结构映射为 $F\iota\colon F(Fi) \to Fi$。因此，必须存在一个唯一的代数态射从初始代数映射到它：



这个 h 是 ι 的逆元。为了看到这一点，我们考虑组合 $\iota \circ h$。它是以下图的底部箭头：



这是原始图与一个平凡交换的图的粘贴。因此整个矩形交换。我们可以将其解释为 $\iota \circ h$ 是 (i, ι) 到其自身的代数态射。但已经存在这样一个代数态射——即恒等态射。因此，根据从初始代数映射出的唯一性，这两个必须相等：

$$\iota \circ h = id_i$$

知道这一点后，我们现在可以回到前面的图，该图表示：

$$h \circ \iota = F\iota \circ Fh$$

由于 F 是一个函子，它将复合映射为复合，将恒等映射为恒等。因此右边等式等于：

$$F(\iota \circ h) = F(id_i) = id_{Fi}$$

我们已经证明了 h 是 ι 的逆元，这意味着 ι 是一个同构。换句话说：

$$Fi \ \ddot{o} \ i$$

我们将此等式解释为表示 i 是 F 的不动点（同构意义上的）。F 在 i 上的作用"没有改变它"。

可能有许多不动点，但这个是*最小不动点 (Least Fixed Point)*，因为有一个代数态射从它映射到任何其他不动点。一个自函子 F 的最小不动点表示为 μF，因此我们写作：

$$i = \mu F$$

Fixed point in Haskell Haskell 中的不动点

让我们考虑不动点的定义如何作用于我们最初的例子，该例子由自函子定义：

```
data ExprF x = ValF Int | PlusF x x
```

它的不动点是一个数据结构，其定义属性是 `ExprF` 作用于它会重现它。如果我们称这个不动点为 `Expr`，那么不动点方程变为（伪 Haskell 代码）：

```
Expr = ExprF Expr
```

展开 `ExprF` 我们得到：

```
Expr = ValF Int | PlusF Expr Expr
```

将其与递归定义（实际 Haskell 代码）比较：

```
data Expr = Val Int | Plus Expr Expr
```

我们得到了一个递归数据结构作为不动点方程的解。

在 Haskell 中，我们可以为任何函子（或仅仅是类型构造函数）定义一个不动点数据结构。正如我们稍后将看到的，这并不总是给我们初始代数的载体。它只适用于具有“叶子”组件的函子。

我们称 `Fix f` 为函子 `f` 的不动点。象征性地，不动点方程可以写作：

$$f\,(\mathrm{Fix}\,f) \ddot{\circ} \mathrm{Fix}\,f$$

或代码中，

```
data Fix f where
In :: f (Fix f) -> Fix f
```

数据构造函数 `In` 正是载体为 `Fix f` 的初始代数的结构映射。其逆映射为：

```
out :: Fix f -> f (Fix f)
out (In x) = x
```

Haskell 标准库中包含了一个更惯用的定义：

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

为了创建类型 `Fix f` 的项，我们经常使用“智能构造函数 (Smart Constructors)”。例如，对于函子 `ExprF`，我们会定义：

```
val :: Int -> Fix ExprF
val n = In (ValF n)

plus :: Fix ExprF -> Fix ExprF -> Fix ExprF
plus e1 e2 = In (PlusF e1 e2)
```

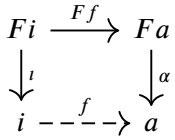
并使用它生成如下的表达式树：

```
e9 :: Fix ExprF
e9 = plus (plus (val 2) (val 3)) (val 4)
```

12.4 Catamorphisms 卡塔态射

作为程序员，我们的目标是能够在递归数据结构上执行计算——即“折叠 (Fold)”它。我们现在已经有了所有的组成部分。

数据结构定义为某个函子的一个不动点。这个函子的代数定义了我们想要执行的操作。我们已经看到不动点和代数结合在以下图中：



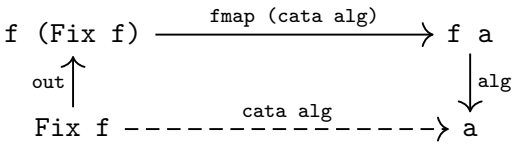
它定义了代数 (a, α) 的卡塔态射 f。

最后的一条信息是 Lambek 引理，它告诉我们 ι 可以反转，因为它是一个同构。这意味着我们可以将这个图解读为：

$$f = \alpha \circ F f \circ \iota^{*1}$$

并将其解释为 f 的递归定义。

让我们使用 Haskell 表示法重新绘制这个图。卡塔态射依赖于代数，所以对于载体为 `a` 和求值器为 `alg` 的代数，我们将有卡塔态射 `cata alg`。



简单地跟随箭头，我们得到这个递归定义：

```
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

这里发生的事情是：我们将这个定义应用于某个 `Fix f`。每个 `Fix f` 都是通过将 `In` 应用于一个函子类型的 `Fix f` 获得的：

```
data Fix f where
In :: f (Fix f) -> Fix f
```

函数 `out` “剥离”了数据构造函数 `In`：

```
out :: Fix f -> f (Fix f)
out (In x) = x
```

我们现在可以通过 `fmap` 对其应用 `cata alg` 来求值函子类型的 `Fix f`。这是一个递归应用。其思想是，函子内部的树比原始树小，因此递归最终会终止。它在命中叶子时终止。

在此步骤之后，我们得到一个函子类型的值，我们对其应用求值器 `alg`，以获得最终结果。

这种方法的强大之处在于所有递归都被封装在一个数据类型和一个库函数中：我们有 `Fix` 的定义和卡塔态射 `cata`。库的客户只提供非递归部分：函子和代数。这些要容易处理得多。我们已经将一个复杂问题分解为更简单的组成部分。

Examples 示例

我们可以立即将此构造应用于我们之前的示例。你可以检查：

```
cata eval e9
```

求值为 9，而

```
cata pretty e9
```

求值为字符串 "2 + 3 + 4"。

有时我们希望多行显示树并进行缩进。这需要将一个深度计数器传递给递归调用。有一个巧妙的技巧是使用函数类型作为载体：

```
pretty' :: Algebra ExprF (Int -> String)
pretty' (ValF n) i = indent i ++ show n
pretty' (PlusF f g) i = f (i + 1) ++ "\n" ++
indent i ++ "+" ++ "\n" ++
g (i + 1)
```

辅助函数 indent 复制空格字符：

```
indent n = replicate (n * 2) ' '
```

结果：

```
cata pretty' e9 0
```

当打印时，看起来像这样：

```
2
+
3
+
4
```

让我们尝试为其他熟悉的函子定义代数。自函子 Maybe 的不动点：

```
data Maybe x = Nothing | Just x
```

经过一些重命名后，相当于自然数类型：

```
data Nat = Z | S Nat
```

此函子的代数包括载体 a 的选择和一个求值器：

```
alg :: Maybe a -> a
```

Maybe 的映射由两件事决定：对应于 Nothing 的值和一个对应于 Just 的函数 a->a。在我们讨论自然数类型时，我们将其称为 init 和 step。现在我们可以看到，Nat 的消除规则就是此代数的卡塔态射。

Lists as initial algebras 列表作为初始代数

我们之前见到的列表类型等价于以下函子的一个不动点，该函子由列表内容的类型 `a` 参数化：

```
data ListF a x = NilF | ConsF a x
```

此函子的代数是一个映射：

```
alg :: ListF a c -> c
alg NifF = init
alg (ConsF a c) = step (a, c)
```

由值 `init` 和函数 `step` 确定：

```
init :: c
step :: (a, c) -> c
```

此代数的卡塔态射是列表递归器：

```
recList :: c -> ((a, c) -> c) -> (List a -> c)
```

其中 `(List a)` 可以与不动点 `Fix (ListF a)` 识别。

我们之前见过一个反转列表的递归函数。它通过将元素追加到列表末尾来实现，这非常低效。使用卡塔态射重写这个函数很容易，但问题仍然存在。

另一方面，预置元素是廉价的。更好的算法是遍历列表，在先进先出的队列中累积元素，然后一个一个地弹出它们并将它们预置到新列表中。

队列机制可以通过闭包组合实现：每个闭包都是一个记住其环境的函数。这里是载体为函数类型的代数：

```
revAlg :: Algebra (ListF a) ([a]->[a])
revAlg NilF = id
revAlg (ConsF a f) = \as -> f (a : as)
```

在每一步中，此代数创建一个新函数。这个函数在执行时会将先前的函数 `f` 应用于一个列表，该列表是当前元素 `a` 预置到函数的参数 `as` 中的结果。生成的闭包记住当前元素 `a` 和先前的函数 `f`。

此代数的卡塔态射累积一个这样的闭包队列。要反转一个列表，我们将此代数的卡塔态射结果应用于空列表：

```
reverse :: Fix (ListF a) -> [a]
reverse as = (cata revAlg as) []
```

这种技巧是 `foldl` 函数的核心。在使用它时应谨慎，因为存在堆栈溢出的危险。

列表如此常见，以至于它们的消除器（称为“折叠”）被包含在标准库中。但可能存在无数种递归数据结构，每种结构由其自身的函子生成，并且我们可以在所有这些结构

上使用相同的卡塔态射。

值得一提的是，列表构造在任何带有余积的单对象范畴中都有效。我们可以将列表函子替换为更通用的形式：

$$Fx = I + a \otimes x$$

其中 I 是单位对象，$\otimes$ 是张量积。固定点方程的解：

$$L_a \ddot{o} I + a \otimes L_a$$

可以形式化写作一个级数：

$$L_a = I + a + a \otimes a + a \otimes a \otimes a + ...$$

我们将此解释为列表的定义，其可以是空的 I、单个元素 a、两个元素的列表 $a \otimes a$ 等等。

顺便提一下，如果你足够认真地观察，这个解可以通过一系列形式变换获得：

$$\begin{aligned} & L_a \ddot{o} I + a \otimes L_a \\ L_a * a \otimes & L_a \ddot{o} I \\ (I * a) \otimes & L_a \ddot{o} I \\ & L_a \ddot{o} I/(I * a) \\ & L_a \ddot{o} I + a + a \otimes a + a \otimes a \otimes a + ... \end{aligned}$$

其中最后一步使用了几何级数和的公式。诚然，中间步骤没有意义，因为在对象上没有定义减法或除法，但最终结果是有意义的，正如我们稍后将看到的，这可以通过考虑一串对象的余极限 (Colimit) 来严格证明。

12.5 Initial Algebra from Universality 从普遍性构造初始代数

在 Set（集合范畴）中，另一种看待初始代数的方法是将其视为一组卡塔态射 (Catamorphisms) 的集合，整体上暗示了一个底层对象的存在。与其将 μF 视为一组树，不如将其视为从代数映射到其载体的一组函数。

在某种程度上，这只是 Yoneda 引理的另一种表现：每个数据结构都可以通过映射进来或映射出去来描述。在这种情况下，映射进来的是递归数据结构的构造函数，而映射出去的是所有可以应用于它的卡塔态射。

首先，让我们明确`cata`定义中的多态性：

```
cata :: Functor f => forall a. Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

然后交换参数的顺序。我们得到：

```
cata' :: Functor f => Fix f -> forall a. Algebra f a -> a
cata' (In x) = \alg -> alg (fmap (flip cata' alg) x)
```

函数flip反转函数的参数顺序：

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f b a = f a b
```

这为我们提供了一个从Fix f到一组多态函数的映射。

相反，给定一个类型为：

```
forall a. Algebra f a -> a
```

的多态函数，我们可以重构Fix f：

```
uncata :: Functor f => (forall a. Algebra f a -> a) -> Fix f
uncata alga = alga In
```

事实上，这两个函数cata'和uncata是彼此的逆元，建立了Fix f与多态函数类型之间的同构：

```
data Mu f = Mu (forall a. Algebra f a -> a)
```

我们现在可以将Mu f替换到任何使用Fix f的地方。

在Mu f上进行折叠 (Folding) 很容易，因为Mu本身携带了一组卡塔态射：

```
cataMu :: Algebra f a -> (Mu f -> a)
cataMu alg (Mu h) = h alg
```

你可能想知道如何为列表之类的结构构造Mu f类型的项。这可以通过递归完成：

```
fromList :: forall a. [a] -> Mu (ListF a)
fromList as = Mu h
where h :: forall x. Algebra (ListF a) x -> x
h alg = go as
where
go [] = alg NilF
go (n: ns) = alg (ConsF n (go ns))
```

要编译此代码，你需要使用语言指示符：

```
{-# language ScopedTypeVariables #-}
```

这将类型变量a引入where子句的作用域。

Exercise 12.5.1. 编写一个测试，获取一个整数列表，将其转换为Mu形式，并使用cataMu计算其总和。

12.6 Initial Algebra as a Colimit 初始代数作为余极限

通常情况下，在代数范畴中，初始对象不一定存在。但如果它存在，Lambek 引理告诉我们，它是这些代数自函子的一个不动点。这个不动点的构造有点神秘，因为它涉及递归结的绑扎。

粗略地说，不动点是在我们无限次应用函子之后达到的。然后，再应用它一次也不会改变任何东西。无穷加一仍然是无穷。如果我们一步一步地进行，这个想法可以被精确化。为简单起见，我们考虑集合范畴中的代数，它具有所有好的性质。

在我们的例子中，构建递归数据结构的实例总是从叶子开始。叶子是函子定义中不依赖于类型参数的部分：列表的`NilF`，树的`ValF`，`Maybe`的`Nothing`等。

如果我们将我们的函子 F 应用于初始对象——空集 0，我们可以将它们分离出来。由于空集没有元素，类型 $F0$ 的实例仅为叶子。

实际上，类型`Maybe Void`的唯一居住者是通过`Nothing`构造的。类型`ExprF Void`的唯一居住者是`ValF n`，其中`n`是一个`Int`。

换句话说，$F0$ 是函子 F 的“叶子类型”。叶子是深度为一的树。对于`Maybe`函子来说，只有一个。此函子的叶子类型是单例：

```
m1 :: Maybe Void
m1 = Nothing
```

在第二次迭代中，我们将 F 应用于上一步中的叶子，并得到深度至多为二的树。它们的类型是 $F(F0)$。

例如，这些是类型`Maybe(Maybe Void)`的所有项：

```
m2, m2' :: Maybe (Maybe Void)
m2 = Nothing
m2' = Just Nothing
```

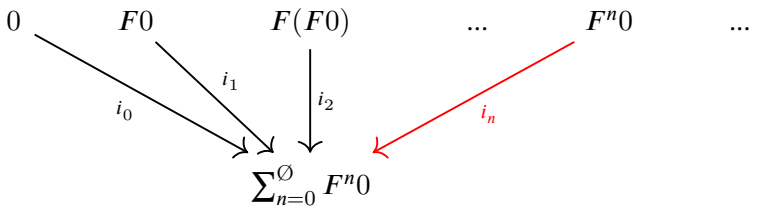
我们可以继续这个过程，在每一步中添加更深的树。在第 n 次迭代中，类型 F^n0（F 的 n 次应用于初始对象）描述了深度至多为 n 的所有树。然而，对于每个 n，仍然有无限多的深度大于 n 的树未被覆盖。

如果我们知道如何定义 $F^{\emptyset}0$，我们将覆盖所有可能的树。我们可以尝试的下一步是将所有这些部分树相加，并构造一个无限和类型。就像我们定义两个类型的和一样，我们可以定义许多类型的和，包括无限多的类型。

无限和（即余积）：

$$\sum_{n=0}^{\emptyset} F^n 0$$

就像有限和一样，只不过它有无限多个构造函数 i_n：



它具有通用的映射属性，类似于两个类型的和，只不过有无限多个情况。（显然，我们不能在 Haskell 中表达它。）

要构造深度为 n 的树，我们首先从 F^n0 中选择它，并使用第 n 个构造函数 i_n 将其注入和中。

只有一个问题：相同的树形状也可以通过任何 F^m0（$m > n$）构造。

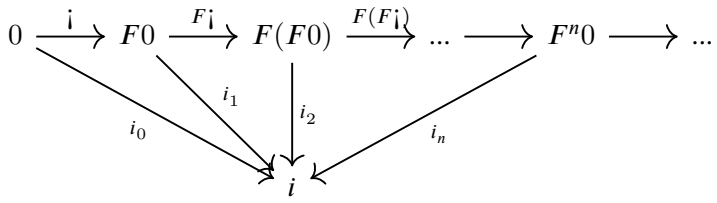
实际上，我们已经看到叶子`Nothing`出现在`Maybe Void`和`Maybe (Maybe Void)`中。事实上，它出现在`Maybe`作用于`Void`的任何非零次幂中。

类似地，`Just Nothing`出现在从二次幂开始的所有幂中。

`Just (Just (Nothing))`出现在从三次幂开始的所有幂中，以此类推……

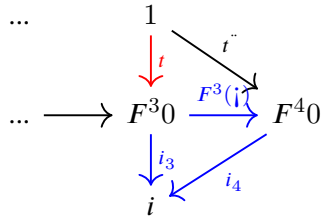
但是有一种方法可以消除所有这些重复。诀窍是用余极限替代和。我们可以构造一个链（这些链被称为 ω-链）。我们称这个链为 Γ，其余极限为 i：

$$i = \operatorname{Colim} \Gamma$$



这几乎与和相同，只不过在余锥 (Cocone) 的基部有额外的箭头。这些箭头是唯一箭头 ¡ 的累积提升，从初始对象到 $F0$（我们在 Haskell 中称之为`absurd`）。这些箭头的效果是将无限多个相同树的副本缩减为一个代表。

例如，考虑一个深度为 3 的树。它可以首先作为类型 F^30 的元素找到，即箭头 $t\colon 1 \to F^30$。它通过复合 $i_3 \circ t$ 注入余极限 i。



同一树形也可以在 $F^4 0$ 中找到，作为复合 $t^{\cdot\cdot} = F^3(¡)\circ t$。它通过复合 $i_4 \circ t^{\cdot\cdot} = i_4 \circ F^3(¡)\circ t$ 注入余极限。

然而，这次我们有一个余锥的通用三角形：

$$i_4 \circ F^3(¡) = i_3$$

这意味着：

$$i_4 \circ t^{\cdot\cdot} = i_4 \circ F^3(¡)\circ t = i_3 \circ t$$

在余极限中，这两棵树的副本已被识别。你可以确信这个过程消除了所有重复。

The proof 证明

我们可以直接证明 $i = \mathrm{Colim}\,\Gamma$ 是初始代数。然而，我们必须做一个假设：函子 F 必须保留 ω-链的余极限。$F\Gamma$ 的余极限必须等于 Fi。

$$\mathrm{Colim}(F\Gamma) \ddot{\circ} Fi$$

幸运的是，这个假设在 Set 中成立[1]。

这里是证明的概要：为了证明同构，我们首先构造箭头 $i \to Fi$，然后构造相反方向的箭头 $\imath\colon Fi \to i$。我们将跳过证明它们是彼此逆元的步骤。然后，我们将通过构造一个到任意代数的卡塔态射来证明 $(i, \imath)$ 的普遍性。

所有后续证明都遵循一个简单的模式。我们从定义余极限的通用余锥开始。然后，我们基于相同的链构造另一个余锥。由于通用性，必须存在一个从余极限到这个新余锥顶点的唯一箭头。

我们使用这个技巧来构造 $i \to Fi$ 的映射。如果我们能够从链 Γ 构造到 $\mathrm{Colim}(F\Gamma)$ 的余锥，那么根据通用性，必须存在一个从 i 到 $\mathrm{Colim}(F\Gamma)$ 的箭头。后者，由于我们的假设 F 保留余极限，是同构于 Fi 的。因此，我们将拥有一个 $i \to Fi$ 的映射。

为了构造这个余锥，首先注意到 $\mathrm{Colim}(F\Gamma)$ 根据定义是 $F\Gamma$ 的余锥的顶点。

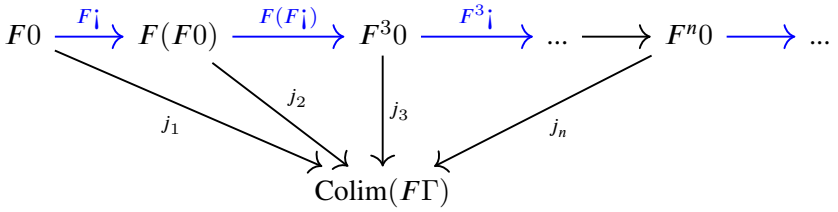
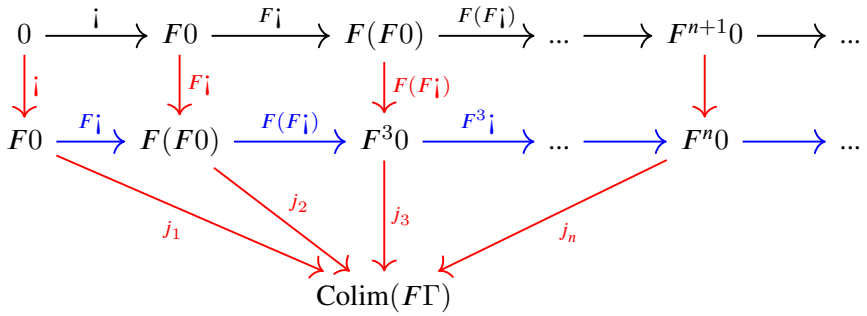


图 $F\Gamma$ 与 Γ 相同，只不过它缺少了链起始处的裸初始对象。

[1]这是 Set 中的余极限由集合的不相交并集构成的结果。

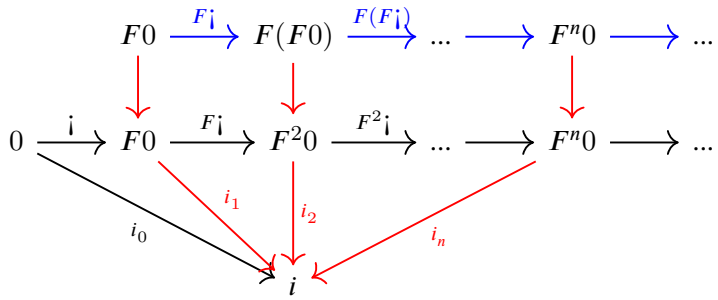
我们寻找的从 Γ 到 $\mathrm{Colim}(F\Gamma)$ 的余锥的辐条在下面的图中标记为红色：



因为 $i = \mathrm{Colim}\,\Gamma$ 是基于 Γ 的通用余锥的顶点，所以必须存在一个从其映射到 $\mathrm{Colim}(F\Gamma)$ 的唯一映射，这就是我们所说的 Fi。这是我们寻找的映射：

$$i \to Fi$$

接下来，注意链 $F\Gamma$ 是 Γ 的子链，因此可以嵌入其中。这意味着我们可以通过 Γ（的子链）构造一个从 $F\Gamma$ 到顶点 i 的余锥（下图中的红色箭头）。



从 $\mathrm{Colim}(F\Gamma)$ 的通用性可以得出存在一个映射

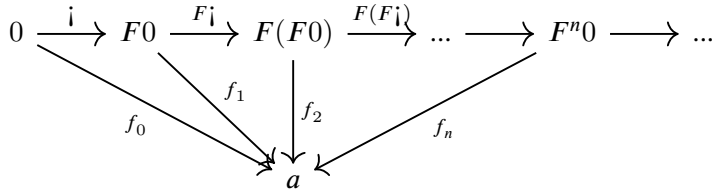
$$\mathrm{Colim}(F\Gamma) \to i$$

因此我们有了另一个方向的映射：

$$\iota\colon Fi \to i$$

这表明 i 是代数的载体。事实上，可以证明这两个映射是彼此的逆元，正如我们从 Lambek 引理中所预期的那样。

为了证明 (i, ι) 确实是初始代数，我们必须构造一个从它到任意代数 $(a, \alpha\colon Fa \to a)$ 的映射。再次，我们可以使用通用性，只要我们能够从 Γ 构造到 a 的余锥。



这个余锥的第零辐条从 0 到 a，所以它只是 $f_0 = ¡$。

第一个辐条 $F0 \to a$ 是 $f_1 = \alpha \circ F f_0$，因为 $F f_0 \colon F0 \to Fa$ 和 $\alpha \colon Fa \to a$。

第三个辐条 $F(F0) \to a$ 是 $f_2 = \alpha \circ F f_1$。依此类推...

从 i 到 a 的唯一映射就是我们的卡塔态射。通过一些更详细的图追踪，可以证明它确实是代数态射。

请注意，这种构造仅在我们可以通过创建函子的叶子来“启动”该过程的情况下才有效。另一方面，如果 $F0$ ö 0，那么就没有叶子，并且所有进一步的迭代将不断地生成 0。

Chapter 13

Coalgebras（余代数）

Coalgebras（余代数）就是在对偶范畴中的代数（algebras）。到此为止，本章结束！

不过，也许事情没有那么简单……正如我们之前所看到的，我们工作的范畴并不具有关于对偶性的对称性。特别是，如果我们比较终对象（terminal objects）和初对象（initial objects），它们的性质并不对称。我们的初对象没有入射箭头，而终对象除了拥有唯一的入射箭头外，还有许多出射箭头。

由于初代数（initial algebras）是从初对象构造出来的，因此我们可能预期终余代数（terminal coalgebras）——作为它们的对偶，因此由终对象生成——不仅仅是它们的镜像，还会有一些有趣的变化。

我们已经看到代数的主要应用是在处理递归数据结构时：在对它们进行折叠（folding）。与此对应，余代数的主要应用是在生成或展开（unfolding）递归的、类似树的数据结构。展开是通过一种称为 anamorphism（异态射）的方式来完成的。

我们使用 catamorphisms（猫态射）来修剪树，而使用 anamorphisms（异态射）来生成它们。

我们无法凭空生成信息，所以一般来说，无论是 catamorphism 还是 anamorphism，都会减少输入中包含的信息量。

例如，当你对整数列表求和后，就无法恢复原始的列表了。

同样地，如果你使用 anamorphism 生成递归数据结构，种子（seed）必须包含最终在树中呈现的所有信息。你没有获得新信息，但优势在于你拥有的信息现在以一种更方便进一步处理的形式存储了起来。

13.1 Coalgebras from Endofunctors（由自函子生成的余代数）

一个 endofunctor（自函子）F 的 coalgebra（余代数）是一个包含载体（carrier）a 和结构映射（structure map）的对：一个从 $a \to Fa$ 的箭头。

在 Haskell 中，我们定义：

```
type Coalgebra f a = a -> f a
```

我们通常将载体视为生成数据结构的种子类型，无论是列表还是树。

例如，这里有一个可以用来创建二叉树的函子，整数存储在节点中：

```
data TreeF x = LeafF | NodeF Int x x
deriving (Show, Functor)
```

我们甚至不需要为它定义Functor的实例——deriving子句告诉编译器为我们生成规范的实例（以及允许将其转换为String的Show实例，如果我们想显示它的话）。

一个 coalgebra（余代数）是一个函数，它接受载体类型的种子并生成一个充满新种子的函子。这些新种子然后可以用于递归地生成子树。

这里是一个为函子TreeF定义的 coalgebra（余代数），它将整数列表作为种子：

```
split :: Coalgebra TreeF [Int]
split [] = LeafF
split (n : ns) = NodeF n left right
where
(left, right) = partition (<= n) ns
```

如果种子为空，它会生成一片叶子；否则它会创建一个新节点。这个节点存储列表的头部，并用两个新种子填充节点。库函数partition使用用户定义的谓词分割列表，这里是(<= n)，小于或等于n。结果是一个满足谓词的列表对，和一个不满足的列表对。

你可以确信，递归地应用这个 coalgebra（余代数）会创建一个二叉排序树。我们稍后将使用这个 coalgebra 实现一个排序。

13.2 Category of Coalgebras（余代数的范畴）

通过类似代数态射（algebra morphisms）的方式，我们可以定义余代数态射（coalgebra morphisms）为满足交换条件的载体之间的箭头。

给定两个余代数 (a, α) 和 (b, β)，如果箭头 $f : a \to b$ 使得下图交换，那么它就是一个余代数态射：

$$\begin{array}{ccc} a & \xrightarrow{f} & b \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ Fa & \xrightarrow{Ff} & Fb \end{array}$$

其含义是，无论我们是先映射载体，然后应用余代数 β，还是先应用余代数 α，然后对其内容应用通过函子提升的箭头 Ff，结果都是相同的。

余代数态射可以组合，且恒等箭头自动是余代数态射。可以很容易地看出，余代数和代数一样，也形成一个范畴。

然而，这次我们感兴趣的是这个范畴中的终对象——终余代数。如果终余代数 (t, τ) 存在，它就满足 Lambek 引理（Lambek's lemma）的对偶。

Exercise 13.2.1. *Lambek's lemma:* 证明终余代数 (t, τ) 的结构映射 τ 是同构的。提示：证明过程与初代数的证明相对偶。

作为 Lambek 引理的结果，终代数的载体是该函子的一个不动点（fixed point）。

$$Ft \ddot{o} t$$

τ 和 τ^{*1} 作为这种同构的证据。

这也意味着 (t, τ^{*1}) 是一个代数；就像 (i, ι^{*1}) 是一个余代数一样，假设 (i, ι) 是初代数。

我们之前已经看到，初代数的载体是一个不动点。原则上，可能有许多相同自函子的不同不动点。初代数是最小的不动点，而终余代数是最大的。

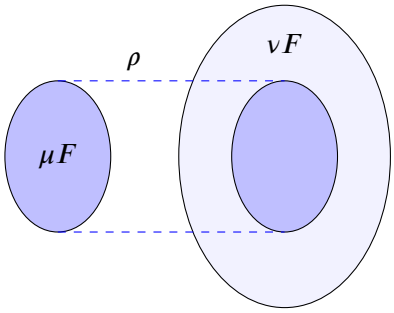
自函子 F 的最大不动点记为 νF，因此我们有：

$$t = \nu F$$

我们还可以看到，必然存在一个从初代数到终余代数的唯一代数态射（catamorphism）。这是因为终余代数也是一个代数。

同样，必然存在一个从初代数（它也是一个余代数）到终余代数的唯一余代数态射。事实上，可以证明，在这两种情况下，它是同一个基础态射 $\rho\colon \mu F \to \nu F$。

在集合范畴中，初代数的载体集合是终余代数的载体集合的子集，函数 ρ 将前者嵌入后者。



我们稍后会看到，在 Haskell 中，由于惰性求值的原因，情况更为微妙。但至少对于那些具有叶节点成分的函子——即它们在初对象上的作用是非平凡的——Haskell 的固定点类型可作为初代数和终余代数的载体。

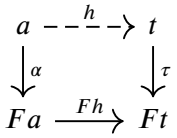
```
data Fix f where
In :: f (Fix f) -> Fix f
```

Exercise 13.2.2. 证明在 Set 中的恒等函子中，每个对象都是一个不动点，空集是最小的不动点，而单集合是最大的。提示：最小的不动点必须有箭头指向所有其他不动点，而最大的不动点必须有箭头从所有其他不动点指向它。

Exercise 13.2.3. 证明空集是 Set 中恒等函子的初代数的载体。对偶地，证明单集合是这个函子的终余代数。提示：证明这些唯一箭头确实是（余）代数态射。

13.3 Anamorphisms（异态射）

终余代数 (t, τ) 通过其泛性质（universal property）定义：从任何余代数 (a, α) 到 (t, τ) 的唯一余代数态射 h 被称为异态射（anamorphism）。作为一个余代数态射，它使下图交换：



就像代数一样，我们可以使用 Lambek 引理来“求解”h：

$$h = \tau^{*1} \circ Fh \circ \alpha$$

这种解法被称为 anamorphism（异态射），有时用“镜片括号”（lens brackets）$[\![\alpha]\!]$ 表示。

由于终余代数（与初代数一样）是函子的一个不动点，上述递归公式可以直接翻译为 Haskell 代码：

```
ana :: Functor f => Coalgebra f a -> a -> Fix f
ana coa = In . fmap (ana coa) . coa
```

这里是该公式的解释：给定类型为a的种子，我们首先对其应用余代数coa。这会给我们一个充满种子的函子。我们通过递归地应用异态射使用fmap扩展这些种子。然后我们应用构造器In得到最终结果。

例如，我们可以将异态射应用于我们之前定义的split余代数：ana split接受一个整数列表并创建一个排序树。

然后我们可以使用一个 catamorphism 来将这个树折叠成一个排序列表。我们定义以下代数：

```
toList :: Algebra TreeF [Int]
toList LeafF = []
toList (NodeF n ns ms) = ns ++ [n] ++ ms
```

它将左侧列表与单元素枢轴和右侧列表连接起来。要对列表进行排序，我们将 anamorphism 与 catamorphism 结合起来：

```
qsort = cata toList . ana split
```

这给了我们一个（非常低效的）快速排序实现。我们将在下一节中进一步讨论这个问题。

Infinite data structures（无限数据结构）

在研究代数时，我们依赖于具有叶节点成分的数据结构——即当自函子作用于初对象时，会产生与初对象不同的结果。构造递归数据结构时，我们必须从某个地方开始，这意味着首先构造叶子。

对于余代数，我们可以自由地放弃这个要求。我们不再需要“手工”构造递归数据结构——我们有 anamorphisms 来为我们做这件事。一个没有叶子的自函子是完全可以接受的：它的余代数将生成无限的数据结构。

在 Haskell 中，由于其惰性求值，无限数据结构是可表示的。只有显式要求的那些部分才会被计算，剩余部分则保持在挂起状态。

要在严格语言中实现无限数据结构，必须使用函数来表示值——这在 Haskell 中是幕后完成的（这些函数称为“thunks”）。

让我们看一个简单的例子：一个无限的值流。要生成它，我们首先定义一个看起来很像我们用来生成列表的函子，除了它缺少叶节点成分（空列表构造器）。你可能会将其识别为乘积函子，第一成分固定为流的有效负载：

```
data StreamF a x = StreamF a x
deriving Functor
```

一个无限流是这个函子的固定点。

```
type Stream a = Fix (StreamF a)
```

这是一个简单的余代数，它使用单个整数`n`作为种子：

```
step :: Coalgebra (StreamF Int) Int
step n = StreamF n (n+1)
```

它将当前种子存储为有效负载，并用`n + 1`为下一个流生成种子。

这个余代数的异态射，当种子为零时，生成所有自然数的流。

```
allNats :: Stream Int
allNats = ana step 0
```

在非惰性语言中，这个异态射将永远运行，但在 Haskell 中它是瞬时的。只有当我们想要检索一些数据时，才会逐步支付成本，例如，使用这些访问器：

```
head :: Stream a -> a
head (In (StreamF a _)) = a

tail :: Stream a -> Stream a
tail (In (StreamF _ s)) = s
```

13.4 Hylomorphisms（合态射）

anamorphism 的输出类型是函子的固定点，这与 catamorphism 的输入类型相同。在 Haskell 中，它们都由相同的数据类型`Fix f`描述。因此我们可以将它们组合在一起，就像我们在实现快速排序时所做的那样。事实上，我们可以将余代数与代数结合在一个递归函数中，称为**合态射**（hylomorphism）：

```
hylo :: Functor f => Algebra f b -> Coalgebra f a -> a -> b
hylo alg coa = alg . fmap (hylo alg coa) . coa
```

我们可以将快速排序重写为一个 hylomorphism：

```
qsort = hylo toList split
```

注意，在 hylomorphism 的定义中没有固定点的踪迹。从概念上讲，余代数用于从种子生成（展开）递归数据结构，而代数用于将其折叠成`b`类型的值。但由于 Haskell 的惰性求值，中间数据结构不必在内存中完全实体化。这在处理非常大的中间树时尤其重要。只有当前正在遍历的分支被评估，一旦它们被处理完毕，它们就会被传递给垃圾收集器。

在 Haskell 中，hylomorphisms 是递归回溯算法的一个方便替代，而递归回溯算法在命令式语言中非常难以正确实现。我们利用了一个事实，即设计数据结构比跟踪复杂的控制流和记住递归算法中的位置要容易得多。

通过这种方式，数据结构可以用于可视化复杂的控制流。

The impedance mismatch（阻抗不匹配）

我们已经看到，在集合范畴中，初代数不一定与终余代数重合。例如，恒等函子具有空集作为其初代数的载体，而具有单集合作为其终余代数的载体。

我们还有其他没有叶节点成分的函子，例如流函子。这样的函子的初代数也是空集。

在 Set 中，初代数是终余代数的子集，合态射只能定义在这个子集中。这意味着我们只能在特定的余代数的 anamorphism 落入这个子集时使用合态射。在这种情况下，由于初代数在终余代数中的嵌入是单射的，我们可以找到初代数中的相应元素并对其应用 catamorphism。

然而，在 Haskell 中，我们有一种类型`Fix f`，它同时结合了初代数和终余代数。这就是将 Haskell 类型简单解释为值的集合的局限性。

让我们考虑这个简单的流代数：

```
add :: Algebra (StreamF Int) Int
add (StreamF n sum) = n + sum
```

没有什么能阻止我们使用合态射来计算所有自然数的和：

```
sumAllNats :: Int
sumAllNats = hylo add step 1
```

这是一个完全有效的 Haskell 程序，通过了类型检查器。那么，当我们运行它时，它会生成什么值呢？（提示：它不是 *1/12。）答案是：我们不知道，因为这个程序永远不会终止。它进入了无限递归，并最终耗尽了计算机的资源。

这是现实计算中函数之间的集合模型无法模拟的一个方面。一些计算机函数可能永远不会终止。

递归函数在形式上通过“域理论”（domain theory）描述为部分定义函数的极限。如果函数未对特定的参数值定义，则该函数被认为返回一个底值 ⊥。如果我们将底值作为每种类型的特殊元素（这些类型称为*提升类型*（lifted types）），我们可以说我们的函数`sumAllNats`返回了一个`Int`类型的底值。一般来说，针对无限类型的 catamorphisms 不会终止，因此我们可以将它们视为返回底值。

然而，应该注意的是，底值的引入复杂了 Haskell 的范畴解释。特别是，许多依赖于唯一性的映射的泛化构造不再如预期那样工作。

“底线”是，Haskell 代码应该被视为范畴概念的一个说明，而不是严格证明的来源。

13.5 Terminal Coalgebra from Universality（从泛性质推导的终余代数）

异态射的定义可以看作是终余代数的泛性质的一种表达。这里是它的定义，带有显式的泛量化：

```
ana :: Functor f => forall a. Coalgebra f a -> (a -> Fix f)
ana coa = In . fmap (ana coa) . coa
```

它告诉我们，给定任何余代数，从其载体到终余代数的载体`Fix f`都有一个映射。我们知道，通过 Lambek 引理，这个映射实际上是一个余代数态射。

让我们将这个定义的参数展开：

```
ana :: Functor f => forall a. (a -> f a, a) -> Fix f
ana (coa, x) = In (fmap (curry ana coa) (coa x))
```

我们可以将此公式用作终余代数载体的替代定义。我们可以将`Fix f`替换为我们正在定义的类型——让我们称之为`Nu f`。类型签名：

```
forall a. (a -> f a, a) -> Nu f
```

告诉我们，我们可以从一个对`(a -> f a, a)`构造出`Nu f`的一个元素。它看起来就像一个数据构造器，除了它在`a`上是多态的。

具有多态构造器的数据类型称为*存在类型*（existential types）。在伪代码中（非实际的 Haskell），我们可以定义`Nu f`为：

```
data Nu f = Nu (exists a. (Coalgebra f a, a))
```

将其与最小不动点代数的定义进行比较：

```
data Mu f = Mu (forall a. Algebra f a -> a)
```

要构造一个存在类型的元素，我们可以选择最方便的类型——我们有构造器所需的数据的类型。

例如，我们可以通过选择`Int`作为方便类型，并提供以下对来构造类型`Nu (StreamF Int)`的一个项：

```
nuArgs :: (Int -> StreamF Int Int, Int)
nuArgs =  (\n -> StreamF n (n+1) , 0)
```

存在类型的客户端无法知道其构造时所使用的类型是什么。他们只知道这种类型“存在”——因此得名。如果他们想要使用一个存在类型，他们必须以一种不敏感于构造时所做选择的方式使用它。实际上，这意味着存在类型必须同时携带生成器和消费者。

在我们的示例中确实如此：生成器只是`a`类型的值，而消费者是`a -> f a`函数。

天真地说，客户端对这一对能做的唯一操作是在不知道`a`是什么类型的情况下，将函数应用于值。但如果`f`是一个函子，他们可以做得更多。他们可以通过将提升的函数应用于`f a`的内容来重复这一过程，依此类推。他们最终得到无限流中包含的所有信息。

在 Haskell 中有几种定义存在数据类型的方法。我们可以直接将异态射的未展开版本用作数据构造器：

```
data Nu f where
Nu :: forall a f. (a -> f a, a) -> Nu f
```

注意，在 Haskell 中，如果我们显式地量化了一个类型，所有其他类型变量也必须量化：这里是类型构造器`f`（然而，`Nu f`在`f`上不是存在的，因为它是一个显式参数）。

我们还可以完全省略量化：

```
data Nu f where
Nu :: (a -> f a, a) -> Nu f
```

这是因为不属于类型构造器的参数的类型变量自动被视为存在类型。

我们还可以使用更传统的形式：

```
data Nu f = forall a. Nu (a -> f a, a)
```

（这个版本需要对`a`进行量化。）

在撰写本文时，有一个提案引入关键字`exists`到 Haskell 中，这样可以使以下定义生效：

```
data Nu f = Nu (exists a. (a -> f a, a))
```

（稍后我们将看到，存在数据类型对应于范畴论中的余端（coends）。）

Nu f的构造器实际上就是异态射的未展开版本：

```
anaNu :: Coalgebra f a -> a -> Nu f
anaNu coa a = Nu (coa, a)
```

如果我们得到了一个Nu (StreamF a)形式的流，我们可以使用访问函数提取其元素。这个函数提取第一个元素：

```
head :: Nu (StreamF a) -> a
head (Nu (unf, s)) =
let (StreamF a _) = unf s
in a
```

这个函数推进流：

```
tail :: Nu (StreamF a) -> Nu (StreamF a)
tail (Nu (unf, s)) =
let (StreamF _ s') = unf s
in Nu (unf, s')
```

你可以在无限整数流上测试它们：

```
allNats = Nu nuArgs
```

13.6 Terminal Coalgebra as a Limit（极限表示的终余代数）

在范畴论中，我们不惧怕无限——我们让它们有意义。

表面上看，认为我们可以通过将函子 F 无限次应用于某个对象（比如终对象 1）来构造终余代数的想法毫无意义。但这个想法非常具有说服力：再应用一次 F 就像在无限上加一——仍然是无限。因此，天真地认为这是 F 的一个不动点：

$$F(F^{\emptyset}1) \ddot{o} F^{\emptyset+1}1 \ddot{o} F^{\emptyset}1$$

要将这种宽松的推理转化为严格的证明，我们必须驯服这个无限，这意味着我们必须定义某种极限程序。

例如，让我们考虑乘积函子：

$$F_a x = a \quad x$$

它的终余代数是一个无限流。我们通过从终对象 1 开始来近似它。下一步是：

$$F_a 1 = a \quad 1 \ddot{o} a$$

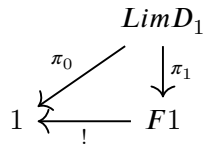
我们可以将其想象为一个长度为一的流。我们可以继续：

$$F_a(F_a 1) = a \quad (a \quad 1) \,\ddot{\circ}\, a \quad a$$

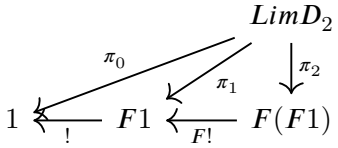
一个长度为二的流，依此类推。

这看起来很有前途，但我们需要的是一个能够结合所有这些近似值的对象。我们需要一种方法将下一个近似值与前一个粘合在一起。

回想一下前面练习中的“行走箭头”图的极限。这个极限具有与图中起始对象相同的元素。特别是，考虑单箭头图 $\boldsymbol{D}_1$ 的极限：

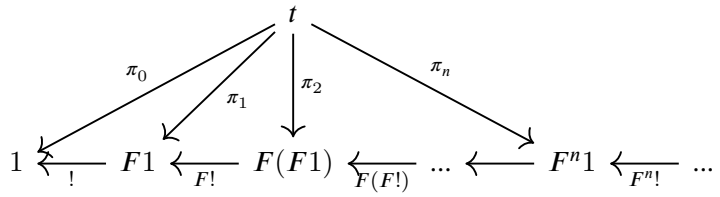


（! 是指向终对象 1 的唯一态射）。这个极限具有与 $\boldsymbol{F}1$ 相同的元素。同样，双箭头图 $\boldsymbol{D}_2$ 的极限：



具有与 $\boldsymbol{F}(\boldsymbol{F}1)$ 相同的元素。

我们可以继续将此图扩展到无限。事实证明，这个无限链的极限是我们所需的终余代数的固定点载体。



这个事实的证明可以通过将初代数的类似证明反转箭头得到。

Chapter 14

Monads（单子）

轮子、陶罐和木房子有什么共同点？它们都因为中心的空洞而变得有用。

老子说：“有之以为利，无之以为用。”

`Maybe`函子、列表函子和读者函子有什么共同点？它们的中心也都有空洞。

当单子（monads）在编程背景下被解释时，如果你专注于函子，很难看到共同的模式。要理解单子，你必须深入到函子内部，阅读代码之间的含义。

14.1 Programming with Side Effects（带有副作用的编程）

到目前为止，我们一直在讨论的编程主要是基于集合之间的函数（除了非终止的情况）。在编程中，这些函数被称为*全函数*（total）和*纯函数*（pure）。

全函数对其参数的所有值都有定义。

纯函数纯粹基于其参数实现，若是闭包函数，纯函数只基于捕获的值——它无法访问外部世界，更不能修改它。

然而，大多数现实世界的程序都必须与外部世界交互：它们读取和写入文件，处理网络数据包，提示用户输入数据，等等。大多数编程语言通过允许副作用来解决这个问题。副作用是指任何打破函数全性或纯性的行为。

不幸的是，命令式语言采用的这种“散弹枪”方法使得对程序进行推理变得极其困难。在组合带有副作用的计算时，人们必须小心地逐案推理副作用的组合。更糟糕的是，大多数副作用不仅隐藏在特定函数的实现（而不是接口）中，还隐藏在它所调用的所有函数的实现中，甚至是递归调用的函数中。

纯函数式语言（如 Haskell）采用的解决方案是将副作用编码到纯函数的返回类型中。令人惊讶的是，这对于所有相关的副作用都是可行的。

这个想法是，用返回类型为`a->b`的计算代替带有副作用的计算，我们使用一个函数`a -> f b`，其中类型构造器`f`编码了适当的副作用。在这一点上，`f`没有任何限制。它

甚至不必是一个Functor，更不必是一个单子。这将在我们讨论副作用组合时再来详细说明。

下面是常见副作用及其纯函数版本的列表。

Partiality（部分性）

在命令式语言中，部分性通常通过异常来编码。当一个函数使用“错误”的值调用时，它会抛出异常。在某些语言中，异常的类型使用特殊语法编码在函数的签名中。

在 Haskell 中，部分计算可以通过返回Maybe函子内的结果来实现。当一个函数使用“错误”的参数调用时，它会返回Nothing，否则会将结果包装在Just构造器中。

如果我们想要编码更多关于失败类型的信息，可以使用Either函子，其中Left通常传递错误数据（通常是一个简单的String）；而Right封装了实际的返回结果，如果有的话。

调用返回Maybe类型的函数的调用者无法轻易忽略异常情况。为了提取值，他们必须对结果进行模式匹配，并决定如何处理Nothing。这与某些命令式语言中使用空指针编码错误情况的“穷人版Maybe”形成对比。

Logging（记录）

有时一个计算必须在某个外部数据结构中记录一些值。记录或审计是一个特别危险的副作用，尤其是在并发程序中，多个线程可能会尝试同时访问同一个日志。

一个简单的解决方案是让函数返回计算值，并与要记录的项配对。换句话说，将类型为a -> b的记录计算替换为纯函数：

```
a -> Writer w b
```

其中Writer函子是一个薄薄的积的封装：

```
newtype Writer w a = Writer (a, w)
```

其中w是日志的类型。

然后由调用者负责提取要记录的值。这是一个常见的技巧：让函数提供所有的数据，让调用者处理副作用。

Environment（环境）

有些计算需要只读访问存储在环境中的某些外部数据。只读环境可以简单地作为附加参数传递给函数，而不是由计算秘密访问。如果我们有一个需要访问某些环境e的计算a -> b，我们将其替换为一个函数(a, e) -> b。起初，这似乎不符合在返回类型中编码副作用的模式。然而，这样的函数总是可以柯里化为以下形式：

```
a -> (e -> b)
```

该函数的返回类型可以编码在读者函子中，它本身由环境类型e参数化：

```
newtype Reader e a = Reader (e -> a)
```

这是一个延迟副作用的示例。该函数：

```
a -> Reader e a
```

不想处理副作用，因此它将责任委托给调用者。你可以将其视为生成一个稍后执行的脚本。函数runReader扮演了这个脚本的一个非常简单的解释器角色：

```
runReader :: Reader e a -> e -> a
runReader (Reader h) e = h e
```

State（状态）

最常见的副作用与访问和潜在地修改某些共享状态有关。不幸的是，共享状态是并发错误的臭名昭著的根源。这是面向对象语言中的一个严重问题，因为有状态的对象可以透明地在多个客户端之间共享。在 Java 中，此类对象可以提供单独的互斥锁，但代价是性能下降和死锁的风险。

在函数式编程中，我们显式地处理状态操作：我们将状态作为附加参数传递，并返回与返回值配对的修改后的状态。我们将一个有状态的计算a -> b替换为：

```
(a, s) -> (b, s)
```

其中s是状态的类型。像以前一样，我们可以将这样的函数柯里化，以将其转换为以下形式：

```
a -> (s -> (b, s))
```

该返回类型可以封装在以下函子中：

```
newtype State s a = State (s -> (a, s))
```

调用这样的函数的调用者应该通过提供初始状态并调用帮助函数，即解释器runState，来检索结果和修改后的状态：

```
runState :: State s a -> s -> (a, s)
runState (State h) s = h s
```

注意，模块化构造器解包，runState是完全合法的函数应用。

Nondeterminism（非确定性）

想象一下进行量子实验，测量电子的自旋。半数时间自旋向上，半数时间自旋向下。结果是不确定的。一种描述方式是使用多世界解释：当我们进行实验时，宇宙分裂成两个

宇宙，每个结果对应一个。

函数的不确定性意味着它每次调用都会返回不同的结果。我们可以使用多世界解释来模拟这种行为：我们让函数一次返回所有可能的结果。实际上，我们会选择一个（可能是无限的）结果列表：

我们用纯函数替换一个非确定性计算`a -> b`，该函数返回一个充满结果的函子——这次是列表函子：

```
a -> [b]
```

同样，调用者决定如何处理这些结果。

Input/Output（输入/输出）

这是最棘手的副作用，因为它涉及与外部世界的交互。显然，我们无法在计算机程序中模拟整个世界。因此，为了保持程序的纯粹性，交互必须在程序外部发生。技巧是让程序生成一个脚本。然后将该脚本传递给运行时以执行。运行时是运行程序的有副作用的虚拟机。

这个脚本本身位于不透明的、预定义的`IO`函子中。程序无法访问此函子中的值：没有`runIO`函数。相反，程序生成的`IO`值在程序结束后执行，至少概念上是如此。

实际上，由于 Haskell 的惰性求值，I/O 的执行与程序的其余部分交织在一起。构成程序主体的纯函数根据需要进行求值——需求由`IO`脚本的执行驱动。如果不是因为 I/O，什么都不会被求值。

Haskell 程序生成的`IO`对象称为`main`，其类型签名为：

```
main :: IO ()
```

这是包含单位的`IO`函子——意味着：除了输入/输出脚本之外没有其他有用的值。

我们很快会讨论如何创建`IO`操作。

Continuation（续延）

我们已经看到，作为 Yoneda 引理的结果，我们可以将类型为`a`的值替换为一个处理该值的函数。该处理器称为续延。调用处理器被视为计算的副作用。在纯函数的上下文中，我们将其编码为：

```
a -> Cont r b
```

其中`Cont r`是以下函子：

```
newtype Cont r a = Cont ((a -> r) -> r)
```

调用此函数的调用者负责提供续延，即函数`k :: a -> r`，并检索结果：

```
runCont :: Cont r a -> (a -> r) -> r
runCont (Cont f) k = f k
```

这是Cont r的Functor实例：

```
instance Functor (Cont r) where
-- f :: a -> b
-- k :: b -> r
fmap f c = Cont (\k -> runCont c (k . f))
```

请注意，这是一个协变函子，因为类型a处于双重否定位置。

在笛卡尔闭范畴中，续延由自函子生成：

$$K_r a = r^{r^a}$$

14.2 Composing Effects（组合副作用）

现在我们知道如何使用一个既产生值又产生副作用的函数进行一次巨大飞跃，接下来的问题是弄清楚如何将这个飞跃分解为较小的适合人类的步骤。或者，换句话说，如何将这些较小的步骤组合成一个更大的步骤。

命令式语言中副作用计算的组合方式是使用常规函数组合来处理值，并让副作用随意组合。

当我们将带有副作用的计算表示为纯函数时，我们面临的问题是如何组合两个形式为g :: a -> f b和h :: b -> f c的函数。

在所有感兴趣的情况下，类型构造器f恰好是一个Functor，因此在接下来的讨论中我们将假设这一点。

一种天真的方法是解包第一个函数的结果，将该值传递给下一个函数，然后在旁边组合这两个函数的副作用，并将它们与第二个函数的结果结合起来。即使是我们到目前为止研究的情况，这也不是总能实现的，更不用说对于一个任意的类型构造器。

为了进行讨论，看看我们如何对Maybe函子实现这一点是有启发性的。如果第一个函数返回Just，我们通过模式匹配提取内容并将其传递给下一个函数。

但是，如果第一个函数返回Nothing，我们就没有值来调用第二个函数。我们必须短路它，并直接返回Nothing。因此，组合是可能的，但这意味着通过根据第一个调用的副作用跳过第二个调用来修改控制流。

对于某些函子，副作用的组合是可能的，而对于其他函子则不是。我们如何描述这些“好”的函子？

要使一个函子编码可组合的副作用，我们至少必须能够实现以下多态高阶函数：

```
composeWithEffects :: Functor f =>
(b -> f c) -> (a -> f b) -> (a -> f c)
```

这非常类似于常规函数组合：

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

因此自然会问，在什么范畴中，前者定义了箭头的组合。让我们看看构造这样一个范畴还需要什么。

这个新范畴中的对象与以前一样是 Haskell 类型。但是箭头 $a \nrightarrow b$ 是由一个 Haskell 函数实现的：

```
g :: a -> f b
```

然后我们可以使用`composeWithEffects`来实现这些箭头的组合。

为了构造一个范畴，我们要求这个组合是结合的。我们还需要每个对象`a`的恒等箭头。这是箭头 $a \nrightarrow a$，因此它对应于一个 Haskell 函数：

```
idWithEffects :: a -> f a
```

它必须相对于`composeWithEffects`的行为像恒等箭头。

从另一种角度来看，这个箭头允许你向任何类型为`a`的值添加一个琐碎的副作用。这是一个组合在一起时对任何其他副作用都不起作用的副作用。

我们刚刚定义了一个单子！经过一些重命名和重新排列，我们可以将其写为一个类型类：

```
class Functor m => Monad m where
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
return :: a -> m a
```

中缀操作符`<=<`取代了`composeWithEffects`函数。函数`return`是我们新范畴中的恒等箭头。（这不是你会在 Haskell 的`Prelude`中找到的单子的定义，但正如我们很快会看到的那样，它与之等价。）

作为练习，让我们定义`Maybe`的`Monad`实例。‘鱼‘操作符`<=<`将两个函数：

```
f :: a -> Maybe b
g :: b -> Maybe c
```

组合成一个类型为`a -> Maybe c`的函数。这个组合的单位，`return`，将一个值封装在`Just`构造器中。

```
instance Monad Maybe where
g <=< f = \a -> case f a of
Nothing -> Nothing
Just b -> g b
```

```
        return = Just
```

你可以很容易地说服自己，范畴定律是满足的。特别是return <=< g与g相同，f <=< return与f相同。结合律的证明也很简单：如果任何一个函数返回Nothing，结果就是Nothing；否则，这只是一个简单的函数组合，而函数组合是结合的。

我们刚刚定义的这个范畴称为*Kleisli 范畴*，适用于单子m。函数a -> m b称为*Kleisli 箭头*。它们使用<=<组合，恒等箭头称为return。

前一节中的所有函子都是Monad实例。如果你将它们视为类型构造器，甚至函子，很难看到它们之间的相似之处。它们的共同点是它们可以用来实现可组合的 Kleisli 箭头。

正如老子所说：组合发生在事物之间。当我们专注于事物时，我们常常会忽略事物之间的空隙。

14.3 Alternative Definitions（替代定义）

使用 Kleisli 箭头定义单子的优点在于，单子定律只是范畴的结合律和单位律。还有两种等价的单子定义，一种是数学家喜欢的，另一种是程序员喜欢的。

首先，让我们注意到，在实现鱼操作符时，我们会得到两个函数作为参数。函数唯一有用的地方就是可以应用到一个参数上。当我们应用第一个函数f :: a -> m b时，我们得到一个类型为m b的值。这时我们可能会陷入困境，但好在m是一个函子。函子的特性使我们能够将第二个函数g :: b -> m c应用到m b上。实际上，g的提升由m处理，类型为：

```
        m b -> m (m c)
```

这几乎是我们要找的结果，如果我们能够将m(m c)压平为m c。这种压平操作称为join。换句话说，如果我们得到了：

```
        join ::  m (m a) -> m a
```

我们就可以实现<=<：

```
        g <=< f = \a -> join (fmap g (f a))
```

or, using point free notation:

```
        g <=< f = join . fmap g . f
```

反过来，我们也可以通过<=<来实现join：

```
        join = id <=< id
```

后者可能不太容易理解，直到你意识到最右边的id应用于m (m a)，而最左边的id应用于m a。我们将 Haskell 函数：

```
m (m a) -> m (m a)
```

解释为 Kleisli 范畴中的箭头 $m(ma) \twoheadrightarrow ma$。类似地，函数：

```
m a -> m a
```

实现了 Kleisli 箭头 $ma \twoheadrightarrow a$。它们的 Kleisli 组合产生了一个 Kleisli 箭头 $m(ma) \twoheadrightarrow a$，即一个 Haskell 函数：

```
m (m a) -> m a
```

这引导我们得出了一个等价的定义，即使用join和return来定义单子：

```
class Functor m => Monad m where
join :: m (m a) -> m a
return :: a -> m a
```

这仍然不是你在标准 Haskell 的Prelude中会看到的定义。由于鱼操作符<=<是点操作符的推广，使用它相当于无点编程。它让我们在不命名中间值的情况下组合箭头。虽然有些人认为无点编程更优雅，但大多数程序员发现它们难以理解。

然而，函数组合实际上是分两个步骤进行的：我们应用第一个函数，然后将第二个函数应用于结果。显式地命名中间结果通常有助于理解发生了什么。

要对 Kleisli 箭头执行相同操作，我们必须知道如何将第二个 Kleisli 箭头应用于命名的单子值——第一个 Kleisli 箭头的结果。执行此操作的函数称为 *bind*，并作为一个中缀操作符表示：

```
(>>=) :: m a -> (a -> m b) -> m b
```

显然，我们可以用bind来实现 Kleisli 组合：

```
g <=< f = \a -> (f a) >>= g
```

反过来，也可以用 Kleisli 箭头来实现bind：

```
ma >>= k = (k <=< id) ma
```

这引导我们得出以下定义：

```
class Monad m where
(>>=) :: m a -> (a -> m b) -> m b
return :: a -> m a
```

这几乎就是你会在Prelude中看到的定义，除了一个附加的约束。这个约束声明了每个Monad实例也是Applicative实例。我们将在讨论单子函子时推迟讨论 Applicative。

我们也可以使用bind实现join：

```
join  :: (Monad m) => m (m a) -> m a
join mma =  mma >>= id
```

Haskell 函数id从m a到m a，或作为一个 Kleisli 箭头，$ma \twoheadrightarrow a$。

有趣的是，用bind定义的Monad自动成为一个函子。它的提升函数称为liftM:

```
liftM :: Monad m => (a -> b) -> (m a -> m b)
liftM f ma = ma >>= (return . f)
```

14.4 Monad Instances（单子实例）

我们现在已经准备好为我们用于副作用的函子定义单子实例了。这将允许我们组合副作用。

Partiality（部分性）

我们已经看到通过 Kleisli 组合实现的Maybe单子版本。这里是更常见的使用bind的实现:

```
instance Monad Maybe where
Nothing >>= k = Nothing
(Just a) >>= k = k a
return = Just
```

将琐碎的副作用添加到任何值意味着将其封装在Just中。

Logging（记录）

为了组合生成日志的函数，我们需要一种方法来组合单个日志条目。这就是为什么记录器单子:

```
newtype Writer w a = Writer (a, w)
```

要求日志的类型是Monoid的实例。这允许我们追加日志，并创建一个琐碎的副作用：一个空日志。

```
instance Monoid w => Monad (Writer w) where
(Writer (a, w)) >>= k = let (Writer (b, w')) = k a
in Writer (b, mappend w w')
return a = Writer (a, mempty)
```

let子句用于引入局部绑定。在这里，应用k的结果进行模式匹配，局部变量b和w'被初始化。let/in结构是一个表达式，其值由in子句的内容给出。

Environment（环境）

读者单子是从环境到返回类型的函数的一个薄封装:

```
newtype Reader e a = Reader (e -> a)
```

这里是Monad实例：

```
instance Monad (Reader e) where
ma >>= k = Reader (\e -> let a = runReader ma e
in runReader (k a) e)
return a = Reader (\e -> a)
```

读取器单子的bind的实现创建了一个函数，该函数接受环境作为参数。这个环境被使用两次，第一次是运行ma来获取a的值，然后用k a生成的值进行求值。

return的实现忽略了环境。

Exercise 14.4.1. 定义以下数据类型的*Functor*和*Monad*实例：

```
newtype E e a = E (e -> Maybe a)
```

提示：你可以使用这个方便的函数：

```
runE :: E e a -> e -> Maybe a
runE (E f) e = f e
```

State（状态）

像读取器一样，状态单子也是一个函数类型：

```
newtype State s a = State (s -> (a, s))
```

它的bind类似，除了k作用于a的结果必须用修改后的状态s'运行。

```
instance Monad (State s) where
st >>= k = State (\s -> let (a, s') = runState st s
in runState (k a) s')

return a = State (\s -> (a, s))
```

将bind应用于恒等式给出了join的定义：

```
join :: State s (State s a) -> State s a
join mma = State (\s -> let (ma, s') = runState mma s
in runState ma s')
```

注意，我们本质上是将第一个runState的结果传递给第二个runState，只是我们必须解构第二个以便它可以接受一个对偶：

```
join mma = State (\s -> (uncurry runState) (runState mma s))
```

在这种形式下，它很容易转换为无点记法：

```
join mma = State (uncurry runState . runState mma)
```

有两个基本的 Kleisli 箭头（第一个，概念上，来自终端对象()），我们可以用它们构造任意的有状态计算。第一个检索当前状态：

```
get :: State s s
get = State (\s -> (s, s))
```

第二个修改它：

```
set :: s -> State s ()
set s = State (\_ -> ((), s))
```

许多单子带有它们自己的预定义的基本 Kleisli 箭头库。

Nondeterminism（非确定性）

对于列表单子，让我们考虑如何实现join。它必须将一个列表的列表转换为单个列表。这可以通过使用库函数concat连接所有内部列表来完成。从那里，我们可以推导出bind的实现。

```
instance Monad [] where
as >>= k = concat (fmap k as)
return a = [a]
```

在这里，return构造了一个单例列表。因此，非确定性的琐碎版本就是确定性。

命令式语言中使用嵌套循环做的事情，我们可以在 Haskell 中使用列表单子来做。将as看作是绑定内循环运行的结果，并将k视为外循环中运行的代码。

在许多方面，Haskell 的列表更像是在命令式语言中称为迭代器或生成器的东西。由于惰性求值，列表的元素很少同时存储在内存中，因此你可以将 Haskell 列表概念化为一个指向头部的指针和一个推进到尾部的配方。或者你可以将列表看作是一个协程，它按需生成序列的元素。

Continuation（续延）

续延单子的bind实现：

```
newtype Cont r a = Cont ((a -> r) -> r)
```

需要一些逆向思维，因为控制的固有反转——“不要打电话给我们，我们会打电话给你”的原则。

bind的结果类型为Cont r b。为了构造它，我们需要一个函数，该函数接受k :: b -> r作为参数：

```
ma >>= fk = Cont (\k -> ...)
```

我们有两个可供使用的成分：

```
ma :: Cont r a
fk :: a -> Cont r b
```

我们希望运行ma，为此我们需要一个接受a的续延。

```
runCont ma (\a -> ...)
```

一旦我们有了a，我们就可以执行我们的fk。结果类型为Cont r b，因此我们可以用我们的续延k :: b -> r来运行它。

```
runCont (fk a) k
```

总的来说，这个复杂的过程产生了以下实现：

```
instance Monad (Cont r) where
ma >>= fk = Cont (\k -> runCont ma (\a -> runCont (fk a) k))
return a = Cont (\k -> k a)
```

正如我之前提到的，组合续延不是心脏虚弱者能干的事情。然而，它只需要实现一次——在续延单子的定义中。从那以后，do记法会使其余部分相对简单。

Input/Output（输入/输出）

IO单子的实现嵌入在语言中。基本的 I/O 原语通过库提供。它们以 Kleisli 箭头或IO对象的形式存在（概念上，Kleisli 箭头来自终端对象()）。

例如，以下对象包含从标准输入读取一行的命令：

```
getLine :: IO String
```

没有办法从中提取字符串，因为它还不存在；但程序可以通过进一步的 Kleisli 箭头处理它。

IO单子是终极的拖延症患者：其 Kleisli 箭头的组合堆积了一个又一个任务，等待由 Haskell 运行时执行。

要输出一个字符串并在其后加上换行符，你可以使用这个 Kleisli 箭头：

```
putStrLn :: String -> IO ()
```

结合这两者，你可以构造一个简单的main对象：

```
main :: IO ()
main = getLine >>= putStrLn
```

它会回显你输入的字符串。

14.5 Do Notation（do 记法）

值得重复一遍，单子在编程中的唯一目的是让我们将一个大的 Kleisli 箭头分解为多个较小的箭头。

这可以直接通过无点风格使用 Kleisli 组合<=<来完成，或者通过命名中间值并将它们绑定到 Kleisli 箭头上使用>>=来完成。

一些 Kleisli 箭头在库中定义，另一些是足够可重用的以保证脱线实现，但在实践中，大多数是作为单次内联 lambda 表达式实现的。

这里有一个简单的例子：

```
main :: IO ()
main =
getLine >>= \s1 ->
getLine >>= \s2 ->
putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

它使用了一个类型为`String->IO ()`的即席 Kleisli 箭头，由 lambda 表达式定义：

```
\s1 ->
getLine >>= \s2 ->
putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

此 lambda 表达式的主体进一步通过另一个即席 Kleisli 箭头分解：

```
\s2 -> putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

这种构造非常常见，因此有一种称为`do`记法的特殊语法，可以减少很多样板代码。例如，上面的代码可以写成：

```
main = do
s1 <- getLine
s2 <- getLine
putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

编译器会自动将其转换为一系列嵌套的 lambda 表达式。通常，行`s1<-getLine`被读取为：“`s1` 获取了`getLine`的结果。”

这里是另一个例子：一个使用列表单子生成从两个列表中取出所有可能的元素对的函数。

```
pairs :: [a] -> [b] -> [(a, b)]
pairs as bs = do
```

```
        a <- as
        b <- bs
        return (a, b)
```

注意do块中的最后一行必须生成一个单子值——这里通过使用return来实现。

大多数命令式语言缺乏泛型定义单子的抽象能力，因此它们试图硬编码一些更常见的单子。例如，它们实现了异常作为Either单子的替代，或者并发任务作为续延单子的替代。一些语言，如 C++，引入了模仿 Haskell 的do记法的协程。

Exercise 14.5.1. *实现一个适用于任何单子的函数：*

```
ap :: Monad m => m (a -> b) -> m a -> m b
```

提示：使用do记法提取函数和参数。使用return返回结果。

Exercise 14.5.2. *使用bind操作符和 lambda 重写pairs函数。*

14.6 Continuation Passing Style（续延传递风格）

我之前提到过，do 语法糖使得使用续延（continuations）更加自然。续延最重要的应用之一就是将程序转换为 CPS（Continuation Passing Style，续延传递风格）。CPS 转换在编译器构造中非常常见。另一个非常重要的应用是将递归转换为迭代。

深度递归程序的常见问题是它们可能会耗尽运行时堆栈。函数调用通常会通过在堆栈上推入函数参数、本地变量以及返回地址开始。因此，深度嵌套的递归调用可能会迅速耗尽（通常是固定大小的）运行时堆栈，导致运行时错误。这是为什么命令式语言更喜欢使用循环而非递归，以及为什么大多数程序员在学习递归之前先学习循环的原因。然而，即使在命令式语言中，当涉及到遍历递归数据结构（如链表或树）时，递归算法也是更自然的选择。

然而，使用循环的问题在于它们需要使用变异（mutation）。通常会有某种计数器或指针在每次循环时进行递增和检查。这就是为什么纯函数式语言（如 Haskell）避免使用变异，并使用递归代替循环的原因。但由于循环更加高效且不会消耗运行时堆栈，编译器会尝试将递归调用转换为循环。在 Haskell 中，所有尾递归函数都会被转换为循环。

Tail recursion and CPS（尾递归与 CPS）

尾递归（Tail recursion）意味着递归调用发生在函数的最末尾。函数不会对尾调用的结果执行任何附加操作。例如，下面的程序不是尾递归的，因为它需要将 i 加到递归调用的结果上：

```haskell
sum1 :: [Int] -> Int
sum1 [] = 0
sum1 (i : is) = i + sum1 is
```

相反，下面的实现是尾递归的，因为递归调用 `go` 的结果被直接返回而没有进一步的修改：

```haskell
sum2 = go 0
where go n [] = n
go n (i : is) = go (n + i) is
```

编译器可以轻松地将后者转换为循环。它不会进行递归调用，而是会用 `n + i` 覆盖第一个参数 `n` 的值，用指向列表头部的指针覆盖指向其尾部的指针，然后跳转到函数的开始处。

然而需要注意的是，这并不意味着 Haskell 编译器不会巧妙地优化第一个实现。它只意味着第二个实现是尾递归的，因此保证会被转换为循环。

实际上，通过执行 CPS 转换，总是可以将递归转换为尾递归。这是因为续延封装了计算的剩余部分，因此它总是函数中的最后一个调用。

要在实践中了解其工作原理，请考虑一个简单的树遍历。我们定义一个存储字符串的树，其中节点和叶子都存储字符串：

```haskell
data Tree = Leaf String
| Node Tree String Tree
```

为了将这些字符串连接起来，我们使用遍历，首先递归进入左子树，然后进入右子树：

```haskell
show :: Tree -> String
show (Node lft s rgt) =
let ls = show lft
rs = show rgt
in ls ++ s ++ rs
```

这显然不是一个尾递归函数，而且如何将其转换为尾递归也并不明显。然而，我们几乎可以机械地使用 continuation monad（续延单子）重写它：

```haskell
showk :: Tree -> Cont r String
showk (Leaf s) = return s
showk (Node lft s rgt) = do
ls <- showk lft
rs <- showk rgt
return (ls ++ s ++ rs)
```

我们可以使用 trivial continuation `id` 来运行结果：

```
show :: Tree -> String
show t = runCont (showk t) id
```

此实现自动为尾递归。通过展开 do 语法糖，我们可以清楚地看到这一点：

```
showk :: Tree -> (String -> r) -> r
showk (Leaf s) k = k s
showk (Node lft s rgt) k =
showk lft (\ls ->
showk rgt (\rs ->
k (ls ++ s ++ rs)))
```

让我们分析一下这段代码。函数调用自己，传递左子树 `lft` 以及以下 continuation（续延）：

```
\ls ->
showk rgt (\rs ->
k (ls ++ s ++ rs))
```

这个 lambda 进一步调用 `showk`，传递右子树 `rgt` 以及另一个 continuation：

```
\rs -> k (ls ++ s ++ rs)
```

这个最内层的 lambda 具有访问所有三个字符串（左、中、右）的权限。它们连接后，将结果传递给最外层的 continuation `k`。

在每种情况下，对 `showk` 的递归调用都是最后的调用，其结果会立即返回。此外，结果的类型是泛型类型 `r`，这本身就保证了我们无法对其执行任何操作。当我们最终运行 `showk` 的结果时，我们将 `id` 传递给它（实例化为 `String` 类型）：

```
show :: Tree -> String
show t = runCont (showk t) id
```

Using named functions（使用命名函数）

假设我们的编程语言不支持匿名函数。是否有可能用命名函数替代 lambda？当我们讨论伴随函子定理（adjoint functor theorem）时，我们之前已经做过类似的工作。我们注意到由 continuation monad 生成的 lambda 是闭包——它们从环境中捕获一些值。如果我们想用命名函数替换它们，就必须显式地传递环境。

我们用对名为 `next` 的函数的调用替换第一个 lambda，并以包含三个值 `(s, rgt, k)` 的元组的形式传递必要的环境：

```
showk :: Tree -> (String -> r) -> r
showk (Leaf s) k = k s
showk (Node lft s rgt) k =
showk lft (next (s, rgt, k))
```

这三个值分别是树当前节点中的字符串、右子树和外部 continuation。

函数 `next` 对 `showk` 进行递归调用，传递给它右子树以及名为 `conc` 的 continuation：

```
next :: (String, Tree, String -> r) -> String -> r
next (s, rgt, k) ls = showk rgt (conc (ls, s, k))
```

同样，`conc` 显式捕获包含两个字符串和外部 continuation 的环境。它执行字符串的连接并将结果传递给外部 continuation：

```
conc :: (String, String, String -> r) -> String -> r
conc (ls, s, k) rs = k (ls ++ s ++ rs)
```

最后，我们定义 trivial continuation：

```
done :: String -> String
done s = s
```

用来提取最终结果：

```
show t = showk t done
```

Defunctionalization（去函数化）

CPS（续延传递风格）需要使用高阶函数。如果这是个问题，例如在实现分布式系统时，我们可以始终使用伴随函子定理来去函数化我们的程序。

第一步是创建所有相关环境的总和，包括我们在 `done` 中使用的空环境：

```
data Kont = Done
| Next String Tree Kont
| Conc String String Kont
```

请注意，该数据结构可以重新解释为列表或栈。它可以看作是以下求和类型的元素列表：

```
data Sum = Next' String Tree  | Conc' String String
```

该列表是我们实现递归算法所需的运行时堆栈的版本。

因为我们只关心生成字符串作为最终结果，所以我们要近似 `String -> String` 函数类型。此函数类型是定义它的伴随关系的近似余单元（counit）（参见伴随函子定理）：

```
apply :: (Kont, String) -> String
apply (Done, s) = s
apply (Next s rgt k, ls) = showk rgt (Conc ls s k)
apply (Conc ls s k, rs) = apply (k, ls ++ s ++ rs)
```

现在可以在不使用高阶函数的情况下实现 showk 函数：

```
showk :: Tree -> Kont -> String
showk (Leaf s) k = apply (k, s)
showk (Node lft s rgt) k = showk lft (Next s rgt k)
```

要提取结果，我们调用它并传递 Done：

```
showTree t = showk t Done
```

14.7 Monads Categorically（范畴论中的 Monad）

在范畴论中，Monad 首先出现在代数的研究中。特别地，bind 运算符可用于实现非常重要的替换操作。

Substitution（替换）

考虑这个简单的表达式类型。它由类型 x 参数化，我们可以用它来命名变量：

```
data Ex x = Val Int
| Var x
| Plus (Ex x) (Ex x)
deriving (Functor, Show)
```

例如，我们可以构造一个表达式 $(2 + a) + b$：

```
ex :: Ex Char
ex = Plus (Plus (Val 2) (Var 'a')) (Var 'b')
```

我们可以为 Ex 实现 Monad 实例：

```
instance Monad Ex where
Val n >>= k = Val n
Var x >>= k = k x
Plus e1 e2 >>= k =
let x = e1 >>= k
y = e2 >>= k
in (Plus x y)
```

```
return x = Var x
```

现在假设你想通过将变量 a 替换为 $x_1 + 2$，将变量 b 替换为 x_2 来进行替换（为简化起见，我们不考虑其他字母）。此替换由 Kleisli 箭头 `sub` 表示：

```
sub :: Char -> Ex String
sub 'a' = Plus (Var "x1") (Val 2)
sub 'b' = Var "x2"
```

如你所见，我们甚至能够将用于命名变量的类型从 `Char` 更改为 `String`。

当我们将此 Kleisli 箭头绑定到 `ex` 时：

```
ex' :: Ex String
ex' = ex >>= sub
```

我们得到的正是与表达式 $(2 + (x_1 + 2)) + x_2$ 对应的树。

Monad as a monoid（Monad 作为单子）

让我们分析使用 `join` 定义的 Monad：

```
class Functor m => Monad m where
join :: m (m a) -> m a
return :: a -> m a
```

我们有一个 endofunctor（自函子）`m` 和两个多态函数。

在范畴论中，定义 Monad 的函子通常用 T 表示（可能是因为 Monad 最初被称为“三元组”）。这两个多态函数成为自然变换。第一个对应于 `join`，将 T 的“平方”（即 T 与自身的复合）映射到 T：

$$\mu : T \circ T \to T$$

（当然，只有自-函子才能以这种方式平方。）

第二个对应于 `return`，将恒等函子映射到 T：

$$\eta : \mathrm{Id} \to T$$

与我们之前在单子范畴中定义的单子相比较：

$$\mu : m \otimes m \to m$$
$$\eta : I \to m$$

相似性是惊人的。这就是为什么我们经常称自然变换 μ 为 *monadic multiplication*（单子乘法）。但是在哪个范畴中，函子的复合可以视为张量积呢？

进入 endofunctor 的范畴。该范畴中的对象是 endofunctor，而箭头是自然变换。

但该范畴还有更多的结构。我们知道，任何两个 endofunctor 都可以复合。如果我们想将 endofunctor 视为对象，该复合如何解释？一种将两个对象结合并产生第三个对象的操作类似于张量积。张量积唯一的条件是它在两个参数中都是函子性的。也就是说，给定一对箭头：

$$\alpha \colon T \to T''$$
$$\beta \colon S \to S''$$

我们可以将其提升到张量积的映射：

$$\alpha \otimes \beta \colon T \otimes S \to T'' \otimes S''$$

在 endofunctor 的范畴中，箭头是自然变换，所以，如果我们将 $\otimes$ 替换为 $\circ$，则提升是复合映射：

$$\alpha \circ \beta \colon T \circ T'' \to S \circ S''$$

但这只是自然变换的水平复合（现在你明白为什么它用圆圈表示了）。

这个 monoidal category（单子范畴）中的单位对象是恒等 endofunctor，并且单位律“在鼻子上”成立，意味着

$$\mathrm{Id} \circ T = T = T \circ \mathrm{Id}$$

我们不需要任何 unitor。我们也不需要任何 associator，因为函子复合是自动关联的。

单位对象和 associator 均为恒等态射的 monoidal category 被称为*严格 monoidal category*。

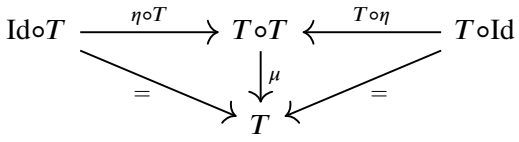
然而需要注意的是，复合不是对称的，因此这不是对称 monoidal category。

总而言之，一个 monad 是 endofunctor 的 monoidal category 中的一个 monoid。

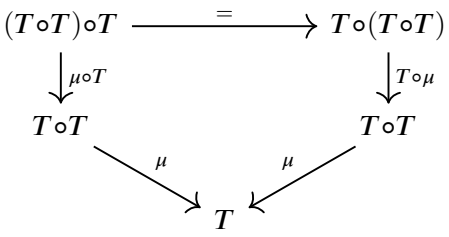
一个 monad (T, η, μ) 由 endofunctor 范畴中的一个对象组成——即 endofunctor T；和两个箭头——即自然变换：

$$\eta \colon \mathrm{Id} \to T$$
$$\mu \colon T \circ T \to T$$

为了成为一个 monoid，这些箭头必须满足 monoidal laws。以下是单位律（unitor 替换为严格等式）：



这是结合律：



我们使用 whiskering notation（掺合记法）来表示 $\mu \circ T$ 和 $T \circ \mu$ 的水平复合。

这些是使用 μ 和 η 表示的 monad laws。它们可以直接翻译为 `join` 和 `return` 的 laws。它们也等价于使用箭头 $a \to Tb$ 构建的 Kleisli 范畴的 laws。

14.8 Free Monads（自由 Monad）

Monad 允许我们指定可能产生副作用的操作序列。这种序列不仅告诉计算机做什么，还告诉它如何去做。但有时需要更多的灵活性：我们希望将“做什么”与“如何做”分开。自由 Monad 允许我们生成操作序列而无需承诺特定的 Monad 来执行它。这类似于定义自由 Monoid（列表），它让我们推迟选择应用的代数；或者在将其编译为可执行代码之前创建一个抽象语法树（AST）。

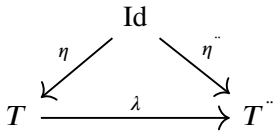
自由构造被定义为遗忘函子的左伴随。因此，我们首先需要定义遗忘 Monad 的含义。由于 Monad 是一个带有额外结构的自函子，我们想遗忘这个结构。我们取一个 Monad (T, η, μ)，并仅保留 T。但为了将这种映射定义为一个函子，我们首先需要定义 Monad 的范畴。

Category of monads（Monads 的范畴）

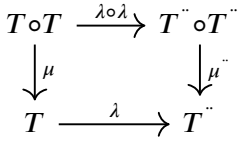
$\mathrm{Mon}(\mathcal{C})$ 范畴中的对象是 Monad (T, η, μ)。我们可以定义两个 Monad (T, η, μ) 和 $(T^{¨}, \eta^{¨}, \mu^{¨})$ 之间的箭头为两个自函子之间的自然变换：

$$\lambda \colon T \to T^{¨}$$

然而，由于 Monad 是带有结构的自函子，我们希望这些自然变换能够保留结构。单位的保留意味着以下图表必须交换：



乘法的保留意味着以下图表必须交换：



从另一个角度来看，$\mathrm{Mon}(\mathcal{C})$ 是 monoidal category $([\mathcal{C},\mathcal{C}],\circ,\mathrm{Id})$ 中 monoid 的范畴。

Free monad（自由 Monad）

既然我们有了 Monad 的范畴，我们就可以定义遗忘函子：

$$U : \mathrm{Mon}(\mathcal{C}) \to [\mathcal{C},\mathcal{C}]$$

它将每个三元组 (T,η,μ) 映射为 T，并将每个 Monad 态射映射为底层自然变换。

我们希望自由 Monad 由此遗忘函子的左伴随生成。问题是这个左伴随并不总是存在。通常，这是与规模问题有关：Monad 往往会导致规模爆炸。最终，自由 Monad 对于一些，但不是所有的自函子存在。因此，我们不能通过伴随关系定义自由 Monad。幸运的是，在大多数感兴趣的情况下，自由 Monad 可以定义为代数的固定点。

这个构造类似于我们如何将自由 Monoid 定义为列表函子的初始代数：

```
data ListF a x = NilF | ConsF a x
```

或更一般的形式：

$$F_a x = I + a \otimes x$$

然而这一次，定义 Monad 作为 monoid 的 monoidal category 是 endofunctor 的范畴 $([\mathcal{C},\mathcal{C}],\circ,\mathrm{Id})$。在这个范畴中的自由 Monoid 是高阶“列表”函子的初始代数，它将函子映射为函子：

$$\Phi_F G = \mathrm{Id} + F \circ G$$

这里，两个函子的coproduct 是按点定义的。在对象上的定义：

$$(F+G)a = Fa + Ga$$

在箭头上的定义：

$$(F + G)f = Ff + Gf$$

（我们使用 coproduct 的函子性形成两个态射的 coproduct。我们假设 C 是余笛卡尔的，即所有 coproduct 都存在。）

初始代数是该运算符的（最小）固定点，或递归方程的解：

$$L_F \ddot{o} \mathrm{Id} + F \circ L_F$$

此公式在两个函子之间建立了自然同构。从右到左，$\mathrm{Id} + F \circ L_F \to L_F$，我们有一个从和的映射，相当于一对自然变换：

$$\mathrm{Id} \to L_F$$
$$F \circ L_F \to L_F$$

当翻译成 Haskell 时，这些变换的组件变成了两个构造函数。我们定义以下由函子 `f` 参数化的递归数据类型：

```
data FreeMonad f a where
Pure :: a -> FreeMonad f a
Free :: f (FreeMonad f a) -> FreeMonad f a
```

如果我们将函子 `f` 视为值的容器，构造函数 `Free` 会取一个 `(FreeMonad f a)` 的函子并将其存储起来。因此，类型 `FreeMonad f a` 的任意值是一个树，其中每个节点都是分支的函子，每个叶子包含一个类型为 `a` 的值。

由于此定义是递归的，因此它的 `Functor` 实例也是递归的：

```
instance Functor f => Functor (FreeMonad f) where
fmap g (Pure a) = Pure (g a)
fmap g (Free ffa) = Free (fmap (fmap g) ffa)
```

这里，外层 `fmap` 使用 `f` 的 `Functor` 实例，而内层的 `(fmap g)` 递归到分支。

很容易看出 `FreeMonad` 是一个 `Monad`。monadic unit `eta` 只是恒等函子的一个简单封装：

```
eta :: a -> FreeMonad f a
eta a = Pure a
```

monadic multiplication 或 `join` 是递归定义的：

```
mu :: Functor f => FreeMonad f (FreeMonad f a) -> FreeMonad f a
mu (Pure fa) = fa
mu (Free ffa) = Free (fmap mu ffa)
```

因此，FreeMonad f 的 Monad 实例为：

```
instance Functor f => Monad (FreeMonad f) where
return a = eta a
m >>= k = mu (fmap k m)
```

我们也可以直接定义 bind：

```
(Pure a)   >>= k = k a
(Free ffa) >>= k = Free (fmap (>>= k) ffa)
```

自由 Monad 累积 monadic 动作在树状结构中，而不承诺任何特定的评估策略。这个树可以使用代数“解释”。但这次它是 endofunctor 范畴中的代数，因此它的载体是一个 endofunctor G，结构映射 α 是自然变换 $\Phi_F G \to G$：

$$\alpha \colon \mathrm{Id} + F \circ G \to G$$

此自然变换作为一个从和的映射，相当于一对自然变换：

$$\lambda \colon \mathrm{Id} \to G$$
$$\rho \colon F \circ G \to G$$

我们可以将其翻译为 Haskell 中的一对多态函数：

```
type MAlg f g a = (a -> g a, f (g a) -> g a)
```

由于自由 Monad 是初始代数，因此存在唯一的映射——catamorphism，从它到任何其他代数。回忆一下我们是如何为常规代数定义 catamorphism 的：

```
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

out 部分解包固定点的内容。这里我们可以通过对自由 Monad 的两个构造函数进行模式匹配来实现这一点。如果它是一个叶子，我们将 λ 应用于它。如果它是一个节点，我们递归处理其内容，并将结果应用于 ρ：

```
mcata :: Functor f => MAlg f g a -> FreeMonad f a -> g a
mcata (l, r) (Pure a) = l a
mcata (l, r) (Free ffa) =
r (fmap (mcata (l, r)) ffa)
```

许多树状 Monad 实际上是简单函子的自由 Monad。

Exercise 14.8.1. 一个（非空）*rose tree* 定义为：

```
data Rose a = Leaf a | Rose [Rose a]
deriving Functor
```

实现 *Rose a* 和 *FreeMonad [] a* 之间的相互转换。

Exercise 14.8.2. 实现二叉树和 *FreeMonad Bin a* 之间的相互转换，其中：

```
data Bin a = Bin a a
```

Exercise 14.8.3. 找到一个函子，其自由 *Monad* 等价于列表 *Monad [a]*。

Stack calculator example（堆栈计算器示例）

作为一个示例，我们来看看一个实现为嵌入式领域特定语言（EDSL）的堆栈计算器。我们将使用自由 Monad 来累积用此语言编写的简单命令。

命令由函子 `StackF` 定义。将参数 `k` 视为 continuation。

```
data StackF k  = Push Int k
| Top (Int -> k)
| Pop k
| Add k
deriving Functor
```

例如，`Push` 旨在将整数压入堆栈，然后调用 continuation `k`。

此函子的自由 Monad 可以被视为一棵树，大多数分支只有一个子节点，因此形成列表。例外是 `Top` 节点，它有许多子节点，每个子节点对应于一个 `Int` 值。

这是该函子的自由 Monad：

```
type FreeStack = FreeMonad StackF
```

为了创建特定领域的程序，我们将定义一些辅助函数。有一个泛型函数将一个函子的值提升为自由 Monad：

```
liftF :: (Functor f) => f r -> FreeMonad f r
liftF fr = Free (fmap (Pure) fr)
```

我们还需要一系列“智能构造函数”，它们是自由 Monad 的 Kleisli 箭头：

```
push :: Int -> FreeStack ()
push n = liftF (Push n ())

pop :: FreeStack ()
pop = liftF (Pop ())
```

```
top :: FreeStack Int
top = liftF (Top id)

add :: FreeStack ()
add = liftF (Add ())
```

由于自由 Monad 是 Monad，因此我们可以方便地使用 `do` 语法结合 Kleisli 箭头。例如，这里有一个玩具程序，它将两个数字相加并返回它们的和：

```
calc :: FreeStack Int
calc = do
push 3
push 4
add
x <- top
pop
return x
```

为了执行该程序，我们需要定义一个代数，其载体是一个 endofunctor。由于我们要实现一个基于堆栈的计算器，我们将使用状态函子的一个版本。其状态是一个整数列表。状态函子定义为一个函数类型；在这里，它是一个接受列表并返回新列表与类型参数 `k` 的函数：

```
data StackAction k = St ([Int] -> ([Int], k))
deriving Functor
```

要运行该动作，我们将函数应用于堆栈：

```
runAction :: StackAction k -> [Int] -> ([Int], k)
runAction (St act) ns = act ns
```

我们将代数定义为自由 Monad 的两个构造函数（`Pure` 和 `Free`）对应的一对多态函数：

```
runAlg :: MAlg StackF StackAction a
runAlg = (stop, go)
```

第一个函数终止程序的执行并返回一个值：

```
stop :: a -> StackAction a
stop a = St (\xs -> (xs, a))
```

第二个函数对命令的类型进行模式匹配。每个命令携带一个 continuation。这个 continuation 必须与一个（可能已修改的）堆栈一起运行。每个命令以不同的方式修改堆栈：

```
        go :: StackF (StackAction k) -> StackAction k
        go (Pop k)    = St (\ns -> runAction k (tail ns))
        go (Top ik)   = St (\ns -> runAction (ik (head ns)) ns)
        go (Push n k) = St (\ns -> runAction k (n: ns))
        go (Add k)    = St (\ns -> runAction k
        ((head ns + head (tail ns)): tail (tail ns)))
```

例如，Pop 丢弃堆栈的顶部。Top 从堆栈顶部获取一个整数，并用它选择要执行的分支。它通过将函数 ik 应用于该整数来实现这一点。Add 将堆栈顶部的两个数字相加并将结果压入堆栈。

注意，我们定义的代数不涉及递归。从操作中分离递归是自由 Monad 方法的优势之一。相反，递归被一次编码在 catamorphism 中。

这是一个可以用来运行我们的玩具程序的函数：

```
        run :: FreeMonad StackF k -> ([Int], k)
        run prog = runAction (mcata runAlg prog) []
```

显然，使用部分函数 head 和 tail 使我们的解释器变得脆弱。格式不正确的程序会导致运行时错误。更健壮的实现将使用允许错误传播的代数。

使用自由 Monad 的另一个优势是相同的程序可以使用不同的代数进行解释。

Exercise 14.8.4. *实现一个“漂亮打印机”来显示使用我们的自由 Monad 构建的程序。提示：实现使用 Const 函子作为载体的代数：*

```
        showAlg :: MAlg StackF (Const String) a
```

14.9 单胚函子 (Monoidal Functors)

我们已经看到了几个单胚范畴 (monoidal categories) 的例子。这类范畴具有某种二元运算，例如笛卡尔积、和、复合（在自函子的范畴中），等等。它们还具有一个特殊的对象，作为该二元运算的单位元。在严格单胚范畴 (strict monoidal categories) 中，单位和结合律是严格满足的，而在其他情况下，它们则是满足同构条件。

每当我们有某种结构的多个实例时，我们可能会问自己：是否有一个关于这些东西的整体范畴？在这种情况下：单胚范畴是否构成了自己的一个范畴？为了使这成为可能，我们需要定义单胚范畴之间的箭头。

一个*单胚函子 (monoidal functor)* F 从单胚范畴 $(\mathcal{C}, \otimes, i)$ 映射到另一个单胚范畴 $(\mathcal{D}, \oplus, j)$ 时，会将张量积 (tensor product) 映射到张量积，将单位元映射到单位元——

所有这些都是在同构条件下进行的：

$$Fa \oplus Fb \ddot{\circ} F(a \otimes b)$$
$$j \ddot{\circ} Fi$$

这里，左侧是目标范畴中的张量积和单位元，右侧是源范畴中的对应物。

如果所讨论的两个单胚范畴不是严格的，即单位和结合律仅在同构条件下满足，则还需要额外的相容性条件，以确保单位元和结合元被正确映射。

以单胚函子为箭头的单胚范畴构成的范畴称为 MonCat。实际上，它是一个 2-范畴，因为可以定义在单胚函子之间的结构保持自然变换。

松弛单胚函子 (Lax Monoidal Functors)

单胚范畴的一个好处是它们允许我们定义单子 (monoids)。你可以轻易地发现，单胚函子将单子映射到单子。但实际上，你并不需要单胚函子的全部功能来实现这一点。让我们考虑为了将单子映射到单子，函数子需要满足的最低要求。

让我们从单胚范畴 $(C, \otimes, i)$ 中的一个单子 (m, μ, η) 开始。考虑一个将 m 映射到 Fm 的函子 F。我们希望 Fm 在目标单胚范畴 $(\mathcal{D}, \oplus, j)$ 中也是一个单子。为此，我们需要找到两个映射：

$$\ddot{\eta} : j \to Fm$$
$$\ddot{\mu} : Fm \oplus Fm \to Fm$$

并使其满足单胚律。

因为 m 是一个单子，我们有可用的原始映射的提升：

$$F\eta : Fi \to Fm$$
$$F\mu : F(m \otimes m) \to Fm$$

为了实现 $\ddot{\eta}$ 和 $\ddot{\mu}$，我们还需要两个额外的箭头：

$$j \to Fi$$
$$Fm \oplus Fm \to F(m \otimes m)$$

一个单胚函子会提供这样的箭头。然而，为了我们要完成的工作，这些箭头并不需要是可逆的。

一个*松弛单胚函子 (lax monoidal functor)* 是一个配备有态射 ϕ_i 和自然变换 ϕ_{ab} 的函子：

$$\phi_i : j \to Fi$$
$$\phi_{ab} : Fa \oplus Fb \to F(a \otimes b)$$

它们满足适当的单位性和结合性条件。

这样的函子将单子 (m, μ, η) 映射到单子 $(Fm, \mu^{\cdot\cdot}, \eta^{\cdot\cdot})$，其中：

$$\eta^{\cdot\cdot} = F\eta \circ \phi_i$$
$$\mu^{\cdot\cdot} = F\mu \circ \phi_{ab}$$

松弛单胚函子的最简单例子是一个保留通常笛卡尔积的自函子。我们可以在 Haskell 中将其定义为一个类型类：

```
class Monoidal f where
unit  :: f ()
(>*<) :: f a -> f b -> f (a, b)
```

与 ϕ_{ab} 对应的我们有一个中缀运算符，根据 Haskell 的惯例，它以柯里化形式编写。

Exercise 14.9.1. *实现列表函子的 `Monoidal` 实例。*

函子强度 (Functorial Strength)

还有另一种方式，函子可以与单胚结构进行交互，这在编程时显而易见。我们理所当然地认为函数能够访问环境。这样的函数被称为闭包 (closures)。

例如，这里有一个从环境中捕获变量 `a` 并将其与其参数配对的函数：

```
\x -> (a, x)
```

这个定义在孤立情况下没有意义，但在环境中包含变量 `a` 时，它是有意义的，例如：

```
pairWith :: Int -> (String -> (Int, String))
pairWith a = \x -> (a, x)
```

通过调用 `pairWith 5` 返回的函数 ‘‘闭合’’ 了环境中的 5。

现在考虑以下修改，它返回一个包含闭包的单例列表：

```
pairWith' :: Int -> [String -> (Int, String)]
pairWith' a = [\x -> (a, x)]
```

作为程序员，你会很惊讶如果这不工作。但我们在这里所做的是非常微妙的：我们将环境 ‘‘偷渡’’ 到列表函子之下。根据我们的 演算 (lambda calculus) 模型，一个闭包是从环境和函数参数的积中得到的一个态射。这里的 lambda 实际上是一个 `(Int, String)` 类型的函数，它在一个列表函子内部定义，但它捕获了定义在列表之外的值 `a`。

让我们能够将环境偷渡到函子之下的属性称为*函子强度 (functorial strength)* 或者*张量强度 (tensorial strength)*，可以在 Haskell 中实现为：

```haskell
strength :: Functor f => (e, f a) -> f (e, a)
strength (e, as) = fmap (e, ) as
```

符号 `(e, )` 被称为元组截断 *(tuple section)*，相当于对对构造器 (pair constructor) 的偏应用：`(,) e`。

在范畴论中，一个自函子 F 的强度定义为一个将张量积 (tensor product) 偷渡到函子中的自然变换：

$$\sigma\colon a \otimes F(b) \to F(a \otimes b)$$

还有一些附加条件，以确保它与单胚范畴中的单位元和结合元良好地配合。

我们能够为任意函子实现 `strength`，这意味着在 Haskell 中，每个函子都是强的。这就是为什么我们在函子内部访问环境时不必担心。

更重要的是，在 Haskell 中，每个 monad 由于是函子，因此也是强的。这也是为什么每个 monad 都自动是 `Monoidal` 的。

```haskell
instance Monad m => Monoidal m where
unit = return ()
ma >*< mb = do
a <- ma
b <- mb
return (a, b)
```

如果你将这段代码解糖 (desugar) 以使用 monadic 绑定和 lambda，你会注意到最终的 `return` 需要访问定义在外部环境中的 `a` 和 `b`。如果没有 monad 是强的，这将是不可能的。

然而，在范畴论中，并非每个单胚范畴中的自函子都是强的。现在的神秘咒语是，我们所工作的范畴是自丰 (self-enriched) 的，并且在 Haskell 中定义的每个自函子都是丰的 (enriched)。我们将在讨论丰范畴 (enriched categories) 时回到这一点。在 Haskell 中，强度归结为我们总是可以对部分应用的对构造器 `(a, )` 进行 `fmap`。

应用函子 (Applicative Functors)

在编程中，应用函子的想法源自以下问题：一个函子让我们可以提升一个变量的函数。我们如何提升一个有两个或多个变量的函数?

通过类似于 `fmap` 的方式，我们想要有一个函数：

```haskell
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

一个有两个参数的函数——在这里是其柯里化形式——是一个返回函数的单参数函数。所以，假设 `f` 是一个函子，我们可以对 `liftA2` 的第一个参数进行 `fmap`，其类型为：

```
        a -> (b -> c)
```

应用于第二个参数 (f a) 来得到：

```
        f (b -> c)
```

问题是，我们不知道如何将 f (b -> c) 应用于剩下的参数 (f b)。

允许我们这么做的函子类称为 Applicative。事实证明，一旦我们知道如何提升一个二参数函数，我们就可以提升任意数量的参数函数，除了零个参数。一个零参数函数只是一个值，所以提升它意味着实现一个函数：

```
        pure :: a -> f a
```

这里是 Haskell 的定义：

```
        class Functor f => Applicative f where
        pure  :: a -> f a
        (<*>) :: f (a -> b) -> f a -> f b
```

将函数应用于带参数的函子定义为一个中缀运算符 <*>，通常称为 “splat”。

还有一个 fmap 的中缀版本：

```
        (<$>) :: Functor f => (a -> b) -> f a -> f b
```

可以用它来实现这个简洁的 liftA2：

```
        liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
        liftA2 g as bs = g <$> as <*> bs
```

两个运算符都绑定到左侧，这使得这个语法模仿了常规函数应用。

应用函子还必须满足一组定律：

```
pure id <*> v = v                            -- 恒等性 (Identity)
pure f <*> pure x = pure (f x)               -- 同态性 (Homomorphism)
u <*> pure y = pure ($ y) <*> u              -- 交换性 (Interchange)
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- 组合性 (Composition)
```

Exercise 14.9.2. 实现 liftA3，一个使用应用函子提升一个三参数函数的函数。

闭函子 (Closed Functors)

如果你细看 splat 运算符的定义：

```
(<*>) :: f (a -> b) -> (f a -> f b)
```

你可能会将它看作是从函数对象映射到函数对象。

当你考虑一个在两个闭范畴之间的函子时，这一点变得更加清晰。你可以从源范畴中的一个函数对象 b^a 开始，并将函子 F 应用于它：

$$F(b^a)$$

或者，你可以将这两个对象 a 和 b 进行映射，并在目标范畴中构造它们之间的一个函数对象：

$$(Fb)^{Fa}$$

如果我们要求这两种方式是同构的，我们就得到了一个严格的*闭函子 (closed functor)* 的定义。但正如单胚函子的情况一样，我们对松弛版本更感兴趣，它配备了一个单向的自然变换：

$$F(b^a) \to (Fb)^{Fa}$$

如果 F 是一个自函子，这直接转换为 splat 运算符的定义。

松弛闭函子的完整定义包括单胚单位的映射和一些相容性条件。总的来说，一个应用函子是一个松弛闭函子。

在一个闭笛卡尔范畴 (closed cartesian category) 中，指数与笛卡尔积通过柯里化伴随 (currying adjunction) 相关联。因此，在这样的范畴中，松弛单胚和松弛闭（应用）自函子是相同的。

我们可以在 Haskell 中轻松表达这一点：

```
instance (Functor f, Monoidal f) => Applicative f where
pure a = fmap (const a) unit
fs <*> as = fmap apply (fs >*< as)
```

其中 `const` 是一个忽略其第二个参数的函数：

```
const :: a -> b -> a
const a b = a
```

而 `apply` 是非柯里化的函数应用：

```
apply :: (a -> b, a) -> b
apply (f, a) = f a
```

反过来，我们有：

```
instance Applicative f => Monoidal f where
unit = pure ()
as >*< bs = (,) <$> as <*> bs
```

在后者中，我们使用了对构造器 `(,)` 作为一个双参数函数。

Monad 和 Applicative

由于在笛卡尔闭范畴中，每个 monad[1]都是松弛单胚的，因此它自动是 applicative 的。我们可以直接通过实现 `ap` 来显示这一点，后者具有与 splat 运算符相同的类型签名：

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap fs as = do
f <- fs
a <- as
return (f a)
```

这一联系在 Haskell 的 `Monad` 定义中通过将 `Applicative` 作为其超类来表示：

```
class Applicative m => Monad m where
(>>=)       :: forall a b. m a -> (a -> m b) -> m b
return      :: a -> m a
return      = pure
```

注意到 `return` 作为 `pure` 的默认实现。

反过来并不成立：并非每个 `Applicative` 都是一个 `Monad`。标准的反例是列表函子的 `Applicative` 实例，它使用拉链 (zipping)：

```
instance Applicative [] where
pure = repeat
fs <*> as = zipWith apply fs as
```

当然，列表函子也是一个 monad，所以基于此还有另一个 `Applicative` 实例。它的 splat 运算符将每个函数应用于每个参数。

在编程中，monad 比 applicative 更强大。这是因为 monadic 代码让你可以检查一个 monadic 值的内容并根据它进行分支。即使对于 `IO` monad 也是如此，后者否则并不提供提取值的方法。在这个例子中，我们根据一个 `IO` 对象的内容进行分支：

```
main :: IO ()
main = do
s <- getLine
if s == "yes"
then putStrLn "Thank you!"
else putStrLn "Next time."
```

当然，值的检查被推迟到 `IO` 的运行时解释器获取这段代码时才进行。

使用 splat 运算符的 Applicative 组合不允许计算的一部分检查另一部分的结果。这种限制可以转化为优势。没有依赖关系的情况下，可以并行运行计算。Haskell 的并行库

[1]正确的咒语是“每个丰 monad (enriched monad)”

广泛使用了应用式编程。

另一方面，monads 允许我们使用非常方便的 `do` 语法，这比应用语法更具可读性。幸运的是，有一个语言扩展 `ApplicativeDo`，它指示编译器在解释 `do` 块时选择性地使用应用结构，前提是没有依赖关系。

Exercise 14.9.3. 验证列表函子的 *zip* 实例的 `Applicative` 定律。

Chapter 15

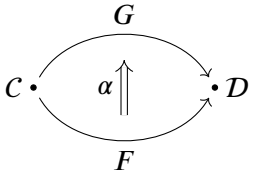
Monads and Adjunctions(单子与伴随)

15.1 String Diagrams(字符串图)

一条线将一个平面划分开。我们可以将其视为将平面分开，也可以将其视为连接平面的两半。

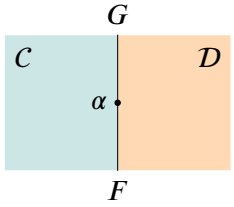
一个点将一条线划分开。我们可以将其视为分离两条半线，也可以将其视为将它们连接在一起。

这是一个图，其中两个范畴被表示为点，两个函子作为箭头，自然变换则作为双箭头。



但是，同样的概念可以通过将范畴绘制为平面的区域，函子作为区域之间的线，自然变换作为连接线段的点来表示。

这个概念是，函子总是在一对范畴之间运行，因此可以将其绘制为它们之间的边界。自然变换总是在一对函子之间运行，因此可以将其绘制为连接两段线段的点。



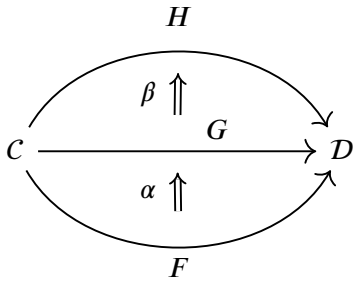
这是一个字符串图（*string diagram*）的示例。你从下到上、从左到右读取这样的图（想象一下 (x, y) 坐标系）。

这个图的底部显示了从 $\mathcal{C}$ 到 $\mathcal{D}$ 的函子 F。图的顶部显示了在同一对范畴之间的函子 G。转变发生在中间，自然变换 α 将 F 映射到 G。

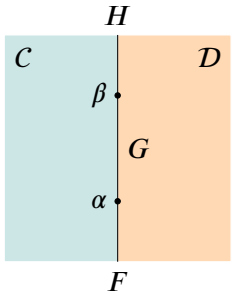
在 Haskell 中，这个图表示为两个自函子之间的多态函数：

```
alpha :: forall x. F x -> G x
```

到目前为止，使用这种新的视觉表示似乎并没有带来太多好处。但让我们将其应用于更有趣的东西：自然变换的垂直组合：



相应的字符串图显示了两个范畴以及它们之间由两个自然变换连接的三个函子。



如你所见，你可以通过从下到上扫描字符串图来重建原始图。

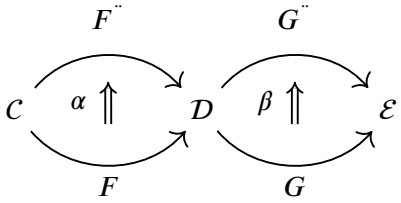
在 Haskell 中，我们将处理三个自函子，以及 `beta` 在 `alpha` 之后的垂直组合：

```
alpha :: forall x. F x -> G x
beta  :: forall x. G x -> H x
```

使用常规函数组合来实现：

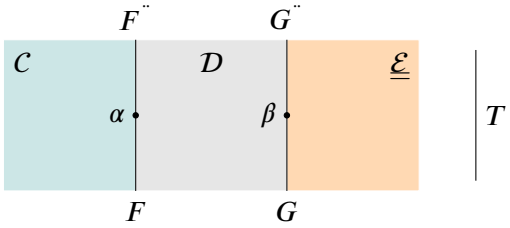
```
beta_alpha :: forall x. F x -> H x
beta_alpha = beta . alpha
```

让我们继续研究自然变换的水平组合：



这次我们有三个范畴，因此我们将有三个区域。

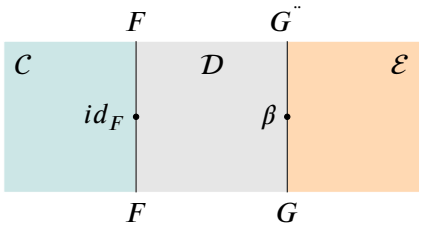
字符串图的底部对应于函子 $G \circ F$（按此顺序）的组合。顶部对应于 $G'' \circ F''$。一个自然变换 α 连接 F 到 F''；另一个自然变换 β 连接 G 到 G''。



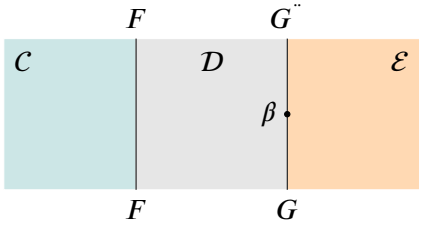
这个新的系统中，平行的垂直线对应于函子的组合。

你可以认为自然变换的水平组合发生在图的中间的想象水平线上。但是，如果有人在绘制图时不够仔细，并且其中一个点比另一个点略高呢？事实证明，由于交换律，点的确切位置无关紧要。

但首先，让我们说明 whiskering：这是一个水平组合，其中一个自然变换是恒等的。我们可以这样绘制它：



但是，实际上，恒等可以插入到垂直线的任何位置，因此我们甚至不需要绘制它。以下图表示 $\beta \circ F$ 的 whiskering。



在 Haskell 中，当 `beta` 是一个多态函数时：

```
beta :: forall x. G x -> G' x
```

我们将这个图读作：

```
beta_f :: forall x. G (F x) -> G' (F x)
beta_f = beta
```

理解到类型检查器将实例化多态函数 beta 为正确的类型。

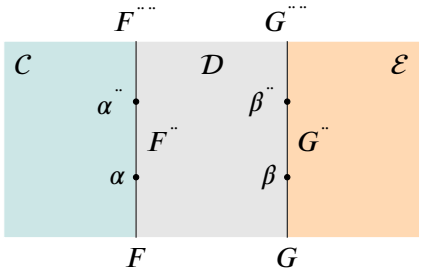
类似地，你可以轻松想象 $G \circ \alpha$ 的图及其 Haskell 实现：

```
g_alpha :: forall x. G (F x) -> G (F' x)
g_alpha = fmap alpha
```

其中：

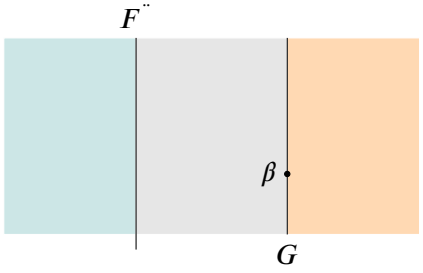
```
alpha :: forall x. F x -> F' x
```

以下是对应于交换律的字符串图：



这个图故意模棱两可。我们应该先进行自然变换的垂直组合，然后再进行水平组合吗？还是应该水平组合 $\beta \circ \alpha$ 和 $\beta'' \circ \alpha''$，然后再垂直组合结果？交换律告诉我们，这无关紧要：结果是相同的。

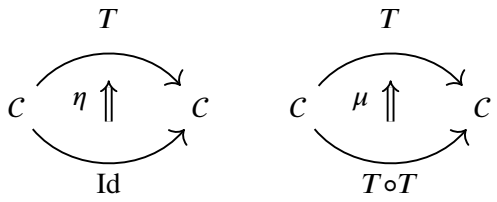
现在尝试在此图中将一对自然变换替换为恒等变换。如果你替换 α'' 和 β''，你会得到 $\beta \circ \alpha$ 的水平组合。如果你用恒等自然变换替换 α'' 和 β，并将 β'' 重命名为 β，你会得到一个图，其中 α 相对于 β 向下移动，依此类推。



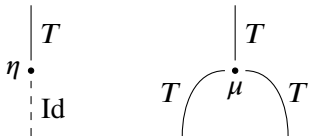
交换律告诉我们，所有这些图都是相等的。我们可以像珠子穿在线上一样自由滑动自然变换。

String diagrams for the monad(单子的字符串图)

一个单子定义为一个配备了两个自然变换的自函子，如下图所示：



由于我们只处理一个范畴，当将这些图转换为字符串图时，我们可以省略范畴的命名（和着色），只绘制字符串。

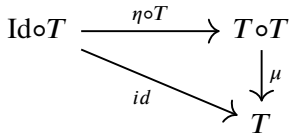


在第一个图中，通常会跳过表示恒等函子的虚线。η 点可以自由地将 T 线注入到图中。两个 T 线可以通过 μ 点连接。

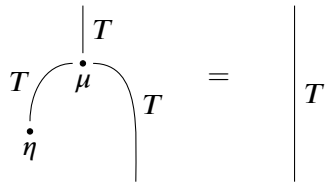
字符串图在表达单子律时特别有用。例如，我们有左恒等律：

$$\mu \circ (\eta \circ T) = id$$

这可以用一个交换图来表示：

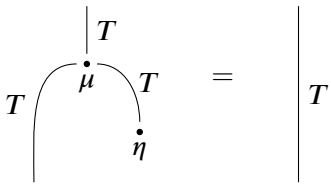


相应的字符串图表示通过这个图的两条路径的相等性：

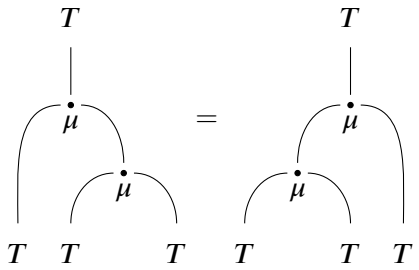


你可以将此等式视为通过拉扯上下字符串，使 η 附属物缩回到直线中。

同样有一个对称的右恒等律：



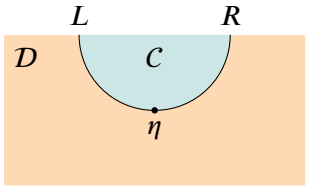
最后，这是关联律的字符串图表示：



String diagrams for the adjunction(伴随函子的字符串图)

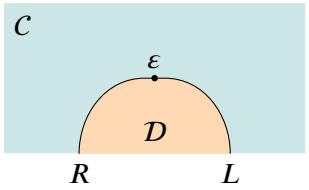
正如我们之前讨论的，伴随函子是两个函子之间的关系，$L\colon \mathcal{D} \to \mathcal{C}$ 和 $R\colon \mathcal{C} \to \mathcal{D}$。它可以通过一对自然变换来定义，伴随单元 η 和伴随余单元 ε，它们满足三角恒等式。

伴随单元可以通过一个“杯”形图来说明：



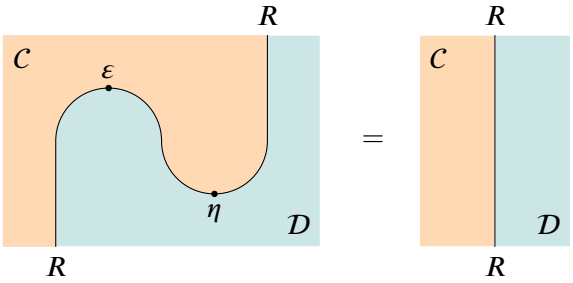
图底部的恒等函子从图中省略了。η 点将其下方的恒等函子转变为其上方的组合 $R \circ L$。

类似地，伴随余单元可以通过一个“帽”形字符串图来可视化，顶部的隐含恒等函子：



三角恒等式可以使用字符串图轻松表达。它们也很直观，因为你可以想象从两边拉动字符串以拉直曲线。

例如，这是第一个三角恒等式，有时称为之字形恒等式：



从下到上读取左图，会得到一系列映射：

$$Id_D \circ R \xrightarrow{\eta \circ R} R \circ L \circ R \xrightarrow{R \circ \varepsilon} R \circ Id_C$$

这必须等于右侧，它可以被解释为 R 上的（隐形）恒等自然变换。

在 R 是自函子的情况下，我们可以将第一个图直接翻译为 Haskell。R 对伴随单元 η 的 whiskering 导致多态函数 unit 在 R x 上实例化。伴随余单元 ε 的 whiskering 导致 counit 由函子 R 提升。垂直组合翻译为函数组合：

```
triangle :: forall x. R x -> R x
triangle = fmap counit . unit
```

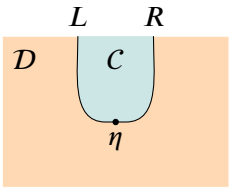
Exercise 15.1.1. *Draw the string diagrams for the second triangle identity and translate them to Haskell.*

15.2 Monads from Adjunctions（从伴随构造单子）

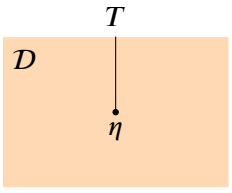
你可能已经注意到，η 这个符号既用于伴随单元，也用于单子单元。这不是巧合。

乍一看，这似乎是将苹果与橙子进行比较：伴随是由两个类别之间的两个函子定义的，而单子是由在一个类别上操作的一个自函子定义的。然而，两个反向函子的组合是一个自函子，并且伴随的单元将恒等自函子映射到自函子 $R \circ L$。

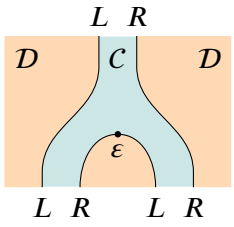
对比这个图：



与定义单子单元的这个图：



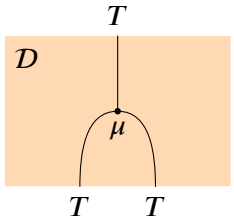
事实证明，对于任何伴随 $L \dashv R$，自函子 $T = R \circ L$ 是一个单子，其乘法 μ 由以下图定义：



从下到上读取这个图，我们得到以下变换（想象将其水平切片在点处）：

$$R \circ L \circ R \circ L \xrightarrow{R \circ \varepsilon \circ L} R \circ L$$

与单子的 μ 定义比较：



我们得到了单子 $R \circ L$ 的 μ 定义，其为 ε 的双 whiskering：

$$\mu = R \circ \varepsilon \circ L$$

以 ε 为基础定义 μ 的字符串图可以始终转换为 Haskell 代码。单子的乘法，或称为`join`，变为：

```
join :: forall x. T (T x) -> T x
join = fmap counit
```

其中`fmap`对应于由自函子`T`（定义为组合 $R \circ L$）提升的函数。注意在这种情况下，$\mathcal{D}$ 是 Haskell 类型和函数的范畴，但 $\mathcal{C}$ 可以是一个外部范畴。

为了完成这个图，我们可以使用字符串图，通过三角等式推导出单子定律。诀窍是将单子定律中的所有字符串替换为成对的平行字符串，然后根据规则重新排列它们。

总结来说，每个具有单元 η 和余单元 ε 的伴随 $L \dashv R$ 定义了一个单子 $(R \circ L, \eta, R \circ \varepsilon \circ L)$。

稍后我们将看到，反之，另一个组合 $L \circ R$ 定义了一个余单子。

Exercise 15.2.1. *Draw string diagrams to illustrate monadic laws (unit and associativity) for the monad derived from an adjunction.* （请绘制字符串图，来说明从伴随推导出的单子的单子定律（单位元和结合律））

15.3 Examples of Monads from Adjunctions（从伴随推导出的单子示例）

我们将通过几个示例来展示从伴随生成的一些在编程中使用的单子。在讨论单子转换器时，我们将进一步展开这些示例。

大多数示例涉及将函子离开 Haskell 类型和函数的范畴，尽管生成单子的循环最终成为自函子。这就是为什么在 Haskell 中经常无法表达这样的伴随的原因。

为了进一步复杂化，存在很多与显式命名数据构造函数相关的簿记工作，这是类型推断所必需的。这有时可能会掩盖底层公式的简单性。

Free monoid and the list monad（自由幺半群与列表单子）

列表单子由我们之前见过的自由幺半群伴随生成。该伴随的单元 $\eta_X : X \to U(FX)$ 将集合 X 的元素注入为自由幺半群 FX 的生成元，之后 U 提取底层集合。

在 Haskell 中，我们将自由幺半群表示为列表类型，其生成元是单元素列表。η_X 将 X 的元素映射为这些单元素列表：

```
return x = [x]
```

为了实现余单元 $\varepsilon_M : F(UM) \to M$，我们取一个幺半群 M，忘记其乘法，并使用其元素集作为新的自由幺半群的生成元。余单元在 M 上的分量是从自由幺半群返回到 M 的幺半群同态，或者在 Haskell 中是`[m]->m`。事实证明，这种幺半群同态是一个特殊的归纳同态。

首先，回顾一下 Haskell 实现的一般列表归纳同态：

```
foldMap :: Monoid m => (a -> m) -> ([a] -> m)
foldMap f = foldr mappend mempty . fmap f
```

在这里，我们将`(a -> m)`解释为从`a`到幺半群`m`底层集合的常规函数。结果被解释为从`a`生成的自由幺半群（即`a`的列表）到`m`的*幺半群同态*。这只是伴随的一个方向：

$$\mathrm{Set}(a, Um) \ddot{\mathrm{o}} \mathrm{Mon}(Fa, m)$$

为了获得作为幺半群同态的余单元`[m]->m`，我们将`foldMap`应用于恒等式。结果是`(foldMap id)`，或者在`foldr`的术语中：

```
epsilon = foldr mappend mempty
```

这是一个幺半群同态，因为它将空列表映射到幺半群单元，并将连接映射到幺半群乘积。

单子乘法或`join`由余单元的 whiskering 给出：

$$\mu = U \circ \varepsilon \circ F$$

你可以很容易地说服自己，左 whiskering 在这里并没有做太多，因为它只是通过遗忘函子提升了一个幺半群同态（它保留了函数而忘记了其保持结构的特殊属性）。

右边的 F 的 whiskering 更有趣。这意味着分量 μ_X 对应于 ε 在 FX 上的分量，这是从集合 X 生成的自由幺半群。这个自由幺半群由以下定义：

```
mempty = []
mappend = (++)
```

这给出了join的定义：

```
join = foldr (++) []
```

正如预期的那样，这与concat相同：在列表单子中，乘法是连接。

The currying adjunction and the state monad（柯里化伴随与状态单子）

状态单子由我们用来定义指数对象的柯里化伴随生成。左函子由与某个固定对象 s 的乘积定义：

$$L_s a = a \quad s$$

例如，我们可以将其实现为 Haskell 类型：

```
newtype L s a = L (a, s)
```

右函子是指数化，以同一个对象 s 为参数化：

$$R_s c = c^s$$

在 Haskell 中，它是一个薄封装的函数类型：

```
newtype R s c = R (s -> c)
```

单子由这两个函子的组合给出。在对象上：

$$(R_s \circ L_s) a = (a \quad s)^s$$

在 Haskell 中我们会写成：

```
newtype St s a = St (R s (L s a))
```

如果你展开这个定义，很容易在其中识别出State函子：

```
newtype State s a = State (s -> (a, s))
```

伴随 $L_s \dashv R_s$ 的单元是一个映射：

$$\eta_a \colon a \to (a \quad s)^s$$

它可以在 Haskell 中实现为：

```
unit :: a -> R s (L s a)
unit a = R (\s -> L (a, s))
```

你可能已经在其中识别出状态单子的return的一个薄薄版本：

```
return :: a -> State s a
return a = State (\s -> (a, s))
```

这里是伴随在 c 上的余单元的分量：

$$\varepsilon_c : c^s \quad s \to c$$

它可以在 Haskell 中实现为：

```
counit :: L s (R s a) -> a
counit (L ((R f), s))= f s
```

在剥离了数据构造器后，它等价于apply，或runState的非柯里化版本。

单子乘法 μ 由 ε 的 whiskering 从两侧给出：

$$\mu = R_s \circ \varepsilon \circ L_s$$

以下是其转换为 Haskell 的版本：

```
mu :: R s (L s (R s (L s a))) -> R s (L s a)
mu = fmap counit
```

右边的 whiskering 除了选择自然变换的分量之外什么也不做。这是 Haskell 的类型推断引擎自动完成的。左边的 whiskering 是通过提升自然变换的分量完成的。同样，类型推断会选择正确的fmap实现——这里它等同于前置组合。

与join的实现进行比较：

```
join :: State s (State s a) -> State s a
join mma = State (fmap (uncurry runState) (runState mma))
```

注意runState的双重使用：

```
runState :: State s a -> s -> (a, s)
runState (State h) s = h s
```

当它被非柯里化时，它的类型签名变为：

```
uncurry runState :: (State s a, s) -> (a, s)
```

这与counit的类型签名相同。

当部分应用时，runState只是剥离了数据构造器，暴露了底层函数类型：

```
        runState st :: s -> (a, s)
```

M-sets and the writer monad（M-集与 writer 单子）

writer 单子：

```
        newtype Writer m a = Writer (a, m)
```

由一个幺半群m参数化。该幺半群用于累积日志条目。我们将使用的伴随涉及该幺半群的 M-集范畴。

一个M-集是一个集合 S，我们在其上定义幺半群 M 的作用。这样的作用是一个映射：

$$a\colon M \quad S \to S$$

我们经常使用作用的柯里化版本，幺半群元素位于下标位置。因此 a_m 成为 $S \to S$ 的函数。

这个映射必须满足一些约束。幺半群单位 1 的作用不能改变集合，因此它必须是恒等函数：

$$a_1 = id_S$$

并且两个连续的作用必须组合成其幺半群乘积的作用：

$$a_{m_1} \circ a_{m_2} = a_{m_1 \cdot m_2}$$

这种乘法顺序的选择定义了所谓的*左作用*。（右作用将右侧的两个幺半群元素交换。）

M-集形成一个范畴 MSet。对象是对 $(S, a\colon M \quad S \to S)$，箭头是*协变映射*，即保持作用的集合之间的函数。

函数 $f\colon S \to R$ 是从 (S, a) 到 (R, b) 的*协变映射*，如果以下图对每个 m Ë M 都交换：

$$\begin{array}{ccc} S & \xrightarrow{f} & R \\ \downarrow{\scriptstyle a_m} & & \downarrow{\scriptstyle b_m} \\ S & \xrightarrow{f} & R \end{array}$$

换句话说，我们可以先进行作用 a_m，然后映射集合；或者先映射集合，然后进行相应的作用 b_m，效果是相同的。

从 MSet 到 Set 有一个遗忘函子 U，它将集合 S 分配给对 (S, a)，从而遗忘了作用。

与之对应的是一个自由函子 F。它在集合 S 上的作用产生一个 M-集。它是一个 S 和 M 的笛卡尔积的集合，其中 M 被视为元素的集合（换句话说，遗忘函子作用在幺半群上的结果）。这个 M-集的元素是对 $(x$ Ë S, m Ë $M)$，自由作用由以下定义：

$$\phi_n\colon (x, m) \mapsto (x, n \cdot m)$$

保持元素 x 不变，仅乘以 m 分量。

为了表明 F 是 U 的左伴随，我们必须构造以下自然同构：

$$\mathrm{MSet}(FS, Q) \ddot{\mathrm{o}} \mathrm{Set}(S, UQ)$$

对于任何集合 S 和任何 M-集 Q。如果我们将 Q 表示为对 (R, b)，则伴随右侧的元素是普通函数 $u\colon S \to R$。我们可以使用此函数在左侧构造一个协变映射。

这里的技巧是注意到这样的协变映射 $f\colon FS \to Q$ 由其在形式 $(x, 1)$ Ë FS 的元素上的作用完全确定，其中 1 是幺半群单位。

事实上，从协变条件可以得出：

$$\begin{array}{ccc} (x,1) & \xrightarrow{\ f\ } & r \\ \Big\downarrow{\scriptstyle \phi_m} & & \Big\downarrow{\scriptstyle b_m} \\ (x, m\cdot 1) & \xrightarrow{\ f\ } & r \end{array}$$

或者：

$$f(\phi_m(x,1)) = f(x,m) = b_m(f(x,1))$$

因此，每个函数 $u\colon S \to R$ 唯一地定义了从 FS 到 Q 的协变映射 $f\colon FS \to Q$，给出：

$$f(x,m) = b_m(ux)$$

这个伴随的单元 $\eta_S\colon S \to U(FS)$ 将元素 x 映射为对 $(x,1)$。将此与 writer 单子的`return`定义进行比较：

```
return a = Writer (a, mempty)
```

余单元由一个协变映射给出：

$$\varepsilon_Q\colon F(UQ) \to Q$$

左侧是通过取 Q 的底层集合并将其与 M 的底层集合取积构造的 M-集。Q 的原始作用被遗忘并由自由作用取代。余单元的显然选择是：

$$\varepsilon_Q\colon (x,m) \mapsto a_m x$$

其中 x 是（底层集合的）Q 的一个元素，而 a 是在 Q 中定义的作用。

单子乘法 μ 由余单元的 whiskering 给出。

$$\mu = U \circ \varepsilon \circ F$$

这意味着在 ε_Q 的定义中用自由作用取代 Q 中的一个自由 M-集。换句话说，我们用 (x,m) 取代 x，用 ϕ_n 取代 a_n。（与 U 的 whiskering 不会改变任何东西。）

$$\mu_S\colon ((x,m),n) \mapsto \phi_n(x,m) = (x, n\cdot m)$$

将此与 writer 单子的`join`定义进行比较：

```
join :: Monoid m => Writer m (Writer m a) -> Writer m a
join (Writer ( Writer (x, m), n)) = Writer (x, mappend n m)
```

Pointed objects and the Maybe monad（有点对象与Maybe单子）

有点对象是具有指定元素的对象。由于选择一个元素是使用来自终端对象的箭头完成的，因此有点对象的范畴是使用对 $(a, p\colon 1 \to a)$ 定义的，其中 a 是 $\mathcal{C}$ 中的对象。

这些对之间的态射是保持点的 $\mathcal{C}$ 中的箭头。因此，从 $(a, p\colon 1 \to a)$ 到 $(b, q\colon 1 \to b)$ 的态射是箭头 $f\colon a \to b$，使得 $q = f \circ p$。这个范畴也被称为余片范畴，写作 $1/\mathcal{C}$。

显然，有一个遗忘函子 $U\colon 1/\mathcal{C} \to \mathcal{C}$，它遗忘了点。

它的左伴随是一个自由函子 F，将对象 a 映射到对 $(1 + a, \text{Left})$。换句话说，F 使用余积自由地为对象添加一个点。

Either单子类似地通过将 1 替换为固定对象 e 来构造。

Exercise 15.3.1. *Show that $U \circ F$ is the Maybe monad.* （证明 $U \circ F$ 是Maybe单子）

The continuation monad（续延单子）

续延单子是根据集合范畴中的一对反变函子定义的。我们不需要修改伴随的定义来处理反变函子。只需要选择其中一个端点的对偶范畴即可。

我们将左函子定义为：

$$L_Z\colon \mathrm{Set}^{op} \to \mathrm{Set}$$

它将一个集合 X 映射到 Set 中的同态集合：

$$L_Z X = \mathrm{Set}(X, Z)$$

这个函子由另一个集合 Z 参数化。右函子由基本相同的公式定义：

$$R_Z\colon \mathrm{Set} \to \mathrm{Set}^{op}$$

$$R_Z X = \mathrm{Set}^{\mathrm{op}}(Z, X) = \mathrm{Set}(X, Z)$$

组合 $R \circ L$ 可以在 Haskell 中写成`((x -> r) -> r)`，这与定义续延单子的（协变）自函子相同。

15.4 Monad Transformers（Monad 变换器）

假设你想要组合多个效果，例如，将状态（state）和可能的失败（failure）结合在一起。一个选择是从头定义你自己的 monad。你可以定义一个函子（functor）：

```
newtype MaybeState s a = MS (s -> Maybe (a, s))
deriving Functor
```

并配合使用一个函数来提取结果（或者报告失败）：

```
runMaybeState :: MaybeState s a -> s -> Maybe (a, s)
runMaybeState (MS h) s = h s
```

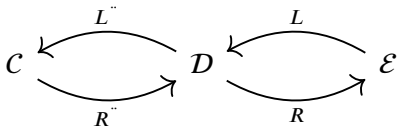
接着，你可以为它定义一个 Monad 实例：

```
instance Monad (MaybeState s) where
return a = MS (\s -> Just (a, s))
ms >>= k = MS (\s -> case runMaybeState ms s of
Nothing -> Nothing
Just (a, s') -> runMaybeState (k a) s')
```

并且，如果你足够勤奋的话，检查它是否满足 monad 定律。

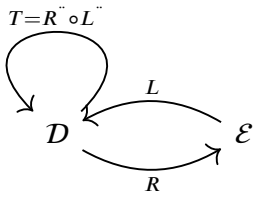
没有一种通用的方法可以组合 monad。从这个意义上说，monad 不是可组合的（composable）。然而，我们知道伴随函子（adjunctions）是可组合的。我们还看到了如何通过伴随函子得到 monad，并且我们很快就会看到，每个 monad 都可以通过这种方式获得。所以，如果我们可以匹配伴随函子，那么它们生成的 monad 将自动组合。

考虑两个可组合的伴随函子：



在这张图中有三个 monad。存在“内在的”monad $R\tilde{}\circ L\tilde{}$ 和“外在的”monad $R\circ L$ 以及复合的 $R\circ R\tilde{}\circ L\tilde{}\circ L$。

如果我们将内在的 monad 称为 $T = R\tilde{}\circ L\tilde{}$，那么 $R\circ T\circ L$ 就是复合 monad，称为“monad 变换器”（monad transformer），因为它将 monad T 转换为一个新的 monad。



在我们的例子中，我们可以将 `Maybe` 视为内在的 monad：

$$Ta = 1 + a$$

它通过生成状态 monad 的外在伴随函子 $L_s \dashv R_s$ 进行转换：

$$L_s a = a \quad s$$

$$R_s c = c^s$$

结果是：

$$(R_s \circ T \circ L_s) a = (1 + a \quad s)^s$$

或者，在 Haskell 中：

```
        s -> Maybe (a, s)
```

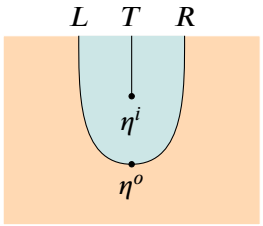
这与我们定义的 `MaybeState` monad 一致。

通常情况下，内在 monad T 是由它的单位元（unit）η^i 和乘法（multiplication）μ^i 定义的（上标 i 代表“内在的”）。外在伴随函子由它的单位元 η^o 和余单位元（counit）ε^o 定义。

复合 monad 的单位元是一个自然变换：

$$\eta \colon Id \to R \circ T \circ L$$

通过以下字符串图（string diagram）表示：



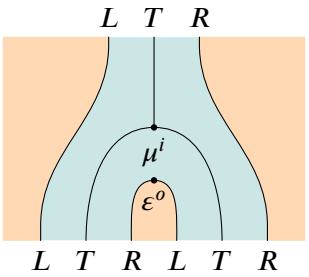
这是经过唇形变换的内在单位 $R \circ \eta^i \circ L$ 和外在单位 η^o 的垂直组合。在具体构成上：

$$\eta_a = R(\eta^i_{La}) \circ \eta^o_a$$

复合 monad 的乘法是一个自然变换：

$$\mu \colon R \circ T \circ L \circ R \circ T \circ L \to R \circ T \circ L$$

通过以下字符串图表示：



这是经过多重唇形变换的外在余单位元：

$$R \circ T \circ \varepsilon^o \circ T \circ L$$

然后是经过唇形变换的内在乘法 $R \circ \mu^i \circ L$。在具体构成上：

$$\mu_c = R(\mu^i_{Lc}) \circ (R \circ T)(\varepsilon^o_{(T \circ L)c})$$

State Monad Transformer（状态 monad 变换器）

让我们逐步解析这些方程，以应用于状态 monad 变换器。状态 monad 由柯里化伴随（currying adjunction）生成。左函子 L_s 是乘积函子 `(a, s)`，而右函子 R_s 是指数函子（exponential），又称为读者函子（reader functor）`(s -> a)`。

如前所述，外在余单位元 ε^o_a 是函数应用：

```
counit :: (s -> a, s) -> a
counit (f, x) = f x
```

而单位元 η^o_a 是柯里化对偶构造函数：

```
unit :: a -> s -> (a, s)
unit x = \s -> (x, s)
```

我们将保持内在 monad (T, η^i, μ^i) 的任意性。在 Haskell 中，我们将这个三元组称为 `m`, `return`, 和 `join`。

通过将状态 monad 变换器应用于 monad T，我们得到的复合 monad 是 $R \circ T \circ L$ 或者在 Haskell 中：

```
newtype StateT s m a = StateT (s -> m (a, s))
```

```
runStateT :: StateT s m a -> s -> m (a, s)
runStateT (StateT h) s = h s
```

monad 变换器的单位元是 η^o 和 $R \circ \eta^i \circ L$ 的垂直组合。在具体构成上：

$$\eta_a = R(\eta^i_{La}) \circ \eta^o_a$$

这个公式中有很多细节，让我们逐步分析。从右边开始：我们有伴随的单位元 a 分量，这是从 a 到 $R(La)$ 的箭头。在 Haskell 中，这是函数 `unit`。

```
unit :: a -> s -> (a, s)
```

让我们在某个 `x :: a` 上计算这个函数。结果是另一个函数 `s -> (a, s)`。我们将这个函数作为参数传递给 $R(\eta^i_{La})$。

η^i_{La} 是内在 monad 的 `return` 在 La 上的分量。这里，La 是类型 `(a, s)`。所以我们将多态函数 `return :: a -> m a` 实例化为函数 `(a, s) -> m (a, s)`。（类型推断器会为我们自动执行这个操作。）

接着，我们用 R 提升这个 `return` 的分量。这里，R 是指数 $(*)^s$，所以它通过后组合（post-composition）来提升函数。它将 `return` 后组合到任何传递给它的函数中。在我们

的例子中，这是由 unit 生成的函数。注意，类型匹配：我们正在将 (a, s) -> m (a, s) 后组合到 s -> (a, s) 之后。

我们可以将这个组合的结果写为：

```
return x = StateT (return . \s -> (x, s))
```

或者，内联函数组合：

```
return x = StateT (\s -> return (x, s))
```

我们插入了数据构造函数 StateT 以使类型检查器满意。这是复合 monad 的 return，是以内在 monad 的 return 表示的。

同样的推理可以应用于某个 a 处复合 μ 的分量公式：

$$\mu_a = R(\mu^i_{La}) \circ (R \circ T)(\varepsilon^o_{(T \circ L)a})$$

内在 μ^i 是 monad m 的 join。应用 R 将其转化为后组合。

外在 ε^o 是取自 $T(La)$ 或 m (a, s) 的函数应用。它是一个类型的函数：

```
(s -> m (a, s), s) -> m (a, s)
```

插入适当的数据构造函数后，可以将其写为 uncurry runStateT：

```
uncurry runStateT :: (StateT s m a, s) -> m (a, s)
```

应用 $(R \circ T)$ 使用函子 R 和 T 的组合来提升 ε 的分量。前者作为后组合实现，后者是 monad m 的 fmap。

将这些组合在一起，我们得到了状态 monad 变换器的 join 的无点公式（point-free formula）：

```
join :: StateT s m (StateT s m a) -> StateT s m a
join mma = StateT (join . fmap (uncurry runStateT) . runStateT mma)
```

这里，部分应用的 (runStateT mma) 从参数 mma 中剥离了数据构造函数：

```
runStateT mma :: s -> m (a, x)
```

我们之前的 MaybeState 示例现在可以使用 monad 变换器重写为：

```
type MaybeState s a = StateT s Maybe a
```

可以通过将 StateT monad 变换器应用于身份函子（identity functor）来恢复普通的 State monad，身份函子在库中定义了一个 Monad 实例（注意，在这个定义中最后的类型变量 a 被省略了）：

```
type State s = StateT s Identity
```

其他 monad 变换器遵循相同的模式。它们在 monad 变换器库（Monad Transformer Library，MTL）中定义。

15.5 Monad Algebras（Monad 代数）

每个伴随函子生成一个 monad，到目前为止，我们已经能够为我们感兴趣的所有 monad 定义伴随函子。但是，是否每个 monad 都由伴随函子生成？答案是肯定的，并且通常每个 monad 都有许多伴随函子——事实上，每个 monad 都有一整个类别的伴随函子。

为 monad 寻找一个伴随函子类似于因式分解。我们希望将一个函子表示为两个其他函子的组合，即 $T = R \circ L$。问题的复杂性在于，这种因式分解还需要找到适当的中间类别。我们将通过研究 monad 的代数来找到这样的类别。

monad 由一个自函子（endofunctor）定义，我们知道可以为自函子定义代数。数学家通常将 monad 视为生成表达式的工具，而将代数视为评估这些表达式的工具。然而，monad 生成的表达式对这些代数施加了一些兼容性条件。

例如，你可能会注意到，monad 的单位元 $\eta_a \colon a \to Ta$ 具有看起来像是代数结构映射 $\alpha \colon Ta \to a$ 的逆的类型签名。当然，η 是为每种类型定义的自然变换，而代数具有固定的载体类型。然而，我们可以合理地期望，η 可能会撤销 α 的作用。

考虑前面表达式 monad `Ex` 的示例。这个 monad 的一个代数是选择载体类型，比如 `Char`，以及一个箭头：

```
alg :: Ex Char -> Char
```

由于 `Ex` 是一个 monad，它定义了一个单位元，或者说 `return`，它是一个多态函数，可以用来从值生成简单的表达式。`Ex` 的单位元是：

```
return x = Var x
```

我们可以为任意类型实例化这个单位元，特别是为我们代数的载体类型。我们合理地要求，评估 `Var c`（其中 `c` 是一个字符）应该返回相同的 `c`。换句话说，我们希望：

```
alg . return = id
```

这个条件将立即消除很多代数，例如：

```
alg (Var c) = 'a' -- 与 monad Ex 不兼容
```

我们想要施加的第二个条件是，与 monad 兼容的代数应该尊重替换。monad 允许我们使用 `join` 扁平化嵌套表达式。代数允许我们评估这些表达式。

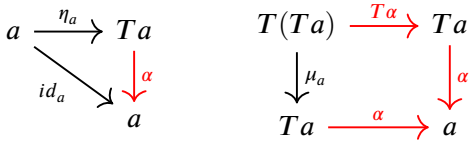
有两种方式可以做到这一点：我们可以将代数应用于一个扁平化的表达式，或者我们可以先应用它到内部表达式（使用 `fmap`），然后再评估结果表达式。

```
alg (join mma) = alg (fmap alg mma)
```

其中 `mma` 是嵌套类型 `Ex (Ex Char)`。

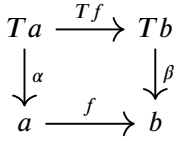
在范畴理论中，这两个条件定义了 monad 代数（monad algebra）。

我们说 $(a, \alpha : Ta \to a)$ 是 monad (T, μ, η) 的一个 *monad* 代数，如果以下图表交换：



这些规律有时被称为 monad 代数的单位定律和乘法定律。

由于 monad 代数只是代数的特例，它们构成了代数的子范畴。回忆一下，代数态射（algebra morphisms）是满足以下条件的箭头：



根据这个定义，我们可以将第二个 monad 代数图表重新解释为断言 monad 代数的结构映射 α（底部箭头）也是一个代数态射，从 (Ta, μ_a) 到 (a, α)。这将在接下来的内容中发挥重要作用。

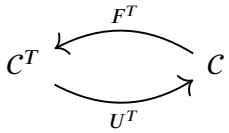
Eilenberg-Moore 范畴（Eilenberg-Moore Category）

给定 monad T 在 $\mathcal{C}$ 上的 monad 代数的范畴称为 Eilenberg-Moore 范畴，记作 $\mathcal{C}^T$。事实证明，它是一个很好的选择，可以让我们将 monad T 因式分解为一对伴随函子的组合。

过程如下：我们定义一对函子，证明它们形成了一个伴随关系，然后证明由这个伴随关系生成的 monad 就是原来的 monad。

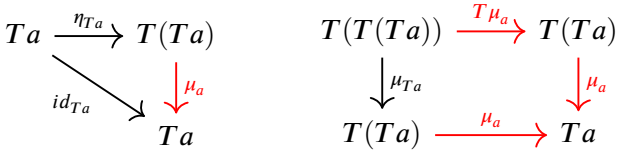
首先，有一个显而易见的遗忘函子（forgetful functor），我们称之为 U^T，从 $\mathcal{C}^T$ 到 $\mathcal{C}$。它将一个代数 (a, α) 映射到它的载体 a，并将代数态射视为载体之间的常规态射。

更有趣的是，有一个自由函子（free functor）F^T，它是 U^T 的左伴随函子。



在对象上，F^T 将 $\mathcal{C}$ 的一个对象 a 映射到一个 *monad* 代数，即 $\mathcal{C}^T$ 中的一个对象。对于这个代数的载体，我们选择 Ta 而不是 a。对于结构映射，即映射 $T(Ta) \to Ta$，我们选择 monad 乘法的分量 $\mu_a : T(Ta) \to Ta$。

很容易检查这个代数 (Ta, μ_a) 确实是一个 monad 代数——所需的交换条件遵循 monad 定律。实际上，将代数 (Ta, μ_a) 代入 monad 代数的图表中，我们得到（代数部分用红色表示）：



第一个图表只是左侧 monad 单位定律的分量。箭头 η_{Ta} 对应于 $\eta \circ T$ 的唇形变换。第二个图表是 μ 的结合律（associativity）与两个唇形变换 $\mu \circ T$ 和 $T \circ \mu$ 用分量表示。

为了证明我们有一个伴随关系，我们将定义两个自然变换来作为伴随关系的单位和余单位。

对于伴随关系的单位，我们选择 monad T 的单位元 η。它们具有相同的签名——在分量中，$\eta_a : a \to U^T(F^T a)$。

余单位是一个自然变换：

$$\varepsilon : F^T \circ U^T \to Id$$

余单位在 (a, α) 处的分量是从 a 生成的自由代数（即 (Ta, μ_a)）回到 (a, α) 的代数态射。正如我们之前所见，α 本身就是这样的态射。因此，我们可以选择 $\varepsilon_{(a,\alpha)} = \alpha$。

这些定义的 η 和 ε 的三角恒等式遵循 monad 和 monad 代数的单位定律。

对于所有伴随函子而言，复合 $U^T \circ F^T$ 是一个 monad。我们将展示这个 monad 与我们最初的 monad 相同。实际上，在对象上，复合 $U^T(F^T a)$ 首先将 a 映射到一个自由 monad 代数 (Ta, μ)，然后遗忘结构映射。净结果是将 a 映射到 Ta，这正是原来的 monad 所做的。

在箭头上，它使用 T 提升箭头 $f : a \to b$。从 (Ta, μ_a) 到 (Tb, μ_b) 的箭头 Tf 是一个代数态射的事实源自 μ 的自然性：

$$\begin{array}{ccc} T(Ta) & \xrightarrow{T(Tf)} & T(Tb) \\ \downarrow{\scriptstyle \mu_a} & & \downarrow{\scriptstyle \mu_b} \\ Ta & \xrightarrow{Tf} & Tb \end{array}$$

最后，我们必须证明 monad $U^T \circ F^T$ 的单位元和余单位与我们原始 monad 的单位元和余单位相同。

单位元是相同的，构造上就是这样。

$U^T \circ F^T$ 的 monad 乘法由伴随关系单位的唇形变换 $U^T \circ \varepsilon \circ F^T$ 给出。在分量中，这意味着在 (Ta, μ_a) 处实例化 ε，这给了我们 μ_a（U^T 对箭头的作用是平凡的）。这确实是原始 monad 的乘法。

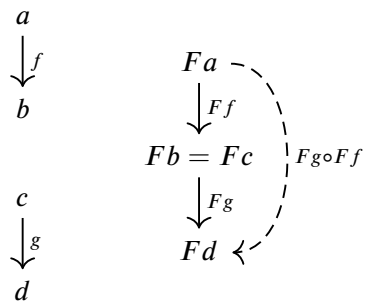
我们已经证明，对于任何 monad T，我们都可以定义 Eilenberg-Moore 范畴和一对伴随函子，这些函子可以分解这个 monad。

Kleisli 范畴（Kleisli Category）

在每个 Eilenberg-Moore 范畴中，都有一个较小的 Kleisli 范畴在努力挣脱出来。这个较小的范畴是我们在前一节中构造的自由函子的像（image）。

尽管表面上看，自由函子的像并不一定定义一个子范畴。诚然，它将恒等映射到恒等，并将组合映射到组合。然而，如果源范畴中的两个箭头不能组合，而在目标范畴中却可以组合，问题就会出现。如果第一个箭头的目标被映射到与第二个箭头的源相同的

对象，那么这种情况就可能发生。在下面的例子中，Ff 和 Fg 是可组合的，但它们的组合 $Fg \circ Ff$ 可能不在第一个范畴的像中。

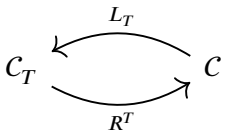


然而，自由函子 F^T 将不同的对象映射到不同的自由代数中，因此它的像确实是 $\mathcal{C}^T$ 的一个子范畴。

我们之前已经遇到过 Kleisli 范畴。有多种构造相同范畴的方法，其中最简单的一种是用 Kleisli 箭头（Kleisli arrows）描述 Kleisli 范畴。

一个 monad (T, η, μ) 的 Kleisli 范畴记作 $\mathcal{C}_T$。它的对象与 $\mathcal{C}$ 的对象相同，但 $\mathcal{C}_T$ 中从 a 到 b 的一个箭头由 $\mathcal{C}$ 中的一个箭头表示，这个箭头从 a 到 Tb。你可能认出它是我们之前定义的 Kleisli 箭头 `a -> m b`。由于 T 是一个 monad，这些 Kleisli 箭头可以使用“鱼”（fish）操作符 `<=<` 进行组合。

为了建立伴随关系：



我们定义左函子 $L_T : \mathcal{C} \to \mathcal{C}_T$ 在对象上为恒等。在箭头上，我们仍然需要定义它的作用。它应将一个常规箭头 $f : a \to b$ 映射到一个从 a 到 b 的 Kleisli 箭头。这个 Kleisli 箭头 $a \twoheadrightarrow b$ 由 $\mathcal{C}$ 中的一个箭头表示，从 a 到 Tb。这样的箭头总是存在，它是复合 $\eta_b \circ f$：

$$L_T f : a \xrightarrow{f} b \xrightarrow{\eta_b} Tb$$

右函子 $R_T : \mathcal{C}_T \to \mathcal{C}$ 在对象上定义为将 Kleisli 范畴中的 a 映射到 $\mathcal{C}$ 中的对象 Ta。给定一个 Kleisli 箭头 $a \twoheadrightarrow b$，它由一个箭头 $g : a \to Tb$ 表示，R_T 将其映射为一个箭头 $R_T a \to R_T b$，即 $\mathcal{C}$ 中的一个箭头 $Ta \to Tb$。我们将这个箭头视为复合 $\mu_b \circ Tg$：

$$Ta \xrightarrow{Tg} T(Tb) \xrightarrow{\mu_b} Tb$$

为了建立这个伴随关系，我们将展示同构集的同构性：

$$\mathcal{C}_T(L_T a, b) \ddot{\circ} \mathcal{C}(a, R_T b)$$

左侧的一个元素是一个 Kleisli 箭头 $a \twoheadrightarrow b$，由 $f : a \to Tb$ 表示。我们可以在右侧找到相同的箭头，因为 $R_T b$ 是 Tb。因此，同构性存在于 $\mathcal{C}^T$ 中的 Kleisli 箭头与 $\mathcal{C}$ 中表示它们的箭头之间。

复合 $R_T \circ L_T$ 等于 T，实际上，可以证明这个伴随关系生成了原来的 monad。

通常，可能有许多生成相同 monad 的伴随关系。伴随函子本身形成了一个 2-范畴，因此可以使用伴随态射（2-范畴中的 1-胞腔）比较伴随函子。事实证明，Kleisli 伴随关系是在所有生成给定 monad 的伴随关系中初始的对象。对偶地，Eilenberg-Moore 伴随关系是终极的。

Chapter 16

Comonads(余单子)

如果它的发音更容易，我们可能会将副作用称为“ntext”，因为副作用的对偶是“context”（上下文）。

就像我们使用 Kleisli 箭头来处理副作用一样，我们使用 co-Kleisli 箭头来处理上下文。

让我们从一个熟悉的例子开始，即将环境作为上下文。我们之前通过对箭头进行柯里化构造了一个 Reader Monad（读取器单子）：

```
    (a, e) -> b
```

然而，这次我们将其视为一个 co-Kleisli 箭头，它是一个来自“上下文化”参数的箭头。

与单子类似，我们也对 co-Kleisli 箭头的组合感兴趣。对于携带环境的箭头来说，这相对容易：

```
composeWithEnv :: ((b, e) -> c) -> ((a, e) -> b) -> ((a, e) -> c)
composeWithEnv g f = \(a, e) -> g (f (a, e), e)
```

实现一个在这种组合下作为身份的箭头也很简单：

```
idWithEnv :: (a, e) -> a
idWithEnv (a, e) = a
```

这强烈暗示存在一个类别，在这个类别中 co-Kleisli 箭头作为态射。

Exercise 16.0.1. 证明使用 *composeWithEnv* 的 *co-Kleisli* 箭头的组合是结合的。

16.1 Comonads in Programming(编程中的余单子)

如果一个函子 w（可以看作是一个风格化的倒置 m）支持 co-Kleisli 箭头的组合，那么它就是一个余单子：

```
class Functor w => Comonad w where
(=<=) :: (w b -> c) -> (w a -> b) -> (w a -> c)
extract :: w a -> a
```

这里，组合以中缀操作符的形式书写。组合的单位被称为extract，因为它从上下文中提取一个值。

让我们尝试使用我们的例子。将环境作为对的第一个组件传递是方便的。然后，余单子由对构造器((,) e)的部分应用给出。

```
instance Comonad ((,) e) where
g =<= f = \ea -> g (fst ea, f ea)
extract = snd
```

与单子一样，co-Kleisli 组合可以用于无点风格的编程。但我们也可以使用join的对偶，称为duplicate：

```
duplicate :: w a -> w (w a)
```

或者使用bind的对偶，称为extend：

```
extend :: (w a -> b) -> w a -> w b
```

下面是如何用duplicate和fmap来实现 co-Kleisli 组合的：

```
g =<= f = g . fmap f . duplicate
```

Exercise 16.1.1. *实现duplicate和extend之间的互相转换。*

The Stream Comonad(流余单子)

余单子处理更大规模，有时是无限的上下文的例子很有趣。这里有一个无限流：

```
data Stream a = Cons a (Stream a)
deriving Functor
```

如果我们将这样的流视为类型a的值在一个无限尾部的上下文中，我们可以为它提供一个Comonad实例：

```
instance Comonad Stream where
extract (Cons a as) = a
duplicate (Cons a as) = Cons (Cons a as) (duplicate as)
```

这里，extract返回流的头部，而duplicate将一个流变成一个流的流，在这个流中，每一个连续的流都是前一个的尾部。

直观上，duplicate为迭代设置了舞台，但它以非常通用的方式进行。每个子流的头部可以解释为原始流中的未来“当前位置”。

这使得进行一个遍历这些流头部的计算变得容易。但这并不是余单子的力量所在。它让我们能够进行需要任意前瞻的计算。这种计算不仅需要访问连续子流的头部，还需要访问它们的尾部。

这就是extend的作用：它将给定的 co-Kleisli 箭头f应用于由duplicate生成的所有流：

```
extend f (Cons a as) = Cons (f (Cons a as)) (extend f as)
```

这里有一个计算流前五个元素平均值的 co-Kleisli 箭头的例子：

```
avg :: Stream Double -> Double
avg  = (/5). sum . stmTake 5
```

它使用了一个帮助函数，该函数提取前n个元素：

```
stmTake :: Int -> Stream a -> [a]
stmTake 0 _ = []
stmTake n (Cons a as) = a : stmTake (n - 1) as
```

我们可以使用extend在整个流上运行avg来平滑局部波动。电气工程师可能会将其识别为一个简单的低通滤波器，其中extend实现了卷积。它生成了原始流的一个滑动平均值。

```
smooth :: Stream Double -> Stream Double
smooth = extend avg
```

余单子对于结构化计算在空间上或时间上扩展的数据结构中很有用。这些计算足够局部以定义“当前位置”，但需要从邻近位置收集信息。信号处理或图像处理是很好的例子。模拟，在这些模拟中，微分方程必须在体积内迭代求解：气候模拟、宇宙学模型或核反应等等。康威的生命游戏也是一个很好的余单子方法测试场。

有时在连续的数据流上进行计算是方便的，推迟采样直到最后一步。这里是一个信号的例子，它是时间的函数（由Double表示）：

```
data Signal a = Sig (Double -> a) Double
```

第一个组件是由时间函数实现的连续a流。第二个组件是当前时间。

这是连续流的Comonad实例：

```
instance Comonad Signal where
extract (Sig f x) = f x
duplicate (Sig f x) = Sig (\y -> Sig f (x - y)) x
extend g (Sig f x) = Sig (\y -> g (Sig f (x - y))) x
```

这里，extend将滤波器g :: Signal a -> a卷积到整个流上。

Exercise 16.1.2. *为双向流实现Comonad实例：*

```haskell
data BiStream a = BStr [a] [a]
```

假设两个列表都是无限的。提示：将第一个列表视为过去（逆序）；将第二个列表的头部视为现在，其尾部视为未来。

Exercise 16.1.3. 为之前练习中的*BiStream*实现低通滤波器。它在三个值上进行平均：当前值、立即过去的一个值和立即未来的一个值。对于电气工程师：实现一个高斯滤波器。

16.2 Comonads Categorically(范畴论中的余单子)

我们可以通过反转单子定义中的箭头来获得余单子的定义。我们的duplicate对应于反转的join，而extract是反转的return。

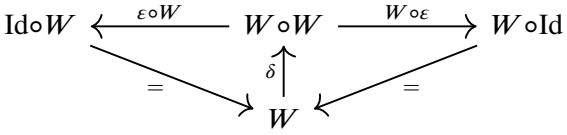
因此，余单子是一个配备了两个自然变换的自函子 W：

$$\delta \colon W \to W \circ W$$

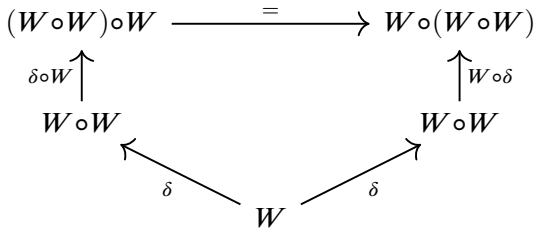
$$\varepsilon \colon W \to \mathrm{Id}$$

这些变换（分别对应duplicate和extract）必须满足与单子相同的恒等式，只不过箭头是反转的。

这些是余单子的余单元定律：



这就是结合律：



Comonoids(余半群)

我们已经看到了单子定律如何从半群定律中得出。我们可以期望余单子定律应当来自半群的对偶版本。

确实，一个余半群是一个对象 w，它在一个单元范畴 $(\mathcal{C}, \otimes, I)$ 中配备了两个称为余乘法和余单元的态射：

$$\delta \colon w \to w \otimes w$$

$$\varepsilon \colon w \to I$$

我们可以用自函子的复合替换张量积，用恒等函子替换单位对象，以获得作为自函子类别中的余半群的余单子定义。

在 Haskell 中，我们可以为笛卡尔积定义Comonoid类型类：

```
class Comonoid w where
split   :: w -> (w, w)
destroy :: w -> ()
```

与它们的兄弟 monoids（单半群）相比，comonoids（余半群）谈论较少，主要是因为它们被视为理所当然。在笛卡尔范畴中，每个对象都可以通过使用对角线映射 $\Delta_a \colon a \to a\ a$ 来进行余乘法，并使用到终对象的唯一箭头来进行余单元，从而成为一个 comonoid。

在编程中，这是我们不加思索就会做的事情。余乘法意味着能够复制一个值，余单元意味着能够放弃一个值。

在 Haskell 中，我们可以轻松地为任何类型实现Comonoid实例：

```
instance Comonoid w where
split w   = (w, w)
destroy w = ()
```

事实上，我们在使用函数的参数两次或根本不使用它时根本不会多想。但如果我们想要明确，我们可以将这样的函数：

```
f x = x + x
g y = 42
```

写成：

```
f x = let (x1, x2) = split x
in x1 + x2
g y = let () = destroy y
in 42
```

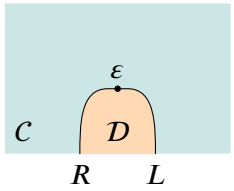
然而，在某些情况下，复制或丢弃变量是不合适的。这种情况出现在参数是外部资源（例如文件句柄、网络端口或在堆上分配的内存块）时。这些资源应具有明确的生命周期，从分配到释放。追踪可以轻松复制或丢弃的对象的生命周期是非常困难的，并且是编程错误的一个著名来源。

基于笛卡尔范畴的编程模型将始终存在此问题。解决方案是改用不支持对象复制或销毁的单元（闭）范畴。这样的范畴是线性类型的自然设置。在线性类型中，元素被用在Rust中，并且在写作本文时，正在尝试在 Haskell 中使用。在 C++ 中，有一些结构可以模仿线性，例如unique_ptr和移动语义。

16.3 Comonads from Adjunctions(从伴随构造余单子)

我们已经看到，两个函子 $L\colon \mathcal{D} \to \mathcal{C}$ 和 $R\colon \mathcal{C} \to \mathcal{D}$ 之间的伴随 $L \dashv R$ 会产生一个单子 $R \circ L\colon \mathcal{D} \to \mathcal{D}$。另一个复合 $L \circ R$，它是 $\mathcal{C}$ 中的一个自函子，结果是一个余单子。

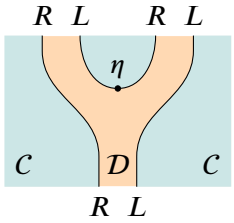
伴随的余单元成为余单子的余单元。这可以通过以下弦图来说明：



余乘法由 η 的 whiskering（缠绕）给出：

$$\delta = L \circ \eta \circ R$$

如下弦图所示：



与之前一样，余单子定律可以从三角恒等式中导出。

Costate Comonad(余状态余单子)

我们已经看到，State Monad（状态单子）可以从乘积与指数之间的柯里化伴随构造中生成。左函子被定义为与某个固定对象 s 的乘积：

$$L_s a = a \times s$$

右函子是以相同对象 s 为参数的指数：

$$R_s c = c^s$$

组合 $L_s \circ R_s$ 生成一个称为余状态余单子或存储余单子的余单子。

在 Haskell 中，右函子将函数类型`s->c`分配给`c`，而左函子将`c`与`s`配对。复合的结果是自函子：

```
data Store s c = St (s -> c) s
```

或者，使用 GADT 表示法：

```
data Store s c where
St :: (s -> c) -> s -> Store s c
```

函子实例通过后组合函数到Store的第一个组件来实现：

```
instance Functor (Store s) where
fmap g (St f s) = St (g . f) s
```

这个伴随的余单元成为余单子的extract，即函数应用：

```
extract :: Store s c -> c
extract (St f s) = f s
```

这个伴随的单位是自然变换 η: Id $\to$ $R_s \circ L_s$。我们已经使用它作为状态单子的return。这是它在c处的分量：

```
unit :: c -> (s -> (c, s))
unit c = \s -> (c, s)
```

要得到duplicate，我们需要将 η 进行缠绕：

$$\delta = L_s \circ \eta \circ R_s$$

缠绕在右边意味着使用eta在对象 $R_s c$ 上的分量，而缠绕在左边意味着使用 L_s 来提升此分量。由于 Haskell 翻译缠绕是一个棘手的过程，我们一步步分析。

为了简单起见，让我们将类型s固定为Int。我们将左函子封装在一个newtype中：

```
newtype Pair c = P (c, Int)
deriving Functor
```

并保持右函子为类型同义词：

```
type Fun c = Int -> c
```

伴随的单位可以写成使用显式forall的自然变换：

```
eta :: forall c. c -> Fun (Pair c)
eta c = \s -> P (c, s)
```

我们现在可以实现余乘法作为eta的缠绕。在右侧的缠绕通过使用eta在Fun c上的分量来编码，而在左侧的缠绕通过使用Pair函子的fmap来提升此分量。我们使用语言扩展TypeApplications使其明确使用哪个fmap：

```
delta :: forall c. Pair (Fun c) -> Pair (Fun (Pair (Fun c)))
delta = fmap @Pair eta
```

这可以更明确地重写为：

```
delta (P (f, s)) = P (\s' -> P (f, s'), s)
```

因此，Comonad实例可以写成：

```
instance Comonad (Store s) where
extract (St f s) = f s
duplicate (St f s) = St (St f) s
```

存储余单子是一个有用的编程概念。为了理解它，让我们再次考虑s为Int的情况。

我们将Store Int c的第一个组件，函数f :: Int -> c，解释为一个虚拟的无限值流的访问器，每个整数对应一个值。

第二个组件可以解释为当前索引。确实，extract使用这个索引来检索当前值。

有了这个解释，duplicate生成一个无限流的流，每个流都由不同的偏移量进行移位，而extend则对这个流进行卷积。当然，惰性保存了问题：只有我们明确要求的值才会被计算。

还要注意到，我们之前的Signal余单子的例子也可以通过Store Double复现。

Exercise 16.3.1. *使用存储余单子实现一个元胞自动机。这是描述规则 110 的 co-Kleisli 箭头：*

```
step :: Store Int Cell -> Cell
step (St f n) =
case (f (n-1), f n, f (n+1)) of
(L, L, L) -> D
(L, D, D) -> D
(D, D, D) -> D
_ -> L
```

一个元胞可以是活的也可以是死的：

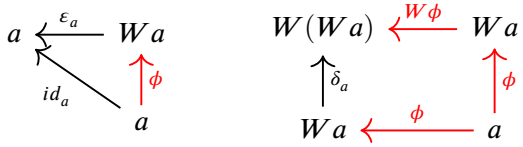
```
data Cell = L | D
deriving Show
```

运行此自动机的几代。提示：使用 Prelude 中的iterate函数。

Comonad Coalgebras(余单子余代数)

与单子代数对偶，我们有余单子余代数。给定一个余单子 (W, ε, δ)，我们可以构造一个余代数，它由一个承载对象 a 和一个箭头 $\phi\colon a \to W\,a$ 组成。为了使这个余代数能够与余单子很好地组合，我们要求能够提取用 ϕ 注入的值；并且 ϕ 作用于 ϕ 结果上的提升

与重复相同：



与单子代数一样，余单子余代数形成一个类别。给定 C 中的余单子 (W, ε, δ)，它的余单子余代数形成一个类别，称为 Eilenberg-Moore 范畴（有时前缀为 co-）C^W。

有一个 co-Kleisli 子范畴 C^W，记为 C_W。

给定一个余单子 W，我们可以使用 C^W 或 C_W 构造一个伴随构造，复现余单子 W。这个构造与单子的完全类比。

Lenses(透镜)

Store 余单子的余代数特别有趣。我们先做一些重命名。让我们称载体为`s`，状态为`a`。

```
data Store a s = St (a -> s) a
```

余代数由一个函数给出：

```
phi :: s -> Store a s
```

它等价于一对函数：

```
set :: s -> a -> s
get :: s -> a
```

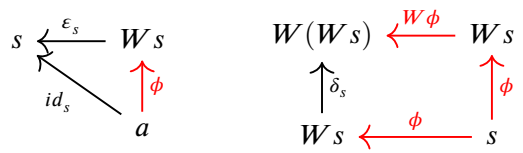
这样一对被称为透镜：`s`称为源，`a`为焦点。

在这种解释下，`get`让我们提取焦点，而`set`则用新值替换焦点以生成一个新的`s`。

透镜最初被引入以描述数据库记录中数据的检索和修改。然后，它们在处理数据结构时得到了应用。透镜客观化了对一个更大对象的部分进行读写访问的思想。例如，透镜可以聚焦于对的一部分或记录的一个特定组件。我们将在下一章讨论透镜和光学。

让我们将余单子余代数的定律应用于透镜。为简单起见，让我们省略方程中的数据构造函数。我们得到以下简化定义：

```
phi s = (set s, get s)
epsilon (f, a) = f a
delta (f, a) = (\x -> (f, x), a)
```



第一个定律告诉我们，应用`get`的结果来应用`set`的结果是恒等的：

```
set s (get  s) = s
```

这被称为透镜的 set/get 定律。当你用相同的焦点替换焦点时，不应改变任何东西。

第二个定律需要将`fmap phi`应用于`phi`的结果：

```
fmap phi (set s, get s) = (phi . set s, get s)
```

这应该等同于应用`delta`：

```
delta (set s, get s) = (\x -> (set s, x), get s)
```

比较两者，我们得到：

```
phi . set s = \x -> (set s, x)
```

让我们将其应用于某个`a`：

```
phi (set s a) = (set s, a)
```

使用`phi`的定义，我们得到：

```
(set (set s a), get (set s a)) = (set s, a)
```

我们有两个等式。第一个组件是函数，所以我们将它们应用于某个`a'`，得到 set/set 透镜定律：

```
set (set s a) a' = set s a'
```

将焦点设置为`a`，然后将其覆盖为`a'`与直接将焦点设置为`a'`是相同的。

第二个组件给我们 get/set 定律：

```
get (set s a) = a
```

在我们将焦点设置为`a`后，`get`的结果是`a`。

满足这些定律的透镜被称为合法透镜。它们是存储余单子的余代数。

Chapter 17

Ends and Coends（极限与余极限）

17.1 Profunctors（泛函）

在范畴论的高度抽象中，我们会遇到一些模式，这些模式离其起源如此遥远，以至于我们难以将其具象化。更为抽象的模式通常与其具体实例的相似度更低，这使得我们理解起来更加困难。

从 a 到 b 的箭头（arrow）相对容易理解。我们对它有一个非常熟悉的模型：一个函数，它消耗 a 的元素并生成 b 的元素。态射集（hom-set）是一组这样的箭头的集合。

函子（functor）是范畴之间的箭头。它消耗来自一个范畴的对象和箭头，并生成另一个范畴中的对象和箭头。我们可以将其视为一种使用源范畴提供的材料来构建这些对象（和箭头）的配方。尤其是，我们通常将自函子（endofunctor）视为构建材料的容器。

泛函（profunctor）将一对对象 $\langle a, b\rangle$ 映射到一个集合 $P\langle a, b\rangle$，并将一对箭头

$$\langle f : s \to a, g : b \to t\rangle$$

映射到一个函数：

$$P\langle f, g\rangle : P\langle a, b\rangle \to P\langle s, t\rangle$$

泛函是结合了许多其他抽象元素的抽象。因为它是一个从 $\mathcal{C}^{op}\ \ \mathcal{C}$ 到 Set 的函子，我们可以将其视为从一对对象中构造一个集合，并从一对箭头（其中一个箭头的方向相反）中构造一个函数。然而，这并不有助于我们进行想象。

幸运的是，我们有一个关于泛函的良好模型：同态函子（hom-functor）。在变动对象时，两者之间的箭头集表现得像一个泛函。并且明显的是，改变源对象和目标对象的同态集（hom-set）之间存在区别。

因此，我们可以将任意泛函视为同态函子的泛化。泛函在已有的同态集基础上，为对象之间提供了额外的桥梁。

然而，泛函 $\mathcal{C}(a, b)$ 的元素与集合 $P\langle a, b\rangle$ 的元素之间有一个很大的区别。前者的元素是箭头，而箭头可以组合（compose）。对于泛函如何组合，并没有一目了然的方法。

当然，可以认为泛函通过提升箭头（lifting of arrows）来泛化组合——不过不是在泛函之间，而是在同态集与泛函之间。例如，我们可以用箭头 $f : s \to a$ 来“预组合”$P\langle a, b\rangle$，以得到 $P\langle s, b\rangle$：

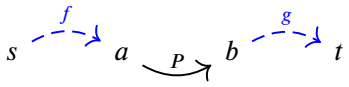
$$P\langle f, id_b\rangle : P\langle a, b\rangle \to P\langle s, b\rangle$$

类似地，我们可以用 $g : b \to t$“后组合”它：

$$P\langle id_a, g\rangle : P\langle a, b\rangle \to P\langle a, t\rangle$$

这种异质的组合方式，接受由一个箭头和一个泛函元素组成的可组合对，并生成一个泛函的元素。

泛函可以通过提升一对箭头在两侧进行扩展：



Collages（拼接）

没有理由将泛函限制在单一范畴上。我们可以轻松定义一个在两个范畴之间的泛函，例如 $P : \mathcal{C}^{op} \times \mathcal{D} \to \mathrm{Set}$。这样的泛函可以通过生成从 $\mathcal{C}$ 中的对象到 $\mathcal{D}$ 中的对象的同态集（hom-set）来将两个范畴连接在一起。

两个范畴 $\mathcal{C}$ 和 $\mathcal{D}$ 的拼接（collage）是一个范畴，其对象是来自两个范畴的对象（不相交的并集）。对于两个对象 x 和 y 之间的同态集，要么是 $\mathcal{C}$ 中的同态集（如果两个对象都在 $\mathcal{C}$ 中）；要么是 $\mathcal{D}$ 中的同态集（如果两个对象都在 $\mathcal{D}$ 中）；要么是集合 $P\langle x, y\rangle$（如果 x 在 $\mathcal{C}$ 中，y 在 $\mathcal{D}$ 中）。否则，同态集是空的。

态射的组合是通常的组合方式，除非其中一个态射是 $P\langle x, y\rangle$ 的元素。这种情况下，我们需要提升我们尝试进行前组合或后组合的态射。

很容易看出，拼接确实是一个范畴。新的态射跨越了拼接的两边，有时被称为异态射（heteromorphisms）。它们只能从 $\mathcal{C}$ 到 $\mathcal{D}$，而不能反向进行。

从这个角度看，一个从 $\mathcal{C}^{op} \times \mathcal{C}$ 到 Set 的泛函应该真正被称为一个内泛函（endo-profunctor）。它定义了 $\mathcal{C}$ 与其自身的拼接。

Exercise 17.1.1. 证明从两个范畴的拼接到一个有两个对象和一个箭头（以及两个恒等箭头）的“步行箭头”范畴（*walking arrow category*）之间存在一个函子。

Exercise 17.1.2. 证明，如果从 $\mathcal{C}$ 到步行箭头范畴存在一个函子，那么 $\mathcal{C}$ 可以分解成两个范畴的拼接。

Profunctors as relations（作为关系的泛函）

在微观层面上，泛函看起来像一个同态函子，并且集合 $P\langle a, b\rangle$ 的元素看起来像个别箭头。但是，当我们放大观察时，可以将泛函视为对象之间的关系。这些关系不是普通的关系；它们是*证明相关关系*（proof-relevant relations）。

为了更好地理解这一概念，让我们考虑一个普通的函子 $F\colon \mathcal{C} \to \mathrm{Set}$（换句话说，一个余预层（co-presheaf））。可以将其解释为，它定义了 $\mathcal{C}$ 对象的一个*证明相关子集*（proof-relevant subset），即映射到非空集合的那些对象。Fa 中的每个元素都被视为 a 是此子集成员的证明。另一方面，如果 Fa 是一个空集合，则 a 不是该子集的成员。

我们可以对泛函应用相同的解释。如果集合 $P\langle a, b\rangle$ 为空，我们说 b 与 a 不相关。如果它非空，我们说该集合中的每个元素代表了 b 与 a 相关的一个证明。然后，我们可以将泛函视为一种证明相关的关系。

请注意，我们对这种关系没有做任何假设。它不必是自反的，因为 $P\langle a, a\rangle$ 可能为空（实际上，$P\langle a, a\rangle$ 仅对内泛函有意义）。它也不必是对称的。

由于同态函子是（内）泛函的一个例子，这种解释使我们可以从新的角度看待同态函子：作为范畴中对象之间的一种内建的证明相关关系。如果两个对象之间有箭头，它们是相关的。请注意，这种关系是自反的，因为 $\mathcal{C}(a, a)$ 从不为空：至少它包含恒等态射。

此外，正如我们之前所见，同态函子与泛函相互作用。如果 a 通过 P 与 b 相关，并且同态集 $\mathcal{C}(s, a)$ 和 $\mathcal{D}(b, t)$ 是非空的，那么 s 与 t 通过 P 自动相关。因此，泛函是与它们所操作的范畴结构兼容的证明相关关系。

我们知道如何将同态函子与泛函组合在一起，但我们如何组合两个泛函呢？我们可以从关系的组合中得到线索。

假设你想为手机充电，但你没有充电器。要将你连接到充电器，你只需有一个拥有充电器的朋友。任何朋友都可以。你将拥有朋友的关系与拥有充电器的关系组合在一起，以获得可以为手机充电的关系。你能够为手机充电的证明是一对证明：一个是友情的证明，一个是拥有充电器的证明。

一般而言，我们说，如果存在一个中间对象与两者都相关，那么两个对象通过组合关系相关。

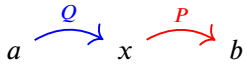
Profunctor composition in Haskell（Haskell 中的泛函组合）

关系的组合可以翻译为 Haskell 中的泛函组合。首先回顾泛函的定义：

```
class Profunctor p where
dimap :: (s -> a) -> (b -> t) -> (p a b -> p s t)
```

理解泛函组合的关键在于它需要中间对象的*存在*。对于对象 b 通过组合 $P \diamond Q$ 与对

象 a 相关，必须存在一个对象 x 来弥合这个差距：



这可以在 Haskell 中使用存在类型进行编码。给定两个泛函p和q，它们的组合是一个新的泛函Procompose p q：

```
data Procompose p q a b where
Procompose ::  q a x -> p x b -> Procompose p q a b
```

我们使用GADT来表达对象x的存在性。数据构造函数的两个参数可以看作是一对证明：一个证明x与a相关，另一个证明b与x相关。然后，这对证明构成了b与a相关的证明。

存在类型可以被视为和类型的推广。我们正在对所有可能的类型x进行求和。就像有限求和可以通过注入其中一个备选项来构造（想想Either的两个构造函数），存在类型可以通过为x选择一个特定类型并将其注入Procompose的定义中来构造。

就像从和类型映射出需要一对函数，每个备选项一个；从存在类型映射出需要一个函数族，每个类型一个。比如，从Procompose映射出的函数由一个多态函数定义：

```
mapOut :: Procompose p q a b -> (forall x. q a x -> p x b -> c) -> c
mapOut (Procompose qax pxb) f = (f qax pxb)
```

泛函的组合本身也是一个泛函，如以下实例所示：

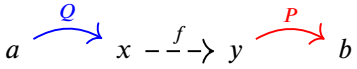
```
instance (Profunctor p, Profunctor q) => Profunctor (Procompose p q)
where
dimap l r (Procompose qax pxb) =
Procompose (dimap l id qax) (dimap id r pxb)
```

这只是说你可以通过将第一个泛函扩展到左边，将第二个泛函扩展到右边来扩展组合泛函。

在 Haskell 中这个泛函组合定义之所以能正常工作，是由于参数多态性。语言通过类型限制了泛函，使其能够正常工作。然而，一般来说，如果仅对中间对象进行简单求和，会导致过度计数，因此在范畴论中我们必须对此进行补偿。

17.2 Coends（余极限）

在泛函组合的简单定义中，过度计数发生在两个候选中间对象之间通过一个态射连接时：



我们可以在右边扩展 Q，通过提升 $Q\langle id, f\rangle$，并使用 y 作为中间对象；或者我们可以在左边扩展 P，通过提升 $P\langle f, id\rangle$，并使用 x 作为中介。

为了避免双重计数，我们必须在应用于泛函时调整我们的和类型定义。得到的构造被称为余极限（coend）。

首先，让我们重新表述问题。我们试图对所有对象 x 在乘积上求和：

$$P\langle a, x\rangle \quad Q\langle x, b\rangle$$

双重计数发生是因为我们可以在两个泛函之间打开一个间隙，只要有一个态射可以填充其中。因此，我们真正关注的是一个更通用的乘积：

$$P\langle a, x\rangle \quad Q\langle y, b\rangle$$

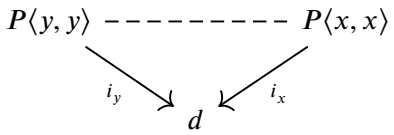
重要的是，如果我们固定端点 a 和 b，这个乘积是 $\langle y, x\rangle$ 中的一个泛函。这可以通过一些排列（同构）轻松看出：

$$Q\langle y, b\rangle \quad P\langle a, x\rangle$$

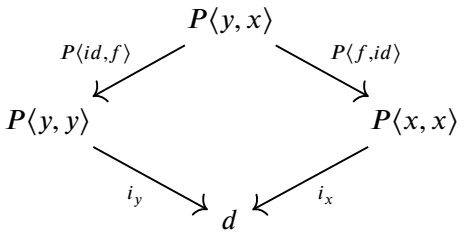
我们感兴趣的是这个泛函的对角部分的和，也就是说，当 x 等于 y 时。

因此，让我们看看如何定义一个泛函 P 的对角项的和。实际上，这种构造适用于任何函子 $P\colon \mathcal{C}^{op} \quad \mathcal{C} \to \mathcal{D}$，而不仅仅是 Set 值的泛函。

对角项的和由注入定义；在这种情况下，每个对象 $\mathcal{C}$ 有一个注入。这里我们只展示其中的两个，虚线代表所有其他的注入：



如果我们在定义一个和，我们会使其成为一个带有这些注入的泛函对象。但由于我们处理的是两个变量的函子，我们希望识别那些通过“扩展”一些公共祖先（这里是 $P\langle y, x\rangle$）而相关的注入。我们希望下图在存在连接态射 $f\colon x \to y$ 时交换：



这个图被称为余楔形图（co-wedge），其交换条件称为余楔条件。对于每个 $f\colon x \to y$，我们要求：

$$i_x \circ P\langle f, id_y\rangle = i_y \circ P\langle id_x, f\rangle$$

通用余楔被称为余极限（coend）。

由于余极限将和推广到可能的无限域，我们用积分符号表示它，上方标出“积分变量”：

$$\int^{x\colon C} P\langle x, x\rangle$$

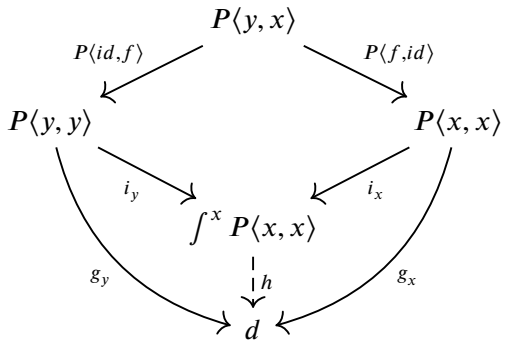
通用性意味着，任何配备了满足余楔条件的箭头族 $g_x\colon P\langle x, x\rangle \to d$ 的对象 d，都会有一个唯一的从余极限映射出的映射：

$$h\colon \int^{x\colon C} P\langle x, x\rangle \to d$$

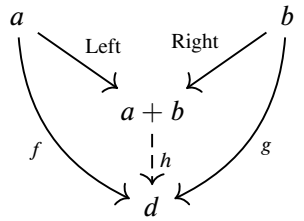
可以通过注入 i_x 因式分解所有的 g_x：

$$g_x = h \circ i_x$$

如图所示：



将其与两个对象之和的定义进行比较：



正如和被定义为通用的余图形（cospan），余极限被定义为通用的余楔形图。

特别是，如果你要构造一个 Set 值泛函的余极限，你会从所有集合 $P\langle x, x\rangle$ 的和（带鉴别的并集）开始。然后你会识别出所有满足余楔条件的和元素。你会将 $P\langle x, x\rangle$ 中的元素 a 与 $P\langle y, y\rangle$ 中的元素 b 相识别，每当存在 $P\langle y, x\rangle$ 中的元素 c 和态射 $f\colon x \to y$，使得：

$$P\langle id, f\rangle(c) = b$$

且

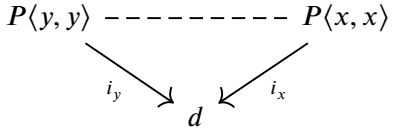
$$P\langle f, id\rangle(c) = a$$

请注意，在离散范畴中（它只是一个对象集，不存在它们之间的箭头），余楔条件是平凡的（除了恒等箭头外没有其他 f），所以余极限只是对角对象 $P\langle x, x\rangle$ 的直接和（余并）。

Extranatural transformations（超自然变换）

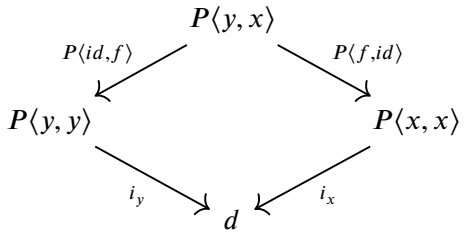
在目标范畴中由源范畴的对象参数化的箭头族通常可以合并为两个函子之间的单一自然变换。

在余楔的定义中，注入构成了一个由对象参数化的函数族，但它们不能很好的契合自然变换的定义。



问题在于函子 $P\colon \mathcal{C}^{op}\ \mathcal{C} \to \mathcal{D}$ 在第一个参数中是反变的，在第二个参数中是协变的；因此，其对角部分在对象上定义为 $x \mapsto P\langle x, x\rangle$，但它并非自然变换。

我们可以利用的最接近的自然性（naturality）是余楔条件：



的确，与自然性方块一样，它涉及的是态射 $f\colon x \to y$ 的提升（这里有两种不同的方式）与变换 i 的分量之间的交互。

当然，标准的自然性条件涉及的是函子对。在这里，变换的目标是一个固定的对象 d。但我们总是可以将其重新解释为常量泛函 $\Delta_d\colon \mathcal{C}^{op}\ \mathcal{C} \to \mathcal{D}$ 的输出。因此，为了推广自然性，我们将 Δ_d 替换为任意泛函 Q。

我们将余楔条件重新解释为更一般的**超自然变换**（extranatural transformation）的特例。超自然变换是一个箭头族：

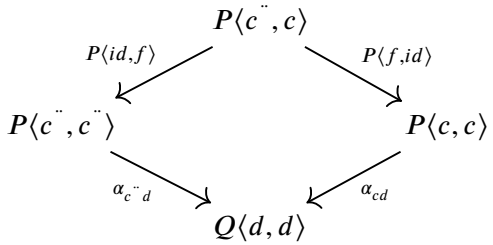
$$\alpha_{cd}\colon P\langle c, c\rangle \to Q\langle d, d\rangle$$

在两个函子之间：

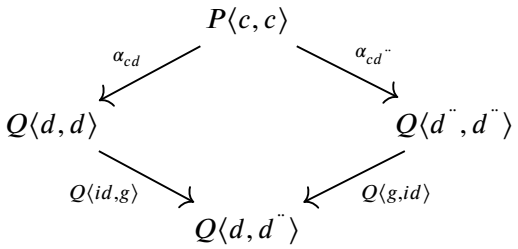
$$P\colon \mathcal{C}^{op}\ \mathcal{C} \to \mathcal{E}$$

$$Q\colon \mathcal{D}^{op}\ \mathcal{D} \to \mathcal{E}$$

在 c 中的超自然性意味着，对于任何态射 $f\colon c \to c^{\cdot\cdot}$，以下图表交换：



在 d 中的超自然性意味着，对于任何态射 $g\colon d \to d^{\cdot\cdot}$，以下图表交换：



基于此定义，我们可以将余楔条件重新解释为从泛函 P 到常量泛函 Δ_d 的映射的超自然性。

现在我们可以将余极限定义为对 (c, i) 的配对，其中 c 是配备了从 P 到 Δ_c 的超自然变换 i 的对象，并且在此类对中是通用的。

通用性意味着，对于任何配备了从 P 到 Δ_d 的超自然变换 α 的对象 d，存在一个唯一的态射 $h\colon c \to d$，它通过 i 的所有分量对 α 的所有分量进行因式分解：

$$\alpha_x = h \circ i_x$$

我们将这个对象 c 称为余极限，并将其写为：

$$c = \int^x P\langle x, x\rangle$$

Exercise 17.2.1. 验证对于从 P 到 Δ_d 的超自然变换，第一个超自然性菱形等价于余楔条件，第二个条件是平凡的。

Profunctor composition using coends（使用余极限的泛函组合）

有了余极限的定义，我们现在可以正式定义两个泛函的组合：

$$(P \diamond Q)\langle a, b\rangle = \int^{x\colon C} Q\langle a, x\rangle \quad P\langle x, b\rangle$$

将其与以下内容进行比较：

```
data Procompose p q a b where
Procompose ::  q a x -> p x b -> Procompose p q a b
```

在 Haskell 中我们不需要担心余楔条件的原因类似于所有参数多态函数自动满足自然性条件的原因。余极限是使用箭头族定义的；在 Haskell 中，所有这些注入都是由单个多态函数定义的：

```
data Coend p where
Coend ::  p x x -> Coend p
```

参数多态性（parametricity）然后强制执行余楔条件。

余极限在处理泛函时引入了一个新的抽象层次。使用余极限进行计算通常利用其映射出属性。要定义从余极限到某个对象 d 的映射：

$$\int^x P\langle x, x\rangle \to d$$

只需定义从函子的对角项到 d 的一族函数：

$$g_x \colon P\langle x, x\rangle \to d$$

满足余楔条件。这个技巧结合 Yoneda 引理可以带来很大的好处。我们将在接下来的内容中看到一些例子。

Exercise 17.2.2. 为以下泛函对定义一个*Profunctor*实例：

```
newtype ProPair q p a b x y = ProPair (q a y, p x b)
```

提示：保持前四个参数固定：

```
instance (Profunctor p, Profunctor q) => Profunctor (ProPair q p a b)
```

Exercise 17.2.3. 泛函组合可以使用余极限表示：

```
newtype CoEndCompose p q a b = CoEndCompose (Coend (ProPair q p a b))
```

为*CoEndCompose*定义一个*Profunctor*实例。

Colimits as coends（作为余极限的余极限）

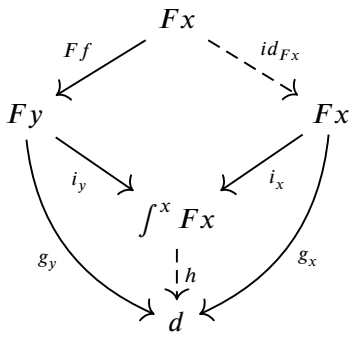
一个忽略其参数之一的二元函数等价于一个一元函数。类似地，一个忽略其参数之一的泛函等价于一个函子。反过来，给定一个函子 F，我们可以构造一个平凡的泛函。它在对象上的作用由以下公式给出：

$$P\langle x, y\rangle = Fy$$

它在箭头对上的作用忽略了其中一个箭头：

$$P\langle f, g\rangle = Fg$$

对于任意 $f \colon x \to y$，我们这种泛函的余极限的定义简化为以下图表：



在缩小恒等箭头之后，原始的余楔形图变成了余锥形图（co-cone），通用条件变成了余极限的定义。这就证明了将余极限符号用于余极限的合理性：

$$\int^x Fx = \mathrm{colim} F$$

函子 F 定义了目标范畴中的一个图形。这个模式是整个源范畴。

如果我们考虑一个离散范畴，其中泛函是一个（可能是无限的）矩阵，余极限是其对角线元素的和（余并）。沿着一条轴恒定的泛函对应于每一行相同的矩阵（每行由一个向量 Fx 给出）。这样的矩阵的对角线元素之和等于向量 Fx 的所有元素之和。

$$\begin{pmatrix} Fa & Fb & Fc & \dots \\ Fa & Fb & Fc & \dots \\ Fa & Fb & Fc & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

在非离散范畴中，这个和则推广为余极限。

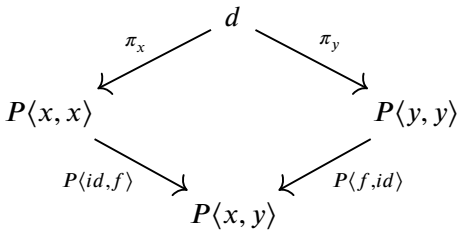
17.3 Ends（余极限）

就像余极限（coend）推广了伴随态射（profunctor）的对角元素之和一样，它的对偶概念，极限（end），推广了乘积的概念。乘积通过它的投影来定义，极限也是如此。

我们在定义乘积时使用的广义“跨越”（span）的概念可以推广为一个对象 d，该对象有一系列投影，每个对象 x 对应一个投影：

$$\pi_x : d \to P\langle x, x\rangle$$

与余契（co-wedge）相对的是所谓的契（wedge）：



对于每一个箭头 $f : x \to y$，我们要求：

$$P\langle f, id_y\rangle \circ \pi_y = P\langle id_x, f\rangle \circ \pi_x$$

极限（end）是一个具有普遍性质的契（universal wedge）。我们为其使用积分符号，这次积分变量写在符号的下方：

$$\int_{x:\,C} P\langle x, x\rangle$$

你可能会想知道，为什么在微积分中，基于乘法的积分而不是加法的积分很少被使用。那是因为我们总是可以通过对数将乘法转换为加法。但在范畴论中，我们没有这种奢侈的选择，因此极限和余极限同样重要。

总结一下，极限是一个具备一系列态射（projections）的对象：

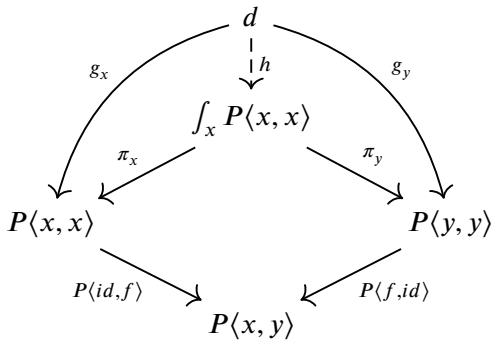
$$\pi_a \colon \left(\int_x P\langle x, x\rangle \right) \to P\langle a, a\rangle$$

并满足契（wedge）条件。

它在此类对象中具有普遍性；也就是说，对于任何其他对象 d，如果其带有一系列满足契条件的箭头 g_x，那么存在唯一的态射 h，使得 g_x 的这系列态射通过 π_x 的这系列态射来分解：

$$g_x = \pi_x \circ h$$

图解如下：



同样地，我们可以说极限是一个 (e, π) 对，其中 $e = \int_x P\langle x, x\rangle$ 是一个对象，$\pi \colon \Delta_d \to e$ 是一个额外自然变换（extranatural transformation），并且在所有此类对中具有普遍性。契条件实际上是额外自然性条件（extranaturality condition）的特例。

如果你要构建一个 Set-值的伴随态射的极限，你可以从包含类别 C 中所有对象 x 的 $P\langle x, x\rangle$ 的巨大积开始，然后剪除不满足契条件的元组。

具体来说，假设使用单例集（singleton set）1 代替 d。族 g_x 将从每个集合 $P\langle x, x\rangle$ 中选择一个元素。这会给你一个巨大的元组。然后你将剔除大部分的元组，只保留那些满足契条件的元组。

同样地，在 Haskell 中，由于参数化（parametricity）的原因，契条件自动得到满足，并且对一个伴随态射 `p` 的极限定义简化为：

```
type End p = forall x. p x x
```

Haskell 中的 `End` 实现并没有展示它是 `Coend` 的对偶这一事实。这是因为在撰写本文时，Haskell 还没有内置的存在类型语法。如果有的话，`Coend` 可以实现为：

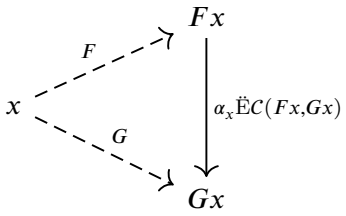
```
type Coend p = exists x. p x x
```

`Coend` 和 `End` 之间的存在/普遍性二元对立意味着构造 `Coend` 很容易——你只需要选择一个类型 `x`，对于该类型你有一个类型 `p x x` 的值。另一方面，要构造一个 `End`，你

必须提供整个族的值 `p x x`，对于每个类型 `x` 都有一个。换句话说，你需要一个以 `x` 为参数的多态公式。多态函数的定义是这种公式的典型示例。

Natural transformations as an end（作为余极限的自然变换）

极限的最有趣的应用是简洁地定义自然变换（natural transformations）的集合。考虑两个函子 F 和 G，它们在两个范畴 $\mathcal{B}$ 和 $\mathcal{C}$ 之间移动。它们之间的自然变换是一个在 $\mathcal{C}$ 中的箭头族 α_x。你可以将其视为从每个同态集合 $\mathcal{C}(Fx, Gx)$ 中选择一个元素 α_x。



我们知道，映射 $\langle a, b\rangle \to \mathcal{C}(a, b)$ 定义了一个伴随态射。事实证明，对于任何一对函子，映射 $\langle a, b\rangle \to \mathcal{C}(Fa, Gb)$ 也表现得像一个伴随态射。此伴随态射对一对箭头 $\langle f : s \to a, g : b \to t\rangle$ 的作用是一个函数：

$$\mathcal{C}(Fa, Gb) \to \mathcal{C}(Fs, Gt)$$

该函数通过以下方式实现组合：

$$Fs \xrightarrow{Ff} Fa \xrightarrow{h} Gb \xrightarrow{Gg} Gt$$

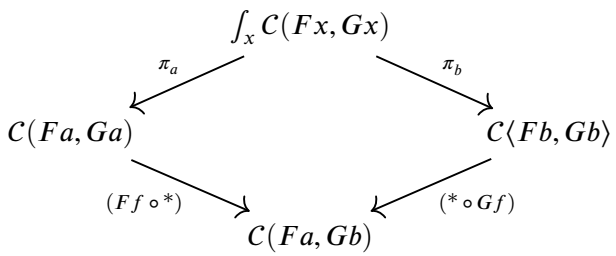
其中 h 是 $\mathcal{C}(Fa, Gb)$ 的一个元素。

这个伴随态射的对角部分是自然变换分量的完美候选。在实际操作中，这个极限：

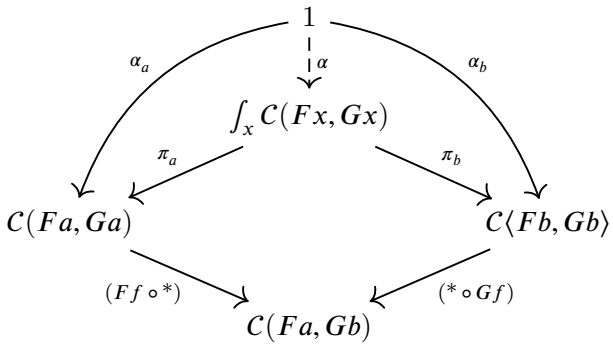
$$\int_{x:\,\mathcal{B}} \mathcal{C}(Fx, Gx)$$

定义了从 F 到 G 的自然变换的集合。

为了证明这一点，让我们检查契条件。将我们的伴随态射代入，可以得到：



我们可以通过单例集实例化普遍条件，选择集合 $\int_x \mathcal{C}(Fx, Gx)$ 的一个单一元素：



我们从同态集合 $\mathcal{C}(Fa, Ga)$ 中选择了分量 α_a，从 $\mathcal{C}(Fb, Gb)$ 中选择了分量 α_b。契条件最终归结为：

$$Ff \circ \alpha_a = \alpha_b \circ Gf$$

对于任何 $f : a \to b$，这正是自然性条件。因此，极限中的任何元素 α 都自动是一个自然变换。

自然变换的集合，或者说函子范畴中的同态集合，由极限给出：

$$[\mathcal{C}, \mathcal{D}](F, G) \ \ddot{\circ} \int_{x\colon \mathcal{B}} \mathcal{C}(Fx, Gx)$$

在 Haskell 中，这与我们之前的定义是一致的：

```
type Natural f g = forall x. f x -> g x
```

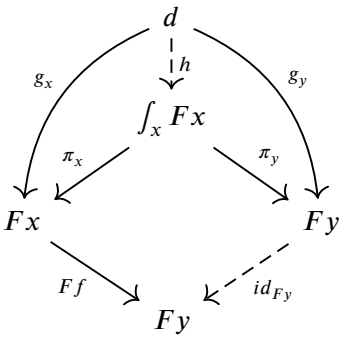
正如我们前面讨论的那样，要构造一个 `End`，我们必须提供一个以类型为参数的整个族的值。在这里，这些值就是多态函数的分量。

Limits as ends（极限作为余极限）

正如我们能够将余极限表示为余极限，我们也可以将极限表示为极限。同样地，我们定义了一个忽略其第一个参数的平凡伴随态射：

$$P\langle x, y\rangle = Fy$$
$$P\langle f, g\rangle = Fg$$

定义极限的普遍条件成为定义普遍锥体的条件：



因此我们可以使用极限符号表示极限：

$$\int_x Fx = \lim F$$

Exercise 17.3.1. 一个乘积是从两个对象的范畴 2 到某个范畴的函子的极限。证明它可以被定义为一个余极限。提示：在 2 中没有非恒等态射。

17.4 Continuity of the Hom-Functor（同态函子的连续性）

在范畴论中，如果一个函子 F 保持极限（也就是说，它保持极限和余极限），我们就称它是连续函子（continuous functor）。这意味着，如果你在源范畴中有一个图表，那么无论是先使用 F 映射图表，然后在目标范畴中取极限，还是在源范畴中取极限然后使用 F 映射这个极限，结果是一样的。

同态函子是一个在其第二个参数中连续的函子的例子。由于乘积是最简单的极限示例，这意味着特别地：

$$\mathcal{C}(x, a \quad b) \ \ddot{\text{o}} \ \mathcal{C}(x, a) \quad \mathcal{C}(x, b)$$

左边将同态函子应用于乘积（一个跨越的极限）。右边映射图表，这里仅是一个对象对，并在目标范畴中取乘积（极限）。对于同态函子来说，目标范畴是 Set，因此这只是一个笛卡尔乘积。通过乘积的普遍性质，这两边是同构的：映射到乘积中的映射是通过映射到两个对象的对定义的。

同态函子在第一个参数中的连续性是相反的：它将余极限映射到极限。再一次，最简单的余极限示例是和，因此我们有：

$$\mathcal{C}(a + b, x) \ \ddot{\text{o}} \ \mathcal{C}(a, x) \quad \mathcal{C}(b, x)$$

这来自和的普遍性：从和映射出的映射由从两个对象映射出的对定义。

可以证明，极限可以表示为极限，余极限可以表示为余极限。因此，通过同态函子的连续性，我们总是可以将积分符号从同态集合中拉出。通过与乘积的类比，我们有极限的映射公式：

$$\mathcal{D}\left(d, \int_a P\langle a, a\rangle\right) \ \ddot{\text{o}} \ \int_a \mathcal{D}(d, P\langle a, a\rangle)$$

通过与和的类比，我们有余极限的映射公式：

$$\mathcal{D}\left(\int^a P\langle a, a\rangle, d\right) \ \ddot{\text{o}} \ \int_a \mathcal{D}(P\langle a, a\rangle, d)$$

请注意，在这两种情况下，右边的表达式是一个余极限。

17.5 Fubini Rule（富比尼法则）

富比尼法则（Fubini rule）在微积分中指出了我们可以交换双重积分中积分顺序的条件。事实证明，我们也可以类似地交换双重极限和余极限的顺序。对于形式为 $P\colon \mathcal{C}\ \ \mathcal{C}^{op}\ \ \mathcal{D}\ \ \mathcal{D}^{op} \to \mathcal{E}$ 的函子，富比尼法则适用于以下表达式（只要它们存在）是同构的：

$$\int_{c\colon \mathcal{C}} \int_{d\colon \mathcal{D}} P\langle c,c\rangle\langle d,d\rangle \cong \int_{d\colon \mathcal{D}} \int_{c\colon \mathcal{C}} P\langle c,c\rangle\langle d,d\rangle \cong \int_{\langle c,d\rangle\colon \mathcal{C}\ \mathcal{D}} P\langle c,c\rangle\langle d,d\rangle$$

在最后一个极限中，函子 P 被重新解释为 $P\colon (\mathcal{C}\ \ \mathcal{D})^{op}\ \ (\mathcal{C}\ \ \mathcal{D}) \to \mathcal{E}$

类似的规则也适用于余极限。

17.6 Ninja Yoneda Lemma（Ninja 代数引理）

在表达了自然变换的集合作为极限之后，我们现在可以重新写代数引理（Yoneda lemma）。这是原始的表述：

$$[\mathcal{C}, \mathrm{Set}](\mathcal{C}(a,*), F) \cong Fa$$

在这里，F 是一个从 $\mathcal{C}$ 到 Set（一个反预层）的（协变）函子，同态函子 $\mathcal{C}(a,*)$ 也是。将自然变换的集合表达为极限，我们得到：

$$\int_{x\colon \mathcal{C}} \mathrm{Set}(\mathcal{C}(a,x), Fx) \cong Fa$$

类似地，我们有一个适用于逆变函子（一个预层）G 的代数引理：

$$\int_{x\colon \mathcal{C}} \mathrm{Set}(\mathcal{C}(x,a), Gx) \cong Ga$$

这些版本的代数引理以极限的形式表达，通常半开玩笑地称之为 Ninja 代数引理。因为“积分变量”是显式的，这使得它们在复杂公式中更容易使用。

还有一组对偶的 Ninja 共代数引理，它们使用余极限。对于协变函子，我们有：

$$\int^{x\colon \mathcal{C}} \mathcal{C}(x,a)\ \ Fx \cong Fa$$

对于逆变函子，我们有：

$$\int^{x\colon \mathcal{C}} \mathcal{C}(a,x)\ \ Gx \cong Ga$$

物理学家可能会注意到这些公式与涉及狄拉克 δ 函数的积分的相似性——严格来说，δ 是一种分布。这就是为什么伴随态射有时被称为分布器（distributors），遵循“分布器对于函子就像分布对于函数”的格言。工程师可能会注意到同态函子与冲激函数的相似性。

这种直觉通常被表达为我们可以在这个公式中执行“对 x 的积分”，从而将被积函数中的 x 替换为 a。

如果 $\mathcal{C}$ 是一个离散范畴，余极限简化为和（余积），同态函子简化为单位矩阵（克罗内克 δ）。共代数引理变为：

$$\sum_j \delta_i^j v_j = v_i$$

事实上，许多线性代数可以直接翻译为 Set-值函子的理论。你可以经常将这些函子视为向量空间中的向量，其中同态函子形成一个基。伴随态射变为矩阵，余极限用于乘以这些矩阵，计算它们的迹，或将向量与矩阵相乘。

在澳大利亚，伴随态射还有一个名称，叫做双模（bimodules）。这是因为通过伴随态射提升态射的行为类似于集上的左作用和右作用。

我们现在将继续证明共代数引理，它相当具有指导意义，因为它使用了一些常见的技巧。最重要的是，我们依赖于代数引理的推论，它说，如果从两个对象映射到任意对象的所有映射是同构的，那么这两个对象本身是同构的。在我们的例子中，我们将考虑映射到任意集合 S 的映射，并表明：

$$\mathrm{Set}\left(\int^{x:\,\mathcal{C}} \mathcal{C}(x,a) \quad Fx, S\right) \ddot{\mathrm{o}}\ \mathrm{Set}(Fa, S)$$

使用同态函子的余连续性，我们可以将积分符号提取出来，将余极限替换为极限：

$$\int_{x:\,\mathcal{C}} \mathrm{Set}\left(\mathcal{C}(x,a) \quad Fx, S\right)$$

由于集合范畴是笛卡尔闭的，我们可以对积进行柯里化：

$$\int_{x:\,\mathcal{C}} \mathrm{Set}\left(\mathcal{C}(x,a), S^{Fx}\right)$$

我们现在可以使用代数引理“对 x 进行积分”。结果是 S^{Fa}。最后，在 Set 中，指数对象与同态集合同构：

$$S^{Fa} \ddot{\mathrm{o}}\ \mathrm{Set}(Fa, S)$$

由于 S 是任意的，我们得出结论：

$$\int^{x:\,\mathcal{C}} \mathcal{C}(x,a) \quad Fx \ddot{\mathrm{o}}\ Fa$$

Exercise 17.6.1. 证明共代数引理的逆变版本。

Yoneda lemma in Haskell（Haskell 中的代数引理）

我们已经在 Haskell 中实现了代数引理。我们现在可以将其以极限的形式重写。我们首先定义一个将在极限下使用的伴随态射。它的类型构造器接收一个函子 `f` 和一个类型 `a`，并生成一个在 `x` 上逆变，在 `y` 上协变的伴随态射：

```
data Yo f a x y = Yo ((a -> x) -> f y)
```

代数引理建立了这个伴随态射下的极限与通过对 a 应用函子 f 获得的类型之间的同构。这种同构通过一对函数来实现：

```
yoneda :: Functor f => End (Yo f a) -> f a
yoneda (Yo g) = g id

yoneda_1 :: Functor f => f a -> End (Yo f a)
yoneda_1 fa = Yo (\h -> fmap h fa)
```

类似地，共代数引理使用下面这个伴随态射上的余极限：

```
data CoY f a x y = CoY (x -> a) (f y)
```

同构由一对函数表示。第一个函数说明，如果你有一个函数 x -> a 和一个包含 x 的函子，你就可以使用 fmap 创建一个包含 a 的函子：

```
coyoneda :: Functor f => Coend (CoY f a) -> f a
coyoneda (Coend (CoY g fa)) = fmap g fa
```

你可以做到这一点，而不需要知道存在类型 x 的任何信息。

第二个函数说明，如果你有一个包含 a 的函子，你可以通过将它（连同恒等函数）注入到存在类型中创建一个余极限：

```
coyoneda_1 :: Functor f => f a -> Coend (CoY f a)
coyoneda_1 fa = Coend (CoY id fa)
```

17.7 Day Convolution（Day 卷积）

电气工程师熟悉卷积的概念。我们可以通过移位其中一个流并求其与另一个流的乘积和来卷积两个流：

$$(f \star g)(x) = \int^{\varnothing}_{*\varnothing} f(y)g(x * y)dy$$

这个公式几乎可以直接翻译为范畴论。我们可以从用余极限替换积分开始。问题是，我们不知道如何对对象进行减法操作。然而，我们确实知道如何在共笛卡尔范畴中对它们进行加法操作。

请注意，这两个函数的参数之和等于 x。我们可以通过引入狄拉克 δ 函数或“冲激函数”（$\delta(a + b * x)$）来强制执行这一条件。在范畴论中，我们使用同态函子 $C(a + b, x)$ 来实现同样的效果。因此，我们可以定义两个 Set-值函子的卷积：

$$(F \star G)x = \int^{a,b} C(a + b, x) \quad Fa \quad Gb$$

非正式地说，如果我们可以定义减法为共积的右伴随函数，我们可以写成：

$$\int^{a,b} C(a+b,x) \quad Fa \quad Gb \ö \int^{a,b} C(a,b*x) \quad Fa \quad Gb \ö \int^{b} F(b*x) \quad Gb$$

共积没有什么特别之处，因此，通常情况下，Day 卷积在具有张量积的任何单胚范畴中都可以定义：

$$(F \star G)x = \int^{a,b} C(a \otimes b, x) \quad Fa \quad Gb$$

实际上，Day 卷积为一个单胚范畴 $(\mathcal{C}, \otimes, I)$ 赋予了 $[\mathcal{C}, \mathrm{Set}]$ 的共预层范畴一个单胚结构。简单来说，如果你可以在 $\mathcal{C}$ 中对对象进行乘法，你就可以在 $\mathcal{C}$ 上对 Set-值函子进行乘法。

很容易证明 Day 卷积是关联的（最多是同构），并且 $\mathcal{C}(I,*)$ 作为单位对象。例如，我们有：

$$(\mathcal{C}(I,*) \star G)x = \int^{a,b} C(a \otimes b, x) \quad C(I,a) \quad Gb \ö \int^{b} C(I \otimes b, x) \quad Gb \ö Gx$$

因此，Day 卷积的单位是对单胚单位采取的 Yoneda 函子，这引出了一个字谜般的口号，“Day 的 ONE 是 ONE 的 Yoneda。”

如果张量积是对称的，那么相应的 Day 卷积也是对称的（最多是同构）。

在笛卡尔闭范畴的特殊情况下，我们可以使用柯里化调整来简化公式：

$$(F \star G)x = \int^{a,b} C(a \quad b, x) \quad Fa \quad Gb \ö \int^{a,b} C(a, x^b) \quad Fa \quad Gb \ö \int^{b} F(x^b) \quad Gb$$

在 Haskell 中，基于乘积的 Day 卷积可以使用存在类型来定义：

```
data Day f g x where
Day :: ((a, b) -> x) -> f a -> g b -> Day f g x
```

如果我们将函子视为值的容器，Day 卷积告诉我们如何将两个不同的容器组合成一个——只需给出一个将两个不同值组合成一个的函数。

Exercise 17.7.1. *为 `Day` 定义 `Functor` 实例。*

Exercise 17.7.2. *为 `Day` 实现关联器。*

```
assoc :: Day f (Day g h) x -> Day (Day f g) h x
```

Applicative functors as monoids（作为单胚的应用函子）

我们之前看到了应用函子作为松弛单胚函子的定义。事实证明，就像单子一样，应用函子也可以定义为单胚。

回想一下，单胚是一个单胚范畴中的对象。我们感兴趣的范畴是 $[\mathcal{C}, \mathrm{Set}]$ 的共预层范畴。如果 $\mathcal{C}$ 是笛卡尔的，那么共预层范畴相对于 Day 卷积是单胚的，单位对象是 $\mathcal{C}(I, *)$。此范畴中的单胚是一个函子 F，它带有两个自然变换，分别作为单位和乘法：

$$\eta \colon \mathcal{C}(I, *) \to F$$

$$\mu \colon F \star F \to F$$

特别是在一个笛卡尔闭范畴中，其中单位是终对象，$\mathcal{C}(1, a)$ 与 a 同构，并且单位在 a 处的分量是：

$$\eta_a \colon a \to Fa$$

你可能会认出这个函数就是 `pure` 在 `Applicative` 定义中的作用。

```
pure :: a -> f a
```

让我们考虑从中选择 μ 的自然变换的集合。我们将其写为一个极限：

$$\mu \ \ddot{\mathrm{E}} \int_x \mathrm{Set}\Big((F \star F)x, Fx\Big)$$

代入 Day 卷积的定义，我们得到：

$$\int_x \mathrm{Set}\Big(\int^{a,b} \mathcal{C}(a \quad b, x) \quad Fa \quad Fb, Fx \Big)$$

我们可以使用同态函子的余连续性来提取余极限：

$$\int_{x,a,b} \mathrm{Set}\Big(\mathcal{C}(a \quad b, x) \quad Fa \quad Fb, Fx\Big)$$

然后我们可以在 Set 中使用柯里化调整：

$$\int_{x,a,b} \mathrm{Set}\Big(\mathcal{C}(a \quad b, x), \mathrm{Set}(Fa \quad Fb, Fx)\Big)$$

最后，我们应用代数引理来对 x 进行积分：

$$\int_{a,b} \mathrm{Set}\Big(Fa \quad Fb, F(a \quad b)\Big)$$

结果是从中选择松弛单胚函子的第二部分的自然变换的集合：

```
(>*<) :: f a -> f b -> f (a, b)
```

Free Applicatives（自由应用函子）

我们刚刚了解到，应用函子是单胚范畴中的单胚：

$$([\mathcal{C}, \mathrm{Set}], \mathcal{C}(I, *), \star)$$

自然会问，这个范畴中的自由单胚是什么。

就像我们对自由单子所做的那样，我们将构造一个自由应用函子作为初始代数，或者列表函子的最小不动点。回想一下，列表函子被定义为：

$$\Phi_a x = 1 + a \otimes x$$

在我们的情况下，它变为：

$$\Phi_F G = \mathcal{C}(I, *) + F \star G$$

它的不动点由以下递归公式给出：

$$A_F \ddot{\circ} \mathcal{C}(I, *) + F \star A_F$$

将此公式翻译为 Haskell 时，我们观察到从单位 `()->a` 到元素 `a` 的函数是同构的。

对应于 A_F 定义中的两个加数，我们得到两个构造函数：

```
data FreeA f x where
DoneA :: x -> FreeA f x
MoreA :: ((a, b) -> x) -> f a -> FreeA f b -> FreeA f x
```

我将 Day 卷积的定义内联化了：

```
data Day f g x where
Day :: ((a, b) -> x) -> f a -> g b -> Day f g x
```

最简单的方式展示 `FreeA f` 是一个应用函子是通过 `Monoidal`：

```
class Monoidal f where
unit  :: f ()
(>*<) :: f a -> f b -> f (a, b)
```

由于 `FreeA f` 是列表的泛化，自由应用函子的 `Monoidal` 实例泛化了列表连接的概念。我们对第一个列表进行模式匹配，导致两个情况。

在第一种情况下，代替一个空列表，我们有 `DoneA x`。将其附加到第二个参数中不会改变列表的长度，但会修改存储在其中的值的类型。它将每个值与 `x` 配对：

```
(DoneA x) >*< fry = fmap (x,) fry
```

第二种情况是一个列表，其头部 `fa` 是 `a` 的函子，尾部 `frb` 是类型 `FreeA f b`。这两者通过一个函数 `abx :: (a, b) -> x` 粘合在一起。

```
(MoreA abx fa frb) >*< fry = MoreA (reassoc abx) fa (frb >*< fry)
```

为了生成结果，我们使用对 `>*<` 的递归调用连接两个尾部，并将 `fa` 附加到它上面。为了将这个头部粘合到新的尾部，我们必须提供一个重新关联这些对的函数：

```
reassoc :: ((a, b)-> x) -> (a, (b, y)) -> (x, y)
reassoc abx (a, (b, y)) = (abx (a, b), y)
```

完整的实例如下：

```
instance Functor f => Monoidal (FreeA f) where
unit = DoneA ()
(DoneA x) >*< fry = fmap (x,) fry
(MoreA abx fa frb) >*< fry = MoreA (reassoc abx) fa (frb >*< fry)
```

一旦我们有了 `Monoidal` 实例，就很容易生成 `Applicative` 实例：

```
instance Functor f => Applicative (FreeA f) where
pure a = DoneA a
ff <*> fx = fmap app (ff >*< fx)

app :: (a -> b, a) -> b
app (f, a) = f a
```

Exercise 17.7.3. 为自由应用函子定义 *Functor* 实例。

17.8 伴随态射的双范畴（The Bicategory of Profunctors）

既然我们已经知道如何使用余极限（coends）来组合伴随态射（profunctors），那么问题就来了：是否存在一个以这些伴随态射作为态射的范畴？答案是肯定的，只要我们稍微放松一下规则。问题在于，伴随态射组合的范畴定律并不能严格满足，而是只能满足同构意义上的定律。

例如，我们可以尝试展示伴随态射组合的结合律。我们从以下开始：

$$((P \diamond Q) \diamond R)\langle s,t\rangle = \int^b \left(\int^a P\langle s,a\rangle \quad Q\langle a,b\rangle \right) \quad R\langle b,t\rangle$$

然后经过一些转换，得到：

$$(P \diamond (Q \diamond R))\langle s,t\rangle = \int^a P\langle s,a\rangle \quad \left(\int^b Q\langle a,b\rangle \quad R\langle b,t\rangle \right)$$

我们使用了乘积的结合性和富比尼定理（Fubini theorem）中可以交换余极限的顺序。两者都只能在同构的意义上成立。我们不能获得严格的结合律。

恒等伴随态射实际上是同态函子（hom-functor），它可以符号化地表示为 $\mathcal{C}(*,=)$，其中两个参数为占位符。例如：

$$(\mathcal{C}(*,=) \diamond P)\,\langle s,t\rangle = \int^a C(s,a) \quad P\langle a,t\rangle \,\ddot{\text{o}}\, P\langle s,t\rangle$$

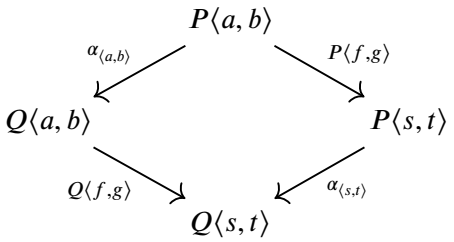
这是（逆变）Ninja 共代数引理（ninja co-Yoneda lemma）的结果，这也是一个同构，而不是一个等式。

一个范畴，如果其范畴定律仅在同构意义上满足，则称为双范畴（bicategory）。请注意，这样的范畴必须配备 2-态射（2-cells）——即态射之间的态射，这在 2-范畴的定义中已经见过。我们需要这些 2-态射来定义 1-态射之间的同构。

一个双范畴 Prof 具有（小）范畴作为对象，伴随态射作为 1-态射，自然变换作为 2-态射。

由于伴随态射是从 $\mathcal{C}^{op} \;\; \mathcal{D} \to \mathrm{Set}$ 的函子，适用于它们的自然变换的标准定义是一个由 $\mathcal{C}^{op} \;\; \mathcal{D}$ 的对象参数化的函数族，这些对象本身是对象的对。

对于两个伴随态射 P 和 Q 之间的变换 $\alpha_{\langle a,b\rangle}$，其自然性条件如下形式：



对于每对箭头：

$$\langle f : s \to a, g : b \to t \rangle$$

双范畴中的单子（Monads in a Bicategory）

我们之前已经看到，范畴、函子和自然变换构成了一个 2-范畴 Cat。让我们关注一个对象，即一个 0-态射 Cat 中的范畴 $\mathcal{C}$。从这个对象出发并终止于它的 1-态射构成了一个常规范畴，在本例中，它是函子范畴 $[\mathcal{C}, \mathcal{C}]$。这个范畴中的对象是外部 2-范畴 Cat 的端 1-态射（endo-1-cells）。它们之间的箭头是外部 2-范畴的 2-态射。

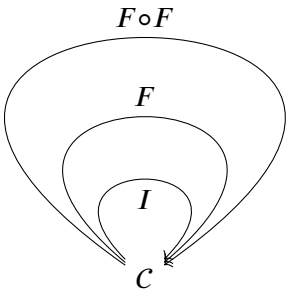
这个端 1-态射范畴自动配备了单胚结构。我们简单地定义张量积为 1-态射的组合——所有具有相同源和目标的 1-态射可以组合。单胚单位对象是恒等 1-态射 I。在 $[\mathcal{C}, \mathcal{C}]$ 中，这个乘积是端函子的组合，单位是恒等函子。

如果我们现在只关注一个端 1-态射 F，我们可以“平方”它，即使用单胚乘积将其自身相乘。换句话说，使用 1-态射组合来创建 $F \circ F$。我们说，如果我们可以找到 2-态射：

$$\mu : F \circ F \to F$$

$$\eta : I \to F$$

使得它们像乘法和单位一样，使得结合性和单位图表交换，那么 F 就是一个单子。



事实上，一个单子可以在任意双范畴中定义，而不仅仅是 2-范畴 Cat 中。

在 Prof 中的前箭头作为单子（Prearrows as Monads in Prof）

由于 Prof 是一个双范畴，我们可以在其中定义一个单子。它是一个端伴随态射（端 1-态射）：

$$P\colon \mathcal{C}^{op}\ \ \mathcal{C} \to \mathrm{Set}$$

配备两个自然变换（2-态射）：

$$\mu\colon P \diamond P \to P$$

$$\eta\colon \mathcal{C}(*,=) \to P$$

满足结合性和单位条件。

让我们将这些自然变换视为余极限中的元素。例如：

$$\mu \in \int_{\langle a,b\rangle} \mathrm{Set}\Big(\int^{x} P\langle a,x\rangle\ \ P\langle x,b\rangle, P\langle a,b\rangle\Big)$$

通过余连续性（co-continuity），这等价于：

$$\int_{\langle a,b\rangle,x} \mathrm{Set}\Big(P\langle a,x\rangle\ \ P\langle x,b\rangle, P\langle a,b\rangle\Big)$$

同样，单位自然变换是：

$$\eta \in \int_{\langle a,b\rangle} \mathrm{Set}(\mathcal{C}(a,b), P\langle a,b\rangle)$$

在 Haskell 中，这样的伴随态射单子被称为前箭头（pre-arrows）：

```
class Profunctor p => PreArrow p where
(>>>) :: p a x -> p x b -> p a b
arr   :: (a -> b) -> p a b
```

一个 `Arrow` 是一个 `PreArrow`，它同时也是一个 Tambara 模块。我们将在下一章讨论 Tambara 模块。

17.9 存在透镜（Existential Lens）

范畴论俱乐部的第一条规则是你不谈论对象的内部结构。

范畴论俱乐部的第二条规则是，如果你必须谈论对象的内部结构，请只使用箭头。

Haskell 中的存在透镜（Existential Lens in Haskell）

什么意味着一个对象是复合的——即它具有部分？至少，你应该能够检索该对象的某个部分。如果你还能用一个新的部分替换这个部分，那就更好了。这几乎定义了一个透镜（lens）：

```
get :: s -> a
set :: s -> a -> s
```

这里，`get` 从整体 `s` 中提取部分 `a`，而 `set` 用一个新的 `a` 替换该部分。透镜定律有助于加强这一概念。而且这一切都是通过箭头完成的。

描述复合对象的另一种方法是说它可以被分割为焦点和剩余部分。诀窍在于，虽然我们想知道焦点的类型，但我们不关心剩余部分的类型。我们只需要知道剩余部分可以与焦点结合以重新构建整个对象。

在 Haskell 中，我们使用存在类型来表达这个想法：

```
data LensE s a where
LensE :: (s -> (c, a), (c, a) -> s) -> LensE s a
```

这告诉我们存在某种未指定的类型 `c`，使得 `s` 可以被分割成一个 `(c, a)`，并可以从中重建。

可以从这种存在形式中推导出 `get`/`set` 版本的透镜。

```
toGet :: LensE s a -> (s -> a)
toGet (LensE (l, r)) = snd . l

toSet :: LensE s a -> (s -> a -> s)
toSet (LensE (l, r)) s a = r (fst (l s), a)
```

注意，我们不需要知道剩余部分的类型。我们利用了存在透镜包含了 `c` 的生产者和消费者这一事实，而我们只是调解它们之间的关系。

提取“裸”剩余部分是不可能的，正如以下代码无法编译所证明的那样：

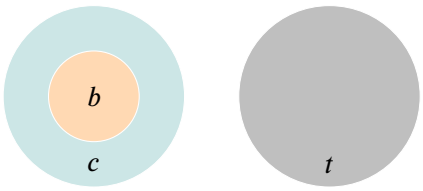
```
getResidue :: Lens s a -> c
getResidue (Lens (l, r)) = fst . l
```

范畴论中的存在透镜（Existential Lens in Category Theory）

我们可以很容易地将透镜的新定义翻译到范畴论中，使用余极限（coend）表达存在类型：

$$\int^{c} \mathcal{C}(s, c \quad a) \quad \mathcal{C}(c \quad a, s)$$

事实上，我们可以将其推广为类型变化透镜，其中焦点 a 可以被替换为新类型 b 的焦点。将 a 替换为 b 会产生一个新的复合对象 t：



透镜现在由两对对象参数化：$\langle s, t\rangle$ 为外部对象对，$\langle a, b\rangle$ 为内部对象对。存在剩余 c 仍然是隐藏的：

$$\mathcal{L}\langle s, t\rangle\langle a, b\rangle = \int^{c} \mathcal{C}(s, c \quad a) \quad \mathcal{C}(c \quad b, t)$$

余极限下的乘积是一个在 y 上协变、在 x 上逆变的伴随态射的对角部分：

$$\mathcal{C}(s, y \quad a) \quad \mathcal{C}(x \quad b, t)$$

Exercise 17.9.1. 证明：

$$\mathcal{C}(s, y \quad a) \quad \mathcal{C}(x \quad b, t)$$

是一个在 $\langle x, y\rangle$ 上的伴随态射。

Haskell 中的类型变化透镜（Type-changing Lens in Haskell）

在 Haskell 中，我们可以定义一个类型变化透镜为以下存在类型：

```
data LensE s t a b where
LensE :: (s -> (c, a)) -> ((c, b) -> t) -> LensE s t a b
```

和之前一样，我们可以使用存在透镜来获取和设置焦点：

```
toGet :: LensE s t a b -> (s -> a)
toGet (LensE l r) = snd . l

toSet :: LensE s t a b -> (s -> b -> t)
toSet (LensE l r) s a = r (fst (l s), a)
```

这两个函数，`s->(c, a)` 和 `(c, b)->t`，通常被称为前向和后向传递。前向传递可以用于提取焦点 `a`。后向传递提供了一个问题的答案：如果我们希望前向传递的结果是另一个 `b`，我们应该传递给它什么 `t`？

有时我们只是问：如果我们希望通过 `b` 更改焦点，我们应该对输入进行什么更改 `t`？这种观点在使用透镜描述神经网络（neural networks）时特别有用。

透镜的最简单示例作用于乘积。它可以提取或替换乘积的一个分量，将另一个分量视为剩余部分。在 Haskell 中，我们将其实现为：

```
prodLens :: LensE (c, a) (c, b) a b
prodLens = LensE id id
```

在这里，整体的类型是乘积 `(c, a)`。当我们将 `a` 替换为 `b` 时，我们最终得到目标类型 `(c, b)`。由于源和目标已经是乘积，存在透镜定义中的两个函数只是恒等函数。

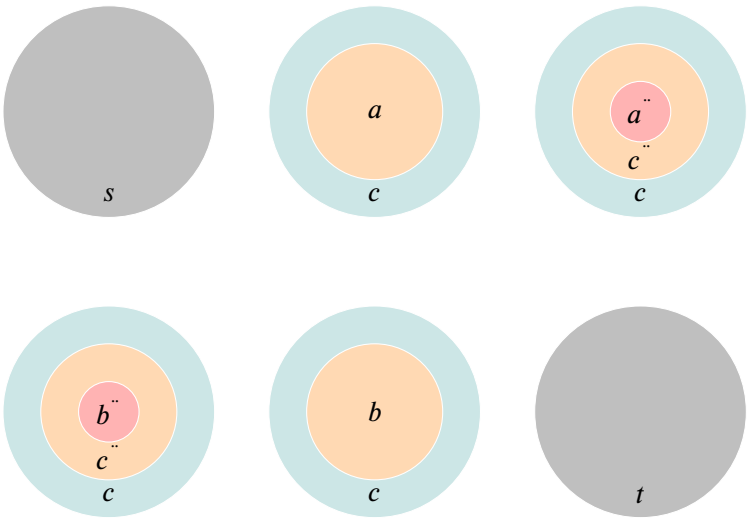
透镜组合（Lens Composition）

使用透镜的主要优点在于它们可以组合。两个透镜的组合使我们可以放大到某个分量的子分量。

假设我们从一个透镜开始，它使我们可以访问焦点 `a` 并将其更改为 `b`。这个焦点是整体的一部分，由源 `s` 和目标 `t` 描述。

我们还有一个内部透镜，它可以访问 `a` 内的焦点 `a'`，并将其替换为 `b'`，从而为我们提供一个新的 `b`。

现在我们可以构造一个组合透镜，使我们能够在 `s` 和 `t` 中访问 `a'` 和 `b'`。诀窍是意识到我们可以将两个剩余部分的乘积作为新的剩余部分：




```
compLens :: LensE a b a' b' -> LensE s t a b -> LensE s t a' b'
compLens (LensE l2 r2) (LensE l1 r1) = LensE l3 r3
```

```
  where l3 = assoc' . bimap id l2  . l1
  r3 = r1 . bimap id r2 . assoc
```

新透镜中的左映射由以下组合给出：

$$s \xrightarrow{l_1} (c, a) \xrightarrow{(id, l_2)} (c, (c'', a'')) \xrightarrow{assoc''} ((c, c''), a'')$$

右映射由以下公式给出：

$$((c, c''), b'') \xrightarrow{assoc} (c, (c'', b'')) \xrightarrow{(id, r_2)} (c, b) \xrightarrow{r_1} t$$

我们使用了乘积的结合性和函子性：

```
assoc :: ((c, c'), b') -> (c, (c', b'))
assoc ((c, c'), b') = (c, (c', b'))

assoc' :: (c, (c', a')) -> ((c, c'), a')
assoc' (c, (c', a')) = ((c, c'), a')

instance Bifunctor (,) where
bimap f g (a, b) = (f a, g b)
```

作为示例，让我们组合两个乘积透镜：

```
l3 :: LensE (c, (c', a')) (c, (c', b')) a' b'
l3 = compLens prodLens prodLens
```

并将其应用于嵌套的乘积：

```
x :: (String, (Bool, Int))
x = ("Outer", (True, 42))
```

我们的组合透镜不仅让我们检索最内层的组件：

```
toGet l3 x
> 42
```

还可以用不同类型的值（这里是 Char）替换它：

```
toSet l3 x 'z'
> ("Outer",(True,'z'))
```

透镜的范畴（Category of Lenses）

由于透镜可以组合，你可能会想知道是否存在一个范畴，其中透镜作为同态集（hom-sets）。

确实存在一个范畴 Lens，其对象是 $\mathcal{C}$ 中对象的对，而从 $\langle s,t\rangle$ 到 $\langle a,b\rangle$ 的箭头是 $\mathcal{L}\langle s,t\rangle\langle a,b\rangle$ 的元素。

存在透镜的组合公式太复杂，以至于在实际操作中难以使用。在下一章中，我们将看到使用 Tambara 模块表示透镜的另一种表示法，其中组合只是函数的组合。

17.10 透镜与纤维丛 (Lenses and Fibrations)

透镜可以通过纤维丛的语言来理解。定义纤维化的投影 p 可以被看作将丛 E 分解为纤维的过程。

在这种观点下，p 扮演了 `get` 的角色：

$$p\colon E \to B$$

其中 B 代表焦点的类型，而 E 代表可以从中提取出焦点的复合类型。

透镜的另一部分 `set` 是一个映射：

$$q\colon E \;\; B \to E$$

让我们看看如何使用纤维化解释它。

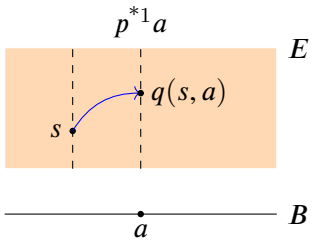
传输定律 (Transport Law)

我们将 q 解释为将丛 E 的一个元素“传输”到新纤维的操作。新纤维由 B 的一个元素指定。

传输的这种特性由 get/set 透镜定律，即传输定律来表达，它说明“你得到的就是你设置的”：

```
get (set s a) = a
```

我们说 $q(s,a)$ 将 s 传输到 a 上的新纤维：

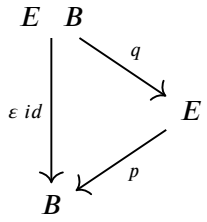


我们可以用 p 和 q 来重新表达这个定律：

$$p \circ q = \pi_2$$

其中 π_2 是从乘积到第二个分量的投影。

等价地，我们可以将其表示为一个交换图：



在这里，我使用了一个余单模余单元 ε 而不是投影 π_2：

$$\varepsilon \colon E \to 1$$

然后应用乘积的单位定律。使用余单模使得这个结构更容易推广到单模范畴中的张量积。

恒等定律 (Identity Law)

这是 set/get 定律或恒等定律。它说明“如果你设置的是你得到的东西，那么不会发生任何变化”：

```
set s (get  s) = s
```

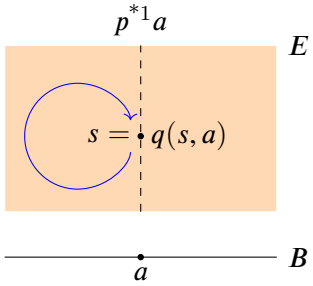
我们可以用一个余单模的余乘法来表达：

$$\delta \colon E \to E \quad E$$

set/get 定律要求以下组合是一个恒等映射：

$$E \xrightarrow{\delta} E \quad E \xrightarrow{id\ p} E \quad B \xrightarrow{q} E$$

下图展示了这个定律在丛中的体现：

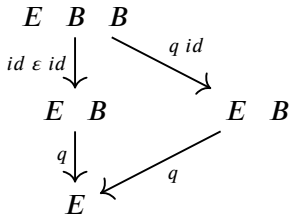


组合定律 (Composition Law)

最后，这是 set/set 定律或组合定律。它说明“最后设置的值获胜”：

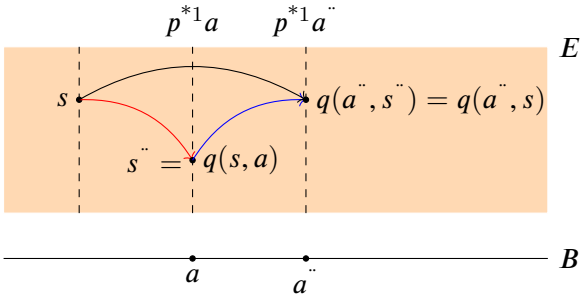
```
  set (set s a) a' = set s a'
```

对应的交换图如下：



同样地，为了去除中间的 B，我使用了余单元而不是从乘积中的投影。

这就是 set/set 定律在丛中的样子：

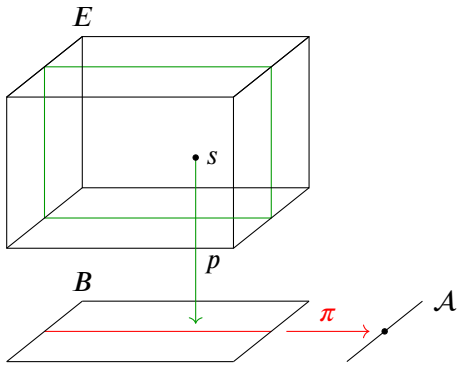


类型转换透镜 (Type-changing Lens)

类型转换透镜将传输泛化为在丛之间操作。我们必须定义一整个丛的家族。我们从一个范畴 $\mathcal{A}$ 开始，其对象定义了我们将用于透镜焦点的类型。

我们构造集合 B 作为所有焦点类型的所有元素的联合集合。B 在 $\mathcal{A}$ 上纤维化——投影 π 将 B 的一个元素发送到其对应的类型。你可以将 B 视为余切范畴 $1/\mathcal{A}$ 的对象集合。

丛的丛 E 是一个在 B 上纤维化的集合，其投影为 p。由于 B 本身在 $\mathcal{A}$ 上纤维化，E 在 $\mathcal{A}$ 上传递地纤维化，投影为复合投影 $\pi \circ p$。正是这种更粗的纤维化将 E 分解为一个丛的家族。每个丛对应于给定焦点类型的复合类型的不同类型。类型转换透镜将在这些丛之间移动。



投影 p 将元素 s Ë E 转换为一个元素 b Ë B，其类型由 πb 给出。这是 `get` 的推广。

对应于 `set` 的传输 q 接收一个元素 s Ë E 和一个元素 b Ë B，并生成一个新元素 t Ë E。重要的是要注意，s 和 t 可以属于 E 的不同子丛。

传输满足以下定律：

get/set 定律（传输）：

$$p(q(b,s)) = b$$

set/get 定律（恒等）：

$$q(p(s),s) = s$$

set/set 定律（组合）：

$$q(c,q(b,s)) = q(c,s)$$

17.11 重要公式 (Important Formulas)

这是一个便捷的（共）端计算速查表。

- Hom-函子连续性 (Continuity of the hom-functor):

$$\mathcal{D}\left(d, \int_a P\langle a,a\rangle\right) \text{ ö } \int_a \mathcal{D}\left(d, P\langle a,a\rangle\right)$$

- Hom-函子共连续性 (Co-continuity of the hom-functor):

$$\mathcal{D}\left(\int^a P\langle a,a\rangle, d\right) \text{ ö } \int_a \mathcal{D}\left(P\langle a,a\rangle, d\right)$$

- 忍者 Yoneda (Ninja Yoneda):

$$\int_x \mathrm{Set}(\mathcal{C}(a,x), Fx) \text{ ö } Fa$$

- 忍者 Co-Yoneda (Ninja co-Yoneda):

$$\int^x \mathcal{C}(x,a) \quad Fx \text{ ö } Fa$$

- 反变函子（预层）的忍者 Yoneda (Ninja Yoneda for contravariant functors):

$$\int_x \mathrm{Set}(\mathcal{C}(x,a), Gx) \text{ ö } Ga$$

- 反变函子（预层）的忍者 Co-Yoneda (Ninja co-Yoneda for contravariant functors):

$$\int^{x} C(a,x) \quad Gx \ \ddot{\text{o}}\ Ga$$

- Day 卷积 (Day Convolution):

$$(F \star G)x = \int^{a,b} C(a \otimes b, x) \quad Fa \quad Gb$$

Chapter 18

Tambara 模块 (Tambara Modules)

在编程中，范畴论的一个不太知名的领域突然获得了新的关注：Tambara 模块在 profunctor optics 中得到了新的应用。它们为 optics 的组合问题提供了一个巧妙的解决方案。我们已经看到，对于透镜 (lenses) 来说，getter 可以通过函数组合来很好地组合，但 setter 的组合则涉及一些复杂操作。存在的表示法帮助不大。而使用 profunctor 表示法，组合变得非常简单。

这种情况类似于图形编程中组合几何变换的问题。例如，如果你尝试组合绕两个不同轴的旋转，新轴和角度的公式会非常复杂。但如果将旋转表示为矩阵，你可以使用矩阵乘法；或者将其表示为四元数 (quaternions)，你可以使用四元数乘法。profunctor 表示法让你可以使用简单的函数组合来组合 optics。

18.1 Tannakian 重建 (Tannakian Reconstruction)

幺半群与其表示 (Monoids and their Representations)

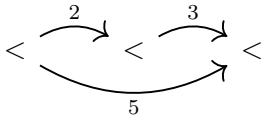
表示论本身就是一门科学。在这里，我们将从范畴论的角度来探讨它。我们将考虑幺半群 (monoids) 而不是群 (groups)。幺半群可以定义为单对象范畴 $\mathcal{M}$ 中的一个特殊对象。如果我们将其对象称为 $<$，那么同态集 $\mathcal{M}(<,<)$ 包含了我们所需的所有信息。

在幺半群中称为“乘积”的操作在这里被形态组合取代。根据范畴的定律，它是结合的，并且具有单位元——即身份形态。

在这个意义上，每个单对象范畴自动是一个幺半群，并且所有幺半群都可以变为单对象范畴。

例如，可以将全体整数的幺半群（加法）看作一个只有一个抽象对象 $<$ 的范畴，并且每个整数对应一个形态。要组合两个这样的形态，你只需将它们的数值相加，如下图

所示：



与零对应的形态自动是身份形态。

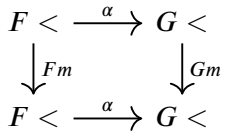
我们可以将幺半群*表示*为集合的变换。这样的表示是由函子 $F\colon \mathcal{M} \to \mathrm{Set}$ 给出的。它将单个对象 $<$ 映射到某个集合 S，并将同态集 $\mathcal{M}(<,<)$ 映射到一组 $S \to S$ 的函数。根据函子定律，它将身份形态映射为身份函数，并将组合映射为组合，因此它保持了幺半群的结构。

如果函子是完全忠实的 (fully faithful)，那么其像包含与幺半群相同的信息，且没有其他信息。然而，通常情况下，函子会“作弊”。同态集 $\mathrm{Set}(S,S)$ 可能包含一些不在 $\mathcal{M}(<,<)$ 像中的函数；而且 $\mathcal{M}$ 中的多个形态可能映射到同一个函数。

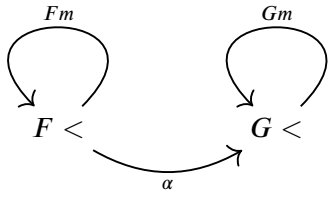
在极端情况下，整个同态集 $\mathcal{M}(<,<)$ 可能被映射到身份函数 id_S。所以，仅仅通过查看集合 S——即函子 F 下的 $<$ 的像——我们无法重建原始的幺半群。

尽管如此，如果我们可以同时查看给定幺半群的所有表示，我们可能还有机会。这些表示形成一个范畴——函子范畴 $[\mathcal{M},\mathrm{Set}]$，也称为 $\mathcal{M}$ 上的余预层范畴 (co-presheaf category)。这个范畴中的箭头是自然变换 (natural transformations)。

由于源范畴 $\mathcal{M}$ 只包含一个对象，自然性条件变得特别简单。自然变换 $\alpha\colon F \to G$ 只有一个分量，即函数 $\alpha\colon F\,< \to G\,<$。给定一个形态 $m\colon <\to<$，自然性方块如下：



这是在两个集合上作用的三个函数之间的关系：



自然性条件告诉我们：

$$Gm \circ \alpha = \alpha \circ Fm$$

换句话说，如果你选择集合 $F\,<$ 中的任意元素 x，你可以使用 α 将它映射到 $G\,<$，然后应用与 m 对应的变换 Gm；或者你可以先应用变换 Fm，然后使用 α 映射结果。结果必须相同。

这样的函数被称为 *等变函数*。我们通常称 Fm 为幺半群在集合 $F\,<$ 上的 *作用*。等变函数通过前置组合或后置组合将一个集合上的作用连接到另一个集合上的对应作用。

幺半群的 Tannakian 重建 (Tannakian Reconstruction of a Monoid)

我们需要多少信息才能从幺半群的表示中重建一个幺半群？仅仅查看集合显然不够，因为任何幺半群都可以在任意集合上表示。但如果我们包括这些集合之间的结构保持函数，我们可能有机会。

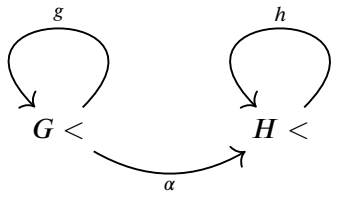
给定一个函子 $F\colon \mathcal{M} \to \mathrm{Set}$，考虑所有函数 $F\ < \to F\ <$ 的集合，即同态集 $\mathrm{Set}(F\ <, F\ <)$。至少其中一些函数是幺半群的作用。这些函数的形式为 Fm，其中 m 是 $\mathcal{M}$ 中的一个箭头。不过，必须注意，这个同态集中可能有许多不对应于作用的函数。

现在让我们看看另一个集合 $G\ <$，它是某个其他函子 G 的像。在它的同态集 $\mathrm{Set}(G\ <, G\ <)$ 中，我们会发现（包括其他的）形式为 Gm 的对应作用。一个等变函数，即 $[\mathcal{M}, \mathrm{Set}]$ 中的自然变换，将连接任何两个相关的作用 Fm 和 Gm。

现在，想象创建一个巨大的元组，其中每个元素是从所有函子 $F\colon \mathcal{M} \to \mathrm{Set}$ 的同态集 $\mathrm{Set}(F\ <, F\ <)$ 中选择的一个函数。我们只对那些彼此连接的元组感兴趣。这里我的意思是：如果我们选择 g Ë $\mathrm{Set}(G\ <, G\ <)$ 和 h Ë $\mathrm{Set}(H\ <, H\ <)$，并且在函子 G 和 H 之间存在一个自然变换（等变函数）α，我们希望这两个函数是相关的：

$$\alpha \circ g = h \circ \alpha$$

或者，图示如下：

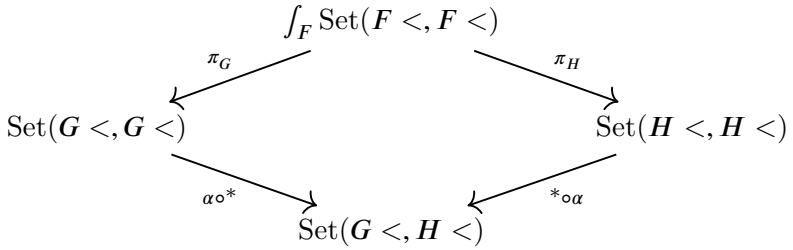


注意，这种相关性也适用于不形式为 $g = Gm$ 和 $h = Hm$ 的 g 和 h 对。

这样的元组正是端的元素：

$$\int_F \mathrm{Set}(F\ <, F\ <)$$

其楔条件 (wedge condition) 提供了我们所寻找的约束。



这里 α 是函子范畴 $[\mathcal{M}, \mathrm{Set}]$ 中的一个形态：

$$\alpha\colon G \to H$$

这个自然变换只有一个分量，我们也称它为 α。它是两个表示之间的等变函数。

这里有一些细节。端下的 profunctor 由下式给出：

$$P\langle G, H\rangle = \mathrm{Set}(G <, H <)$$

它是以下形式的函子：

$$P\colon [\mathcal{M}, \mathrm{Set}]^{op} \quad [\mathcal{M}, \mathrm{Set}] \to \mathrm{Set}$$

考虑它在 $[\mathcal{M}, \mathrm{Set}]$ 中形态对上的作用。给定一对自然变换：

$$\alpha\colon G^{\cdot\cdot} \to G$$
$$\beta\colon H \to H^{\cdot\cdot}$$

它们的提升是一个函数：

$$P\langle\alpha, \beta\rangle\colon P\langle G, H\rangle \to P\langle G^{\cdot\cdot}, H^{\cdot\cdot}\rangle$$

代入我们对 P 的定义，我们有：

$$P\langle\alpha, \beta\rangle\colon \mathrm{Set}(G <, H <) \to \mathrm{Set}(G^{\cdot\cdot} <, H^{\cdot\cdot} <)$$

我们通过与 α 的前置组合和与 β 的后置组合来得到这个函数（这些函数是两个自然变换 α 和 β 的唯一分量）：

$$P\langle\alpha, \beta\rangle = \beta \circ - \circ \alpha$$

即，给定一个函数 f Ë $\mathrm{Set}(G <, H <)$，我们生成一个函数 $\beta \circ f \circ \alpha$ Ë $\mathrm{Set}(G^{\cdot\cdot} <, H^{\cdot\cdot} <)$。

在楔条件中，如果我们选择 g 为 $\mathrm{Set}(G <, G <)$ 的元素，h 为 $\mathrm{Set}(H <, H <)$ 的元素，我们就重现了我们的条件：

$$\alpha \circ g = h \circ \alpha$$

Tannakian 重建定理在这种情况下告诉我们：

$$\int_F \mathrm{Set}(F <, F <) \ddot{\mathrm{o}}\ \mathcal{M}(<, <)$$

换句话说，我们可以从幺半群的表示中恢复幺半群。我们将在更一般的陈述中看到这个定理的证明。

Cayley 定理 (Cayley's Theorem)

在群论中，Cayley 定理指出每个群都同构于一个置换群的子群。一个 *群*只是一个幺半群，其中每个元素都有一个逆元素。置换是将一个集合映射到它自身的双射函数。它们 *置换*集合中的元素。

在范畴论中，Cayley 定理几乎内置于幺半群及其表示的定义中。

在单对象解释和幺半群的更传统的元素集解释之间建立联系非常容易。我们通过构造函子 $F\colon \mathcal{M} \to \mathrm{Set}$ 来实现这一点，该函子将 $<$ 映射到一个特殊集合 S，该集合等于同态集：$S = \mathcal{M}(<,<)$。这个集合的元素与 $\mathcal{M}$ 中的形态相对应。我们将 F 在形态上的作用定义为后置组合：

$$(Fm)n = m \circ n$$

这里 m 是 $\mathcal{M}$ 中的一个形态，而 n 是集合 S 的一个元素，恰好也是 $\mathcal{M}$ 中的一个形态。

我们可以将这种特定表示作为 Set 中幺半群的一种替代定义。我们所需要的只是实现单位元和乘法：

$$\eta\colon 1 \to S$$
$$\mu\colon S \quad S \to S$$

单位元选择与 $\mathcal{M}(<,<)$ 中的 $id_<$ 对应的 S 元素。两个元素 m 和 n 的乘法由与 $m \circ n$ 对应的元素给出。

同时，我们可以将 S 视为 $F\colon \mathcal{M} \to \mathrm{Set}$ 的像，在这种情况下，形成幺半群表示的是 $S \to S$ 的函数。这就是 Cayley 定理的本质：每个幺半群都可以用一组自函子表示。

在编程中，Cayley 定理的最佳应用示例是高效实现列表反转。回想一下反转的简单递归实现：

```
reverse :: [a] -> [a]
reverse [] = []
reverse (a : as) = reverse as ++ [a]
```

它将列表分为头和尾，反转尾部，然后将头组成的单元素列表追加到结果中。问题是每次追加都必须遍历不断增长的列表，导致 $O(N^2)$ 的性能。

然而，请记住，列表是一个（自由）幺半群：

```
instance Monoid [a] where
mempty = []
mappend as bs = as ++ bs
```

我们可以使用 Cayley 定理将这个幺半群表示为列表上的函数：

```
type DList a = [a] -> [a]
```

为了表示一个列表，我们将其转换为一个函数。它是一个函数（闭包），将这个列表 `as` 追加到其参数 `xs` 中：

```
rep :: [a] -> DList a
rep as = \xs -> as ++ xs
```

这种表示称为 *差异列表*。

要将一个函数还原为列表，只需将其应用于一个空列表：

```
unRep :: DList a -> [a]
unRep f = f []
```

很容易验证，空列表的表示是一个恒等函数，而两个列表的连接表示是表示的组合：

```
rep [] = id
rep (xs ++ ys) = rep xs . rep ys
```

所以这正是列表幺半群的 Cayley 表示。

现在我们可以翻译反转算法以生成这个新表示：

```
rev :: [a] -> DList a
rev [] = rep []
rev (a : as) = rev as . rep [a]
```

然后将其还原为列表：

```
fastReverse :: [a] -> [a]
fastReverse = unRep . rev
```

乍一看，这似乎没有做太多，除了在我们的递归算法之上添加了一个转换层。除了新的算法具有 $O(N)$ 而不是 $O(N^2)$ 的性能。要看清这一点，考虑反转一个简单列表 `[1, 2, 3]`。函数 `rev` 将这个列表变成了一个函数组合：

```
rep [3] . rep [2] . rep [1]
```

它以线性时间完成这一操作。函数 `unRep` 从空列表开始执行这个组合。但请注意，每个 `rep` 都会将其参数前置到累积结果中。特别是，最后的 `rep [3]` 执行了：

```
[3] ++ [2, 1]
```

与追加不同，前置是一个常数时间操作，因此整个算法的时间复杂度为 $O(N)$。

另一种看待它的方式是，认识到 `rev` 按照列表元素的顺序从头部开始排列操作队列。但函数队列按先进先出 (FIFO) 顺序执行。

由于 Haskell 的惰性，使用 `foldl` 的列表反转具有类似的性能：

```
reverse = foldl (\as a -> a : as) []
```

这是因为 `foldl` 在返回结果之前，从左到右遍历列表，累积函数（闭包）。然后根据需要按 FIFO 顺序执行它们。

Tannakian 重建的证明 (Proof of Tannakian Reconstruction)

幺半群重建是一个更一般定理的特例，其中我们使用了一个常规范畴而不是单对象范畴。与幺半群的情况一样，我们将重建同态集，只是这次它是两个对象之间的常规同态

集。我们将证明公式：

$$\int_{F\colon [\mathcal{C},\mathrm{Set}]} \mathrm{Set}(Fa, Fb) \ddot{\circ} \mathcal{C}(a,b)$$

技巧是使用 Yoneda 引理来表示 F 的作用：

$$Fa \ddot{\circ} [\mathcal{C},\mathrm{Set}](\mathcal{C}(a,*),F)$$

同样地，我们也可以表示 Fb。我们得到：

$$\int_{F\colon [\mathcal{C},\mathrm{Set}]} \mathrm{Set}([\mathcal{C},\mathrm{Set}](\mathcal{C}(a,*),F),[\mathcal{C},\mathrm{Set}](\mathcal{C}(b,*),F))$$

注意，这里的两个自然变换集是 $[\mathcal{C},\mathrm{Set}]$ 中的同态集。

回想一下 Yoneda 引理的推论，它适用于任何范畴 $\mathcal{A}$：

$$[\mathcal{A},\mathrm{Set}](\mathcal{A}(x,*),\mathcal{A}(y,*)) \ddot{\circ} \mathcal{A}(y,x)$$

我们可以用端 (end) 来表示它：

$$\int_{z\colon \mathcal{C}} \mathrm{Set}(\mathcal{A}(x,z),\mathcal{A}(y,z)) \ddot{\circ} \mathcal{A}(y,x)$$

特别地，我们可以用函子范畴 $[\mathcal{C},\mathrm{Set}]$ 替换 $\mathcal{A}$。我们得到：

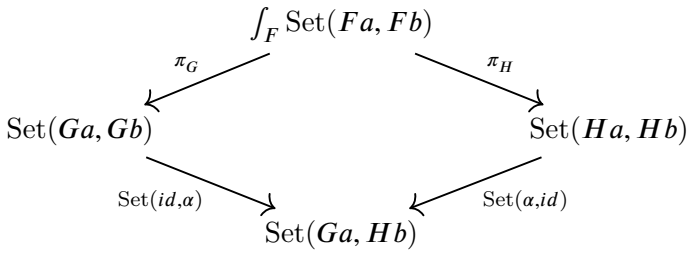
$$\int_{F\colon [\mathcal{C},\mathrm{Set}]} \mathrm{Set}([\mathcal{C},\mathrm{Set}](\mathcal{C}(a,*),F),[\mathcal{C},\mathrm{Set}](\mathcal{C}(b,*),F)) \ddot{\circ} [\mathcal{C},\mathrm{Set}](\mathcal{C}(b,*),\mathcal{C}(a,*))$$

然后我们可以再次应用 Yoneda 引理到右侧，得到：

$$\mathcal{C}(a,b)$$

这正是我们所寻找的结果。

重要的是要意识到函子范畴的结构如何通过楔条件进入端。它通过自然变换实现。每当我们在两个函子 $\alpha\colon G \to H$ 之间有一个自然变换时，以下图表必须交换：



要对 Tannakian 重建有一些直观理解，你可以回忆起 Set 值函子可以解释为与证据相关的子集。函子 $F\colon \mathcal{C} \to \mathrm{Set}$（一个余预层）定义了一个小范畴 $\mathcal{C}$ 的对象子集。我们说某个对象 a 属于这个子集当且仅当 Fa 是非空的。然后 Fa 的每个元素都可以解释为该子集的证据。

但是，除非所讨论的范畴是离散的，并不是所有子集都对应于余预层。特别是，每当存在一个箭头 $f : a \to b$ 时，也存在一个函数 $Ff : Fa \to Fb$。根据我们的解释，这样的函数将 a 属于 F 定义的子集的每个证据映射为 b 属于该子集的证据。因此，余预层定义了与范畴结构兼容的证据相关子集。

让我们以同样的精神重新解释 Tannakian 重建。

$$\int_{F\,:\,[\mathcal{C},\mathrm{Set}]} \mathrm{Set}(Fa, Fb) \,\ddot{\mathrm{o}}\, \mathcal{C}(a, b)$$

左侧的元素是一个证据，表明对于与 $\mathcal{C}$ 的结构兼容的每个子集，如果 a 属于该子集，那么 b 也属于该子集。这只有在存在从 a 到 b 的箭头时才有可能。

Haskell 中的 Tannakian 重建 (Tannakian Reconstruction in Haskell)

我们可以立即将这个结果翻译到 Haskell 中。我们用 `forall` 替换端。左侧变为：

```
forall f. Functor f => f a -> f b
```

右侧则是函数类型 `a->b`。

我们之前已经见过多态函数：它们是为所有类型或某些类型类定义的函数。这里我们有一个为所有函子定义的函数。它说：给我一个装有 `a` 的函子，我将产生一个装有 `b` 的函子——无论你使用什么函子。唯一可以实现它的方法（使用参数多态性）是这个函数秘密地捕获了一个类型为 `a->b` 的函数，并使用 `fmap` 进行应用。

事实上，同构的一方向就是这样：捕获一个函数并在参数上 `fmap` 它：

```
toRep :: (a -> b) -> (forall f. Functor f => f a -> f b)
toRep g fa = fmap g fa
```

另一个方向使用了 Yoneda 技巧：

```
fromRep :: (forall f. Functor f => f a -> f b) -> (a -> b)
fromRep g a = unId (g (Id a))
```

其中恒等函子 (identity functor) 定义如下：

```
data Id a = Id a

unId :: Id a -> a
unId (Id a) = a

instance Functor Id where
fmap g (Id a) = Id (g a)
```

这种重建看起来可能显得微不足道且无意义。为什么有人会想要将函数类型 `a->b` 替换为一个复杂得多的类型：

```
type Getter a b = forall f. Functor f => f a -> f b
```

不过，有趣的是，将 a->b 视为所有 optics 的前身是很有意义的。它是一个透镜，聚焦于 a 中的 b 部分。它告诉我们 a 包含了足够的信息，以某种形式或其他形式构造 b。它是一个“getter”或“访问器 (accessor)”。

显然，函数是可以组合的。不过有趣的是，函子表示也可以组合，并且它们通过简单的函数组合来组合，如下例所示：

```
boolToStrGetter :: Getter Bool String
boolToStrGetter = toRep (show) . toRep (bool 0 1)
```

其他 optics 的组合就不那么简单了，但它们的函子（和 profunctor）表示组合却非常简单。

带伴随的 Tannakian 重建 (Tannakian Reconstruction with Adjunction)

广义的 Tannakian 重建技巧是在某个专业函子范畴 $\mathcal{T}$ 上首先应用遗忘函子，然后定义端。我们假设在两个函子范畴 $\mathcal{T}$ 和 $[\mathcal{C}, \mathrm{Set}]$ 之间存在自由/遗忘伴随 $F \dashv U$：

$$\mathcal{T}(FQ, P) \ddot{\circ} [\mathcal{C}, \mathrm{Set}](Q, UP)$$

其中 Q 是 $[\mathcal{C}, \mathrm{Set}]$ 中的一个函子，P 是 $\mathcal{T}$ 中的一个函子。

我们用于 Tannakian 重建的起点是以下端：

$$\int_{P\colon \mathcal{T}} \mathrm{Set}\Big((UP)a, (UP)s\Big)$$

顺便说一下，参数化为对象 a 的映射 $\mathcal{T} \to \mathrm{Set}$ 由以下公式给出：

$$P \mapsto (UP)a$$

有时称为纤维函子，因此端公式可以解释为两个纤维函子之间的自然变换集。概念上，纤维函子描述了一个对象的“微分邻域”。它将函子映射到集合，但更重要的是，它将自然变换映射到函数。这些函数探测对象所处的环境。特别是 $\mathcal{T}$ 中的自然变换参与定义端的楔条件。（在微积分中，层的茎 (stalks) 扮演着非常类似的角色。）

像以前一样，我们首先应用 Yoneda 引理，得到：

$$\int_{P\colon \mathcal{T}} \mathrm{Set}\Big([\mathcal{C}, \mathrm{Set}]\Big(\mathcal{C}(a, *), UP\Big), [\mathcal{C}, \mathrm{Set}]\Big(\mathcal{C}(s, *), UP\Big)\Big)$$

现在我们可以使用伴随：

$$\int_{P\colon \mathcal{T}} \mathrm{Set}\Big(\mathcal{T}\Big(F\mathcal{C}(a, *), P\Big), \mathcal{T}\Big(F\mathcal{C}(s, *), P\Big)\Big)$$

我们最终得到了两个函子范畴 $\mathcal{T}$ 中自然变换之间的映射。我们可以使用 Yoneda 引理的推论来进行积分，得到：

$$\mathcal{T}\Big(F\mathcal{C}(s, *), F\mathcal{C}(a, *)\Big)$$

我们可以再次应用伴随：

$$\mathrm{Set}\Big(\mathcal{C}(s,*),(U\circ F)\mathcal{C}(a,*)\Big)$$

并再次应用 Yoneda 引理：

$$\Big((U\circ F)\mathcal{C}(a,*)\Big)s$$

最后的观察是，伴随函子的复合 $U\circ F$ 是函子范畴中的一个幺半群。我们将这个幺半群称为 Φ。结果是以下等式，它将作为 profunctor optics 的基础：

$$\int_{P\colon \mathcal{T}} \mathrm{Set}\Big((U\,P)a,(U\,P)s\Big) \ddot{\mathrm{o}} \Big(\Phi\mathcal{C}(a,*)\Big)s$$

右侧是幺半群 $\Phi = U\circ F$ 对可表示函子 $\mathcal{C}(a,*)$ 的作用，然后在 s 处求值。

将其与早期的 Tannakian 重建公式进行比较，特别是如果我们将其重写为以下形式：

$$\int_{F\colon [\mathcal{C},\mathrm{Set}]} \mathrm{Set}(Fa,Fs) \ddot{\mathrm{o}}\, \mathcal{C}(a,*)s$$

请记住，在 optics 的推导中，我们将 a 和 s 替换为 $\mathcal{C}^{op}\ \ \mathcal{C}$ 中的对象对 $\langle a,b\rangle$ 和 $\langle s,t\rangle$。在这种情况下，我们的函子将变为 profunctor。

Chapter 19

Kan Extensions（Kan 扩展）

如果说范畴论不断提高抽象层次，是因为它的核心在于发现模式。一旦发现了模式，接下来就是研究这些模式所形成的更高层次的模式，依此类推。

我们已经看到相同的概念在更高的抽象层次上以越来越简洁的方式重复出现。

例如，我们最初使用泛范构造（universal construction）定义了积（product）。然后我们看到在积的定义中涉及的跨度（span）实际上是自然变换（natural transformations）。这引导我们将积解释为极限（limit）。接着我们又看到，可以用伴随（adjunctions）来定义它。我们最终能够用一个简洁的公式将积与余积（coproduct）结合起来：

$$(+) \dashv \Delta \dashv (\ )$$

老子说过："将欲取之，必先与之。"

Kan 扩展进一步提高了抽象的层次。Mac Lane 曾说："所有的概念都是 Kan 扩展。"

19.1 Closed Monoidal Categories（闭单态范畴）

我们已经看到，如何将一个函数对象定义为范畴积（categorical product）的右伴随（right adjoint）：

$$\mathcal{C}(a \quad b, c) \ddot{\text{o}} \ \mathcal{C}(a, [b, c])$$

这里我使用了一个替代符号 $[b, c]$ 来表示内部同态（internal hom），即指数（exponential）c^b。

两个函子的伴随关系可以被认为是其中一个函子的伪逆（pseudo-inverse）。它们的组合并不形成恒等映射，但它们的组合通过单位（unit）和余单位（counit）与恒等函子相关联。例如，如果你仔细观察，柯里化（currying）伴随的余单位：

$$\varepsilon_{bc} : [b, c] \quad b \to c$$

表明 $[b, c]$ 在某种意义上体现了乘法的逆。它的作用类似于除法中的 c/b：

$$c/b \quad b = c$$

以典型的范畴论方式，我们可能会问：如果我们将积替换为其他东西会怎样？显而易见，将它替换为余积（coproduct）是行不通的（因此我们没有减法的类比）。但也许还有其他行为良好的二元运算可以有一个右伴随。

一种自然的推广积的方式是使用具有张量积（tensor product）$\otimes$ 和单位对象 I 的单态范畴（monoidal category）。如果存在一个伴随关系：

$$\mathcal{C}(a \otimes b, c) \ddot{\circ} \mathcal{C}(a, [b, c])$$

我们将该范畴称为闭单态（closed monoidal）范畴。除非引起混淆，典型的范畴论中滥用符号，我们将继续使用相同的符号（方括号）来表示单态内部同态（monoidal internal hom），就像我们在笛卡尔积（cartesian hom）中所做的那样。还有一种替代的符号表示法，称为棒棒糖（lollipop）符号，用来表示张量积的右伴随：

$$\mathcal{C}(a \otimes b, c) \ddot{\circ} \mathcal{C}(a, b \multimap c)$$

它通常在*线性类型*（linear types）的上下文中使用。

内部同态的定义适用于对称单态范畴（symmetric monoidal category）。如果张量积不是对称的，伴随关系定义了*左闭单态*范畴。左内部同态（left internal hom）是“后乘法”函子 $(* \otimes b)$ 的右伴随。右闭结构则定义为“前乘法”函子 $(b \otimes *)$ 的右伴随。如果两者都定义了，那么该范畴称为*双闭单态范畴*（bi-closed monoidal category）。

Internal Hom for Day Convolution（Day 卷积的内部同态）

作为一个例子，考虑具有 Day 卷积（Day convolution）的协预层（co-presheaves）范畴中的对称单态结构：

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \otimes b, x) \quad Fa \quad Gb$$

我们正在寻找伴随关系：

$$[\mathcal{C}, \mathrm{Set}](F \star G, H) \ddot{\circ} [\mathcal{C}, \mathrm{Set}](F, [G, H]_{\mathrm{Day}})$$

左边的自然变换可以写成 x 上的一个末端（end）：

$$\int_x \mathrm{Set}\Big(\int^{a,b} \mathcal{C}(a \otimes b, x) \quad Fa \quad Gb, Hx \Big)$$

我们可以利用共连续性（co-continuity）将共末端（coend）提取出来：

$$\int_{x,a,b} \mathrm{Set}\Big(\mathcal{C}(a \otimes b, x) \quad Fa \quad Gb, Hx \Big)$$

接下来我们使用 Set 中的柯里化伴随（currying adjunction）：

$$\int_{x,a,b} \mathrm{Set}\Big(Fa, [\mathcal{C}(a \otimes b, x) \quad Gb, Hx]\Big)$$

最后，我们使用同态集的连续性（continuity of the hom-set）将两个末端移到同态集（hom-set）的内部：

$$\int_{a} \mathrm{Set}\Big(Fa, \int_{x,b} [\mathcal{C}(a \otimes b, x) \quad Gb, Hx]\Big)$$

我们发现 Day 卷积的右伴随由以下给出：

$$\Big([G, H]_{\mathrm{Day}}\Big)a = \int_{x,b} \Big(\!\Big(\mathcal{C}(a \otimes b, x), [Gb, Hx]\Big)\!\Big) \text{ ö } \int_{b} [Gb, H(a \otimes b)]$$

最后的转换是应用 Set 中的 Yoneda 引理（Yoneda lemma）。

Exercise 19.1.1. *实现 Day 卷积的内部同态在 Haskell 中的实现。提示：使用类型别名。*

Exercise 19.1.2. *实现以下伴随关系的见证：*

```
ltor :: (forall a. Day f g a -> h a) -> (forall a. f a -> DayHom g h a)
rtol :: Functor h =>
(forall a. f a -> DayHom g h a) -> (forall a. Day f g a -> h a)
```

Powering and Co-powering（幂对象与余幂对象）

在集合范畴（Set）中，内部同态（function object）等价于外部同态（external hom），即两个对象之间的态射集：

$$C^B \text{ ö } Set(B, C)$$

因此我们可以将定义集合范畴中内部同态的柯里化伴随关系改写为：

$$\mathrm{Set}(A \quad B, C) \text{ ö } \mathrm{Set}\Big(A, \mathrm{Set}(B, C)\Big)$$

我们可以将此伴随关系推广到 B 和 C 不再是集合，而是一些范畴 $\mathcal{C}$ 中的对象的情况。任何范畴中的外部同态总是一个集合。然而，左边不再由积定义。相反，它定义了一个集合 A 作用于对象 b 上的行为：

$$\mathcal{C}(A \cdot b, c) \text{ ö } \mathrm{Set}\Big(A, \mathcal{C}(b, c)\Big)$$

这被称为余幂（co-power）。

你可以将这种行为理解为将 A 个 b 相加在一起（取它们的余积）。例如，如果 A 是一个包含两个元素的集合 2，我们得到：

$$\mathcal{C}(2 \cdot b, c) \text{ ö } \mathrm{Set}\Big(2, \mathcal{C}(b, c)\Big) \text{ ö } \mathcal{C}(b, c) \quad \mathcal{C}(b, c) \text{ ö } \mathcal{C}(b + b, c)$$

换句话说，

$$2 \cdot b \ddot{o} b + b$$

从这个意义上说，余幂定义了迭代加法形式的乘法，就像我们在学校里学到的一样。

如果我们将 b 乘以同态集 $\mathcal{C}(b,c)$ 并对所有 b 取共末端，结果同构于 c：

$$\int^{b} \mathcal{C}(b,c) \cdot b \ddot{o} c$$

实际上，由于 Yoneda 引理，两边到任意 x 的映射是同构的：

$$\mathcal{C}\Big(\int^{b} \mathcal{C}(b,c) \cdot b, x\Big) \ddot{o} \int_{b} \mathrm{Set}\Big(\mathcal{C}(b,c), \mathcal{C}(b,x)\Big) \ddot{o} \mathcal{C}(c,x)$$

正如预期的那样，在集合范畴中，余幂简化为笛卡尔积（cartesian product）。

$$\mathrm{Set}(\boldsymbol{A} \cdot \boldsymbol{B}, \boldsymbol{C}) \ddot{o} \mathrm{Set}\Big(\boldsymbol{A}, \mathrm{Set}(\boldsymbol{B}, \boldsymbol{C})\Big) \ddot{o} \mathrm{Set}(\boldsymbol{A} \quad \boldsymbol{B}, \boldsymbol{C})$$

类似地，我们可以将幂对象（power）表达为迭代乘法。我们使用相同的右侧，但这次我们用映射来定义幂：

$$\mathcal{C}(b, \boldsymbol{A} \pitchfork c) \ddot{o} \mathrm{Set}\Big(\boldsymbol{A}, \mathcal{C}(b,c)\Big)$$

你可以将幂理解为将 $\boldsymbol{A}$ 个 c 相乘在一起。实际上，将 $\boldsymbol{A}$ 替换为 2 的结果是：

$$\mathcal{C}(b, 2 \pitchfork c) \ddot{o} \mathrm{Set}\Big(2, \mathcal{C}(b,c)\Big) \ddot{o} \mathcal{C}(b,c) \quad \mathcal{C}(b,c) \ddot{o} \mathcal{C}(b, c \quad c)$$

换句话说：

$$2 \pitchfork c \ddot{o} c \quad c$$

这是一种表示 c^2 的花式写法。

如果我们将 c 乘以同态集 $\mathcal{C}(c^{\cdot\cdot}, c)$ 并对所有 c 取末端，结果同构于 $c^{\cdot\cdot}$：

$$\int_{c} \mathcal{C}(c^{\cdot\cdot}, c) \pitchfork c \ddot{o} c^{\cdot\cdot}$$

这可以从 Yoneda 引理中得出。实际上，从 x 到两边的映射是同构的：

$$\mathcal{C}\Big(x, \int_{c} \mathcal{C}(c^{\cdot\cdot}, c) \pitchfork c\Big) \ddot{o} \int_{c} \mathrm{Set}\Big(\mathcal{C}(c^{\cdot\cdot}, c), \mathcal{C}(x,c)\Big) \ddot{o} \mathcal{C}(x, c^{\cdot\cdot})$$

在集合范畴中，幂简化为指数（exponential），它与同态集是同构的：

$$\boldsymbol{A} \pitchfork \boldsymbol{C} \ddot{o} \boldsymbol{C}^{\boldsymbol{A}} \ddot{o} \mathrm{Set}(\boldsymbol{A}, \boldsymbol{C})$$

这是因为乘法的对称性所致。

$$\mathrm{Set}(\boldsymbol{B}, \boldsymbol{A} \pitchfork \boldsymbol{C}) \ddot{o} \mathrm{Set}(\boldsymbol{A}, \mathrm{Set}(\boldsymbol{B}, \boldsymbol{C})) \ddot{o} \mathrm{Set}(\boldsymbol{A} \quad \boldsymbol{B}, \boldsymbol{C})$$

$$\ddot{o} \mathrm{Set}(\boldsymbol{B} \quad \boldsymbol{A}, \boldsymbol{C}) \ddot{o} \mathrm{Set}(\boldsymbol{B}, \mathrm{Set}(\boldsymbol{A}, \boldsymbol{C}))$$

19.2 Inverting a Functor（反转一个函子）

范畴论的一个方面是通过执行有损转换丢弃信息；另一个方面是恢复丢失的信息。我们已经看到用自由函子（free functors）补偿丢失数据的例子——这是遗忘函子（forgetful functors）的伴随函子。Kan 扩展是另一个例子。两者都弥补了那些不可逆函子丢失的数据。

一个函子可能不可逆的原因有两个。一个是它可能将多个对象或箭头映射到单个对象或箭头上。换句话说，它在对象或箭头上不是单射的（injective）。另一个原因是它的像可能无法覆盖整个目标范畴。换句话说，它在对象或箭头上不是满射的（surjective）。

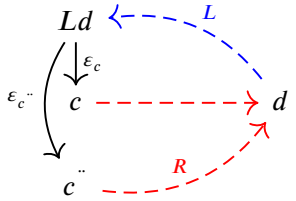
例如，考虑一个伴随 $L \dashv R$。假设 R 不是单射的，它将两个对象 c 和 $c^{\cdot\cdot}$ 合并为一个对象 d

$$Rc = d$$
$$Rc^{\cdot\cdot} = d$$

L 无力逆转这种操作。它无法同时将 d 映射到 c 和 $c^{\cdot\cdot}$。它最多只能将 d 映射到一个“更一般”的对象 Ld，该对象有箭头指向 c 和 $c^{\cdot\cdot}$。这些箭头用于定义伴随的余单位的分量：

$$\varepsilon_c : Ld \to c$$
$$\varepsilon_{c^{\cdot\cdot}} : Ld \to c^{\cdot\cdot}$$

其中 Ld 既是 $L(Rc)$ 也是 $L(Rc^{\cdot\cdot})$



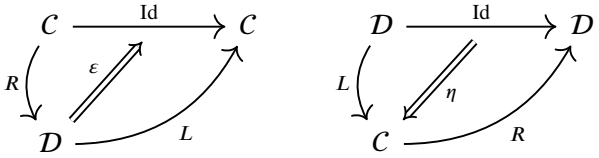
此外，如果 R 在对象上不是满射的，函子 L 必须以某种方式在 R 的像之外的 $\mathcal{D}$ 对象上进行定义。同样，单位和余单位的自然性会限制可能的选择，只要这些对象与 R 的像有箭头连接。

显然，所有这些限制意味着伴随只能在非常特殊的情况下定义。

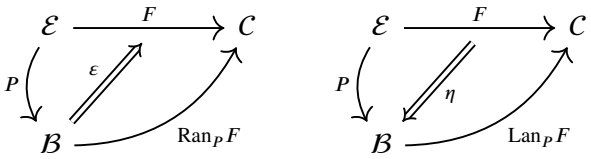
Kan 扩展甚至比伴随还要弱。

如果伴随函子像逆一样工作，Kan 扩展就像分数一样工作。

这在我们重绘定义伴随的余单位和单位的图表时看得最清楚。在第一个图中，L 似乎扮演了 $1/R$ 的角色。在第二个图中，R 则假装自己是 $1/L$。



右 Kan 扩展 $\mathrm{Ran}_P F$ 和左 Kan 扩展 $\mathrm{Lan}_P F$ 通过将恒等函子替换为某个函子 $F\colon \mathcal{E} \to \mathcal{C}$ 来推广这些概念。Kan 扩展扮演的角色类似于分数 F/P。从概念上讲，它们“逆转”P 的作用，并跟随它执行 F 的作用。



正如伴随的情况一样，“逆转”并不完全。组合 $\mathrm{Ran}_P F \circ P$ 并不能还原 F；相反，它通过一个称为余单位（counit）的自然变换 ε 与它相关联。同样，组合 $\mathrm{Lan}_P F \circ P$ 通过单位（unit）η 与 F 相关联。

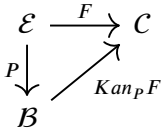
请注意，F 丢弃的信息越多，Kan 扩展“逆转”函子 P 就越容易。在某种意义上，它只需“模 F”逆转 P 即可。

以下是关于 Kan 扩展的直觉。我们从一个函子 F 开始：

$$\mathcal{E} \xrightarrow{F} \mathcal{C}$$

有一个第二个函子 P，它将 $\mathcal{E}$ 压缩到另一个范畴 $\mathcal{B}$ 中。这可能是一个嵌入，它是有损且非满射的。我们的任务是以某种方式*扩展* F 的定义，以涵盖整个 $\mathcal{B}$。

在理想情况下，我们希望以下图表能够交换：



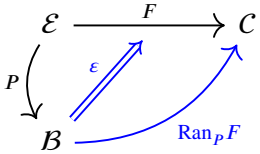
但这会涉及函子之间的相等，这是我们尽量避免的。

下一个最好的选择是要求在这个图中两条路径之间存在一个自然同构。但是即使这样看起来也要求过高。因此，我们最终决定要求一条路径能够变形为另一条路径，这意味着它们之间存在一个单向的自然变换。这种变换的方向区分了右 Kan 扩展和左 Kan 扩展。

19.3 Right Kan Extension（右 Kan 扩展）

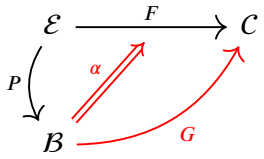
右 Kan 扩展是一个函子 $\mathrm{Ran}_P F$，配备了一个自然变换 ε，称为 Kan 扩展的余单位，定义为：

$$\varepsilon\colon (\mathrm{Ran}_P F) \circ P \to F$$

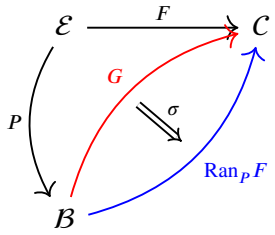


对 $\mathrm{Ran}_P F$ 的要求是，$(\mathrm{Ran}_P F, \varepsilon)$ 在所有类似对 (G, α) 中是泛化的，其中 G 是一个函子 $G\colon \mathcal{B} \to \mathcal{C}$，而 α 是一个自然变换：

$$\alpha\colon G \circ P \to F$$



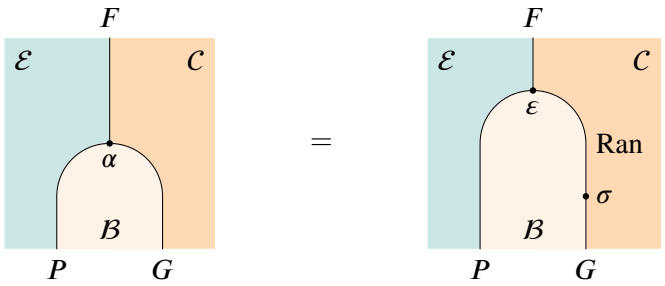
泛化意味着对于任何这样的 (G, α)，都存在唯一的自然变换 $\sigma\colon G \to \mathrm{Ran}_P F$，



它使得 α 可以分解为：

$$\alpha = \varepsilon \cdot (\sigma \circ P)$$

这是一个在自然变换的垂直和水平组合中，其中 $\sigma \circ P$ 是 σ 的卷须（whiskering）。以下是相同的方程式，用线图的方式绘制：



如果沿 P 的右 Kan 扩展对每个函子 F 都定义了，那么这个泛范构造可以推广到一个伴随关系——这次它是在两个函子范畴之间的伴随关系：

$$[\mathcal{E}, \mathcal{C}](G \circ P, F) \ddot{\circ} [\mathcal{B}, \mathcal{C}](G, \mathrm{Ran}_P F)$$

对于任何 α（左侧的元素），都存在唯一的 σ（右侧的元素）。

换句话说，如果右 Kan 扩展对每个 F 都存在，那么它就是函子前组合的右伴随：

$$(* \circ P) \dashv \mathrm{Ran}_P$$

这个伴随关系的余单位在 F 的成分就是 ε。

这在某种程度上类似于柯里化伴随：

$$\mathcal{E}(a \quad b, c) \cong \mathcal{E}(a, [b, c])$$

其中积被函子组合取代。（这个类比并不完美，因为组合只能在自函子范畴中被认为是张量积。）

Right Kan Extension as an End（作为末端的右 Kan 扩展）

回顾一下忍者 Yoneda 引理：

$$Fb \cong \int_e \mathrm{Set}(\mathcal{B}(b, e), Fe)$$

这里，F 是一个协预层，即从 $\mathcal{B}$ 到 Set 的函子。沿 P 的 F 的右 Kan 扩展推广了这个公式：

$$(\mathrm{Ran}_P F)b \cong \int_e \mathrm{Set}\Big(\mathcal{B}(b, Pe), Fe\Big)$$

这适用于协预层。通常我们关心的是 $F\colon \mathcal{E} \to \mathcal{C}$，因此我们需要将 Set 中的同态集替换为幂（power）。因此，右 Kan 扩展由以下末端给出（如果它存在的话）：

$$(\mathrm{Ran}_P F)b \cong \int_e \mathcal{B}(b, Pe) \pitchfork Fe$$

证明基本上是自然而然的：在每一步中只有一种操作可以执行。我们从伴随开始：

$$[\mathcal{E}, \mathcal{C}](G \circ P, F) \cong [\mathcal{B}, \mathcal{C}](G, \mathrm{Ran}_P F)$$

并使用末端重新书写它：

$$\int_e \mathcal{C}\Big(G(Pe), Fe\Big) \cong \int_b \mathcal{C}\Big(Gb, (\mathrm{Ran}_P F)b\Big)$$

我们将我们的公式代入得到：

$$\cong \int_b \mathcal{C}\Big(Gb, \int_e \mathcal{B}(b, Pe) \pitchfork Fe\Big)$$

我们使用同态函子的连续性将末端推到前面：

$$\cong \int_b \int_e \mathcal{C}\Big(Gb, \mathcal{B}(b, Pe) \pitchfork Fe\Big)$$

然后我们使用幂的定义：

$$\int_b \int_e \mathrm{Set}\Big(\mathcal{B}(b, Pe), \mathcal{C}(Gb, Fe)\Big)$$

并应用 Yoneda 引理：

$$\int_e \mathcal{C}\Big(G(Pe), Fe\Big)$$

这个结果确实是伴随关系的左侧。

如果 F 是一个协预层，右 Kan 扩展公式中的幂简化为指数/同态集：

$$(\mathrm{Ran}_P F)b \ddot{o} \int_e \mathrm{Set}\Big(\mathcal{B}(b, Pe), Fe\Big)$$

还要注意的是，如果 P 有一个左伴随，我们称之为 P^{*1}，即：

$$\mathcal{B}(b, Pe) \ddot{o} \mathcal{E}(P^{*1}b, e)$$

我们可以使用忍者 Yoneda 引理来计算末端：

$$(\mathrm{Ran}_P F)b \ddot{o} \int_e \mathrm{Set}\Big(\mathcal{B}(b, Pe), Fe\Big) \ddot{o} \int_e \mathrm{Set}(\mathcal{E}(P^{*1}b, e), Fe) \ddot{o} F(P^{*1}b)$$

得到：

$$\mathrm{Ran}_P F \ddot{o} F \circ P^{*1}$$

由于伴随关系是逆的弱化，这个结果与 Kan 扩展“逆转”P 并跟随 F 的直觉是一致的。

Right Kan extension in Haskell（Haskell 中的右 Kan 扩展）

右 Kan 扩展的末端公式可以直接翻译为 Haskell 代码：

```
newtype Ran p f b = Ran (forall e. (b -> p e) -> f e)
```

右 Kan 扩展的余单位 ε 是一个从`(Ran p f)`和`p`的组合到`f`的自然变换：

```
counit :: forall p f e'. Ran p f (p e') -> f e'
```

为了实现它，我们需要生成一个类型为`(f c')`的值，给定一个多态函数

```
h :: forall e. (p e' -> p e) -> f e
```

我们通过在类型`e = e'`上实例化此函数并用`(p e')`上的身份函数调用它来实现：

```
counit (Ran h) = h id
```

右 Kan 扩展的计算能力来自其泛性质。我们从一个带有自然变换的函子 G 开始：

$$\alpha \colon G \circ P \to F$$

这可以表示为一个 Haskell 数据类型：

```
type Alpha p f g = forall e. g (p e) -> f e
```

泛化性质告诉我们，存在一个唯一的自然变换 σ，从这个函子到相应的右 Kan 扩展：

```
sigma :: Functor g => Alpha p f g -> forall b. (g b -> Ran p f b)
sigma alpha gb = Ran (\b_pe -> alpha $ fmap b_pe gb)
```

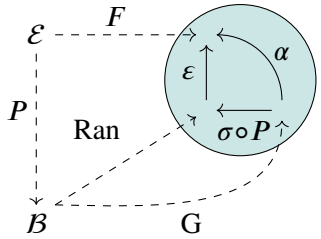
该变换通过余单位 ε 分解 α：

$$\alpha = \varepsilon \cdot (\sigma \circ P)$$

请记住，卷须（whiskering）意味着我们在`b = p c`时实例化`sigma`。然后它跟随`counit`。α 的分解由下式给出：

```
factorize' :: Functor g => Alpha p f g -> forall e. g (p e) -> f e
factorize' alpha gpc = alpha gpc
```

这三个自然变换的成分都是目标范畴 $\mathcal{C}$ 中的态射：



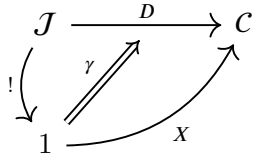
Exercise 19.3.1. *Implement the* `Functor` *instance for* `Ran`.

Limits as Kan extensions（作为 Kan 扩展的极限）

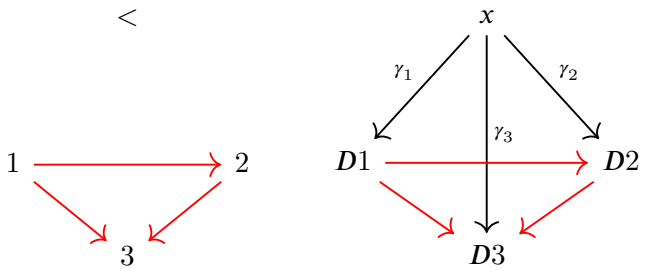
我们之前已经定义了极限（limits）为泛锥（universal cones）。锥的定义涉及两个范畴：定义图形的指标范畴 $\mathcal{J}$ 和目标范畴 $\mathcal{C}$。图（diagram）是一个函子 $D\colon \mathcal{J} \to \mathcal{C}$，它将形状嵌入目标范畴中。

我们可以引入第三个范畴 1：一个包含单个对象和单个恒等箭头的终范畴。然后我们可以使用从该范畴到锥顶点 x 的函子 X。由于 1 在 Cat 中是终范畴，因此我们也有从 $\mathcal{J}$ 到它的唯一函子，我们称之为 !。它将所有对象映射到 1 的唯一对象，并将所有箭头映射到其恒等箭头。

事实证明，D 的极限是沿着 ! 的图 D 的右 Kan 扩展。首先，观察组合 $X \circ !$ 将形状 $\mathcal{J}$ 映射到单个对象 x，因此它充当了常函子 Δ_x 的作用。它选择锥的顶点。一个以 x 为顶点的锥是一个自然变换 γ：

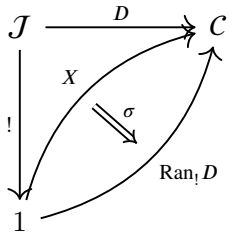


下图说明了这一点。左图中有两个范畴：具有单个对象 $<$ 的 1 和具有三个对象形成图形的 $\mathcal{J}$。右图中有 D 的图像和 $X \circ !$ 的图像，即顶点 x。γ 的三个成分将顶点 x 连接到图上。γ 的自然性确保形成锥边的三角形是交换的。



右 Kan 扩展 $(\mathrm{Ran}_! D, \varepsilon)$ 是泛化的这种锥。$\mathrm{Ran}_! D$ 是一个从 1 到 $\mathcal{C}$ 的函子，因此它选择 $\mathcal{C}$ 中的一个对象。这确实是泛锥的顶点 $\mathrm{Lim} D$。

泛化性意味着对于任何对 (X, γ)，都有一个自然变换 $\sigma\colon X \to \mathrm{Ran}_! D$



它分解 γ。

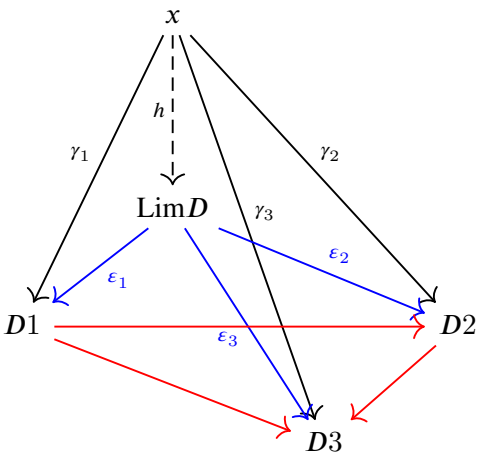
变换 σ 只有一个成分 $\sigma_<$，它是将顶点 x 连接到顶点 $\mathrm{Lim} D$ 的箭头 h。分解如下：

$$\gamma = \varepsilon \cdot (\sigma \circ !)$$

以分量的形式：

$$\gamma_i = \varepsilon_i \circ h$$

它使下图中的三角形交换：



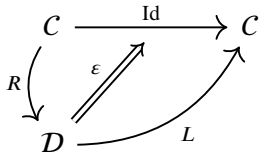
这种泛化条件使 $\mathrm{Lim} D$ 成为图 D 的极限。

Left adjoint as a right Kan extension（作为右 Kan 扩展的左伴随）

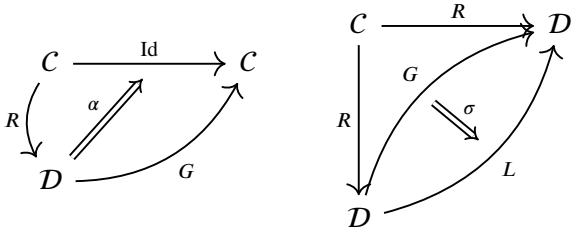
我们开始描述 Kan 扩展作为伴随的推广。观察这些图，如果我们有一对伴随函子 $L \dashv R$，我们希望左函子是沿右函子的恒等函子的右 Kan 扩展。

$$L \ddot{\circ} \operatorname{Ran}_R \mathrm{Id}$$

实际上，Kan 扩展的余单位与伴随的余单位相同：



我们还需要展示泛化性：



为此，我们可以使用伴随的单位：

$$\eta \colon \mathrm{Id} \to R \circ L$$

我们将 σ 构造为复合：

$$G \to G \circ \mathrm{Id} \xrightarrow{G \circ \eta} G \circ R \circ L \xrightarrow{\alpha \circ L} \mathrm{Id} \circ L \to L$$

换句话说，我们将 σ 定义为：

$$\sigma = (\alpha \circ L) \cdot (G \circ \eta)$$

我们可以问反向问题：如果 $\operatorname{Ran}_R \mathrm{Id}$ 存在，它是否自动成为 R 的左伴随？事实证明，为此我们需要一个额外的条件：Kan 扩展必须被 R 保留，即：

$$R \circ \operatorname{Ran}_R \mathrm{Id} \ddot{\circ} \operatorname{Ran}_R R$$

我们将在下一节中看到这个条件的右侧定义了密度余子模。

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（密度余子模）

我们已经看到，每个伴随 $L \dashv F$ 都会产生一个子模 $F \circ L$。事实证明，这个子模是沿 F 的 F 的右 Kan 扩展。有趣的是，即使 F 没有左伴随，Kan 扩展 $\mathrm{Ran}_F F$ 仍然是一个被称为密度余子模的子模，记为 T^F：

$$T^F = \mathrm{Ran}_F F$$

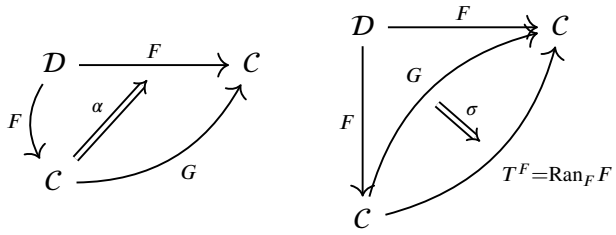
如果我们认真对待 Kan 扩展作为分数的解释，密度余子模将对应于 F/F。对于这种“分数”等于恒等的函子，称为密度。

为了看到 T^F 是一个子模，我们需要定义子模的单位和乘法：

$$\eta \colon \mathrm{Id} \to T^F$$

$$\mu \colon T^F \circ T^F \to T^F$$

这两者都来自泛化性。对于每个 (G, α)，我们都有一个 σ：



为了得到单位，我们用恒等函子 Id 替换 G，并用恒等自然变换替换 α。

为了得到乘法，我们用 $T^F \circ T^F$ 替换 G，并注意到我们有 Kan 扩展的余单位：

$$\varepsilon \colon T^F \circ F \to F$$

我们可以选择 α 类型的：

$$\alpha \colon T^F \circ T^F \circ F \to F$$

作为复合：

$$T^F \circ T^F \circ F \xrightarrow{id \circ \varepsilon} T^F \circ F \xrightarrow{\varepsilon} F$$

或者，使用卷须表示法：

$$\alpha = \varepsilon \cdot (T^F \circ \varepsilon)$$

对应的 σ 给我们子模的乘法。

现在让我们来展示一下，如果我们从一个伴随开始：

$$\mathcal{D}(Lc, d) \ddot{\circ} \mathcal{C}(c, Fd)$$

密度余子模由 $F \circ L$ 给出。让我们从将任意函子 G 映射到 $F \circ L$ 开始：

$$[\mathcal{C}, \mathcal{C}](G, F \circ L) \ddot{\circ} \int_c \mathcal{C}(Gc, F(Lc))$$

我们可以使用 Yoneda 引理重写它：

$$\ddot{o} \int_c \int_d \mathrm{Set}\Big(\mathcal{D}(Lc,d), \mathcal{C}(Gc, Fd)\Big)$$

这里，对 d 的末端求值的效果是将 d 替换为 Lc。我们现在可以使用伴随：

$$\ddot{o} \int_c \int_d \mathrm{Set}\Big(\mathcal{C}(c,Fd), \mathcal{C}(Gc, Fd)\Big)$$

并执行忍者 Yoneda 对 c 的积分以得到：

$$\ddot{o} \int_d \mathcal{C}(G(Fd), Fd)$$

这反过来定义了一组自然变换：

$$\ddot{o}\ [\mathcal{D}, \mathcal{C}](G \circ F, F)$$

由 F 的前置组合是右 Kan 扩展的左伴随：

$$[\mathcal{D}, \mathcal{C}](G \circ F, F)\ \ddot{o}\ [\mathcal{C}, \mathcal{C}](G, \mathrm{Ran}_F F)$$

由于 G 是任意的，我们可以得出结论，$F \circ L$ 确实是密度余子模 $\mathrm{Ran}_F F$。

由于每个子模都可以从某种伴随中得出，因此每个子模都是某种伴随的密度余子模。

article amsmath tikz-cd haskell

Codensity Monad in Haskell (Haskell 中的 Codensity Monad)

将 codensity monad 翻译为 Haskell，我们得到以下代码：

```
newtype Codensity f c = C (forall d. (c -> f d) -> f d)
```

以及提取函数：

```
runCodensity :: Codensity f c -> forall d. (c -> f d) -> f d
runCodensity (C h) = h
```

这看起来非常类似于一个 continuation monad。事实上，如果我们将`f`选为 identity 函子，它就变成了 continuation monad。我们可以将`Codensity`看作是接受一个回调`(c -> f d)`，并在类型为`c`的结果可用时调用它。

下面是 monad 实例：

```
instance Monad (Codensity f) where
return x = C (\k -> k x)
m >>= kl = C (\k -> runCodensity m (\a -> runCodensity (kl a) k))
```

同样，这几乎与 continuation monad 完全相同：

```haskell
instance Monad (Cont r) where
return x = Cont (\k -> k x)
m >>= kl = Cont (\k -> runCont m (\a -> runCont (kl a) k))
```

这就是为什么`Codensity`具有 continuation passing style 的性能优势。由于它将 continuations"内部嵌套"，它可以用来优化由`do`块生成的长链 binds。

这种特性在处理 free monads 时尤为重要，free monads 在树状结构中积累 binds。当我们最终解释一个 free monad 时，这些积累的 binds 需要遍历不断增长的树。对于每一个 bind，遍历从根节点开始。将此与前面反转列表的示例进行比较，该示例通过在 FIFO 队列中累积函数进行了优化。codensity monad 提供了同样类型的性能提升。

Exercise 19.3.3. 实现`Codensity`的`Functor`实例。

Exercise 19.3.4. 实现`Codensity`的`Applicative`实例。

19.4 Left Kan Extension (左 Kan 扩展)

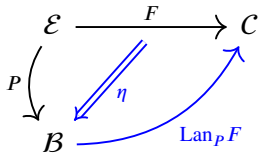
就像右 Kan 扩展被定义为函子预组合的*右*伴随一样，左 Kan 扩展被定义为函子预组合的*左*伴随：

$$[\mathcal{B}, C](\mathrm{Lan}_P F, G) \ \ddot{\mathrm{o}} \ [\mathcal{E}, C](F, G \circ P)$$

（还有*后*组合的伴随：它们被称为 Kan lifts。）

另一种定义 $\mathrm{Lan}_P F$ 的方法是将其定义为一个函子，并带有称为单位元的自然变换：

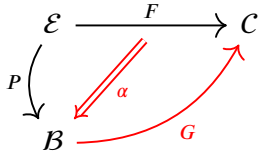
$$\eta \colon F \to \mathrm{Lan}_P F \circ P$$



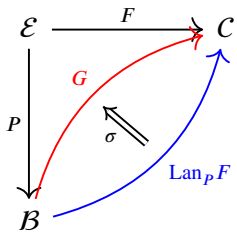
注意，左 Kan 扩展的单位元方向与右 Kan 扩展的余单位元方向相反。

$(\mathrm{Lan}_P F, \eta)$ 是普遍的，这意味着，对于任何其他对 (G, α)，其中

$$\alpha \colon F \to G \circ P$$



存在唯一的映射 $\sigma\colon \mathrm{Lan}_P F \to G$

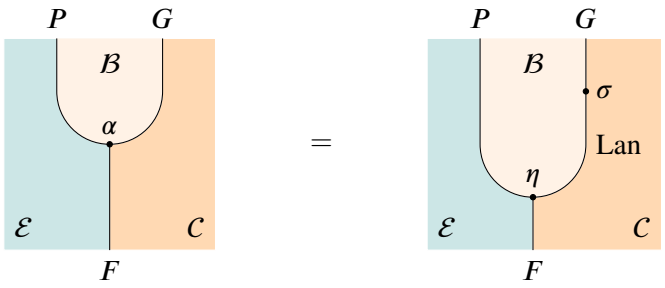


将 α 因式分解：

$$\alpha = (\sigma \circ P) \cdot \eta$$

再次强调，σ 的方向与右 Kan 扩展相反。

使用弦图，我们可以将普遍条件表示为：



这确立了两个自然变换集合之间的一一映射。对于左边的每一个 α，右边都有一个唯一的 σ：

$$[\mathcal{E}, \mathcal{C}](F, G \circ P) \ \ddot{\text{o}} \ [\mathcal{B}, \mathcal{C}](\mathrm{Lan}_P F, G)$$

Left Kan Extension as a Coend (左 Kan 扩展作为共端)

回想一下忍者 co-Yoneda 引理。对于每一个 co-presheaf F，我们有：

$$Fb \ \ddot{\text{o}} \int^{c} \mathcal{B}(c, b) \quad Fc$$

左 Kan 扩展将此公式推广为：

$$(\mathrm{Lan}_P F)\, b \ \ddot{\text{o}} \int^{e} \mathcal{B}(Pe, b) \quad Fe$$

对于一般的函子 $F\colon \mathcal{E} \to \mathcal{C}$，我们将乘积替换为copower：

$$(\mathrm{Lan}_P F)\, b \ \ddot{\text{o}} \int^{e} \mathcal{B}(Pe, b) \cdot Fe$$

只要所涉及的共端存在，我们可以通过考虑映射到某个函子 G 的映射来证明这一公式。我们将自然变换集合表示为对 b 的端：

$$\int_b \mathcal{C}\Big(\int^{e} \mathcal{B}(Pe, b) \cdot Fe, Gb \Big)$$

利用余小连续性，我们将共端拉出，变成一个端：

$$\int_b \int_e \mathcal{C}\Big(\mathcal{B}(Pe, b) \cdot Fe, Gb\Big)$$

然后我们代入 copower 的定义：

$$\int_b \int_e \mathcal{C}\Big(\mathcal{B}(Pe, b), \mathcal{C}(Fe, Gb)\Big)$$

现在我们可以使用 Yoneda 引理对 b 进行积分，将 b 替换为 Pe：

$$\int_e \mathcal{C}(Fe, G(Pe))\Big) \text{ ö } [\mathcal{E}, \mathcal{C}](F, G \circ P)$$

这确实为我们提供了函子预组合的左伴随：

$$[\mathcal{B}, \mathcal{C}](\mathrm{Lan}_P F, G) \text{ ö } [\mathcal{E}, \mathcal{C}](F, G \circ P)$$

在 Set 中，copower 衰减为笛卡尔积，所以我们得到一个更简单的公式：

$$(\mathrm{Lan}_P F)\, b \text{ ö } \int^e \mathcal{B}(Pe, b) \quad Fe$$

注意，如果函子 P 有一个右伴随，我们称之为 P^{*1}：

$$\mathcal{B}(Pe, b) \text{ ö } \mathcal{E}(e, P^{*1} b)$$

我们可以使用忍者 co-Yoneda 引理来得到：

$$(\mathrm{Lan}_P F)\, b \text{ ö } (F \circ P^{*1}) b$$

从而进一步增强了 Kan 扩展逆转 P 并跟随 F 的直观理解。

Left Kan Extension in Haskell (Haskell 中的左 Kan 扩展)

当我们将左 Kan 扩展的公式翻译为 Haskell 时，我们将共端替换为存在类型。符号表示：

```
type Lan p f b = exists e. (p e -> b, f e)
```

这是我们如何使用GADT编码存在性的方式：

```
data Lan p f b where
Lan :: (p e -> b) -> f e -> Lan p f b
```

左 Kan 扩展的单位元是一个从函子f到(Lan p f)之后p组合的自然变换：

```
unit :: forall p f e'.
f e' -> Lan p f (p e')
```

为了实现单位元，我们从一个类型为(f e')的值开始。我们必须得出某个类型e、一个函数p e -> p e'，以及一个类型为(f e)的值。显然的选择是取e = e'并在(p e')上使用身份：

```
unit fe = Lan id fe
```

左 Kan 扩展的计算能力体现在其普遍性上。给定一个函子g以及从f到g之后p组合的自然变换：

```
type Alpha p f g = forall e. f e -> g (p e)
```

存在一个唯一的自然变换 σ，从对应的左 Kan 扩展到g：

```
sigma :: Functor g => Alpha p f g -> forall b. (Lan p f b -> g b)
sigma alpha (Lan pe_b fe) = fmap pe_b (alpha fe)
```

它通过单位元 η 将 α 因式分解：

$$\alpha = (\sigma \circ P) \cdot \eta$$

σ 的 whiskering 意味着将其在b = p e处实例化，因此 α 的因式分解实现如下：

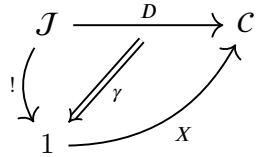
```
factorize :: Functor g => Alpha p f g -> f e -> g (p e)
factorize alpha = sigma alpha . unit
```

Exercise 19.4.1. 实现*Lan*的*Functor*实例。

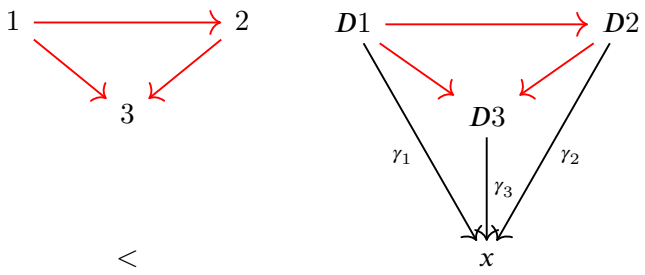
Colimits as Kan Extensions (作为 Kan 扩展的余极限)

就像极限可以定义为右 Kan 扩展一样，余极限可以定义为左 Kan 扩展。

我们从一个定义了余极限形状的索引范畴 $\mathcal{J}$ 开始。函子 D 在目标范畴 $\mathcal{C}$ 中选择了这个形状。圆锥的顶点由来自终端单对象范畴 1 的函子选择。自然变换定义了从 D 到 X 的圆锥：



这里是一个由三个对象和三条态射（不包括身份态射）组成的简单形状的示例。对象 x 是单个对象 $<$ 在函子 X 下的像：



余极限是普遍圆锥，由沿函子 ! 的 D 的左 Kan 扩展给出：

$$\mathrm{Colim}\ D = \mathrm{Lan}_! D$$

Right Adjoint as a Left Kan Extension (作为左 Kan 扩展的右伴随)

我们已经看到，当我们有一个伴随 $L \vdash R$ 时，左伴随与右 Kan 扩展有关。对偶地，如果存在右伴随，它可以表示为 identity 函子的左 Kan 扩展：

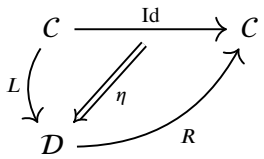
$$R \circ \mathrm{Lan}_L \mathrm{Id}$$

反过来，如果存在 identity 的左 Kan 扩展并且它保持函子 L：

$$L \circ \mathrm{Lan}_L \mathrm{Id} \circ \mathrm{Lan}_L L$$

那么 $\mathrm{Lan}_L \mathrm{Id}$ 是 L 的右伴随。L 沿其自身的左 Kan 扩展称为密度共模子（density comonad）。

Kan 扩展的单位元与伴随的单位元相同：



普遍性的证明与右 Kan 扩展的证明类似。

Exercise 19.4.2. *为密度共模子实现Comonad实例：*

```
data Density f c where
D :: (f d -> c) -> f d -> Density f c
```

Day Convolution as a Kan Extension (Day 卷积作为 Kan 扩展)

我们已经看到 Day 卷积在单对象范畴 $\mathcal{C}$ 上的 co-presheaves 的范畴中定义为张量积：

$$(F \star G)c = \int^{a,b} C(a \otimes b, c) \quad Fa \quad Gb$$

co-presheaves，即 $[\mathcal{C}, \mathrm{Set}]$ 中的函子，也可以使用外部张量积（external tensor product）进行张量。两个对象的外部积并不在同一个范畴中生成一个对象，而是在不同的范畴中选择一个对象。在我们的情况下，两个函子的积最终在 $\mathcal{C} \quad \mathcal{C}$ 上的 co-presheaves 的范畴中：

$$\bar{\otimes} : [\mathcal{C}, \mathrm{Set}] \quad [\mathcal{C}, \mathrm{Set}] \to [\mathcal{C} \quad \mathcal{C}, \mathrm{Set}]$$

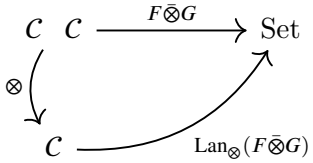
两个 co-presheaves 作用于 $\mathcal{C} \quad \mathcal{C}$ 中的一对对象，其公式为：

$$(F \bar{\otimes} G)\langle a, b \rangle = Fa \quad Gb$$

事实证明，两个函子的 Day 卷积可以表示为沿 $\mathcal{C}$ 中的张量积的外部积的左 Kan 扩展：

$$F \star G \circ \mathrm{Lan}_{\otimes}(F \bar{\otimes} G)$$

图解如下：



确实，使用左 Kan 扩展的共端公式我们得到：

$$(\mathrm{Lan}_{\otimes}(F \bar{\otimes} G))c \cong \int^{\langle a,b \rangle} \mathcal{C}(a \otimes b, c) \cdot (F \bar{\otimes} G)\langle a, b \rangle$$

$$\cong \int^{\langle a,b \rangle} \mathcal{C}(a \otimes b, c) \cdot (Fa \quad Gb)$$

由于两个函子都是 Set 值的，copower 衰减为笛卡尔积：

$$\cong \int^{\langle a,b \rangle} \mathcal{C}(a \otimes b, c) \quad Fa \quad Gb$$

并重现了 Day 卷积的公式。

19.5 Useful Formulas (有用的公式)

- Co-power (copower)：

$$\mathcal{C}(A \cdot b, c) \cong \mathrm{Set}\Big(A, \mathcal{C}(b, c)\Big)$$

- Power (幂)：

$$\mathcal{C}(b, A \pitchfork c) \cong \mathrm{Set}\Big(A, \mathcal{C}(b, c)\Big)$$

- Right Kan Extension (右 Kan 扩展)：

$$[\mathcal{E}, \mathcal{C}](G \circ P, F) \cong [\mathcal{B}, \mathcal{C}](G, \mathrm{Ran}_P F)$$

$$(\mathrm{Ran}_P F)b \cong \int_e \mathcal{B}(b, Pe) \pitchfork Fe$$

- Left Kan Extension (左 Kan 扩展)：

$$[\mathcal{B}, \mathcal{C}](\mathrm{Lan}_P F, G) \cong [\mathcal{E}, \mathcal{C}](F, G \circ P)$$

$$(\mathrm{Lan}_P F)\, b \cong \int^e \mathcal{B}(Pe, b) \cdot Fe$$

- Right Kan Extension in Set (Set 中的右 Kan 扩展)：

$$(\mathrm{Ran}_P F)b \cong \int_e \mathrm{Set}\Big(\mathcal{B}(b, Pe), Fe\Big)$$

- Left Kan Extension in Set (Set 中的左 Kan 扩展)：

$$(\mathrm{Lan}_P F)b \cong \int^e \mathcal{B}(Pe, b) \quad Fe$$

Chapter 20

Enrichment（丰盈）

老子曰:“知足者富。”

20.1 Enriched Categories（丰盈范畴）

这可能会让人感到惊讶，但在没有丰盈范畴的背景知识下，是无法完全解释`Functor`（函子）的 Haskell 定义的。在本章中，我将尝试展示，从概念上来说，丰盈并不是从普通范畴论向前迈出的一大步。

研究丰盈范畴的额外动机来自这样一个事实：大量文献，尤其是网站 nLab，通常以最一般的术语（这通常意味着以丰盈范畴的术语）来描述概念。大多数常见的构造都可以通过更改词汇来翻译，将同态集（hom-sets）替换为同态对象（hom-objects），将 Set 替换为一个幺半范畴 $\mathcal{V}$。

一些丰盈概念，如加权极限和余极限（weighted limits and colimits），本身就非常强大，以至于人们可能会被引诱将 Mac Lane 的格言“所有概念都是 Kan 扩张”替换为“所有概念都是加权（余）极限。”

Set-theoretical foundations（集合论基础）

范畴论在其基础上非常节俭。但它（不情愿地）依赖于集合论。特别是，同态集的概念被定义为两个对象之间箭头的集合，这引入了集合论作为范畴论的前提。诚然，箭头只在*局部小*范畴中形成集合，但这只是聊以自慰，考虑到处理太大的东西而无法成为集合需要更多的理论。

如果范畴论能够自举，例如通过将同态集替换为更一般的对象，那就太好了。这正是丰盈范畴背后的思想。然而，这些同态对象必须来自某个具有同态集的其他范畴，并且在某些时候我们必须回到集合论基础上。然而，能够用不同的东西替换没有结构的同态集，扩大了我们建模更复杂系统的能力。

集合的主要属性是，与对象不同，它们不是原子的：它们有*元素*。在范畴论中，我们有时谈论*广义元素*，它们只是指向某个对象的箭头；或者*全局元素*，它们是来自终对象（有时来自幺元对象 I）的箭头。但最重要的是，集合定义了元素的*相等性*。

我们几乎学到的所有关于范畴的知识都可以翻译到丰盈范畴的领域中。然而，许多范畴推理涉及交换图表，它们表示箭头的相等性。在丰盈环境中，我们没有对象之间的箭头，因此所有这些构造都必须进行修改。

Hom-Objects（同态对象）

乍一看，用对象替换同态集似乎是一个退步。毕竟，集合有元素，而对象只是无形的斑点。然而，同态对象的丰富性体现在它们所属范畴的态射中。从概念上讲，集合是无结构的，意味着它们之间有大量的态射（函数）。态射越少，通常意味着结构越多。

定义丰盈范畴的指导原则是，我们应该能够将普通范畴论作为特例来恢复。毕竟，同态集*是* Set 范畴中的对象。实际上，我们非常努力地将集合的属性表达为函数而不是元素。

话虽如此，用复合和单位定义范畴时会涉及到同态集的*元素*的态射。因此，让我们首先在不诉诸元素的情况下重新表述范畴的基本原理。

箭头的复合可以作为同态集之间的函数来定义：

$$\circ\colon \mathcal{C}(b,c)\ \ \mathcal{C}(a,b) \to \mathcal{C}(a,c)$$

我们可以使用单集合中的函数来代替单位箭头：

$$j_a\colon 1 \to \mathcal{C}(a,a)$$

这表明，如果我们想用来自某个范畴 $\mathcal{V}$ 的对象替换同态集 $\mathcal{C}(a,b)$，我们必须能够*乘*这些对象以定义复合，并且我们需要某种*单位对象*来定义单位。我们可以要求 $\mathcal{V}$ 是笛卡尔的，但实际上，幺半范畴就足够了。正如我们将看到的，幺半范畴的单位和结合律直接转换为复合的单位和结合律。

Enriched Categories（丰盈范畴）

设 $\mathcal{V}$ 是一个具有张量积 $\otimes$、单位对象 I 和结合子及两个单位子的幺半范畴（以及它们的逆）：

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

一个丰盈于 $\mathcal{V}$ 的范畴 $\mathcal{C}$ 具有对象，并且对于任意一对对象 a 和 b，有一个同态对象 $\mathcal{C}(a,b)$。这个同态对象是 $\mathcal{V}$ 中的一个对象。复合使用 $\mathcal{V}$ 中的箭头定义：

$$\circ\colon \mathcal{C}(b,c)\otimes\mathcal{C}(a,b)\to\mathcal{C}(a,c)$$

单位由箭头定义：

$$j_a\colon I\to\mathcal{C}(a,a)$$

结合律由 $\mathcal{V}$ 中的结合子表达：

$$\begin{array}{ccc}
\Big(\mathcal{C}(c,d)\otimes\mathcal{C}(b,c)\Big)\otimes\mathcal{C}(a,b) & \xrightarrow{\quad\alpha\quad} & \mathcal{C}(c,d)\otimes\Big(\mathcal{C}(b,c)\otimes\mathcal{C}(a,b)\Big)\\
\downarrow{\scriptstyle \circ\otimes id} & & \downarrow{\scriptstyle id\otimes\circ}\\
\mathcal{C}(b,d)\otimes\mathcal{C}(a,b) & & \mathcal{C}(c,d)\otimes\mathcal{C}(a,c)\\
& {\scriptstyle\circ}\searrow \quad \swarrow{\scriptstyle\circ} & \\
& \mathcal{C}(a,d) &
\end{array}$$

单位律则由 $\mathcal{V}$ 中的单位子表达：

$$\begin{array}{ccc}
I\otimes\mathcal{C}(a,b) & \xrightarrow{\lambda} & \mathcal{C}(a,b)\\
{\scriptstyle j_b\otimes id}\downarrow & \nearrow{\scriptstyle\circ} & \\
\mathcal{C}(b,b)\otimes\mathcal{C}(a,b) & &
\end{array}
\qquad
\begin{array}{ccc}
\mathcal{C}(a,b)\otimes I & \xrightarrow{\rho} & \mathcal{C}(a,b)\\
{\scriptstyle id\otimes j_a}\downarrow & \nearrow{\scriptstyle\circ} & \\
\mathcal{C}(a,b)\otimes\mathcal{C}(a,a) & &
\end{array}$$

注意，这些都是 $\mathcal{V}$ 中的图表，在这里我们确实有形成同态集的箭头。我们仍然在不同的层次上回到了集合论。

一个丰盈于 $\mathcal{V}$ 的范畴也称为 $\mathcal{V}$-范畴。接下来我们将假设丰盈范畴是对称幺半范畴，因此我们可以形成相反范畴和积 $\mathcal{V}$-范畴。

与 $\mathcal{C}$ 对立的 $\mathcal{V}$-范畴 $\mathcal{C}^{op}$ 通过逆转同态对象获得，即：

$$\mathcal{C}^{op}(a,b)=\mathcal{C}(b,a)$$

对立范畴中的复合涉及逆转同态对象的顺序，因此它只有在张量积是对称的情况下才有效。

我们还可以定义 $\mathcal{V}$-范畴的张量积；同样，前提是 $\mathcal{V}$ 是对称的。两个 $\mathcal{V}$-范畴 $\mathcal{C}\otimes\mathcal{D}$ 的积具有对象对，分别来自每个范畴。这些对之间的同态对象被定义为张量积：

$$(\mathcal{C}\otimes\mathcal{D})(\langle c,d\rangle,\langle c^{¨},d^{¨}\rangle)=\mathcal{C}(c,c^{¨})\otimes\mathcal{D}(d,d^{¨})$$

我们需要张量积的对称性来定义复合。事实上，我们需要在中间交换两个同态对象，然后才能应用两个可用的复合：

$$\circ\colon\Big(\mathcal{C}(c^{¨},c^{¨¨},)\otimes\mathcal{D}(d^{¨},d^{¨¨})\Big)\otimes\Big(\mathcal{C}(c,c^{¨})\otimes\mathcal{D}(d,d^{¨})\Big)\to\mathcal{C}(c,c^{¨¨})\otimes\mathcal{D}(d,d^{¨¨})$$

单位箭头是两个单位的张量积：

$$I_{\mathcal{C}} \otimes I_{\mathcal{D}} \xrightarrow{j_c \otimes j_d} \mathcal{C}(c,c) \otimes \mathcal{D}(d,d)$$

Exercise 20.1.1. *在 $\mathcal{V}$-范畴 $\mathcal{C}^{op}$ 中定义复合和单位。*

Exercise 20.1.2. *证明每个 $\mathcal{V}$-范畴 $\mathcal{C}$ 都有一个底层的普通范畴 $\mathcal{C}_0$，其对象相同，但其同态集由同态对象的（幺半全局）元素给出，即 $\mathcal{V}(I, \mathcal{C}(a,b))$ 的元素。*

Examples（示例）

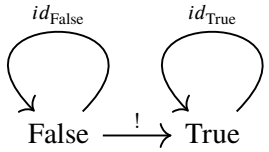
从这个新的视角来看，我们迄今为止研究的普通范畴可以说是通过单子积 和单位集 1 在幺半范畴 (Set, , 1) 中得到了丰盈的。

有趣的是，2-范畴可以被视为在 Cat 上丰盈。实际上，2-范畴中的 1-态射本身就是另一个范畴中的对象。2-态射就是该范畴中的箭头。特别地，小范畴的 2-范畴 Cat 在自身中得到了丰盈。其同态对象是函子范畴，这些对象位于 Cat 中。

Preorders（预序）

丰盈并不总是意味着增加更多的内容。有时它看起来更像是一种贫乏，比如在一个行箭范畴上的丰盈情况。

这个范畴只有两个对象，为了这个构造的目的，我们称它们为 False 和 True。从 False 到 True 有一个箭头（不计算单位箭头），这使得 False 成为初始对象，True 成为终对象。



要将其变为幺半范畴，我们定义张量积，使得：

$$\text{True} \otimes \text{True} = \text{True}$$

其他所有组合都产生False。True 是幺元，因为：

$$\text{True} \otimes x = x$$

在幺半行箭范畴上丰盈的范畴称为*预序*。任何两个对象之间的同态对象 $\mathcal{C}(a,b)$ 可以是 False 或 True。我们解释 True 为 a 在预序中领先于 b，即 $a \leq b$。False 表示这两个对象不相关。

复合的一个重要属性是，如果左边的两个同态对象都是 True，那么右边也必须是 True。（它不能是 False，因为没有箭头从 True 到 False。）在预序解释中，这意味着 $\leq$ 是传递的：

$$b \leq c \text{ á } a \leq b \implies a \leq c$$

通过相同的推理，存在单位箭头：

$$j_a \colon \text{True} \to \mathcal{C}(a, a)$$

意味着 $\mathcal{C}(a, a)$ 总是 True。在预序解释中，这意味着 $\leq$ 是自反的，即 $a \leq a$。

注意，预序不排除循环，特别是可以有 $a \leq b$ 且 $b \leq a$ 但 a 不等于 b。

也可以在不诉诸丰盈的情况下定义预序，作为*薄范畴*——在其中，任何两个对象之间最多有一个箭头。

Self-enrichment（自丰盈）

任何笛卡尔闭范畴 $\mathcal{V}$ 都可以视为自丰盈的。这是因为每个外部同态集 $\mathcal{C}(a, b)$ 都可以被内部同态 b^a（箭头的对象）替换。

实际上，每个*幺半闭范畴* $\mathcal{V}$ 都是自丰盈的。回想一下，在幺半闭范畴中我们有同态函子伴随关系：

$$\mathcal{V}(a \otimes b, c) \ddot{\circ} \mathcal{V}(a, [b, c])$$

此伴随的余单元作为评估态射起作用：

$$\varepsilon_{bc} \colon [b, c] \otimes b \to c$$

要在这个自丰盈的范畴中定义复合，我们需要一个箭头：

$$\circ \colon [b, c] \otimes [a, b] \to [a, c]$$

诀窍是考虑整个同态集，并表明我们始终可以在其中选取一个规范元素。我们从集合开始：

$$\mathcal{V}([b, c] \otimes [a, b], [a, c])$$

我们可以使用伴随关系将其重写为：

$$\mathcal{V}(([b, c] \otimes [a, b]) \otimes a, c)$$

现在我们只需从这个同态集中选取一个元素即可。我们通过构造以下组合来完成：

$$([b, c] \otimes [a, b]) \otimes a \xrightarrow{\alpha} [b, c] \otimes ([a, b] \otimes a) \xrightarrow{id \otimes \varepsilon_{ab}} [b, c] \otimes b \xrightarrow{\varepsilon_{bc}} c$$

我们使用了结合子和伴随关系的余单元。

我们还需要一个箭头来定义单位：

$$j_a \colon I \to [a, a]$$

同样，我们可以将其作为同态集 $\mathcal{V}(I, [a, a])$ 的成员选取。我们使用伴随关系：

$$\mathcal{V}(I, [a, a]) \ddot{\circ} \mathcal{V}(I \otimes a, a)$$

我们知道这个同态集包含左幺子 λ，因此我们可以用它来定义 j_a。

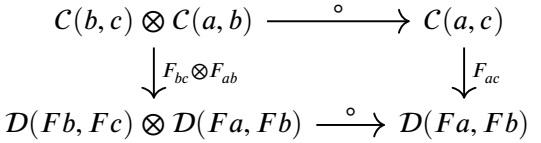
20.2 $\mathcal{V}$-Functors（$\mathcal{V}$-函子）

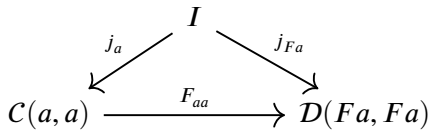
普通函子将对象映射到对象，将箭头映射到箭头。类似地，一个丰盈函子 F 将对象映射到对象，但它必须将同态对象映射到同态对象。只有当源范畴 $\mathcal{C}$ 中的同态对象属于与目标范畴 $\mathcal{D}$ 中的同态对象相同的范畴时，这才有可能。换句话说，这两个范畴必须丰盈于相同的 $\mathcal{V}$。F 在同态对象上的作用使用 $\mathcal{V}$ 中的箭头定义：

$$F_{ab} \colon \mathcal{C}(a,b) \to \mathcal{D}(Fa, Fb)$$

为清晰起见，我们在 F 的下标中指定了对象对。

函子必须保持复合和单位。可以将它们表示为 $\mathcal{V}$ 中的交换图表：





注意我对两个不同的复合使用了相同的符号 $\circ$，对两个不同的单位映射使用了相同的 j。它们的含义可以从上下文中推导出来。

和以前一样，所有的图表都在范畴 $\mathcal{V}$ 中。

The Hom-functor（同态函子）

在一个丰盈于幺半闭范畴 $\mathcal{V}$ 的范畴中，同态函子是一个丰盈函子：

$$\mathrm{Hom}_{\mathcal{C}} \colon \mathcal{C}^{op} \otimes \mathcal{C} \to \mathcal{V}$$

在这里，为了定义一个丰盈函子，我们必须将 $\mathcal{V}$ 视为自丰盈。

很明显，这个函子在（对象对）上的作用是如何进行的：

$$\mathrm{Hom}_{\mathcal{C}}\langle a, b\rangle = \mathcal{C}(a,b)$$

为了定义一个丰盈函子，我们必须定义同态函子对同态对象的作用。在这里，源范畴是 $\mathcal{C}^{op} \otimes \mathcal{C}$，目标范畴是 $\mathcal{V}$，两者都丰盈于 $\mathcal{V}$。让我们考虑从 $\langle a, a^{¨}\rangle$ 到 $\langle b, b^{¨}\rangle$ 的同态对象。同态函子对该同态对象的作用是 $\mathcal{V}$ 中的一个箭头：

$$\mathrm{Hom}_{\langle a,a^{¨}\rangle\langle b,b^{¨}\rangle} \colon (\mathcal{C}^{op} \otimes \mathcal{C})(\langle a, a^{¨}\rangle, \langle b, b^{¨}\rangle) \to \mathcal{V}(\mathrm{Hom}\langle a, a^{¨}\rangle, \mathrm{Hom}\langle b, b^{¨}\rangle)$$

根据积范畴的定义，源是两个同态对象的张量积。目标是 $\mathcal{V}$ 中的内部同态。我们正在寻找的箭头是：

$$\mathcal{C}(b,a) \otimes \mathcal{C}(a^{¨}, b^{¨}) \to [\mathcal{C}(a, a^{¨}), \mathcal{C}(b, b^{¨})]$$

我们可以使用柯里化的同态函子伴随关系来展开内部同态：

$$\Big(\mathcal{C}(b,a)\otimes\mathcal{C}(a^{¨},b^{¨})\Big)\otimes\mathcal{C}(a,a^{¨})\to\mathcal{C}(b,b^{¨})$$

我们可以通过重新排列积并两次应用复合来构造这个箭头。

在丰盈环境中，定义从 a 到 b 的单个态射最接近的方法是使用一个来自单位对象的箭头。我们将一个同态对象的（幺半全局）元素定义为 $\mathcal{V}$ 中的一个态射：

$$f\colon I\to\mathcal{C}(a,b)$$

我们可以定义提升此箭头的含义，使用同态函子。例如，将第一个参数保持为常量，我们可以定义：

$$\mathcal{C}(c,f)\colon\mathcal{C}(c,a)\to\mathcal{C}(c,b)$$

作为以下组合：

$$\mathcal{C}(c,a)\xrightarrow{\lambda^{*1}}I\otimes\mathcal{C}(c,a)\xrightarrow{f\otimes id}\mathcal{C}(a,b)\otimes\mathcal{C}(c,a)\xrightarrow{\circ}\mathcal{C}(c,b)$$

类似地，f 的反变提升：

$$\mathcal{C}(f,c)\colon\mathcal{C}(b,c)\to\mathcal{C}(a,c)$$

可以定义为：

$$\mathcal{C}(b,c)\xrightarrow{\rho^{*1}}\mathcal{C}(b,c)\otimes I\xrightarrow{id\otimes f}\mathcal{C}(b,c)\otimes\mathcal{C}(a,b)\xrightarrow{\circ}\mathcal{C}(a,c)$$

我们在普通范畴论中学习的许多熟悉构造都有其丰盈对应物，产品被张量积取代，Set 被 $\mathcal{V}$ 取代。

Exercise 20.2.1. 什么是两个预序之间的函子?

Enriched co-presheaves（丰盈余预层）

余预层，即 Set-值函子，在范畴论中扮演着重要角色，因此很自然地问在丰盈环境中的对应物是什么。余预层的推广是 $\mathcal{V}$-函子 $\mathcal{C}\to\mathcal{V}$。只有当 $\mathcal{V}$ 可以成为一个 $\mathcal{V}$-范畴，即当它是幺半闭的时，这才有可能。

一个丰盈余预层将 $\mathcal{C}$ 的对象映射到 $\mathcal{V}$ 的对象，并将 $\mathcal{C}$ 的同态对象映射到 $\mathcal{V}$ 的内部同态：

$$F_{ab}\colon\mathcal{C}(a,b)\to[Fa,Fb]$$

特别地，同态函子是一个 $\mathcal{V}$-值 $\mathcal{V}$-函子的例子：

$$\mathrm{Hom}\colon\mathcal{C}^{op}\otimes\mathcal{C}\to\mathcal{V}$$

同态函子是丰盈态射的特例，它被定义为：

$$\mathcal{C}^{op}\otimes\mathcal{D}\to\mathcal{V}$$

Exercise 20.2.2. 张量积是 $\mathcal{V}$ 中的一个函子：

$$\otimes : \mathcal{V} \ \ \mathcal{V} \to \mathcal{V}$$

证明如果 $\mathcal{V}$ 是幺半闭的，张量积定义了一个 $\mathcal{V}$-函子。提示：定义它对内部同态的作用。

Functorial strength and enrichment（函子强度与丰盈）

当我们讨论 monads（幺半群）时，我提到一个使它们在编程中起作用的重要属性。定义 monads 的端函子必须是强大的，这样我们才能在 monadic 代码中访问外部上下文。

事实证明，我们在 Haskell 中定义端函子的方式使它们自动变得强大。原因是强度与丰盈相关，正如我们所见，笛卡尔闭范畴是自丰盈的。让我们从一些定义开始。

在幺半范畴中，端函子的函子强度定义为具有如下分量的自然变换：

$$\sigma_{ab} : a \otimes F(b) \to F(a \otimes b)$$

有一些显而易见的一致性条件使得强度尊重张量积的性质。这是结合性条件：

$$\begin{array}{ccccc}
(a \otimes b) \otimes F(c) & & \xrightarrow{\sigma_{(a\otimes b)c}} & & F((a \otimes b) \otimes c) \\
\downarrow{\scriptstyle \alpha} & & & & \downarrow{\scriptstyle F(\alpha)} \\
a \otimes (b \otimes F(c)) & \xrightarrow{a\otimes\sigma_{bc}} & a \otimes F(b \otimes c) & \xrightarrow{\sigma_{a(b\otimes c)}} & F(a \otimes (b))
\end{array}$$

这是单位条件：

$$\begin{array}{ccc}
I \otimes F(a) & \xrightarrow{\sigma_{Ia}} & F(I \otimes a) \\
& {\scriptstyle \lambda} \searrow & \downarrow{\scriptstyle F(\lambda)} \\
& & F(a)
\end{array}$$

在一般幺半范畴中，这被称为*左强度*，并且有一个对应的右强度定义。在对称幺半范畴中，这两者是等价的。

一个丰盈端函子将同态对象映射到同态对象：

$$F_{ab} : C(a, b) \to C(Fa, Fb)$$

如果我们将幺半闭范畴 $\mathcal{V}$ 视为自丰盈，内部同态就是内部同态，因此丰盈端函子配备了以下映射：

$$F_{ab} : [a, b] \to [Fa, Fb]$$

将其与 Haskell 中的`Functor`定义进行比较：

```
class Functor f where
fmap :: (a -> b) -> (f a -> f b)
```

此定义中涉及的函数类型，(a -> b)和(f a -> f b)，是内部同态。因此，Haskell 的Functor确实是一个丰盈函子。

在 Haskell 中，我们通常不区分外部和内部同态，因为它们的元素集是同构的。这是柯里化伴随关系的一个简单结果：

$$\mathcal{C}(1 \quad b, c) \ddot{\text{o}} \mathcal{C}(1, [b, c])$$

以及终对象是笛卡尔积的单位这一事实。

事实证明，在自丰盈范畴 $\mathcal{V}$ 中，每个强端函子都是自动丰盈的。实际上，为了证明一个函子是丰盈的，我们需要定义内部同态之间的映射，即一个同态集的元素：

$$F_{ab} \ddot{\text{E}} \mathcal{V}([a, b], [Fa, Fb])$$

使用同态伴随，这等同于：

$$\mathcal{V}([a, b] \otimes Fa, Fb)$$

我们可以通过将强度和伴随的余单元（评估态射）组合起来构造此映射：

$$[a, b] \otimes Fa \xrightarrow{\sigma_{[a,b]a}} F([a, b] \otimes a) \xrightarrow{F\epsilon_{ab}} Fb$$

反过来，每个 $\mathcal{V}$ 中的丰盈端函子都是强的。为了展示强度，我们需要定义映射 σ_{ab}，或等价地（通过同态伴随）：

$$a \to [Fb, F(a \otimes b)]$$

回想一下同态伴随关系的单位定义，协同评估态射：

$$\eta_{ab} \colon a \to [b, a \otimes b]$$

我们构造以下复合：

$$a \xrightarrow{\eta_{ab}} [b, a \otimes b] \xrightarrow{F_{b,a\otimes b}} [Fb, F(a \otimes b)]$$

这可以直接翻译成 Haskell：

```
strength :: Functor f => (a, f b) -> f (a, b)
strength = uncurry (\a -> fmap (coeval a))
```

协同评估的定义如下：

```
coeval :: a -> (b -> (a, b))
coeval a = \b -> (a, b)
```

由于柯里化和评估内置在 Haskell 中，我们可以进一步简化这个公式：

```
strength :: Functor f => (a, f b) -> f (a, b)
strength (a, bs) = fmap (a, ) bs
```

20.3 $\mathcal{V}$-Natural Transformations（$\mathcal{V}$-自然变换）

两个从 $\mathcal{C}$ 到 $\mathcal{D}$ 的函子 F 和 G 之间的普通自然变换是从同态集中选择箭头 $\mathcal{D}(Fa, Ga)$。在丰盈环境中，我们没有箭头，因此我们能做的最好的事情就是使用单位对象 I 来进行选择。我们在 a 处定义 $\mathcal{V}$-自然变换的一个分量作为一个箭头：

$$\nu_a \colon I \to \mathcal{D}(Fa, Ga)$$

自然性条件有点棘手。标准的自然性方块涉及到一个任意箭头 $f \colon a \to b$ 的提升，以及以下组合的相等性：

$$\nu_b \circ Ff = Gf \circ \nu_a$$

让我们考虑这个方程中涉及的同态集。我们正在提升一个态射 f Ë $\mathcal{C}(a, b)$。方程两侧的组合都是 $\mathcal{D}(Fa, Gb)$ 的元素。

在左侧，我们有箭头 $\nu_b \circ Ff$。复合本身是从两个同态集的积映射的：

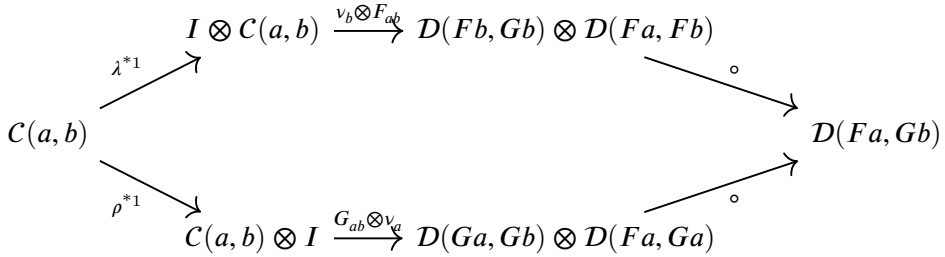
$$\mathcal{D}(Fb, Gb) \quad \mathcal{D}(Fa, Fb) \to \mathcal{D}(Fa, Gb)$$

同样地，在右侧我们有 $Gf \circ \nu_a$，它是一个复合：

$$\mathcal{D}(Ga, Gb) \quad \mathcal{D}(Fa, Ga) \to \mathcal{D}(Fa, Gb)$$

在丰盈环境中，我们必须处理同态对象而不是同态集，并且自然变换的分量是使用单位 I 选择的。我们始终可以使用左或右幺子的逆凭空生成单位。

总而言之，自然性条件表示为以下交换图表：



这也适用于普通范畴，在这里我们可以通过首先从 $\mathcal{C}(a, b)$ 中选择一个 f 来跟踪此图表中的两条路径。然后我们可以使用 ν_b 和 ν_a 选择自然变换的分量。我们还可以使用 F 或 G 提升 f。最后，我们使用复合来重现自然性方程。

如果我们使用早先定义的同态函子对同态对象的全局元素的作用，此图表可以进一步简化。自然变换的分量定义为这样的全局元素：

$$\nu_a \colon I \to \mathcal{D}(Fa, Ga)$$

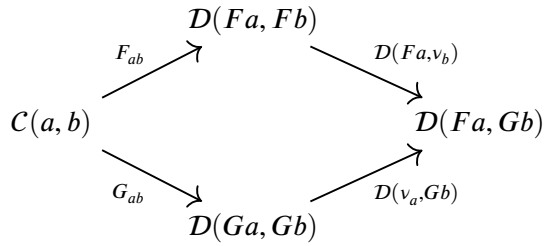
我们可以使用两种这样的提升：

$$\mathcal{D}(d, \nu_b) \colon \mathcal{D}(d, Fb) \to \mathcal{D}(d, Gb)$$

和：

$$\mathcal{D}(\nu_a, d) \colon \mathcal{D}(Ga, d) \to \mathcal{D}(Fa, d)$$

我们得到的东西更像是熟悉的自然性方块：



F 和 G 之间的 $\mathcal{V}$-自然变换形成了一个我们称之为 $\mathcal{V}\text{-nat}(F, G)$ 的集合。

我们之前看到，在普通范畴中，自然变换的集合可以写成一个端：

$$[\mathcal{C}, \mathcal{D}](F, G) \cong \int_a \mathcal{D}(Fa, Ga)$$

事实证明，端和余端可以为丰盈态射定义，因此该公式适用于丰盈自然变换。不同之处在于，它不是定义自然变换的集合 $\mathcal{V}\text{-nat}(F, G)$，而是定义了 $\mathcal{V}$ 中的自然变换的对象 $[\mathcal{C}, \mathcal{D}](F, G)$。

$\mathcal{V}$-态射 $\mathcal{C}^{op} \otimes \mathcal{C} \to \mathcal{V}$ 的端的定义类似于我们在普通态射中看到的定义。例如，端是 $\mathcal{V}$ 中的一个对象 e，它配备了 $\pi \colon e \to P$ 的超自然变换，在这些对象中是普遍的。

20.4 Yoneda Lemma（Yoneda 引理）

普通的 Yoneda 引理涉及一个 Set-值函子 F 和自然变换的集合：

$$[\mathcal{C}, \mathrm{Set}](\mathcal{C}(c, *), F) \cong Fc$$

为了将其推广到丰盈环境，我们将考虑一个 $\mathcal{V}$-值函子 F。和以前一样，只要它是闭的，我们就可以将 $\mathcal{V}$ 视为自丰盈，因此我们可以谈论 $\mathcal{V}$-值 $\mathcal{V}$-函子。

弱版本的 Yoneda 引理处理的是 $\mathcal{V}$-自然变换的集合。因此，我们必须将右侧转换为集合。这是通过获取 Fc 的（幺半全局）元素来完成的。我们得到：

$$\mathcal{V}\text{-nat}(\mathcal{C}(c, *), F) \cong \mathcal{V}(I, Fc)$$

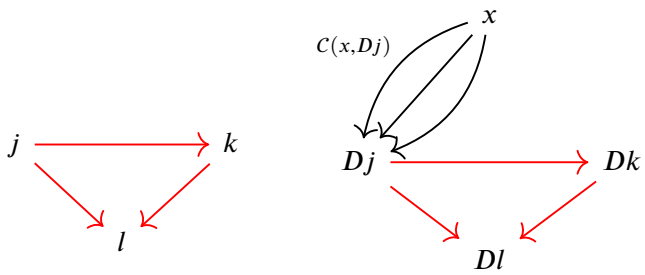
强版本的 Yoneda 引理处理的是 $\mathcal{V}$ 的对象，并使用 $\mathcal{V}$ 中内部同态上的端来表示自然变换的对象：

$$\int_x [\mathcal{C}(c, x), Fx] \cong Fc$$

20.5 Weighted Limits（加权极限）

极限（和余极限）围绕着交换三角形构建，因此它们无法立即翻译到丰盈环境中。问题是，锥体是由“线”，即单个态射构成的。你可以将同态集视为厚线束，每条线的厚度为零。在构造锥体时，您选择同态集中的单根线。我们必须用更厚的东西替换这些线。

考虑一个图表，即从索引范畴 $\mathcal{J}$ 到目标范畴 $\mathcal{C}$ 的函子。锥体的顶点 x 的线是从同态集 $\mathcal{C}(x, Dj)$ 中选择的，其中 j 是 $\mathcal{J}$ 的一个对象。



这种选择的 j’ 线可以描述为从单集合 1 的函数：

$$\gamma_j \colon 1 \to \mathcal{C}(x, Dj)$$

我们可以尝试将这些函数汇集成一个自然变换：

$$\gamma \colon \Delta_1 \to \mathcal{C}(x, D^*)$$

其中 Δ_1 是一个常量函子，将 $\mathcal{J}$ 的所有对象映射到单集合。自然性条件确保形成锥体侧面的三角形是交换的。

顶点为 x 的所有锥体的集合由自然变换的集合给出：

$$[\mathcal{J}, \mathrm{Set}](\Delta_1, \mathcal{C}(x, D^*))$$

此重构使我们更接近丰盈环境，因为它用同态集而不是单个态射来重新表述问题。我们可以从考虑 $\mathcal{J}$ 和 $\mathcal{C}$ 丰盈于 $\mathcal{V}$ 开始，在这种情况下 D 将是一个 $\mathcal{V}$-函子。

只有一个问题：我们如何定义一个常量 $\mathcal{V}$-函子 $\Delta_x \colon \mathcal{C} \to \mathcal{D}$？它对对象的作用是显而易见的：它将 $\mathcal{C}$ 中的所有对象映射到 $\mathcal{D}$ 中的一个对象 x。但是，它对同态对象应该做什么？

一个普通的常量函子 Δ_x 将 $\mathcal{C}(a, b)$ 中的所有态射映射到 $\mathcal{D}(x, x)$ 中的单位。在丰盈环境中，然而，$\mathcal{D}(x, x)$ 是一个没有内部结构的对象。即使它碰巧是单位 I，也无法保证对于每个同态对象 $\mathcal{C}(a, b)$ 我们都可以找到一个箭头到 I；即使有，也不一定是唯一的。换句话说，没有理由相信 I 是终对象。

解决方案是“涂抹奇点”：我们应该使用其他的“加权”函子 $W \colon \mathcal{J} \to \mathrm{Set}$ 来选择一个更厚的“圆柱”而不是使用常量函子选择单根线。顶点为 x 的加权锥体是自然变换集合的元素：

$$[\mathcal{J}, \mathrm{Set}]\,(W, \mathcal{C}(x, D^*))$$

一个加权极限，也称为索引极限 $\lim^W D$，然后定义为普遍加权锥体。它意味着对于具有顶点 x 的任何加权锥体，存在唯一的从 x 到 $\lim^W D$ 的态射来对其进行因子分解。通过定义加权极限的同态集合的自然性来保证因子分解：

$$\mathcal{C}(x, \lim^W D) \ddot{\circ} [\mathcal{J}, \mathrm{Set}](W, \mathcal{C}(x, D^*))$$

常规的非加权极限通常称为锥形极限，它对应于使用常量函子作为权重。

这个定义可以几乎逐字翻译到丰盈环境中，通过将 Set 替换为 $\mathcal{V}$：

$$\mathcal{C}(x, \lim^W D) \ddot{\circ} [\mathcal{J}, \mathcal{V}](W, \mathcal{C}(x, D^*))$$

当然，这个公式中的符号的含义已经改变。两侧现在都是 $\mathcal{V}$ 中的对象。左侧是 $\mathcal{C}$ 中的同态对象，右侧是两个 $\mathcal{V}$-函子之间的自然变换对象。

对偶地，定义一个加权余极限为自然同构：

$$\mathcal{C}(\mathrm{colim}^W D, x) \ddot{\circ} [\mathcal{J}^{op}, \mathcal{V}](W, \mathcal{C}(D^*, x))$$

在这里，余极限由来自对立范畴的函子 $W \colon \mathcal{J}^{op} \to \mathcal{V}$ 加权。

加权（余）极限，在普通范畴和丰盈范畴中，都扮演着根本性角色：它们可以用于重新表述许多熟悉的构造，如（余）端，Kan 扩张等。

20.6 Ends as Weighted Limits（端作为加权极限）

一个端与积或更一般的极限有很多共同点。如果你仔细看，投影 $\pi_x \colon e \to P\langle a, a\rangle$ 形成了锥体的侧面；除了交换三角形，我们还有楔子。事实证明，我们可以将端表达为加权极限。这样表述的优点是它也适用于丰盈环境。

我们已经看到，$\mathcal{V}$-值 $\mathcal{V}$-态射的端可以使用超自然变换的更基本概念来定义。这反过来允许我们定义自然变换的对象，从而使我们能够定义加权极限。我们现在可以继续并扩展端的定义，以在 $\mathcal{V}$-范畴 $\mathcal{D}$ 中与混合变差值的更一般的 $\mathcal{V}$-函子一起使用：

$$P \colon \mathcal{C}^{op} \otimes \mathcal{C} \to \mathcal{D}$$

我们将使用此函子作为 $\mathcal{D}$ 中的图表。

在这一点上，数学家们开始担心规模问题。毕竟我们将整个范畴——平方——嵌入 $\mathcal{D}$ 中的一个单一图表中。为了避免规模问题，我们只假设 $\mathcal{C}$ 是小的；也就是说，它的对象形成一个集合。

我们想对由 P 定义的图表采取加权极限。权重必须是一个 $\mathcal{V}$-函子 $\mathcal{C}^{op} \otimes \mathcal{C} \to \mathcal{V}$。我们有一个总是在我们手边的函子，同态函子 $\mathrm{Hom}_{\mathcal{C}}$。我们将使用它将端定义为加权极限：

$$\int_c P\langle c, c\rangle = \lim^{\mathrm{Hom}} P$$

首先，让我们说服自己这个公式在普通（Set-丰盈）范畴中有效。由于端是通过其映射入属性定义的，让我们考虑从任意对象 d 到加权极限的映射，并使用标准的 Yoneda 技巧来证明同构。根据定义，我们有：

$$\mathcal{D}(d, \lim^{\mathrm{Hom}} P) \ \ddot{\mathrm{o}} \ [\mathcal{C}^{op} \quad \mathcal{C}, \mathrm{Set}](\mathcal{C}(*, =), \mathcal{D}(d, P(*, =))$$

我们可以将自然变换的集合重写为在对象对 $\langle c, c^{¨} \rangle$ 上的端：

$$\int_{\langle c, c^{¨} \rangle} \mathrm{Set}(\mathcal{C}(c, c^{¨}), \mathcal{D}(d, P\langle c, c^{¨} \rangle))$$

使用 Fubini 定理，这等效于迭代端：

$$\int_{c} \int_{c^{¨}} \mathrm{Set}(\mathcal{C}(c, c^{¨}), \mathcal{D}(d, P\langle c, c^{¨} \rangle))$$

我们现在可以应用 ninja-Yoneda 引理对 $c^{¨}$ 进行积分。结果是：

$$\int_{c} \mathcal{D}(d, P\langle c, c \rangle) \ \ddot{\mathrm{o}} \ \mathcal{D}(d, \int_{c} P\langle c, c \rangle)$$

我们使用连续性将端推到同态函子下。由于 d 是任意的，我们得出结论，对于普通范畴：

$$\lim^{\mathrm{Hom}} P \ \ddot{\mathrm{o}} \int_{c} P\langle c, c \rangle$$

这证明了我们使用加权极限在丰盈情况下定义端是合理的。

类似的公式适用于余端，除了我们使用对立范畴中的同态函子 $\mathrm{Hom}_{\mathcal{C}^{op}}$ 作为权重的余极限：

$$\int^{c} P\langle c, c \rangle = \mathrm{colim}^{\mathrm{Hom}_{\mathcal{C}^{op}}} P$$

Exercise 20.6.1. 证明对于普通的 Set-丰盈范畴，余端的加权余极限定义再现了先前的定义。提示：使用余端的映射出属性。

Exercise 20.6.2. 证明，只要双方存在，以下等式在普通（Set-丰盈）范畴中成立（可以推广到丰盈环境）：

$$lim^{W} D \ \ddot{\mathrm{o}} \int_{j \colon \mathcal{J}} W j \pitchfork D j$$

$$colim^{W} D \ \ddot{\mathrm{o}} \int^{j \colon \mathcal{J}} W j \cdot D j$$

提示：使用 *Yoneda* 技巧和幂与共幂的定义进行映射入/出。

20.7 Kan Extensions（Kan 扩张）

我们已经看到如何使用来自奇点范畴 1 的函子将极限和余极限表达为 Kan 扩张。加权极限让我们摆脱了奇点，明智地选择权重使我们能够将 Kan 扩张表达为加权极限。

首先，让我们推导普通的 Set-丰盈范畴的公式。右 Kan 扩张定义为：

$$(\mathrm{Ran}_P F)e \ \ddot{\mathrm{o}} \int_c \mathcal{B}(e, Pc) \pitchfork Fc$$

我们将考虑从任意对象 d 到它的映射。推导遵循几个简单的步骤，主要是通过扩展定义。

我们从以下开始：

$$\mathcal{D}\Big(d, (\mathrm{Ran}_P F)e\Big)$$

并代入 Kan 扩张的定义：

$$\mathcal{D}\Big(d, \int_c \mathcal{B}(e, Pc) \pitchfork Fc\Big)$$

使用同态函子的连续性，我们可以将端拉出：

$$\int_c \mathcal{D}\Big(d, \mathcal{B}(e, Pc) \pitchfork Fc\Big)$$

然后我们使用 pitchfork 的定义：

$$\mathcal{D}(d, A \pitchfork d^{¨}) \ \ddot{\mathrm{o}} \ \mathrm{Set}\Big(A, \mathcal{D}(d, d^{¨})\Big)$$

得到：

$$\int_c \mathcal{D}\Big(d, \mathcal{B}(e, Pc) \pitchfork Fc\Big) \ \ddot{\mathrm{o}} \int_c \mathrm{Set}\Big(\mathcal{B}(e, Pc), \mathcal{D}(d, Fc)\Big)$$

这可以写成一个自然变换的集合：

$$[\mathcal{C}, \mathrm{Set}]\Big(\mathcal{B}(e, P^*), \mathcal{D}(d, F^*)\Big)$$

加权极限也通过自然变换的集合来定义：

$$\mathcal{D}(d, \lim^W F) \ \ddot{\mathrm{o}} \ [\mathcal{C}, \mathrm{Set}]\Big(W, \mathcal{D}(d, F^*)\Big)$$

我们得出最终结果：

$$\mathcal{D}(d, \lim^{\mathcal{B}(e, P^*)} F)$$

由于 d 是任意的，我们可以使用 Yoneda 技巧得出结论：

$$(\mathrm{Ran}_P F)e = \lim^{\mathcal{B}(e, P^*)} F$$

这个公式成为丰盈环境中右 Kan 扩张的定义。

类似地，左 Kan 扩张可以定义为加权余极限：

$$(\mathrm{Lan}_P F)e = \mathrm{colim}^{\mathcal{B}(P^*, e)} F$$

Exercise 20.7.1. 推导普通范畴左 *Kan* 扩张的公式。