

2
2.0

函数式编程之道

Bartosz Milewski

(最后更新: April 27, 2025)

Contents

Contents	i
前言	ii
集合论	ii
约定	iii
1 白板	1
1.1 类型与函数	1
1.2 阴阳	2
1.3 元素	3
1.4 箭头对象	4

2 组合	7
2.1 组合	7
2.2 函数应用	9
2.3 恒等态射	10
2.4 单态射	11
2.5 满同态	13
3 同构	15
3.1 同构对象	15
同构与双射	17
3.2 自然性	18
3.3 用箭头推理	19
反转箭头	22
4 和类型 (Sum Types)	25
4.1 布尔类型 (Bool)	25
示例	27
4.2 枚举类型	28
4.3 和类型 (Sum Types)	29
Maybe 类型	30
逻辑	31
4.4 余积范畴	31
一加零	31
某物加零	32
交换性	32
结合性	33
函子性	33
对称幺半群范畴	34
5 积类型	35
逻辑	36
元组和记录	36
5.1 笛卡尔范畴	37

元组算术	37
函子性	39
5.2 对偶性	39
5.3 么半范畴	41
么半群	42
6 函数类型	45
消去规则	45
引入规则	46
柯里化	47
与 λ 演算的关系	48
肯定前件	49
6.1 和与积的再探	50
和类型	50
积类型	50
函子性的再探	51
6.2 函数类型的函子性	52
6.3 双笛卡尔闭范畴	53
分配律	53
7 递归	57
7.1 自然数	57
引入规则	58
消除规则	58
在编程中	60
7.2 列表	61
消除规则	61
7.3 函子性	62
8 函子	65
8.1 范畴	65
集合范畴	65
对偶范畴	66

积范畴	66
切片范畴	67
余切片范畴	67
8.2 函子 (Functors)	67
范畴之间的函子	68
8.3 编程中的函子	70
自函子	70
双函子	71
逆变函子	72
Profunctors	72
8.4 Hom-函子	73
8.5 函子复合	74
范畴的范畴	75
9 自然变换	77
9.1 Hom-函子之间的自然变换	77
9.2 函子间的自然变换	79
9.3 编程中的自然变换	81
自然变换的垂直复合	82
函子范畴	83
自然变换的水平复合	84
Whiskering (whiskering)	86
交换律	88
9.4 重新审视万有构造	88
选择对象	89
余跨度作为自然变换	89
余跨度的函子性	90
和作为万有余跨度	91
积作为万有跨度	92
指数	93
9.5 极限与余极限	95
等化子	96

余等化子	97
终端对象的存在性	98
9.6 米田引理 (Yoneda Lemma)	99
编程中的米田引理	101
逆变米田引理	102
9.7 Yoneda 嵌入	102
9.8 可表函子	105
猜谜游戏	106
编程中的可表函子	106
9.9 2-范畴 Cat	107
9.10 常用公式	107
10 伴随	109
10.1 柯里化伴随	109
10.2 和与积的伴随	110
对角函子	110
和伴随	111
积伴随	112
分配律	112
10.3 函子之间的伴随关系	113
10.4 极限与余极限作为伴随	115
10.5 伴随的单位与余单位	115
三角恒等式	118
柯里化伴随的单位与余单位	119
10.6 使用通用箭头的伴随	120
逗号范畴	120
通用箭头	121
从伴随中得到的通用箭头	121
从通用箭头构造伴随	122
10.7 伴随函子的性质	122
左伴随函子保持余极限	122
右伴随函子保持极限	124

10.8 Freyd 伴随函子定理	125
预序中的 Freyd 定理	125
解集条件	127
去函数化 (Defunctionalization)	128
10.9 自由/遗忘伴随	130
么半群范畴	131
自由么半群	132
编程中的自由么半群	133
10.10 伴随关系的范畴	134
10.11 抽象层次	135
11 代数	137
11.1 自函子的代数	137
11.2 代数范畴	138
初始代数	139
11.3 Lambek 引理与不动点	140
Haskell 中的不动点	141
11.4 Catamorphisms (Catamorphisms)	141
示例	142
列表作为初始代数	143
11.5 从普遍性看初始代数	145
11.6 初始代数作为余极限	146
证明	148
12 余代数	151
12.1 自函子的余代数	151
12.2 余代数的范畴	152
12.3 变形 (Anamorphisms)	154
无限数据结构	154
12.4 合态射 (Hylomorphisms)	155
阻抗不匹配	156
12.5 从普遍性看终端余代数	156
12.6 终余代数作为极限	158

13 效应	161
13.1 带有副作用的编程	161
部分性	162
日志记录	162
环境	163
状态	163
非确定性	163
输入/输出	164
延续	164
组合有副作用的计算	165
14 应用函子 (Applicative Functors)	167
14.1 并行组合	167
么半群函子	167
应用函子	168
14.2 应用函子实例	170
部分性	170
日志记录	170
环境	170
状态	170
非确定性	171
延续	171
输入/输出	172
解析器	172
并发和并行	173
Do Notation	173
应用函子的组合	173
14.3 范畴论中的么半函子	173
松弛么半函子	174
函子强度	175
闭函子	176
15 单子 (Monads)	177

15.1	副作用的顺序组合	177
15.2	其他定义	179
	作为应用函子的单子	180
15.3	单子实例	181
	部分性	181
	日志记录	182
	环境	182
	状态	182
	非确定性	183
	延续	183
	输入/输出	184
15.4	Do 表示法	185
15.5	延续传递风格	186
	尾递归与 CPS	186
	使用命名函数	187
	去函数化	188
15.6	范畴论中的单子 (Monad)	188
	替换	188
	单子作为么半群	189
15.7	自由单子 (Free Monads)	191
	单子的范畴	191
	自由单子	191
	Haskell 中的自由单子	192
	栈计算器示例	195
16	单子与伴随	197
16.1	弦图	197
	单子的弦图	200
	伴随的弦图	201
16.2	从伴随函子导出的单子	203
16.3	伴随函子生成的单子示例	204
	自由么半群与列表单子	204

柯里化伴随函子与状态单子	205
M-集与 Writer 单子	207
点对象与 Maybe 单子	209
延续单子	209
16.4 单子变换器 (Monad Transformers)	210
状态单子变换器	212
16.5 单子代数	213
Eilenberg-Moore 范畴	215
Kleisli 范畴	216
17 余单子 (Comonads)	219
17.1 编程中的余单子	219
Stream 余单子	220
17.2 余单子的范畴定义	221
余幺半群	222
17.3 伴随函子导出的余单子	223
余状态余单子	223
余单子余代数	225
透镜	225
18 端与余端	229
18.1 Profunctors (函子)	229
Collages (拼贴)	230
Profunctors 作为关系	231
Haskell 中的 Profunctor 组合	231
18.2 余端 (Coends)	232
外自然变换 (Extranatural transformations)	235
使用余端的 Profunctor 复合	236
余极限作为余端	237
18.3 Ends	238
自然变换作为 end	240
极限作为 ends	241
18.4 Hom-函子的连续性	242

18.5 Fubini 规则	242
18.6 忍者米田引理	243
Haskell 中的米田引理	244
18.7 Day 卷积	245
应用函子作为么半群	246
自由应用函子	247
18.8 Profunctor 的双范畴	248
双范畴中的 Monad	249
Prof 中的 Prearrow 作为 Monad	250
18.9 存在性透镜	251
Haskell 中的存在性透镜	251
范畴论中的存在性透镜	251
Haskell 中的类型变化透镜	252
透镜组合	253
透镜的范畴	254
18.10 透镜与纤维化	254
传输律	254
恒等律	255
复合律	256
类型转换透镜	256
18.11 重要公式	257
19 Tambara 模	259
19.1 Tannakian 重构	259
么半群及其表示	259
Cayley 定理	260
么半群的 Tannakian 重构	262
Tannakian 重构的证明	264
Haskell 中的 Tannakian 重构	265
伴随下的 Tannakian 重构	266
19.2 Profunctor 透镜	267
同构 (Iso)	268

Profunctors 和 lenses (Profunctors 和透镜)	269
Tambara 模	269
Profunctor 透镜	271
Haskell 中的 Profunctor 透镜	272
19.3 一般光学	273
棱镜	273
遍历	274
19.4 混合光学	276
20 Kan 扩张	279
20.1 闭么半范畴	279
Day 卷积的内部 hom	280
幂与余幂	281
20.2 函子的逆	282
20.3 右 Kan 扩展	284
右 Kan 扩展作为 end	286
Haskell 中的右 Kan 扩展	287
作为 Kan 扩张的极限	288
左伴随作为右 Kan 延拓	289
余密度单子 (Codensity Monad)	290
Haskell 中的密度单子	292
20.4 左 Kan 扩展	293
左 Kan 扩张作为余端	294
Haskell 中的左 Kan 扩展	295
余极限作为 Kan 扩张	295
右伴随作为左 Kan 扩张	296
Day 卷积作为 Kan 扩张	297
20.5 常用公式	298
21 富化	299
21.1 富化范畴	299
集合论基础	299
Hom-对象	300

富化范畴	300
例子	302
预序	302
自富化	303
21.2 \mathcal{V} -函子	304
Hom-函子	304
充实共预层	305
函子强度与充实	306
21.3 \mathcal{V} -自然变换	308
21.4 Yoneda 引理	309
21.5 加权极限	310
21.6 作为加权极限的端	311
21.7 Kan 扩展	313
21.8 常用公式	314
22 依赖类型	315
22.1 依赖向量	316

前言

大多数编程教材，遵循 Brian Kernighan 的传统，以“Hello World!”开始。想要立即获得让计算机执行命令并打印这些著名文字的满足感是很自然的。但真正掌握计算机编程远不止于此，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。如果你的目标仅仅是学习一项有用且高薪的技能，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用任何你选择的语言编写代码。然而，如果你真的想触及编程的本质，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一致的抽象。借用克劳塞维茨的话：编程只是数学以其他方式的延续。许多范畴论的复杂概念在通过数据类型和函数解释时，对程序员来说变得显而易见。从这个意义上说，范畴论可能对程序员来说比专业数学家更容易理解。

当我遇到新的范畴论概念时，我经常会在维基百科或 nLab 上查找它们，或者重新阅读 Mac Lane 或 Kelly 的章节。这些都是很好的资源，但它们需要一些前置知识以及填补空白的能力。本书的目标之一就是提供继续学习范畴论所需的引导。

范畴论和计算机科学中有许多民间知识在文献中无处可寻。在枯燥的定义和定理中获取有用的直觉是非常困难的。我尽可能提供了缺失的直觉，并不仅解释了“是什么”，还解释了“为什么”。

本书的标题暗指 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试图吸收东方哲学元素的尝试。粗略地说，范畴论之于编程，正如道¹之于西方哲学。许多范畴论的定义在初次阅读时毫无意义，但随着时间的推移，你会学会欣赏它们的深刻智慧。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，尽管最近类型论正在争夺这一地位。从某种意义上说，集合论是数学的汇编语言，因此包含了许多实现细节，这些细节常常掩盖了高层次思想的呈现。

范畴论并不试图取代集合论，它通常用于构建抽象，这些抽象后来用集合来建模。事实上，范畴论的基本定理——Yoneda 引理，将范畴与它们在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纵抽象世界，最终将其投影并采样到数字显示器上。

研究范畴论并不需要精通集合论。但熟悉一些基本概念是必要的。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 ，询问 a 是否是 S 的元素是有意义的。这个陈述写作 $a \in S$ (a 是 S 的成员)。也可能存在一个不包含任何元素的空集。

集合元素的重要属性是它们可以比较相等性。给定两个元素 $a \in S$ 和 $b \in S$ ，我们可以问： a 等于 b 吗？或者我们可以强加条件 $a = b$ ，如果 a 和 b 是通过两种不同的方法选择集合 S 的元素的结果。集合元素的

¹道是 Tao 的现代拼写

相等性是范畴论中所有交换图的本质。

两个集合 $S \times T$ 的笛卡尔积定义为所有元素对 $\langle s, t \rangle$ 的集合，其中 $s \in S$ 且 $t \in T$ 。

函数 $f: S \rightarrow T$ ，从称为 f 的域（**domain**）的源集合到称为 f 的陪域（**codomain**）的目标集合，也定义为一组对。这些是形式为 $\langle s, t \rangle$ 的对，其中 $t = fs$ 。这里 fs 是函数 f 作用于参数 s 的结果。你可能更熟悉函数应用的符号 $f(s)$ ，但在这里我将遵循 **Haskell** 的惯例，省略括号（以及多变量函数的逗号）。

在编程中，我们习惯于函数由一系列指令定义。我们提供一个参数 s 并应用指令以最终产生结果 t 。我们常常担心计算结果可能需要多长时间，或者算法是否终止。在数学中，我们假设对于任何给定的参数 $s \in S$ ，结果 $t \in T$ 是立即可用的，并且是唯一的。在编程中，我们称这样的函数为纯函数和全函数。

约定

我试图在整本书中保持符号的一致性。它大致基于 **nLab** 中流行的风格。

特别是，我决定使用小写字母如 a 或 b 表示范畴中的对象，使用大写字母如 S 表示集合（尽管集合是集合和函数范畴中的对象）。通用范畴的名称如 \mathcal{C} 或 \mathcal{D} ，而特定范畴的名称如 **Set** 或 **Cat**。

编程示例用 **Haskell** 编写。虽然这不是一本 **Haskell** 手册，但语言结构的引入是渐进的，足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。程序片段以以下格式编写：

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

本书的 **Haskell** 代码，包括编程练习的解决方案，可在[GitHub](#)上找到。

Chapter 1

白板

编程始于类型和函数。你可能对类型和函数有一些先入为主的观念：请摒弃它们！它们会蒙蔽你的思维。

不要考虑事物在硬件中是如何实现的。计算机只是众多计算模型中的一种。我们不应过于依赖它。你可以在脑海中，或用纸笔进行计算。物理载体与编程的理念无关。

1.1 类型与函数

引用老子的话¹：可道之道，非常道。换句话说，类型是一个原始概念。它无法被定义。

我们也可以将其称为对象或命题。这些是数学不同领域中用来描述它的术语（分别是类型理论、范畴论和逻辑学）。

可能存在多个类型，因此我们需要一种命名它们的方式。我们可以通过指向它们来命名，但由于我们希望有效地与他人交流，我们通常会给它们命名。因此，我们会谈论类型 a 、 b 、 c ；或 `Int`、`Bool`、`Double` 等。这些只是名称。

类型本身没有意义。它的特殊之处在于它如何与其他类型连接。这些连接由箭头描述。一个箭头有一个类型作为其源，一个类型作为其目标。目标可以与源相同，在这种情况下，箭头会循环。

类型之间的箭头称为函数。对象之间的箭头称为态射。命题之间的箭头称为蕴涵。这些只是数学不同领域中用来描述箭头的术语。你可以互换使用它们。

命题是可能为真的事物。在逻辑学中，我们将两个对象之间的箭头解释为 a 蕴涵 b ，或 b 可以从 a 推导出来。

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》的半传奇作者，这是一部关于道家的经典著作。

两个类型之间可能有多个箭头，因此我们需要给它们命名。例如，这里有一个名为 f 的箭头，从类型 a 指向类型 b

$$a \xrightarrow{f} b$$

一种解释方式是，函数 f 接受一个类型为 a 的参数，并产生一个类型为 b 的结果。或者 f 是一个证明，如果 a 为真，那么 b 也为真。

注意：类型理论、 λ 演算（编程的基础）、逻辑学和范畴论之间的联系被称为 Curry-Howard-Lambek 对应。

1.2 阴阳

对象由其连接定义。箭头是两个对象连接的证明或见证。有时没有证明，对象是断开的；有时有许多证明；有时只有一个证明——两个对象之间的唯一箭头。

唯一是什么意思？这意味着如果你能找到两个这样的箭头，那么它们必须相等。

一个对象如果对每个对象都有一个唯一的出箭头，则称为初始对象。

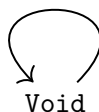
它的对偶是一个对象，对每个对象都有一个唯一的入箭头。它被称为终止对象。

在数学中，初始对象通常用 0 表示，终止对象用 1 表示。

从 0 到任何对象 a 的箭头表示为 i_a ，通常简写为 i 。

从任何对象 a 到 1 的箭头表示为 $!_a$ ，通常简写为 $!$ 。

初始对象是万物的源头。作为一个类型，它在 Haskell 中被称为 **Void**。它象征着万物从中产生的混沌。由于从 **Void** 到万物都有一个箭头，因此从 **Void** 到自身也有一个箭头。



因此，**Void** 生成了 **Void** 和一切其他事物。

终止对象统一了一切。作为一个类型，它被称为 **Unit**。它象征着终极秩序。

在逻辑学中，终止对象象征着终极真理，用 T 或 \top 表示。从任何对象到它都有一个箭头的事实意味着 \top 无论你的假设是什么都是真的。

对偶地，初始对象象征着逻辑上的假、矛盾或反事实。它被写为 **False**，并用倒置的 $\top \perp$ 表示。从它到任何对象都有一个箭头的事实意味着你可以从假前提中证明任何事情。

在英语中，有一种特殊的语法结构用于反事实的蕴涵。当我们说“如果愿望是马，乞丐就会骑马”时，我们的意思是愿望与马之间的等式意味着乞丐能够骑马。但我们知道前提是假的。

编程语言让我们能够相互交流并与计算机交流。有些语言更容易被计算机理解，另一些则更接近理论。我们将使用 Haskell 作为折衷方案。

在 Haskell 中，终止类型的名称是 `()`，一对空括号，发音为 Unit。这个符号稍后会变得有意义。

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。所有这些函数都使用相同的名称： `absurd`。

编程	范畴论	逻辑学
类型	对象	命题
函数	态射（箭头）	蕴涵
<code>Void</code>	初始对象, 0	False \perp
<code>()</code>	终止对象, 1	True \top

1.3 元素

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

在编程和逻辑学中，我们希望我们的初始对象没有结构。因此，我们假设它没有入箭头（除了从它自身循环回来的那个）。因此， `Void` 没有结构。

终止对象具有最简单的结构。从任何对象到它只有一个入箭头：从任何方向探测它只有一种方式。在这方面，终止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。

因为终止对象如此简单，我们可以用它来探测其他更复杂的对象。

如果从终止对象到某个对象 a 有多个箭头，这意味着 a 有一些结构：有不止一种方式来看待它。由于终止对象表现得像一个点，我们可以将每个从它出发的箭头视为选择其目标的不同点或元素。

在范畴论中，我们说 x 是 a 的全局元素，如果它是一个箭头

$$1 \xrightarrow{x} a$$

我们通常会简单地称它为元素（省略“全局”）。

在类型理论中， $x : A$ 表示 x 是类型 A 。

在 Haskell 中，我们使用双冒号符号：

```
x :: A
```

(Haskell 使用大写字母表示具体类型，小写字母表示类型变量。)

我们说 x 是类型 A 的一个项，但从范畴论的角度，我们将其解释为一个箭头 $x : 1 \rightarrow A$ ，即 A 的全局元素。

2

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

由于我们规定没有从任何其他对象到 **Void** 的箭头，因此没有从终止对象到它的箭头。因此，**Void** 没有元素。这就是为什么我们认为 **Void** 是空的。

终止对象只有一个元素，因为它到自身有一个唯一的箭头， $1 \rightarrow 1$ 。这就是为什么我们有时称它为单例。

注意：在范畴论中，并不禁止初始对象有其他对象的入箭头。然而，在我们正在研究的笛卡尔闭范畴中，这是不允许的。

1.4 箭头对象

任何两个对象之间的箭头形成一个集合³。这就是为什么一些集合论知识是学习范畴论的前提。

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。这里， $a \rightarrow b$ 只是我们给这个类型的名称。

如果我们希望函数类型与其他类型一样被对待，我们需要一个对象来表示从 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。我们目前还没有工具来做到这一点，但我们会到达那里。

现在，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”被定义为从终止对象 $()$ 到我们称为 $a \rightarrow b$ 的对象的箭头。

我们在编程中使用的符号往往模糊了这种区别。这就是为什么在范畴论中，我们将箭头对象称为指数，并将其写为 b^a （源对象在指数中）。因此，语句：

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

等价于

²Haskell 类型系统区分 $x :: A$ 和 $x :: () \rightarrow A$ 。然而，它们在范畴语义中表示相同的东西。

³严格来说，这仅在局部小范畴中成立。

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

在逻辑学中，一个箭头 $A \rightarrow B$ 是一个蕴涵：它陈述了“如果 A 则 B ”的事实。一个指数对象 B^A 是对应的命题。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。

给我一个 B^A 的元素，我就知道 B 可以从 A 推导出来。

再次考虑“如果愿望是马，乞丐就会骑马”这句话——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：“有马的人会骑马。乞丐有愿望。既然愿望是马，乞丐就有马。因此乞丐会骑马。”但是，即使你有这个语句的证明，它对你也毫无用处，因为你永远无法证明它的前提：“愿望 = 马”。

Chapter 2

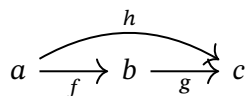
组合

2.1 组合

编程的核心在于组合。借用维特根斯坦的话，可以说：“对于不可分解之物，我们应当保持沉默。”这不是禁令，而是事实陈述。研究、理解和描述的过程就是分解的过程；我们的语言也反映了这一点。

我们构建对象和箭头的词汇表，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得到第三个箭头。



在数学中，我们用一个小圆圈表示组合

$$h = g \circ f$$

我们这样读：“ h 等于 f 之后的 g ”。选择“之后”这个词暗示了动作的时间顺序，在大多数情况下这是一个有用的直觉。

组合的顺序可能看起来是反的，但这是因为我们认为函数是从右边接受参数的。在 **Haskell** 中，我们用点代替圆圈：

```
h = g . f
```

这就是每个程序的精髓。为了实现 h ，我们将其分解为更简单的问题 f 和 g 。这些又可以进一步分解，依此类推。

现在假设我们能够将 g 本身分解为 $j \circ k$ 。我们有

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

我们希望这个分解与

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

相同。我们希望能够说我们将 h 分解为三个更简单的问题

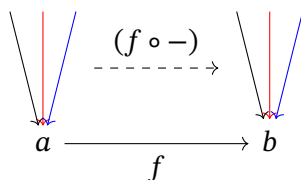
$$h = j \circ k \circ f$$

而不必跟踪哪个分解先进行。这被称为组合的结合性，从现在起我们将假设这一点。

组合产生了两种箭头映射，称为前组合和后组合。

当你后组合一个箭头 h 与一个箭头 f 时，它产生箭头 $f \circ h$ （箭头 f 在箭头 h 之后应用）。当然，你只能将 h 与源为 h 的目标的箭头进行后组合。通过 f 的后组合写为 $(f \circ -)$ ，为 h 留一个空位。正如老子所说：“后组合的用处来自于不存在的东西。”

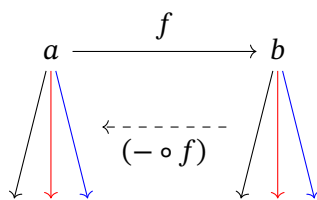
因此，一个箭头 $f: a \rightarrow b$ 诱导了一个箭头映射 $(f \circ -)$ ，它将探测 a 的箭头映射到探测 b 的箭头。



由于对象没有内部结构，当我们说 f 将 a 转换为 b 时，这正是我们的意思。

后组合让我们能够将焦点从一个对象转移到另一个对象。

对偶地，你可以前组合通过 f ，或者应用 $(- \circ f)$ 将源自 b 的箭头映射到源自 a 的箭头（注意方向的变化）。



前组合让我们能够将视角从观察者 b 转移到观察者 a 。

前组合和后组合是箭头的映射。由于箭头形成集合，这些是集合之间的函数。

另一种看待前组合和后组合的方式是，它们是双孔组合运算符 $(- \circ -)$ 的部分应用的结果，我们可以在其中一个孔中预先填充一个固定的箭头。

在编程中，一个出向箭头被解释为从其源中提取数据。一个入向箭头被解释为生成或构造目标。出向箭头定义了接口，入向箭头定义了构造函数。

对于 **Haskell** 程序员，这里是将后组合实现为高阶函数的方式：


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

类似地，前组合的实现如下：

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

做以下练习，以使自己相信焦点和视角的转移是可组合的。

Exercise 2.1.1. 假设你有两个箭头, $f: a \rightarrow b$ 和 $g: b \rightarrow c$ 。它们的组合 $g \circ f$ 诱导了一个箭头映射 $((g \circ f) \circ -)$ 。证明如果你先应用 $(f \circ -)$ ，然后再应用 $(g \circ -)$ ，结果是一样的。符号上：

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

提示：选择一个任意对象 x 和一个箭头 $h: x \rightarrow a$ ，看看你是否得到相同的结果。注意，这里的 \circ 是重载的。在右边，当放在两个后组合之间时，它表示常规的函数组合。

Exercise 2.1.2. 使自己相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. 证明前组合 $(- \circ f)$ 是可组合的，但组合的顺序是相反的：

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

2.2 函数应用

我们已经准备好编写第一个程序了。有句俗话说：“千里之行，始于足下。”考虑从 1 到 b 的旅程。我们的第一步可以从终端对象 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（它们在中间共享对象 a ），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：

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

y

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

$$y = fx$$

让我们将其翻译成 Haskell。我们从 a 的一个元素 x 开始 ($x :: () \rightarrow a$ 的简写)

```
x :: a
```

我们将函数 f 声明为从 a 到 b 的”箭头对象”的一个元素

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

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

```
y :: b
```

并且它被定义为

```
y = f x
```

我们称之为函数对参数的应用，但我们能够纯粹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 恒等态射

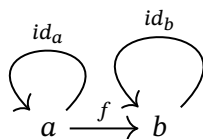
你可以将箭头视为表示变化：对象 a 变为对象 b 。一个回环的箭头表示对象自身的变化。但变化有其对偶面：不变、无为，或者如老子所说的 无为。

每个对象都有一个特殊的箭头，称为恒等态射（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（无论是进入的还是离开的）进行复合时，你得到的仍然是那个箭头。从动作的角度来看，恒等态射什么都不做，也不花费时间。

对象 a 上的恒等态射称为 id_a 。因此，如果我们有一个箭头 $f: a \rightarrow b$ ，我们可以将其与两边的恒等态射进行复合：

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

或者，用图示表示为：



我们可以很容易地检查恒等态射对元素的作用。取一个元素 $x: 1 \rightarrow a$ 并将其与 id_a 复合。结果是：

$$id_a \circ x = x$$

这意味着恒等态射使元素保持不变。

在 Haskell 中，我们对所有恒等函数使用相同的名称 `id`（我们不会在其上加上它所作用的类型的下标）。上述方程，指定了 `id` 对元素的作用，直接翻译为：

```
id x = x
```

这成为函数 `id` 的定义。

我们之前看到，初始对象和终止对象都有唯一的箭头回环到它们自身。现在我们说，每个对象都有一个恒等态射回环到它自身。记住我们关于唯一性的说法：如果你能找到两个这样的箭头，那么它们必须相等。我们必须得出结论，我们之前讨论的这些唯一的回环箭头必须是恒等态射。我们现在可以标记这些图示：



在逻辑中，恒等态射翻译为重言式。这是一个平凡的证明，即“如果 a 为真，那么 a 为真”。它也被称为同一性规则。

如果恒等态射什么都不做，那么我们为什么还要关心它？想象一下，你去旅行，复合了几个箭头，然后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态射进行比较。

有些往返带来了变化，而有些则没有。

更重要的是，恒等态射将允许我们比较对象。它们是同构定义的一个组成部分。

Exercise 2.3.1. $(id_a \circ -)$ 对终止于 a 的箭头做了什么？ $(- \circ id_a)$ 对从 a 出发的箭头做了什么？

2.4 单态射

考虑函数 `even`，它测试输入是否能被 2 整除：

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

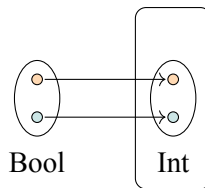
这是一个多对一的函数：所有偶数都被映射到 **True**，所有奇数都被映射到 **False**。关于输入的大部分信息都被丢弃了，我们只关心它的奇偶性，而不是它的实际值。通过丢弃信息，我们得到了一个抽象¹。函数（以及后来的函子）是抽象的典型代表。

与此形成对比的是函数 `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。

不丢弃信息的函数也很有用：它们可以被视为将其源注入到目标中。你可以将源的类型想象为一个被嵌入到目标中的形状。在这里，我们将一个两元素的形状 **Bool** 嵌入到整数类型中。

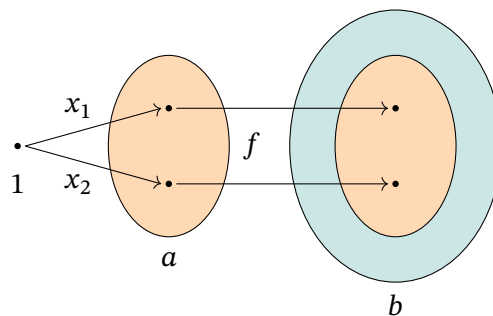


单射函数，或单射，被定义为总是将不同的值分配给不同的参数。换句话说，它们不会将多个元素合并为一个。

这是另一种稍微复杂的说法：单射将两个元素映射到一个元素，仅当这两个元素相等时。

我们可以通过将“元素”替换为从终端对象出发的箭头，将这个定义翻译为范畴论语言。我们会说， $f: a \rightarrow b$ 是一个单射，如果对于任何一对全局元素 $x_1: 1 \rightarrow a$ 和 $x_2: 1 \rightarrow a$ ，以下蕴含成立：

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



¹抽象的字面意思是提取

这个定义的问题在于，并非每个范畴都有终端对象。一个更好的定义是用任意形状替换全局元素。因此，单射性的概念被推广为单态射。

一个箭头 $f: a \rightarrow b$ 是单态射，如果对于任何对象 c 和一对箭头 $g_1: c \rightarrow a$ 和 $g_2: c \rightarrow a$ ，我们有以下蕴含：

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

要证明一个箭头 $f: a \rightarrow b$ 不是单态射，只需找到一个反例： a 中的两个不同形状，使得 f 将它们映射到 b 中的相同形状。

单态射，或简称“monos”，通常用特殊箭头表示，如 $a \hookrightarrow b$ 或 $a \rightarrowtail b$ 。

在范畴论中，对象是不可分割的，因此我们只能通过箭头来讨论子对象。我们说单态射 $a \hookrightarrow b$ 选择了 b 的一个子对象，其形状为 a 。

Exercise 2.4.1. 证明从终端对象出发的任何箭头都是单态射。

2.5 满同态

函数 `injectBool` 是单射的（因此是一个单同态），但它只覆盖了目标集合的一小部分——在无限多个整数中仅覆盖了两个。

```
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`）。覆盖整个目标集合的函数被称为满射。

为了推广单射，我们使用了额外的“映射入”。为了推广满射，我们将使用“映射出”。满射在范畴论中的对应物被称为满同态。

一个箭头 $f: a \rightarrow b$ 是一个满同态，如果对于任意对象 c 和一对箭头 $g_1: b \rightarrow c$ 和 $g_2: b \rightarrow c$ ，我们有以下蕴含关系：

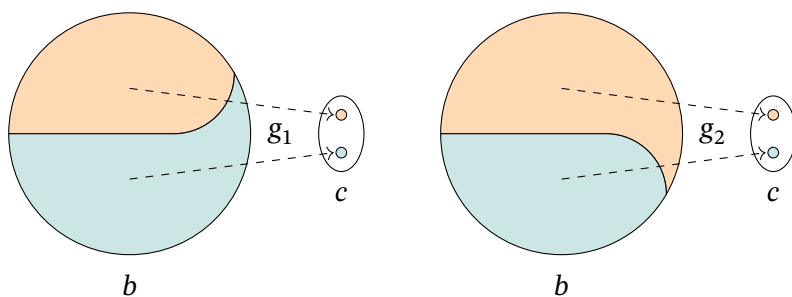
$$g_1 \circ f = g_2 \circ f \implies g_1 = g_2$$

反之，要证明 f 不是满同态，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 f 的复合下一致。

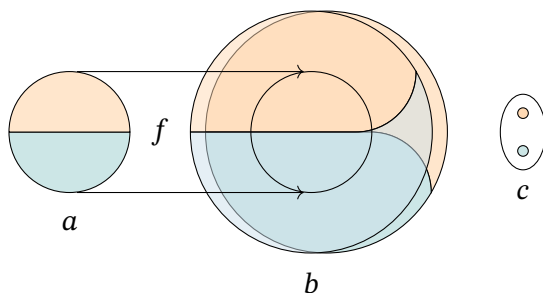
为了理解这个定义，我们需要可视化“映射出”。正如“映射入”一个对象可以被视为定义形状一样，“映射出”一个对象可以被视为定义该对象的属性。

在处理集合时，这一点尤为明显，特别是当目标集合是有限的。你可以将目标集合的元素视为定义一种颜色。所有映射到该元素的源集合元素都被“涂上”特定的颜色。例如，函数 `even` 将所有偶数整数涂上 `True` 颜色，将所有奇数整数涂上 `False` 颜色。

在满同态的定义中，我们有两个这样的映射， g_1 和 g_2 。假设它们只有微小的差异。对象 b 的大部分区域被它们涂成相同的颜色。



如果 f 不是满同态，那么它的像可能只覆盖了 g_1 和 g_2 涂色相同的部分。这两个箭头在 f 的复合下对 a 的涂色一致，尽管它们在整体上是不同的。



当然，这只是一个示意图。在实际的范畴中，我们无法窥视对象的内部。

满同态，简称“epis”，通常用特殊箭头 $a \twoheadrightarrow b$ 表示。

在集合中，既是单射又是满射的函数被称为双射。它在两个集合的元素之间提供了一一对应的可逆映射。在范畴论中，这一角色由同构扮演。然而，一般情况下，一个既是单同态又是满同态的箭头并不一定是同构。

Exercise 2.5.1. 证明任何指向终对象的箭头都是满同态。

Chapter 3

同构

当我们说：

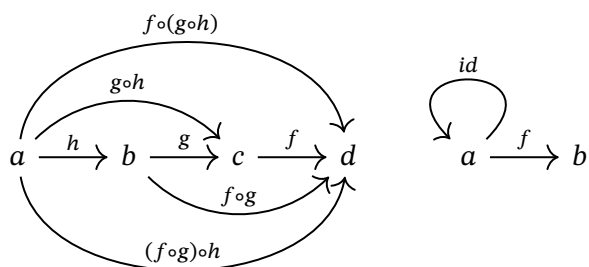
$$f \circ (g \circ h) = (f \circ g) \circ h$$

或：

$$f = f \circ id$$

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，右边的箭头是另一个操作的结果。但结果是相等的。

我们经常通过绘制交换图来说明这种相等性，例如：



因此，我们通过比较箭头来判断它们是否相等。

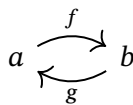
我们不比较对象的相等性¹。我们将对象视为箭头的交汇点，因此如果我们想比较两个对象，我们会观察箭头。

3.1 同构对象

两个对象之间最简单的关系是一个箭头。

¹半开玩笑地说，在范畴论中，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。



有两种可能的往返。一种是 $g \circ f$ ，它从 a 到 a 。另一种是 $f \circ g$ ，它从 b 到 b 。

如果它们都导致恒等箭头，那么我们说 g 是 f 的逆

$$g \circ f = id_a$$

$$f \circ g = id_b$$

我们将其记为 $g = f^{-1}$ （读作 f 的逆）。箭头 f^{-1} 撤销了箭头 f 的作用。

这样的一对箭头称为同构，两个对象称为同构的。

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，而不会改变系统的行为（除了可能的性能）。

在 Haskell 中，我们经常根据其他类型定义类型。如果只是将一个名称替换为另一个名称，我们使用类型同义词。例如：

```
type MyTemperature = Int
```

这让我们在涉及天气的程序中使用 `MyTemperature` 作为整数的更具描述性的名称。这两个类型是相等的——在所有上下文中，一个可以替换为另一个，所有接受 `Int` 参数的函数都可以用 `MyTemperature` 调用。

当我们想要定义一个与现有类型同构但不相等的新类型时，我们可以使用 `newtype`。例如，我们可以将 `Temperature` 定义为：

```
newtype Temperature = Temp Int
```

这里，`Temperature` 是新类型，`Temp` 是一个数据构造函数。数据构造函数只是一个函数²：

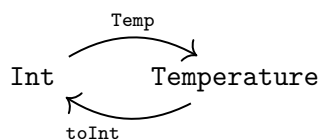
```
Temp :: Int -> Temp
```

我们还可以定义它的逆：

```
toInt :: Temperature -> Int
toInt (Temp i) = i
```

²与常规函数不同，数据构造函数的名称必须以大写字母开头

从而完成同构：



由于这是一个常见的构造，Haskell 提供了特殊语法，可以同时定义这两个函数：

```
newtype Temperature = Temp { toInt :: Int }
```

因为 `Temperature` 是一个新类型，操作整数的函数不会接受它作为参数。这样，程序员可以限制它的使用方式。我们仍然可以使用同构来有选择地允许某些操作，例如在函数 `negate :: Int -> Int` 的应用中。执行此操作的代码是图的直接翻译：

$$\text{Temperature} \xrightarrow{\text{toInt}} \text{Int} \xrightarrow{\text{negate}} \text{Int} \xrightarrow{\text{Temp}} \text{Temperature}$$

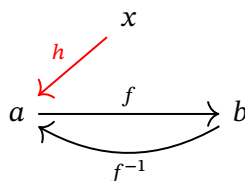
```
invert :: Temperature -> Temperature
```

```
invert = Temp . negate . toInt
```

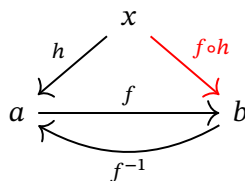
同构与双射

同构的存在告诉我们关于它连接的两个对象的什么信息？

我们说过，对象通过它们与其他对象的交互来描述。因此，让我们从观察者 x 的角度考虑这两个同构对象的样子。取一个从 x 到 a 的箭头 h 。



有一个对应的从 x 到 b 的箭头。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，都有一个对应的探测 a 的箭头。它由 $(f^{-1} \circ -)$ 的作用给出。

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。

我们可以将这两个映射组合起来（见练习2.1.1）形成一个往返。结果与如果我们应用复合 $((f^{-1} \circ f) \circ -)$ 相同。但这等于 $(id_a \circ -)$ ，正如我们从练习2.3.1中所知，它不会改变箭头。

类似地，由 $f \circ f^{-1}$ 引起的往返不会改变箭头 $x \rightarrow b$ 。

这在这两组箭头之间创建了一个“伙伴系统”。想象每个箭头向其伙伴发送一条消息，由 f 或 f^{-1} 决定。然后，每个箭头将恰好收到一条消息，并且这条消息将来自其伙伴。没有箭头会被遗漏，也没有箭头会收到多条消息。数学家称这种伙伴系统为双射或一一对应。

因此，从 x 的角度来看，两个对象 a 和 b 在箭头上看起来完全相同。在箭头上，这两个对象之间没有区别。

两个同构的对象具有完全相同的性质。

特别是，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 $x: 1 \rightarrow a$ ，都有一个对应的元素 $y: 1 \rightarrow b$ ，即 $y = f \circ x$ ，反之亦然。同构对象的元素之间存在双射。

这种无法区分的对象被称为同构的，因为它们具有“相同的形状”。你见过一个，你就见过所有。

我们将这种同构写为：

$$a \cong b$$

在处理对象时，我们使用同构来代替相等。

在经典逻辑中，如果 B 从 A 推出， A 从 B 推出，那么 A 和 B 在逻辑上是等价的。我们经常说 B 为真“当且仅当” A 为真。然而，与逻辑和类型理论之间的先前类比不同，如果你认为证明是相关的，这个类比并不那么直接。事实上，它导致了基础数学的一个新分支的发展，称为同伦类型理论，简称 HoTT。

Exercise 3.1.1. 论证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。

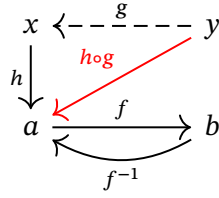
Exercise 3.1.4. 证明前一个练习中的同构是唯一的。

3.2 自然性

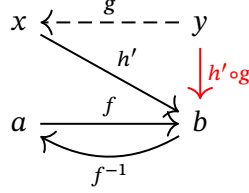
我们已经看到，当两个对象同构时，我们可以使用后复合在它们之间切换焦点：($f \circ -$) 或 ($f^{-1} \circ -$)。

相反，要在不同的观察者之间切换，我们将使用前复合。

确实，从 x 探测 a 的箭头 h 与从 y 探测同一对象的箭头 $h \circ g$ 相关。



类似地，从 x 探测 b 的箭头 h' 对应于从 y 探测它的箭头 $h' \circ g$ 。



在这两种情况下，我们通过应用前复合 $(- \circ g)$ 将视角从 x 切换到 y 。

重要的观察是，视角的变化保留了由同构建立的伙伴关系。如果两个箭头从 x 的角度是伙伴，那么它们从 y 的角度仍然是伙伴。这就像说，先与 g 前复合（切换视角）再与 f 后复合（切换焦点），或者先与 f 后复合再与 g 前复合，结果是一样的。符号上，我们将其写为：

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

我们称之为自然性条件。

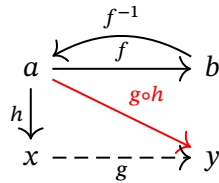
这个方程的意义在你将其应用于态射 $h: x \rightarrow a$ 时显现出来。两边都计算为 $f \circ h \circ g$ 。

$$\begin{array}{ccc} h & \xrightarrow{(- \circ g)} & h \circ g \\ (f \circ -) \downarrow & & \downarrow (f \circ -) \\ f \circ h & \xrightarrow{(- \circ g)} & f \circ h \circ g \end{array}$$

在这里，由于结合性，自然性条件自动满足，但我们很快就会看到它在更不平凡的情况下被推广。

箭头用于广播关于同构的信息。自然性告诉我们，所有对象都能获得一致的观点，与路径无关。

我们还可以反转观察者和被观察者的角色。例如，使用箭头 $h: a \rightarrow x$ ，对象 a 可以探测任意对象 x 。如果存在箭头 $g: x \rightarrow y$ ，它可以将焦点切换到 y 。通过前复合 f^{-1} 将视角切换到 b 。



再次，我们有了自然性条件，这次是从同构对的角度：

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

这种需要采取两个步骤从一个地方移动到另一个地方的情况在范畴论中是典型的。在这里，前复合和后复合的操作可以以任何顺序进行——我们说它们交换。但通常我们采取步骤的顺序会导致不同的结果。我们经常施加交换条件，并说如果一个操作与另一个操作兼容，则这些条件成立。

Exercise 3.2.1. 证明 f^{-1} 的自然性条件的两边，当作用于 h 时，简化为：

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

3.3 用箭头推理

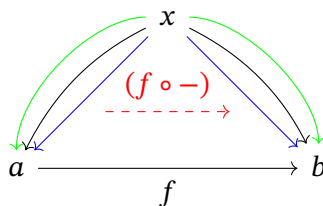
Yoneda 大师说：“看箭头！”

如果两个对象同构，它们具有相同的入箭头集。

如果两个对象同构，它们也具有相同的出箭头集。

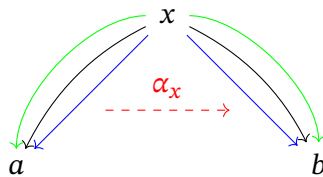
如果你想看两个对象是否同构，看箭头！

当两个对象 a 和 b 同构时，任何同构 f 都会在相应的箭头集之间诱导出——映射 $(f \circ -)$ 。



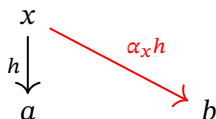
函数 $(f \circ -)$ 将每个箭头 $h: x \rightarrow a$ 映射到箭头 $f \circ h: x \rightarrow b$ 。它的逆 $(f^{-1} \circ -)$ 将每个箭头 $h': x \rightarrow b$ 映射到箭头 $(f^{-1} \circ h')$ 。

假设我们不知道对象是否同构，但我们知道存在一个可逆映射 α_x ，它在每个对象 x 的箭头集之间建立了一一对应关系。换句话说，对于每个 x ， α_x 是箭头的双射。



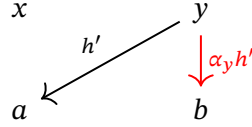
之前，箭头的双射是由同构 f 生成的。现在，箭头的双射由 α_x 给出。这是否意味着这两个对象是同构的？我们可以从映射族 α_x 中构造出同构 f 吗？答案是“是的”，只要映射族 α_x 满足自然性条件。

这是 α_x 在特定箭头 h 上的作用。



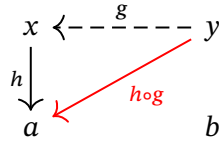
这个映射，连同它的逆 α_x^{-1} ，它将箭头 $x \rightarrow b$ 映射到箭头 $x \rightarrow a$ ，将扮演 $(f \circ -)$ 和 $(f^{-1} \circ -)$ 的角色，如果确实存在同构 f 。映射族 α 描述了一种“人工”的从 a 切换到 b 的方式。

这是从另一个观察者 y 的角度看的相同情况：



注意， y 使用的是同一族中的不同映射 α_y 。

这两个映射， α_x 和 α_y ，每当存在态射 $g: y \rightarrow x$ 时就会纠缠在一起。在这种情况下，与 g 的前复合允许我们从 x 切换到 y 的视角（注意方向）



我们将焦点的切换与视角的切换分开了。前者由 α 完成，后者由前复合完成。自然性在这两者之间施加了兼容性条件。

确实，从某个 h 开始，我们可以先应用 $(- \circ g)$ 切换到 y 的视角，然后应用 α_y 将焦点切换到 b ：

$$\alpha_y \circ (- \circ g)$$

或者我们可以先让 x 使用 α_x 将焦点切换到 b ，然后使用 $(- \circ g)$ 切换视角：

$$(- \circ g) \circ \alpha_x$$

在这两种情况下，我们最终都会从 y 的角度看 b 。我们之前做过这个练习，当 a 和 b 之间存在同构时，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，我们必须施加等效的自然性条件：

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

当作用于某个箭头 $h: x \rightarrow a$ 时，我们希望这个图交换：

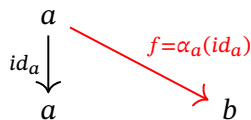
$$\begin{array}{ccc} h & \xrightarrow{(- \circ g)} & h \circ g \\ \downarrow \alpha_x & & \downarrow \alpha_y \\ \alpha_x h & \xrightarrow{(- \circ g)} & (\alpha_x h) \circ g = \alpha_y (h \circ g) \end{array}$$

这样我们就知道，用 $(f \circ -)$ 替换所有 α 会起作用。但这样的 f 存在吗？我们可以从 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 是为任何对象 x 定义的，它也为 a 本身定义。根据定义， α_a 将态射 $a \rightarrow a$ 映射到态射 $a \rightarrow b$ 。我们肯定知道至少存在一个态射 $a \rightarrow a$ ，即恒等态射 id_a 。事实证明，我们寻找的同构 f 由下式给出：

$$f = \alpha_a(id_a)$$

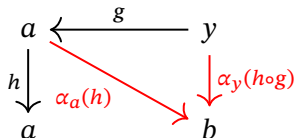
或者，图示为：



让我们验证这一点。如果 f 确实是我们的同构，那么对于任何 x ， α_x 应该等于 $(f \circ -)$ 。为了看到这一点，让我们将自然性条件中的 x 替换为 a 。我们得到：

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

如下图所示：



由于 h 的源和目标都是 a ，这个等式对于 $h = id_a$ 也必须成立

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

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

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

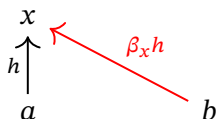
换句话说，对于每个对象 y 和每个态射 $g: y \rightarrow a$ ， $\alpha_y = (f \circ -)$ 。

注意，即使 α_x 是为每个 x 和每个箭头 $x \rightarrow a$ 单独定义的，它最终完全由其在单个恒等箭头上的值决定。这就是自然性的力量！

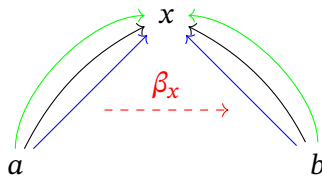
反转箭头

正如老子所说，观察者与被观察者之间的二元性只有在观察者被允许与被观察者交换角色时才能完全成立。

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 $h: a \rightarrow x$ 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，我们能够使用 $(- \circ f^{-1})$ 将视角切换到 b 。这次我们手头有一个变换 β_x 。它在箭头上建立了一一对应关系。



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 通过箭头 $g: x \rightarrow y$ 连接，那么我们还可以选择使用 $(g \circ -)$ 切换焦点。如果我们想同时做两件事：切换视角和切换焦点，有两种方法。自然性要求结果相等：

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

确实，如果我们将 β 替换为 $(- \circ f^{-1})$ ，我们会恢复同构的自然性条件。

Exercise 3.3.1. 使用恒等态的技巧从映射族 β 中恢复 f^{-1} 。

Exercise 3.3.2. 使用前一个练习中的 f^{-1} ，评估任意对象 y 和任意箭头 $g: a \rightarrow y$ 的 $\beta_y g$ 。

正如老子所说：要证明同构，通常更容易在万箭之间定义自然变换，而不是在两个对象之间找到一对箭头。

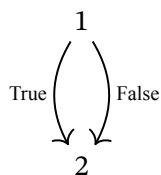
和类型 (Sum Types)

4.1 布尔类型 (Bool)

我们已经知道如何组合箭头。但是如何组合对象呢？

我们已经定义了 0（初始对象）和 1（终止对象）。那么 2 如果不是 1 加 1，又是什么呢？

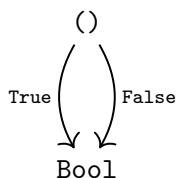
一个 2 是一个具有两个元素的对象：两个从 1 出发的箭头。让我们称其中一个箭头为 **True**，另一个为 **False**。不要将这些名称与初始对象和终止对象的逻辑解释混淆。这两个是箭头。



这个简单的想法可以立即在 Haskell¹中表达为一个类型的定义，传统上称为 **Bool**，以其发明者乔治·布尔 (George Boole, 1815-1864) 命名。

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

它对应于相同的图表（只是有一些 Haskell 的重命名）：



¹这种定义风格在 Haskell 中被称为广义代数数据类型 (Generalized Algebraic Data Types) 或 **GADTs**

正如我们之前所见，元素有一个简写符号，所以这里是一个更紧凑的版本：

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

我们现在可以定义一个类型为`Bool`的项，例如

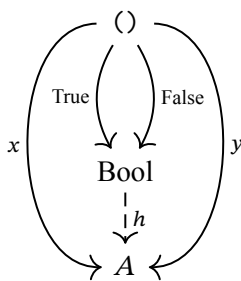
```
x :: Bool
x = True
```

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中的哪一个。

我们在定义`Bool`时使用的函数`True`和`False`被称为数据构造器。它们可以用来构造特定的项，如上面的例子。顺便说一下，在 `Haskell` 中，函数名以小写字母开头，除非它们是数据构造器。

我们对类型`Bool`的定义仍然不完整。我们知道如何构造一个`Bool`项，但我们不知道如何处理它。我们必须能够定义从`Bool`出发的箭头——映射出 `Bool`。

第一个观察是，如果我们有一个从`Bool`到某个具体类型`A`的箭头`h`，那么我们只需通过组合就可以自动得到两个从单位到`A`的箭头`x`和`y`。以下两个（变形的）三角形是可交换的：



换句话说，每个函数`Bool -> A`都会产生一对`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
```

注意使用简写符号表示函数对元素的应用：

```
h True -- 意思是: h . True
```

我们现在准备通过添加一个条件来完成`Bool`的定义，即任何从`Bool`到`A`的函数不仅产生而且等价于一对`A`的元素。换句话说，一对元素唯一地确定了一个从`Bool`出发的函数。

这意味着我们可以用两种方式解释上面的图表：给定`h`，我们可以很容易地得到`x`和`y`。但反过来也是成立的：一对元素`x`和`y`唯一地定义了`h`。

我们这里有一个双射在起作用。这次是一对元素 (x, y) 和一个箭头 h 之间的一一映射。

在 Haskell 中，`h`的这个定义封装在`if`、`then`、`else`构造中。给定

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

我们定义映射出

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

这里，`b`是类型`Bool`的一个项。

一般来说，数据类型使用引入规则创建，并使用消除规则解构。`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`构造定义了消除规则。

给定上述`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`。

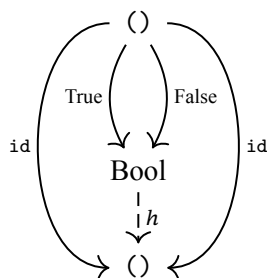
我们永远不应忘记编程的目的：将复杂问题分解为一系列更简单的问题。`Bool`的定义说明了这个想法。每当我们必须构造一个从`Bool`出发的映射时，我们将其分解为构造目标类型的一对元素的两个较小任务。我们将一个较大的问题换成了两个较简单的问题。

示例

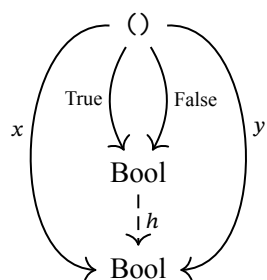
让我们做一些例子。我们还没有定义很多类型，所以我们将仅限于从`Bool`到`Void`、`()`或`Bool`的映射。然而，这些边缘情况可能会为众所周知的结果提供新的见解。

我们已经决定，除了恒等函数外，不能有以`Void`为目标的函数，所以我们不期望有任何从`Bool`到`Void`的函数。确实，我们有零对`Void`的元素。

那么从 **Bool** 到 $()$ 的函数呢？由于 $()$ 是终止的，所以只能有一个从 **Bool** 到它的函数。确实，这个函数对应于从 $()$ 到 $()$ 的单一可能函数对——两者都是恒等函数。到目前为止，一切顺利。



有趣的情况是从 **Bool** 到 **Bool** 的函数。让我们将 **Bool** 代入 **A** 的位置：



我们有多少对从 $()$ 到 **Bool** 的函数 (x, y) 可供使用？只有两个这样的函数，**True** 和 **False**，所以我们可以形成四对。这些是 $(True, True)$ 、 $(False, False)$ 、 $(True, False)$ 和 $(False, True)$ 。因此，只能有四个从 **Bool** 到 **Bool** 的函数。

我们可以使用 **if**、**then**、**else** 构造在 Haskell 中编写它们。例如，最后一个，我们称之为 **not**，定义如下：

```
not :: Bool -> Bool
not b = if b then False else True
```

我们还可以将从 **Bool** 到 **A** 的函数视为箭头对象的元素，或指数对象 A^2 ，其中 **2** 是 **Bool** 对象。根据我们的计数， 0^2 中有零个元素， 1^2 中有一个元素， 2^2 中有四个元素。这正是我们从高中代数中所期望的，其中数字实际上意味着数字。

Exercise 4.1.1. 编写其他三个函数 **Bool** \rightarrow **Bool** 的实现。

4.2 枚举类型

在 0、1、2 之后是什么？一个具有三个数据构造子的对象。例如：

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

如果你厌倦了冗余的语法，这种定义有一个简写形式：

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

这个引入规则允许我们构造类型为`RGB`的项，例如：

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

为了定义从`RGB`出发的映射，我们需要一个更一般的消去模式。就像从`Bool`出发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`和`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，其定义被分成了三种情况。

对于`Bool`，也可以使用相同的语法来代替`if`、`then`、`else`：

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

事实上，还有第三种方式可以使用`case`语句来编写相同的内容：

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

或者甚至

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

在编程时，你可以根据方便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子的类型。例如，一个十进制数字是以下之一：

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

有一个巨大的 Unicode 字符枚举类型叫做 `Char`。它们的构造子有特殊的名称：你可以在两个单引号之间写入字符本身，例如：

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

正如老子所说，万物的模式需要许多年才能完成，因此人们想出了通配符模式，即下划线，它可以匹配任何内容。

因为模式是按顺序匹配的，你应该将通配符模式作为一系列模式中的最后一个：

```
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` 可以被视为在 -2^{29} 到 2^{29} 之间（或更多，取决于实现）的整数枚举。当然，对这样的范围进行穷举模式匹配是不可能的，但原则仍然成立。

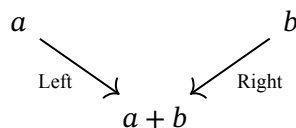
在实践中，用于 Unicode 字符的 `Char` 类型、用于固定精度整数的 `Int` 类型、用于双精度浮点数的 `Double` 类型以及其他几种类型，都是内置在语言中的。

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

4.3 和类型 (Sum Types)

`Bool` 类型可以被看作是 $2 = 1 + 1$ 的和。但没有什么能阻止我们用另一个类型替换 `1`，甚至用不同的类型替换每个 `1`。我们可以通过使用两个箭头来定义一个新的类型 $a + b$ 。我们称它们为 `Left` 和 `Right`。定义

图是引入规则：

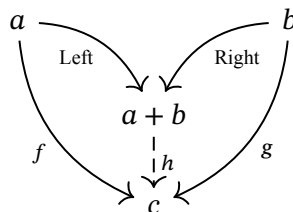


在 Haskell 中，类型 $a + b$ 被称为 `Either a b`。与 `Bool` 类比，我们可以将其定义为

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

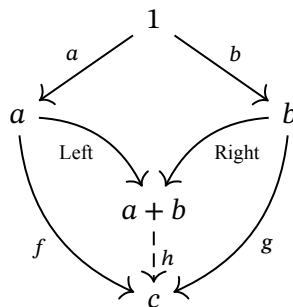
(注意类型变量使用小写字母。)

类似地，从 $a + b$ 到某个类型 c 的映射由以下交换图确定：



给定一个函数 h ，我们只需将其与 `Left` 和 `Right` 组合，就可以得到一对函数 f 和 g 。反过来，这样的一对函数唯一地确定了 h 。这是消除规则。

当我们将这个图转换为 Haskell 时，我们需要选择两个类型的元素。我们可以通过从终端对象定义箭头 a 和 b 来实现。



按照图中的箭头，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等式，从而产生了用于定义 h 的模式匹配语法：

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

(再次注意，类型变量使用小写字母，并且相同字母用于该类型的项。与人类不同，编译器不会因此感到困惑。)

你也可以从右到左阅读这些等式，你会看到和类型的计算规则：用于定义 `h` 的两个函数可以通过将 `h` 应用于 `(Left a)` 和 `(Right b)` 来恢复。

你也可以使用 `case` 语法来定义 `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？它不过是操作箭头的配方。

Maybe 类型

`Maybe` 是一个非常有用的数据类型，它被定义为任何 `a` 的和 `1 + a`。这是它在 Haskell 中的定义：

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

数据构造函数 `Nothing` 是从单元类型出发的箭头，而 `Just` 从 `a` 构造 `Maybe a`。`Maybe a` 与 `Either () a` 同构。它也可以使用简写符号定义为

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

`Maybe` 主要用于编码部分函数的返回类型：这些函数在某些参数值下未定义。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，部分函数通常通过异常（或核心转储）来实现。

逻辑

在逻辑中，命题 `A + B` 被称为择一（*alternative*），或逻辑或。你可以通过提供 `A` 的证明或 `B` 的证明来证明它。其中任何一个都足够。

如果你想证明 `C` 从 `A + B` 得出，你必须为两种可能性做好准备：要么有人通过证明 `A` 来证明 `A + B`（而 `B` 可能为假），要么通过证明 `B` 来证明 `A + B`（而 `A` 可能为假）。在第一种情况下，你必须证明 `C` 从 `A` 得出。在第二种情况下，你需要一个证明 `C` 从 `B` 得出。这些正是 `A + B` 消除规则中的箭头。

4.4 余积范畴

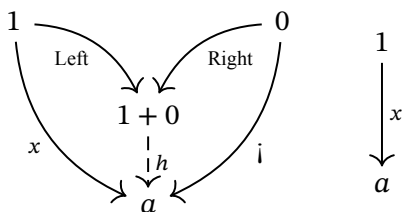
在 Haskell 中，我们可以使用 `Either` 定义任意两种类型的和。一个范畴中如果所有和都存在，并且存在初始对象，则称为余积范畴，而和称为余积。你可能已经注意到，和类型模仿了数字的加法。事实证明，初始对象扮演了零的角色。

一加零

我们首先证明 $1 + 0 \cong 1$ ，即终端对象与初始对象的和同构于终端对象。这类证明的标准方法是使用 Yoneda 技巧。由于和类型是通过映射来定义的，我们应该比较从两边出来的箭头。

Yoneda 论证指出，如果两个对象到任意对象 a 的箭头集合之间存在双射 β_a ，并且这个双射是自然的，那么这两个对象是同构的。

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



我们希望在 $1 + 0$ 出发的箭头和 1 出发的箭头之间建立一一对应关系。箭头 h 由对 (x, i) 决定。由于只有一个 i ， h 和 x 之间存在双射。

我们定义 β_a 将任何由对 (x, i) 定义的 h 映射到 x 。反过来， β_a^{-1} 将 x 映射到对 (x, i) 。但它是一个自然变换吗？

为了回答这个问题，我们需要考虑当我们从 a 转移到通过箭头 $g: a \rightarrow b$ 与之相连的某个 b 时会发生什么。我们现在有两个选择：

- 通过后复合 g 使 h 切换焦点。我们得到一个新的对 $(y = g \circ x, i)$ 。然后用 β_b 跟随它。
- 使用 β_a 将 (x, i) 映射到 x 。然后用后复合 $(g \circ -)$ 跟随它。

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

在 Haskell 中，我们可以定义形成同构的两个函数，但无法直接表达它们是彼此逆的事实。

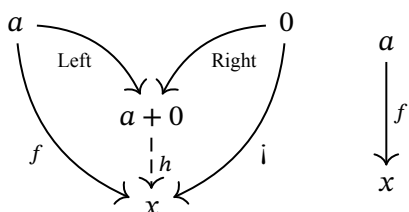
```
f :: Either () Void -> ()
f (Left ()) = ()
f (Right _) = ()
```

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

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

某物加零

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



我们可以通过实现一个（多态的）函数`h`来将这个论证翻译到 Haskell 中，该函数适用于任何类型`a`。

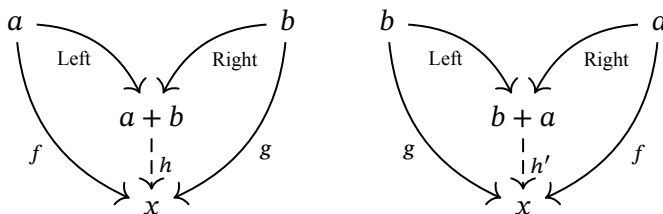
Exercise 4.4.1. 在 Haskell 中实现形成 `(Either a Void)` 和 `a` 之间同构的两个函数。

我们可以使用类似的论证来证明 $0 + a \cong a$ ，但和类型的一个更一般的性质使得这变得不必要。

交换性

在定义和类型的图中存在一个很好的左右对称性，这表明它满足交换律， $a + b \cong b + a$ 。

让我们考虑从这个公式的两边映射出来。你可以很容易地看到，对于每一个由左边的对 (f, g) 决定的 h ，右边都有一个对应的 h' 由对 (g, f) 给出。这建立了箭头的双射。



Exercise 4.4.2. 证明上述定义的双射是自然的。提示： f 和 g 都通过后复合 $k: x \rightarrow y$ 改变焦点。

Exercise 4.4.3. 在 Haskell 中实现见证 `(Either a b)` 和 `(Either b a)` 之间同构的函数。注意这个函数是它自己的逆。

结合性

就像在算术中一样，我们定义的和是结合的：

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

很容易写出左边的映射：

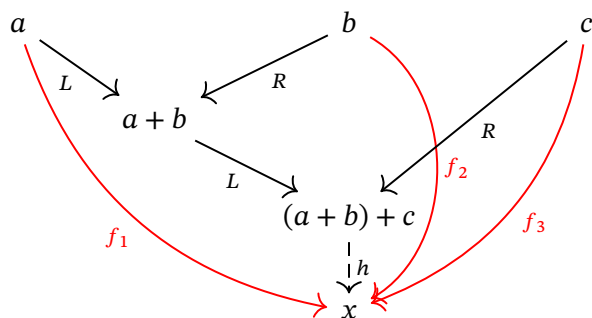
```
h :: Either (Either a b) c -> x
h (Left (Left a)) = f1 a
h (Left (Right b)) = f2 b
h (Right c)        = f3 c
```

注意使用了嵌套模式如`(Left (Left a))`等。映射完全由三个函数定义。同样的函数可以用来定义右边的映射：

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

这建立了定义两个映射出的三函数之间的一一对应关系。这个映射是自然的，因为所有的焦点变化都是通过后复合完成的。因此，两边是同构的。

这段代码也可以用图的形式展示。以下是同构左边的图：



函子性

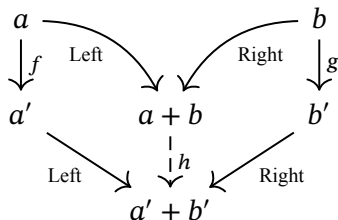
由于和是通过映射出的性质定义的，很容易看出当我们改变焦点时会发生什么：它会“自然地”随着定义积的箭头的焦点变化。但当我们移动这些箭头的源时会发生什么？

假设我们有箭头将 a 和 b 映射到某个 a' 和 b' ：

$$f : a \rightarrow a'$$

$$g : b \rightarrow b'$$

这些箭头与构造函数 `Left` 和 `Right` 的复合可以用来定义和之间的映射：



箭头对 $(\text{Left} \circ f, \text{Right} \circ g)$ 唯一地定义了箭头 $h: a+b \rightarrow a'+b'$ 。这个箭头的符号是：

$$a+b \xrightarrow{\langle f, g \rangle} a'+b'$$

这种将一对箭头提升到和上的性质称为和的函子性。你可以想象它允许你转换和内部两个对象并得到一个新的和。

Exercise 4.4.4. 证明函子性保持复合。提示：取两个可复合的箭头， $g: b \rightarrow b'$ 和 $g': b' \rightarrow b''$ ，并证明应用 $g' \circ g$ 与先应用 g 将 $a+b$ 转换为 $a+b'$ ，然后应用 g' 将 $a+b'$ 转换为 $a+b''$ 得到相同的结果。

Exercise 4.4.5. 证明函子性保持恒等。提示：使用 id_b 并证明它被映射到 id_{a+b} 。

对称幺半群范畴

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，我们称之为余积范畴。

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

正如老子所说，当事物结合在一起形成新事物，并且操作是结合的，并且它有一个中性元素时，我们就知道如何处理万物。

我们定义的和类型满足以下性质：

$$a + 0 \cong a$$

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

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

并且它是函子的。具有这种操作类型的范畴称为对称幺半群范畴。当操作是和（余积）时，它被称为余积范畴。在下一章中，我们将看到另一种称为积范畴的幺半群结构，没有“余”字。

Chapter 5

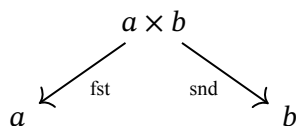
积类型

我们可以使用和类型 (sum types) 来枚举给定类型的可能值，但这种编码方式可能比较浪费。我们仅仅为了编码 0 到 9 的数字，就需要十个构造函数。

```
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，即一个两位的十进制数，我们就能够编码一百个数字。或者，正如老子所说，只需四个数字，你就能编码一万个数字。

以这种方式组合两种类型的数据类型称为积 (product)，或笛卡尔积。其定义性质是消去规则：从 $a \times b$ 出发有两个箭头；一个称为“fst”的箭头指向 a ，另一个称为“snd”的箭头指向 b 。它们被称为 (笛卡尔) 投影。它们让我们可以从积 $a \times b$ 中提取 a 和 b 。



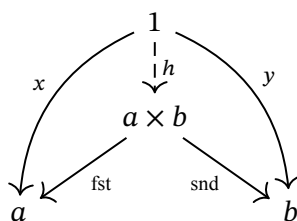
假设有人给了你一个积的元素，即从终对象 1 到 $a \times b$ 的箭头 h 。你可以很容易地通过组合提取出一对元素： a 的元素由

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

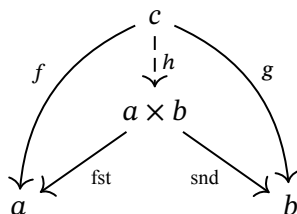
给出， b 的元素由

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

给出。



事实上，给定从任意对象 c 到 $a \times b$ 的箭头，我们可以通过组合定义一对箭头 $f: c \rightarrow a$ 和 $g: c \rightarrow b$



正如我们之前对和类型所做的那样，我们可以将这个想法反过来，用这个图来定义积类型：我们施加一个条件，即一对函数 f 和 g 与从 c 到 $a \times b$ 的映射入一一对应。这是积的引入规则。

特别是，从终对象的映射在 **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)
```

积的内置语法就是这样：一对括号和中间的一个逗号。它既可以用于定义两种类型的积 (A, B) ，也可以用于定义数据构造函数 (a, b) ，它接受两个元素并将它们配对在一起。

我们永远不应忘记编程的目的：将复杂问题分解为一系列更简单的问题。我们在积的定义中再次看到了这一点。每当我们必须构造一个映射入积时，我们将其分解为构造一对函数的两个较小任务，每个函数映射到积的一个组件中。这就像说，要实现一个返回一对值的函数，只需实现两个函数，每个函数返回该对中的一个元素。

逻辑

在逻辑中，积类型对应于逻辑合取。为了证明 $A \times B$ (A 与 B)，你需要提供两者 A 和 B 的证明。这些是目标为 A 和 B 的箭头。消去规则说，如果你有 $A \times B$ 的证明，那么你自动得到 A 的证明（通过 `fst`）和 B 的证明（通过 `snd`）。

元组和记录

正如老子所说，一万个对象的积只是一个有一万个投影的对象。

我们可以在 **Haskell** 中使用元组符号形成任意积。例如，三种类型的积写为 (A, B, C) 。这种类型的项可以从三个元素构造： (a, b, c) 。

在数学家称之为“符号滥用”的情况下，零种类型的积写为 $()$ ，一个空元组，它恰好与终对象或单位类型相同。这是因为积的行为非常类似于数字的乘法，终对象扮演着 1 的角色。

在 **Haskell** 中，我们使用模式匹配语法，而不是为所有元组定义单独的投影。例如，要从三元组中提取第三个组件，我们会写

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

我们使用通配符来忽略我们想要忽略的组件。

老子说：“名者，万物之始也。”在编程中，如果不给特定元组的组件命名，就很难跟踪它们的含义。记录语法允许我们为投影命名。这是用记录风格编写的积的定义：

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

Pair是数据构造函数，**fst**和**snd**是投影。

这是它如何用于声明和初始化特定对的示例：

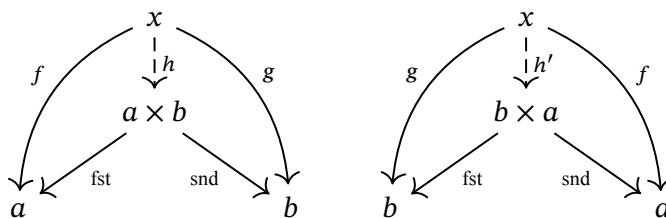
```
ic :: Product Int Char
ic = Pair 10 'A'
```

5.1 笛卡尔范畴

在 **Haskell** 中，我们可以定义任意两种类型的积。一个所有积都存在且终对象存在的范畴称为笛卡尔范畴。

元组算术

积满足的恒等式可以使用映射入性质推导出来。例如，要证明 $a \times b \cong b \times a$ ，考虑以下两个图：



它们表明，对于任何对象 x ，到 $a \times b$ 的箭头与到 $b \times a$ 的箭头一一对应。这是因为这些箭头中的每一个都由相同的对 f 和 g 决定。

你可以检查自然性条件是否满足，因为当你使用 $k: x' \rightarrow x$ 改变视角时，所有从 x 出发的箭头都通过预组合 $(- \circ k)$ 进行了移位。

在 Haskell 中，这种同构可以实现为一个函数，它是它自己的逆函数：

```
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：

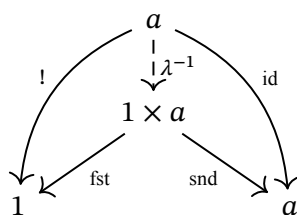
```
swap (x, y) = (y, x)
```

重要的是要记住，积的对称性只是“在同构意义下”。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，但访问它需要修改。

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

$$\lambda: 1 \times a \rightarrow a$$

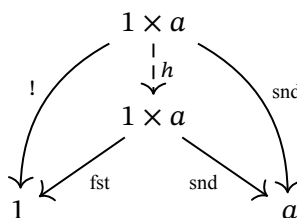
它可以实现为 $\lambda = \text{snd}$ 。它的逆 λ^{-1} 定义为下图中的唯一箭头：



从 a 到 1 的箭头称为 $!$ （发音为 *bang*）。这确实表明

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

我们仍然需要证明 λ^{-1} 是 snd 的左逆。考虑下图：



对于 $h = \text{id}$ ，它显然交换。对于 $h = \lambda^{-1} \circ \text{snd}$ ，它也交换，因为我们有：

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

由于 h 应该是唯一的，我们得出结论：

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

这种使用普遍构造的推理是相当标准的。

以下是一些用 Haskell 编写的其他同构（没有逆的证明）。这是结合律：


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

这是右单位元

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

这两个函数对应于结合子

$$\alpha: (a \times b) \times c \rightarrow a \times (b \times c)$$

和右单位元:

$$\rho: a \times 1 \rightarrow a$$

Exercise 5.1.1. 证明左单位元证明中的双射是自然的。提示，使用箭头 $g: a \rightarrow b$ 改变焦点。

Exercise 5.1.2. 构造一个箭头

$$h: b + a \times b \rightarrow (1 + a) \times b$$

这个箭头是唯一的吗?

提示：它是一个映射入积，所以它由一对箭头给出。这些箭头又映射出和，所以每个箭头由一对箭头给出。

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

Exercise 5.1.3. 重做上一个练习，这次将 h 视为映射出和。

Exercise 5.1.4. 实现一个 *Haskell* 函数 `maybeAB :: Either b (a, b) -> (Maybe a, b)`。这个函数是否由其类型签名唯一定义，还是有一些自由度?

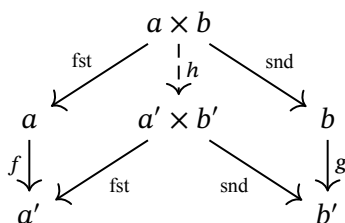
函子性

假设我们有将 a 和 b 映射到某个 a' 和 b' 的箭头:

$$f: a \rightarrow a'$$

$$g: b \rightarrow b'$$

这些箭头与投影 `fst` 和 `snd` 的组合可以用于定义积之间的映射 h :



这个图的简写符号是：

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

积的这种性质称为函子性。你可以想象它允许你转换积内部的两个对象以获得新的积。我们也说函子性让我们提升一对箭头以操作积。

5.2 对偶性

当一个孩子看到箭头时，他知道哪一端指向源 (source)，哪一端指向目标 (target)

$$a \rightarrow b$$

但这可能只是一种先入为主的观念。如果我们称 b 为源 (source) 而 a 为目标 (target)，宇宙会变得非常不同吗？

我们仍然可以将这个箭头与另一个箭头组合

$$b \rightarrow c$$

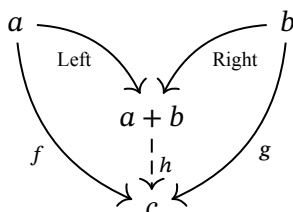
其”目标” b 与 $a \rightarrow b$ 的”源”相同，结果仍然是一个箭头

$$a \rightarrow c$$

只是现在我们会说它从 c 指向 a 。

在这个对偶的宇宙中，我们称为”初始”(initial) 的对象将被称为”终结”(terminal)，因为它是来自所有对象的唯一箭头的”目标”。相反，终结对象将被称为初始。

现在考虑我们用来定义和对象 (sum object) 的图表：



在新的解释中，箭头 h 将从任意对象 c 指向我们称为 $a + b$ 的对象。这个箭头由一对“源”为 c 的箭头 (f, g) 唯一确定。如果我们将 **Left** 重命名为 **fst**，将 **Right** 重命名为 **snd**，我们将得到积对象 (product) 的定义图表。

积对象是箭头反向的和对象。

相反，和对象是箭头反向的积对象。

范畴论中的每个构造都有其对偶。

如果箭头的方向只是一个解释问题，那么在编程中，和类型 (sum types) 与积类型 (product types) 为何如此不同？这种差异可以追溯到我们一开始做出的一个假设：初始对象没有传入的箭头（除了恒等箭头）。这与终结对象有许多传出箭头形成对比，我们使用这些箭头来定义（全局）元素。事实上，我们假设每个感兴趣的对象都有元素，而没有元素的对象与 **Void** 同构。

当我们讨论函数类型时，我们将看到一个更深刻的差异。

5.3 么半范畴

我们已经看到积满足以下简单规则：

$$\begin{aligned} 1 \times a &\cong a \\ a \times b &\cong b \times a \\ (a \times b) \times c &\cong a \times (b \times c) \end{aligned}$$

并且是函子性的。

定义了具有这些性质的运算的范畴称为对称么半范畴¹。我们之前在讨论和与初始对象时已经见过类似的结构。

一个范畴可以同时具有多个么半结构。当不想为么半结构命名时，可以用张量符号代替加号或积符号，用字母 I 代替中性元素。于是，对称么半范畴的规则可以写成：

$$\begin{aligned} I \otimes a &\cong a \\ a \otimes b &\cong b \otimes a \\ (a \otimes b) \otimes c &\cong a \otimes (b \otimes c) \end{aligned}$$

¹严格来说，两个对象的积是定义在同构意义下的，而么半范畴中的积必须精确定义。但我们可以通过选择一个积来得到一个么半范畴

这些同构通常写成可逆箭头的族，称为结合子和单位子。如果么半范畴不是对称的，则存在独立的左单位子和右单位子。

$$\alpha: (a \otimes b) \otimes c \rightarrow a \otimes (b \otimes c)$$

$$\lambda: I \otimes a \rightarrow a$$

$$\rho: a \otimes I \rightarrow a$$

对称性由以下箭头体现：

$$\gamma: a \otimes b \rightarrow b \otimes a$$

函子性允许我们将一对箭头提升：

$$f: a \rightarrow a'$$

$$g: b \rightarrow b'$$

以作用于张量积上：

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

如果我们将态射视为动作，它们的张量积对应于并行执行两个动作。这与态射的串行组合形成对比，后者暗示了它们的时间顺序。

你可以将张量积视为积与和的最低公分母。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。

一些有趣的张量积例子甚至不是对称的。

么半群

么半群是非常简单的结构，配备了一个二元运算和一个单位元。带有加法和零的自然数形成一个么半群。带有乘法和一的自然数也是如此。

直观上，么半群允许你将两个事物组合成另一个事物。还有一个特殊的事物，与任何其他事物组合都会返回相同的事物。这就是单位元。并且组合必须是结合的。

不假设的是组合是对称的，或者存在逆元。

定义么半群的规则让人想起范畴的规则。不同之处在于，在么半群中，任何两个事物都是可组合的，而在范畴中通常不是这样：只有当两个箭头的目标是一个箭头的源时，才能组合它们。除非范畴只包含一个对象，在这种情况下所有箭头都是可组合的。

只有一个对象的范畴称为么半群。组合运算是箭头的组合，单位元是恒等箭头。

这是一个完全有效的定义。然而，在实践中，我们通常对嵌入在更大范畴中的么半群感兴趣。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么半群。

然而，在范畴中，我们更喜欢批量定义运算，而不是查看单个元素。因此，我们从一个对象 m 开始。二元运算是两个参数的函数。由于积的元素是成对的元素，我们可以将二元运算描述为从积 $m \times m$ 到 m 的箭头：

$$\mu : m \times m \rightarrow m$$

单位元素可以定义为从终对象 1 出发的箭头：

$$\eta : 1 \rightarrow m$$

我们可以通过定义一个类型类来直接将这个描述翻译成 Haskell，该类配备了两个方法，传统上称为 `mappend` 和 `mempty`：

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

两个箭头 μ 和 η 必须满足么半群定律，但同样，我们必须批量表述它们，而不依赖于任何元素。

为了表述左单位律，我们首先创建积 $1 \times m$ 。然后使用 η “在 m 中选择单位元”，或者用箭头的话说，将 1 转换为 m 。由于我们在积 $1 \times m$ 上操作，我们必须提升对 $\langle \eta, id_m \rangle$ ，这确保我们“不触碰” m 。最后，我们使用 μ 执行“乘法”。

我们希望结果与 m 的原始元素相同，但不提及元素。因此，我们只需使用左单位子 λ 从 $1 \times m$ 到 m ，而不“搅动物物”。

$$\begin{array}{ccc} 1 \times m & \xrightarrow{\eta \times id_m} & m \times m \\ & \searrow \lambda & \downarrow \mu \\ & & m \end{array}$$

这是右单位的类似定律：

$$\begin{array}{ccc} m \times m & \xleftarrow{id_m \times \eta} & m \times 1 \\ \downarrow \mu & \swarrow \rho & \\ m & & \end{array}$$

为了表述结合律，我们必须从三重积开始并批量操作。这里， α 是结合子，它重新排列积而不“搅动物物”。

$$\begin{array}{ccc} (m \times m) \times m & \xrightarrow{\alpha} & m \times (m \times m) \\ \downarrow \mu \times id & & \downarrow id \times \mu \\ m \times m & \xrightarrow{\mu} & m \times m \\ & \searrow \mu & \swarrow \mu \\ & m & \end{array}$$

注意，我们不必对与对象 m 和 1 一起使用的范畴积做很多假设。特别是我们从未使用过投影。这表明上述定义在任意么半范畴中的张量积上同样适用。它甚至不必是对称的。我们只需假设：存在一个单位对象，积是函子性的，并且它满足单位和结合律的同构。

因此，如果我们将 \times 替换为 \otimes ，将 1 替换为 I ，我们就得到了在任意么半范畴中么半群的定义。

么半群在么半范畴中是一个对象 m ，配备两个态射：

$$\mu: m \otimes m \rightarrow m$$

$$\eta: I \rightarrow m$$

满足单位和结合律：

$$\begin{array}{ccc} 1 \otimes m & \xrightarrow{\eta \otimes id_m} & m \otimes m \xleftarrow{id_m \otimes \eta} m \otimes 1 \\ & \searrow \lambda & \downarrow \mu \swarrow \rho \\ & & m \end{array}$$

$$\begin{array}{ccc} (m \otimes m) \otimes m & \xrightarrow{\alpha} & m \otimes (m \otimes m) \\ \downarrow \mu \otimes id_m & & \downarrow id_m \otimes \mu \\ m \otimes m & \xrightarrow{\mu} & m \otimes m \\ & \searrow \mu \swarrow \mu & \\ & & m \end{array}$$

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

Chapter 6

函数类型

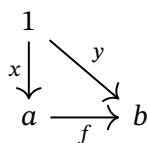
在函数式编程中，还有另一种组合方式处于核心地位。它发生在你将一个函数作为参数传递给另一个函数时。外部函数可以将这个参数作为其自身机制的可插拔部分来使用。例如，它允许你实现一个通用的排序算法，该算法接受任意的比较函数。

如果我们将函数建模为对象之间的箭头，那么将函数作为参数意味着什么？

我们需要一种将函数对象化的方法，以便定义以“箭头对象”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高阶函数。高阶函数是函数式编程的主力军。

消去规则

函数的一个定义特性是它可以应用于一个参数以产生结果。我们已经用组合的方式定义了函数应用：



这里 f 被表示为从 a 到 b 的箭头，但我们希望能够用箭头对象的元素替换 f ，或者如数学家所称的，指数对象 b^a ；或者如我们在编程中所称的，函数类型 $a \rightarrow b$ 。

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：

$$f : 1 \rightarrow b^a$$

$$x : 1 \rightarrow a$$

它应该产生一个元素：

$$y : 1 \rightarrow b$$

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

我们知道，一对元素 (f, x) 等价于积 $b^a \times a$ 的一个元素。因此，我们可以将函数应用定义为单个箭头：

$$\varepsilon_{ab} : b^a \times a \rightarrow b$$

这样，应用的结果 y 由以下交换图定义：

$$\begin{array}{ccc} 1 & & \\ (f, x) \downarrow & \searrow y & \\ b^a \times a & \xrightarrow{\varepsilon_{ab}} & b \end{array}$$

函数应用是函数类型的消去规则。

当某人给你一个函数对象的元素时，你唯一能做的就是使用 ε 将其应用于参数类型的元素。

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某个其他对象 c 构造函数对象 b^a 。这意味着存在一个箭头

$$h : c \rightarrow b^a$$

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

函子性允许我们将一对箭头应用于积以得到另一个积。这里，这对箭头是 (h, id_a) （我们希望将 c 变为 b^a ，但我们不感兴趣修改 a ）

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

我们现在可以跟随这个箭头进行函数应用，以到达 b

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

这个复合箭头定义了一个我们称为 f 的映射：

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

这是对应的图

$$\begin{array}{ccc} c \times a & & \\ h \times id_a \downarrow & \searrow f & \\ b^a \times a & \xrightarrow{\varepsilon} & b \end{array}$$

这个交换图告诉我们，给定一个 h ，我们可以构造一个 f ；但我们也可以要求相反的情况：每个映射 $f: c \times a \rightarrow b$ ，应该唯一地定义一个映射到指数的 $h: c \rightarrow b^a$ 。

我们可以使用这个性质，即两组箭头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，积是使用其映射入性质定义的。另一方面，函数应用被定义为积的映射出。

柯里化

有几种方式可以看待这个定义。一种方式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函数。这并不是一个真正的限制，因为我们总是可以将一个双参数函数实现为从积出发的（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
```

另一方面， h 是一个返回函数的函数：

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

柯里化是这两组箭头之间的同构。

这种同构可以在 **Haskell** 中通过一对（高阶）函数来表示。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

$$h = \text{curry } f$$

当然，这样写时，**curry**和**uncurry**的类型对应于函数对象而不是箭头。这种区别通常被忽略，因为指数的元素与定义它们的箭头之间存在一一对应关系。当我们用终端对象替换任意对象 c 时，这一点很容易看到。我们得到：

$$\begin{array}{ccc} 1 \times a & & \\ \downarrow h \times \text{id}_a & \searrow f & \\ b^a \times a & \xrightarrow{\varepsilon_{ab}} & b \end{array}$$

在这种情况下， h 是对象 b^a 的一个元素， f 是从 $1 \times a$ 到 b 的箭头。但我们知道 $1 \times a$ 与 a 同构，因此，实际上， f 是从 a 到 b 的箭头。

因此，从现在开始，我们将把箭头`->`称为箭头 \rightarrow ，而不做太多区分。这种现象的正确说法是，该范畴是自丰富的。

我们可以将 ε_{ab} 写为 Haskell 函数 `apply`：

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

但这只是一个语法技巧：函数应用是内置在语言中的：`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，但在 Haskell 中则不然。

尽管将函数应用定义为一个单独的函数可能看起来是多余的，但 Haskell 库确实为此提供了一个中缀运算符`$`：

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

不过，技巧在于，常规的函数应用是左结合的，例如，`f x y`与`(f x) y`相同；但美元符号是右结合的，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）双参数的函数；在第二个例子中，它可以是一个单参数的函数。

在 Haskell 中，柯里化无处不在。双参数函数几乎总是写成一个返回函数的函数。由于函数箭头`->`是右结合的，因此不需要为这种类型加括号。例如，对构造函数的签名是：

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

你可以将其视为一个双参数函数返回一个对，或者一个单参数函数返回一个单参数函数，`b->(a, b)`。这样，部分应用这样的函数是可以的，结果是另一个函数。例如，我们可以定义：

```
pairWithTen :: a -> (Int, a)
pairWithTen = pair 10 -- pair 的部分应用
```

与 λ 演算的关系

看待函数对象定义的另一方式是将`c`解释为`f`定义的环境的类型。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：

$$ax^2 + bx + c$$

你可以将其视为由实数三元组 (a, b, c) 和变量 x 参数化，假设 x 是一个复数。三元组是积 $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ 的一个元素。这个积是我们表达式的环境 Γ 。

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

$$f: \Gamma \times \mathbb{C} \rightarrow \mathbb{C}$$

这是一个从积出发的映射，因此我们可以用它来构造函数对象 $\mathbb{C}^{\mathbb{C}}$ 并定义映射 $h: \Gamma \rightarrow \mathbb{C}^{\mathbb{C}}$

$$\begin{array}{ccc} \Gamma \times \mathbb{C} & & \\ \downarrow h \times id_{\mathbb{C}} & \searrow f & \\ \mathbb{C}^{\mathbb{C}} \times \mathbb{C} & \xrightarrow{\varepsilon} & \mathbb{C} \end{array}$$

这个新的映射 h 可以被视为函数对象的构造函数。生成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，三元组参数 (a, b, c) 。

对应于我们原始的表达式 $ax^2 + bx + c$ ，在 $\mathbb{C}^{\mathbb{C}}$ 中有一个特定的函数，我们将其写为：

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

或者，在 **Haskell** 中，用反斜杠替换 λ ，

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

箭头 $h: \Gamma \rightarrow \mathbb{C}^{\mathbb{C}}$ 由箭头 f 唯一确定。这个映射产生一个我们称为 $\lambda x.f$ 的函数。

一般来说，函数对象的定义图变为：

$$\begin{array}{ccc} \Gamma \times a & & \\ \downarrow h \times id_a & \searrow f & \\ b^a \times a & \xrightarrow{\varepsilon} & b \end{array}$$

为表达式 f 提供自由参数的环境 Γ 是表示参数类型的多个对象的积（在我们的例子中，它是 $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ ）。

空环境由终端对象 **1** 表示，即积的单位。在这种情况下， f 只是一个箭头 $a \rightarrow b$ ，而 h 简单地从函数对象 b^a 中选择一个对应于 f 的元素。

重要的是要记住，一般来说，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。

这是我们的例子在 **Haskell** 中的翻译。对应于 f ，我们有一个表达式：

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

如果我们使用`Double`来近似 \mathbb{R} ，我们的环境是积`(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，如`(a :+ 0)`。

对应的箭头 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)
```

肯定前件

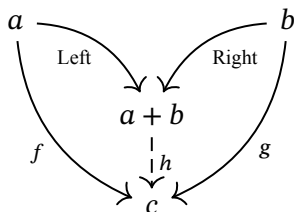
在逻辑中，函数对象对应于蕴涵。从终端对象到函数对象的箭头是该蕴涵的证明。函数应用 ε 对应于逻辑学家所称的肯定前件：如果你有蕴涵 $A \Rightarrow B$ 的证明和 A 的证明，那么这就构成了 B 的证明。

6.1 和与积的再探

当函数获得与其他类型元素相同的地位时，我们就有工具直接将图转换为代码。

和类型

让我们从和的定义开始。



我们说，箭头对 (f, g) 唯一地确定了从和出发的映射 h 。我们可以使用高阶函数简洁地写出它：

```
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
```

这个函数接受一对函数作为参数，并返回一个函数。

首先，我们对对 (f, g) 进行模式匹配以提取 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`，并对其进行案例分析。如果它是用 `Left` 构造的，我们将 f 应用于其内容，否则我们将 g 应用于其内容。

请注意，我们返回的函数是一个闭包。它从其环境中捕获 f 和 g 。

我们实现的函数紧密遵循图，但它不是用通常的 Haskell 风格编写的。Haskell 程序员更喜欢对多参数函数进行柯里化。此外，如果可能，他们更喜欢消除 `lambda`。

这是 Haskell 标准库中相同函数的版本，它被称为（小写）`either`：

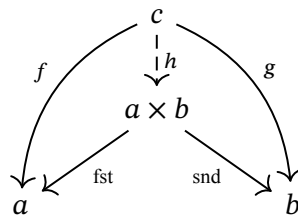
```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x)      = f x
either _ g (Right y)     = g y
```

双射的另一个方向，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)
```

积类型

积类型由其映射入性质对偶定义。



这是该图的直接 Haskell 解读

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

这是用 Haskell 风格编写的风格化版本，作为中缀运算符`&&&`

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

双射的另一个方向由以下给出：

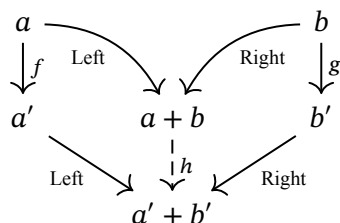
```
fork :: (c -> (a, b)) -> (c -> a, c -> b)
fork h = (fst . h, snd . h)
```

它也紧密遵循图的解读。

函子性的再探

和与积都是函子性的，这意味着我们可以将函数应用于它们的内容。我们准备将这些图转换为代码。

这是和类型的函子性：



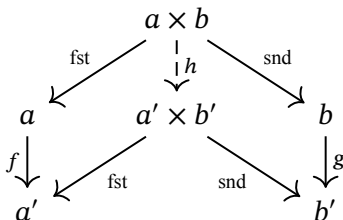
阅读这个图，我们可以立即使用`either`写出 h ：

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

或者我们可以扩展它并称之为`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)
```

类似地，对于积类型：



h 可以写为：

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

或者可以扩展为

```
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,我们都将这个高阶函数称为**bimap**,因为在 **Haskell** 中,和与积都是一个更一般类**Bifunctor**的实例。

6.2 函数类型的函子性

函数类型,或称指数类型,也具有函子性,但有一个转折。我们关注的是从 b^a 到 $b^{a'}$ 的映射,其中带撇的对象通过某些箭头(待确定)与不带撇的对象相关联。

指数类型由其映射入性质定义,因此如果我们寻找

$$k: b^a \rightarrow b^{a'}$$

我们应该绘制以 k 作为映射到 $b^{a'}$ 的图表。我们通过将 b^a 替换为 c 并将带撇的对象替换为不带撇的对象,从原始定义中得到这个图表:

$$\begin{array}{ccc} b^a \times a' & & \\ \downarrow k \times id_a & \searrow g & \\ b^{a'} \times a' & \xrightarrow{\varepsilon} & b' \end{array}$$

问题是:我们能否找到一个箭头 g 来完成这个图表?

$$g: b^a \times a' \rightarrow b'$$

如果我们找到这样的 g ,它将唯一地定义我们的 k 。

思考这个问题的方法是考虑如何实现 g 。它以乘积 $b^a \times a'$ 作为其参数。将其视为一对:一个从 a 到 b 的函数对象元素和一个 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物。但 b^a 需要一个类型为 a 的参数,而我们手头只有 a' 。除非有人给我们一个箭头 $a' \rightarrow a$,否则我们无法做任何事情。这个箭头应用于 a' 将生成 b^a 的参数。然而,应用的结果是类型 b ,而 g 应该产生一个 b' 。同样,我们需要一个箭头 $b \rightarrow b'$ 来完成我们的任务。

这听起来可能很复杂,但归根结底,我们需要在带撇和不带撇的对象之间有两个箭头。转折在于第一个箭头从 a' 到 a ,这感觉与通常的函子性考虑相反。为了将 b^a 映射到 $b^{a'}$,我们需要一对箭头:

$$f: a' \rightarrow a$$

$$g: b \rightarrow b'$$

这在 **Haskell** 中更容易解释。我们的目标是实现一个函数 $a' \rightarrow b'$, 给定一个函数 $h :: a \rightarrow b$ 。

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 $f :: a' \rightarrow a$ 。

由于`h`产生一个`b`，而我们希望返回一个`b'`，我们需要另一个函数`g :: b -> b'`。所有这些都很好地融入一个高阶函数：

```
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`的接口，`dimap`是类型类`Profunctor`的成员。

6.3 双笛卡尔闭范畴

在范畴论中，如果一个范畴对任意两个对象都定义了积和指数，并且具有终对象，则该范畴称为笛卡尔闭范畴。其核心思想是，`hom` 集并不是该范畴的外来概念：范畴在形成 `hom` 集的操作下是“闭”的。

如果该范畴还具有和（余积）和始对象，则称为双笛卡尔闭范畴。

这是建模编程语言所需的最小结构。

使用这些操作构造的数据类型称为代数数据类型。我们拥有类型的加法、乘法和指数运算（但没有减法或除法）；并且满足我们在高中代数中熟悉的所有定律。这些定律在同构的意义下成立。还有一个我们尚未讨论的代数定律。

分配律

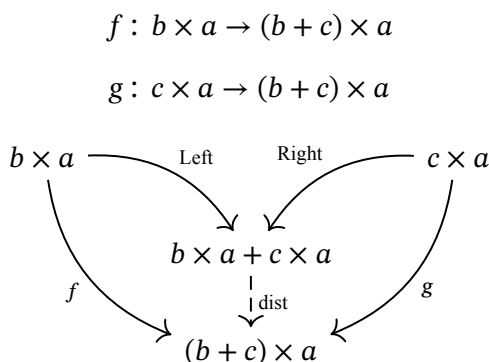
数的乘法对加法具有分配性。在双笛卡尔闭范畴中，我们是否应该期望同样的性质？

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

从左到右的映射很容易构造，因为它同时是一个从和出发的映射和一个到积的映射。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)
```

从左侧的和出发的映射由一对箭头给出：

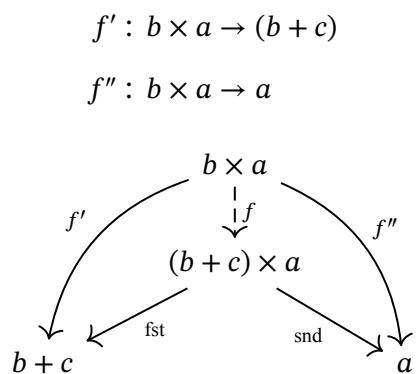


我们在 Haskell 中将其写为：

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

`where` 子句用于引入子函数的定义。

现在我们需要实现 f 和 g 。它们都是到积的映射，因此每个都等价于一对箭头。例如，第一个由以下箭头对给出：



在 Haskell 中：

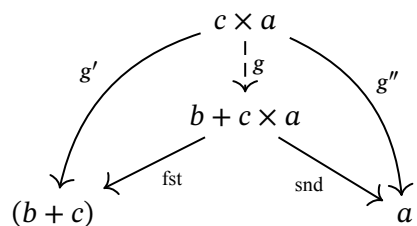
```
f = f' &&& f''
f' :: (b, a) -> Either b c
f'' :: (b, a) -> a
```

第一个箭头可以通过投影第一个分量 b ，然后使用 `Left` 来构造和。第二个箭头只是投影 `snd`：

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

$$f'' = \text{snd}$$

类似地，我们将 g 分解为 g' 和 g'' ：



将这些组合在一起，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

```

它们也可以内联以产生这种简洁的形式：

```

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

```

这种编程风格称为无点风格，因为它省略了参数（点）。出于可读性考虑，Haskell 程序员更喜欢更显式的风格。上述函数通常会实现为：

```

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

```

注意，我们只使用了和与积的定义。同构的另一个方向需要使用指数，因此它仅在双笛卡尔闭范畴中有效。这在直接的 Haskell 实现中并不明显：

```

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。

这是该函数的无点版本：

```
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`curry`和`uncurry`来实现映射。

我们将在稍后回到这个恒等式，那时我们将拥有更强大的工具：伴随。

Exercise 6.3.1. 证明：

$$2 \times a \cong a + a$$

其中 2 是布尔类型。首先用图解法证明，然后实现两个 *Haskell* 函数来见证这个同构。

递归

“道生一。

一生二。

二生三。

三生万物。”

当你站在两面镜子之间时，你会看到自己的倒影，倒影的倒影，倒影的倒影的倒影，依此类推。每个倒影都是根据前一个倒影定义的，但它们共同产生了无限。

递归是一种分解模式，它将单个任务分解为多个步骤，步骤的数量可能是无限的。

递归基于对怀疑的悬置。你面临的任务可能需要任意多个步骤。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之外的所有问题，我将如何迈出最后一步？”

7.1 自然数

自然数对象 N 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出发的箭头来定义一个特殊元素。按照惯例，我们将这个箭头称为 Z ，表示“零”。

$$Z: 1 \rightarrow N$$

但我们必须能够定义无限多个箭头，以说明对于每个自然数，都有一个比它大一的数。

我们可以通过以下方式形式化这一陈述：假设我们知道如何创建一个自然数 $n: 1 \rightarrow N$ 。我们如何迈出下一步，指向下一个数——它的后继？

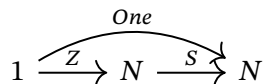
这个下一步可以简单地将 n 与一个从 N 回到 N 的箭头进行后复合。这个箭头不应该是恒等箭头，因为我们希望一个数的后继与该数不同。但一个这样的箭头，我们称之为 S ，表示“后继”，就足够了。

对应于 n 的后继的元素由以下复合给出：

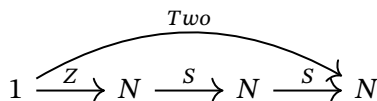
$$1 \xrightarrow{n} N \xrightarrow{S} N$$

(有时我们在单个图中多次绘制同一个对象，以便拉直循环箭头。)

特别地，我们可以将 *One* 定义为 Z 的后继：



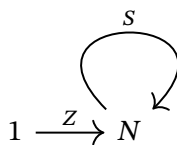
将 *Two* 定义为 Z 的后继的后继：



依此类推。

引入规则

两个箭头 Z 和 S 作为自然数对象 N 的引入规则。其中的一个是递归的： S 使用 N 作为其源和目标。



这两个引入规则直接翻译为 Haskell：

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

它们可以用来定义任意的自然数；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中并不十分有用。然而，它通常用于定义类型级别的自然数，其中每个数字都是其自己的类型。

你可能会在皮亚诺算术的名称下遇到这种构造。

消除规则

引入规则是递归的这一事实在定义消除规则时稍微复杂了一些。我们将遵循前面章节的模式，首先假设我们有一个从 N 出发的映射：

$$h : N \rightarrow a$$

然后看看我们能从中推断出什么。

之前，我们能够将这样的 h 分解为更简单的映射（对于和与积的映射对；对于指数的积的映射）。

N 的引入规则看起来与和的引入规则相似（它要么是 Z ，要么是后继），因此我们期望 h 可以拆分为两个箭头。确实，我们可以通过复合 $h \circ Z$ 轻松得到第一个箭头。这是一个选择 a 元素的箭头。我们称之为 $init$ ：

$$init : 1 \rightarrow a$$

但没有明显的方法找到第二个箭头。

为了看到这一点，让我们扩展 N 的定义：

$$1 \xrightarrow{Z} N \xrightarrow{S} N \xrightarrow{S} N \quad \dots$$

并将 h 和 $init$ 插入其中：

$$\begin{array}{ccccccc} 1 & \xrightarrow{Z} & N & \xrightarrow{S} & N & \xrightarrow{S} & N \quad \dots \\ & \searrow^{init} & \downarrow h & & \downarrow h & & \downarrow h \\ & & a & & a & & a \end{array}$$

直觉上，从 N 到 a 的箭头表示 a 的元素序列 a_n 。第零个元素由

$$a_0 = init$$

给出。下一个元素是

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

接着是

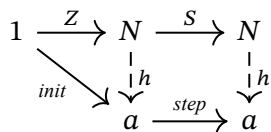
$$a_2 = h \circ S \circ S \circ Z$$

依此类推。

因此，我们用一个箭头 h 替换了无限多个箭头 a_n 。诚然，新的箭头更简单，因为它们表示 a 的元素，但它们的数量是无限的。

问题在于，无论如何看待它，从 N 出发的任意映射都包含无限的信息。

我们必须大幅简化问题。由于我们使用单个箭头 S 生成所有自然数，我们可以尝试使用单个箭头 $a \rightarrow a$ 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 出发的映射称为递归的。并非所有从 N 出发的映射都是递归的。事实上，递归映射非常少；但递归映射足以定义自然数对象。

我们使用上述图作为消除规则。我们规定，每个从 N 出发的递归映射 h 与一对 $init$ 和 $step$ 一一对应。

这意味着求值规则（为给定的 h 提取 $(init, step)$ ）不能为任意箭头 $h: N \rightarrow a$ 制定，只能为那些先前使用 $(init, step)$ 递归定义的箭头制定。

箭头 $init$ 总是可以通过复合 $h \circ Z$ 恢复。箭头 $step$ 是以下方程的解：

$$step \circ h = h \circ S$$

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。

直觉上，对 $init$ 和 $step$ 生成元素序列 a_0, a_1, a_2, \dots 如果两个箭头 h 和 h' 由相同的对 $(init, step)$ 给出，这意味着它们生成的序列是相同的。

因此，如果 h 与 h' 不同，这意味着 N 包含的不仅仅是元素序列 $Z, SZ, S(SZ), \dots$ 例如，如果我们将 -1 添加到 N 中（即让 Z 成为某人的后继），我们可能会有 h 和 h' 在 -1 处不同，但由相同的 $init$ 和 $step$ 生成。唯一性意味着在 Z 和 S 生成的数字之前、之后或之间没有自然数。

我们在这里讨论的消除规则对应于原始递归。我们将在关于依赖类型的章节中看到这个规则的更高级版本，对应于归纳原理。

在编程中

消除规则可以在 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：


```

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其与 n 相加。

在实践中，程序员更喜欢直接实现递归——这种方法相当于内联递归器`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)

```

可以理解为：如果 m 为零，则结果为 n 。否则，如果 m 是某个 k 的后继，则结果是 $k + n$ 的后继。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到第 n 步的所有工作，并将该结果与下一个连续步骤结合起来。

另一方面，递归在处理递归定义的数据结构（如列表或树）时更为自然。

这两种方法是等价的，编译器通常将递归函数转换为循环，这称为尾递归优化。

Exercise 7.1.1. 使用递归器实现一个将`Nat`转换为`Int`的函数。

Exercise 7.1.2. 将加法的柯里化版本实现为从 N 到函数对象 N^N 的映射。提示：在递归器中使用这些类型：

```

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

```

7.2 列表

事物列表要么为空，要么是一个事物后跟一个事物列表。

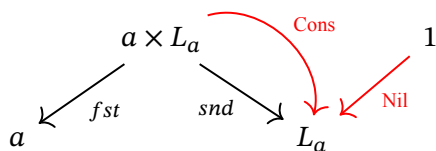
这个递归定义转化为类型 L_a （ a 的列表）的两个引入规则：

$$\text{Nil} : 1 \rightarrow L_a$$

$$\text{Cons} : a \times L_a \rightarrow L_a$$

`Nil`元素描述一个空列表，`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。投影提取使用 `Cons` 构造的列表的头部和尾部。



这个描述可以立即翻译为 Haskell:

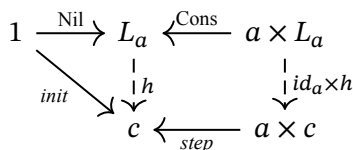
```
data List a where
  Nil  :: List a
  Cons :: (a, List a) -> List a
```

消除规则

假设我们有一个从 a 的列表到某个任意类型 c 的映射:

$$h : L_a \rightarrow c$$

这是我们如何将其插入列表定义的方式:



我们使用积的函子性将 (id_a, h) 应用于积 $a \times L_a$ 。

类似于自然数对象，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:

$$step \circ (id_a \times h) = h \circ Cons$$

同样，并非每个 h 都可以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，或列表的范畴同态。

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`，它生成一个从列表出发的映射。

列表是如此基本的数据类型，以至于 `Haskell` 为其内置了语法。类型`(List a)`写为`[a]`。`Nil`构造函数是一对空的方括号，`[]`，`Cons`构造函数是中缀冒号`(:)`。

我们可以对这些构造函数进行模式匹配。从列表出发的通用映射具有以下形式：

```
h :: [a] -> c
h []      = -- 空列表的情况
h (a : as) = -- 非空列表的头部和尾部的情况
```

对应于列表递归器`recList`，以下是标准库中函数`foldr`（右折叠）的类型签名：

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

以下是一个可能的实现：

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

作为示例，我们可以使用`foldr`计算自然数列表的元素之和：

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

Exercise 7.2.1. 考虑在列表定义中将 a 替换为终端对象时会发生什么。提示：自然数的基一编码是什么？

Exercise 7.2.2. 有多少映射 $h: L_a \rightarrow 1 + a$ ？我们可以使用列表递归器获得所有这些映射吗？`Haskell` 函数的签名如何：

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

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

7.3 函子性

函子性大致意味着能够转换数据结构的“内容”。列表 L_a 的内容是类型 a 。给定一个箭头 $f: a \rightarrow b$ ，我们需要定义列表的映射 $h: L_a \rightarrow L_b$ 。

列表由映射出属性定义，因此让我们将消除规则的目标 c 替换为 L_b 。我们得到：

$$\begin{array}{ccccc}
 1 & \xrightarrow{\text{Nil}_a} & L_a & \xleftarrow{\text{Cons}_a} & a \times L_a \\
 & \searrow \text{init} & \downarrow h & & \downarrow id_a \times h \\
 & & L_b & \xleftarrow{\text{step}} & a \times L_b
 \end{array}$$

由于我们在这里处理两个不同的列表，我们必须区分它们的构造函数。例如，我们有：

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

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

Cons 也是如此。

init 的唯一候选是 Nil_b ，这意味着 h 作用于 a 的空列表时生成 b 的空列表：

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

剩下的就是定义箭头：

$$\text{step} : a \times L_b \rightarrow L_b$$

我们可以猜测：

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

这对应于 Haskell 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
```

你可能会想知道是什么阻止我们选择 $\text{step} = \text{snd}$ ，从而导致：

```
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：当我们对 *id* 应用 `badMap` 时会发生什么？）

Chapter 8

函子

8.1 范畴

到目前为止，我们只见过一个范畴——类型和函数的范畴。因此，让我们快速了解范畴的基本信息。

一个范畴是对象和它们之间的箭头的集合。每一对可组合的箭头都可以组合。组合是结合的，并且每个对象都有一个循环的恒等箭头。

类型和函数形成一个范畴的事实可以在 **Haskell** 中通过定义组合来表达：

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

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

恒等是一个多态的“什么都不做”的函数：

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

你可以很容易地确信这种组合是结合的，并且与 **id** 组合不会对函数产生任何影响。

基于范畴的定义，我们可以想出各种奇怪的范畴。例如，有一个没有对象和箭头的范畴。它空洞地满足所有范畴的条件。还有一个只包含一个对象和一个箭头的范畴（你能猜出这个箭头是什么吗?）。有一个包含两个不连接对象的范畴，还有一个两个对象通过一个箭头连接（加上两个恒等箭头）的范畴，等等。这些是我称之为简笔范畴的例子——只有少量对象和箭头的范畴。

集合范畴

我们也可以去掉一个范畴中的所有箭头（除了恒等箭头）。这种只有对象的范畴被称为离散范畴或集合¹。由于我们将箭头与结构关联起来，集合是一个没有结构的范畴。

集合形成自己的范畴，称为**Set**²。该范畴中的对象是集合，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对的集合。

在最粗略的近似中，我们可以在集合范畴中建模编程。我们通常将类型视为值的集合，将函数视为集合论函数。这并没有错。事实上，我们迄今为止描述的所有范畴构造都有其集合论根源。范畴积是集合的笛卡尔积的推广，和是不交并，等等。

范畴论提供的是更高的精确性：绝对必要的结构与多余细节之间的细微区别。

例如，集合论函数并不符合我们作为程序员使用的函数的定义。我们的函数必须具有底层算法，因为它们必须由某些物理系统（无论是计算机还是人脑）计算。集合论函数比算法多得多。而且一些算法（在图灵完备的语言中）可能永远运行而不会产生结果。

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起点来构造其他范畴。

其中一个范畴称为对偶范畴。这是所有原始箭头都被反转的范畴：在原始范畴中称为箭头的源的东西现在称为其目标，反之亦然。

范畴 \mathcal{C} 的对偶称为 \mathcal{C}^{op} 。我们在讨论对偶性时已经瞥见过这个范畴。 \mathcal{C}^{op} 的对象与 \mathcal{C} 的对象相同。

每当 \mathcal{C} 中有一个箭头 $f: a \rightarrow b$ 时， \mathcal{C}^{op} 中就有一个对应的箭头 $f^{op}: b \rightarrow a$ 。

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

\mathcal{C} 中的终对象是 \mathcal{C}^{op} 中的始对象， \mathcal{C} 中的积是 \mathcal{C}^{op} 中的和，等等。

积范畴

给定两个范畴 \mathcal{C} 和 \mathcal{D} ，我们可以构造一个积范畴 $\mathcal{C} \times \mathcal{D}$ 。该范畴中的对象是对象对 $\langle c, d \rangle$ ，箭头是箭头对。

如果我们在 \mathcal{C} 中有一个箭头 $f: c \rightarrow c'$ ，在 \mathcal{D} 中有一个箭头 $g: d \rightarrow d'$ ，那么在 $\mathcal{C} \times \mathcal{D}$ 中就有一个对应的箭头 $\langle f, g \rangle$ 。这个箭头从 $\langle c, d \rangle$ 到 $\langle c', d' \rangle$ ，两者都是 $\mathcal{C} \times \mathcal{D}$ 中的对象。如果它们的组件分别在 \mathcal{C} 和 \mathcal{D} 中可组合，那么这两个箭头可以组合。恒等箭头是一对恒等箭头。

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。

¹忽略“大小”问题。

²再次忽略“大小”问题，特别是所有集合的集合不存在的问题。

在这两个范畴中，对象都是来自 \mathcal{C} 的对象对。在第一个范畴 $\mathcal{C} \times \mathcal{C}$ 中，从 $\langle a, b \rangle$ 到 $\langle a', b' \rangle$ 的态射是一对 $\langle f: a \rightarrow a', g: b \rightarrow b' \rangle$ 。在第二个范畴 $\mathcal{C}^{op} \times \mathcal{C}$ 中，态射是一对 $\langle f: a' \rightarrow a, g: b \rightarrow b' \rangle$ ，其中第一个箭头方向相反。

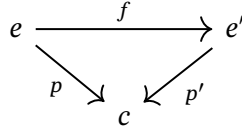
切片范畴

在一个井然有序的宇宙中，对象始终是对象，箭头始终是箭头。但有时箭头的集合可以被视为对象。然而，切片范畴打破了这种整齐的分隔：它们将单个箭头变成对象。

切片范畴 \mathcal{C}/c 描述了特定对象 c 从其范畴 \mathcal{C} 的角度来看的样子。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端。由于其中一端固定为 c ，我们只需要指定另一端。

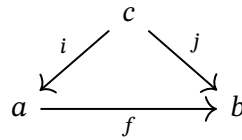
切片范畴 \mathcal{C}/c （也称为上范畴）中的一个对象是一对 $\langle e, p \rangle$ ，其中 $p: e \rightarrow c$ 。

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



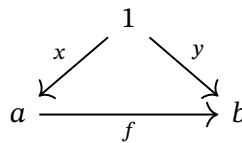
余切片范畴

有一个对偶的概念，即余切片范畴 c/\mathcal{C} ，也称为下范畴。它是一个从固定对象 c 发出的箭头的范畴。该范畴中的对象是 $\langle a, i: c \rightarrow a \rangle$ 对。 c/\mathcal{C} 中的态射是使相关三角形交换的箭头。



特别是，如果范畴 \mathcal{C} 有一个终对象 1 ，那么余切片 $1/\mathcal{C}$ 的对象是 \mathcal{C} 中所有对象的全局元素。

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



特别是，从类型和函数的范畴构造余切片范畴，证明了我们z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 函子 (Functors)

在讨论代数数据类型时，我们已经看到了函子性的例子。其核心思想是，这样的数据类型“记住”了它的创建方式，我们可以通过对它的“内容”应用一个箭头来操作这种记忆。

在某些情况下，这种直觉非常具有说服力：我们将积类型视为一个“包含”其成分的对。毕竟，我们可以使用投影来检索它们。

在函数对象的情况下，这一点就不那么明显了。你可以将函数对象想象为秘密存储所有可能的结果，并使用函数参数来索引它们。从`Bool`出发的函数显然等价于一对值，一个对应`True`，一个对应`False`。将某些函数实现为查找表是一种已知的编程技巧，称为记忆化 (*memoization*)。

尽管对以自然数作为参数的函数进行记忆化并不实际，但我们仍然可以将它们概念化为（无限的，甚至不可数的）查找表。

如果你能将数据类型视为值的容器，那么应用一个函数来转换所有这些值并创建一个转换后的容器是有意义的。当这是可能的时候，我们说该数据类型是函子性的 (*functorial*)。

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由某种类型键控的查找表。如果你想使用另一个相关类型作为键，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有一个箭头是反向的：

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

你正在对一个函数 `h :: a -> b` 应用转换，该函数有一个响应 `a` 类型值的“接收器”，而你希望用它来处理 `a'` 类型的输入。这只有在你有从 `a'` 到 `a` 的转换器时才有可能，即 `f :: a' -> a`。

数据类型“包含”另一种类型的值的想法也可以通过说一个数据类型由另一个类型参数化来表达。例如，类型 `List a` 由类型 `a` 参数化。

换句话说，`List` 将类型 `a` 映射到类型 `List a`。`List` 本身，不带参数，被称为类型构造器 (*type constructor*)。

范畴之间的函子

在范畴论中，类型构造器被建模为对象到对象的映射。它是对象上的函数。这不应与对象之间的箭头混淆，后者是范畴结构的一部分。

事实上，更容易想象的是范畴之间的映射。源范畴中的每个对象都被映射到目标范畴中的一个对象。如果 `a` 是 `C` 中的一个对象，那么在 `D` 中就有对应的对象 `Fa`。

函子映射，或函子 (*functor*)，不仅映射对象，还映射它们之间的箭头。第一个范畴中的每个箭头

$$f: a \rightarrow b$$

在第二个范畴中都有一个对应的箭头：

$$Ff : Fa \rightarrow Fb$$

$$\begin{array}{ccc} a & \xrightarrow{\quad\quad\quad} & Fa \\ \downarrow f & & \downarrow Ff \\ b & \xrightarrow{\quad\quad\quad} & Fb \end{array}$$

我们使用相同的字母，这里是 F ，来命名对象映射和箭头映射。

如果范畴提炼了结构的本质，那么函子就是保持这种结构的映射。源范畴中相关的对象在目标范畴中也是相关的。

范畴的结构由箭头及其组合定义。因此，函子必须保持组合。在一个范畴中组合的：

$$h = g \circ f$$

在第二个范畴中应保持组合：

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

我们可以在 \mathcal{C} 中组合两个箭头并将组合映射到 \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。

$$\begin{array}{ccc} a & \xrightarrow{\quad\quad\quad} & Fa \\ \downarrow f & & \downarrow Ff \\ b & & Fb \\ \downarrow g & & \downarrow Fg \\ c & \xrightarrow{\quad\quad\quad} & Fc \end{array} \quad \begin{array}{c} \text{Left side (in } \mathcal{C} \text{): } g \circ f \text{ maps } a \text{ to } c. \\ \text{Right side (in } \mathcal{D} \text{): } Fg \circ Ff \text{ maps } Fa \text{ to } Fc. \\ \text{Middle (in } \mathcal{D} \text{): } F(g \circ f) \text{ maps } a \text{ to } Fc. \end{array}$$

最后，函子必须保持恒等箭头：

$$Fid_a = id_{Fa}$$

$$\begin{array}{ccc} & & id_{Fa} \\ & \searrow & \uparrow \\ id_a & & F id_a \\ a & \xrightarrow{\quad\quad\quad} & Fa \end{array}$$

这些条件共同定义了函子保持范畴结构的含义。

同样重要的是要意识到哪些条件不是定义的一部分。例如，函子允许将多个对象映射到同一个对象。它也可以将多个箭头映射到同一个箭头，只要端点匹配。

在极端情况下，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必被函子覆盖。在极端情况下，我们可以有一个从单例范畴到任何（非空）范畴的函子。这样的函子选择一个对象及其恒等箭头。

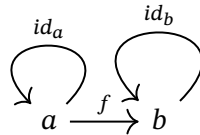
常函子 (*constant functor*) Δ_c 是一个将源范畴中的所有对象映射到目标范畴中的单个对象 c ，并将源范畴中的所有箭头映射到单个恒等箭头 id_c 的函子的例子。

在范畴论中，函子通常用于在一个范畴内创建另一个范畴的模型。它们可以将多个对象和箭头合并为一个，这意味着它们产生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部分内容，这意味着模型被嵌入到更大的环境中。

来自某些极简的、简笔画的范畴的函子可以用于定义更大范畴中的模式。

Exercise 8.2.1. 描述一个源为“行走箭头”范畴的函子。它是一个简笔画范畴，有两个对象和它们之间的一个箭头（加上必需的恒等箭头）。



Exercise 8.2.2. “行走同构”范畴与“行走箭头”范畴类似，只是多了一个从 b 回到 a 的箭头。证明来自这个范畴的函子总是在目标范畴中选择一个同构。

8.3 编程中的函子

自函子 (Endofunctors) 是最容易在编程语言中表达的一类函子。这些函子将一个范畴（在这里，是类型和函数的范畴）映射到其自身。

自函子

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

列表类型构造器 `List` 将任意类型 `a` 映射到类型 `List a`。

`Maybe` 类型构造器将 `a` 映射到 `Maybe a`。

自函子的第二部分是箭头的映射。给定一个函数 `a -> b`，我们希望能够定义一个函数 `List a -> List b` 或 `Maybe a -> Maybe b`。这是我们之前讨论过的这些数据类型的“函子性” (functoriality) 属性。函子性允许我们将任意函数提升 (lift) 为转换后类型之间的函数。

函子性可以在 `Haskell` 中使用类型类 (typeclass) 来表达。在这种情况下，类型类由类型构造器 `f` 参数化（在 `Haskell` 中，我们使用小写字母表示类型构造器变量）。如果存在一个称为 `fmap` 的对应函数映射，我们就说 `f` 是一个 `Functor`：

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

编译器知道 `f` 是一个类型构造器，因为它被应用于类型，如 `f a` 和 `f b`。

为了向编译器证明某个类型构造器是一个 `Functor`，我们必须为其提供 `fmap` 的实现。这是通过定义类型类 `Functor` 的实例（instance）来完成的。例如：

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

函子还必须满足一些定律：它必须保持复合和恒等。这些定律无法在 `Haskell` 中表达，但应由程序员检查。我们之前看到了一个不满足恒等定律的 `badMap` 定义，但它仍会被编译器接受。它将为列表类型构造器 `[]` 定义一个“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微不足道，但它们是其他函子的构建块。

我们有恒等自函子，它将所有对象映射到自身，并将所有箭头映射到自身。

```
newtype Identity a = Identity a
```

Exercise 8.3.2. 证明 `Identity` 是一个 `Functor`。提示：为其实现 `Functor` 实例。

我们还有一个常函子 Δ_c ，它将所有对象映射到单个对象 `c`，并将所有箭头映射到该对象上的恒等箭头。在 `Haskell` 中，它是由目标对象 `c` 参数化的一族函子：

```
data Constant c a = Constant c
```

这个类型构造器忽略其第二个参数。

Exercise 8.3.3. 证明 `(Constant c)` 是一个 `Functor`。提示：类型构造器接受两个参数，但在 `Functor` 实例中，它被部分应用于第一个参数。它在第二个参数上是函子性的。

双函子

我们还看到了接受两个类型作为参数的数据构造器：积和和。它们也是函子性的，但它们不是提升单个函数，而是提升一对函数。在范畴论中，我们将这些定义为从积范畴 $\mathcal{C} \times \mathcal{C}$ 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称为 **Bifunctor** 的单独类的成员。

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

同样，编译器推断 **f** 是一个双参数类型构造器，因为它看到它被应用于两个类型，例如 **f a b**。

为了向编译器证明某个类型构造器是一个 **Bifunctor**，我们定义一个实例。例如，对的双函子性可以定义为：

```
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)
```

逆变函子

来自对偶范畴 \mathcal{C}^{op} 的函子称为逆变函子 (contravariant functors)。它们具有提升反向箭头的属性。常规函子有时称为协变函子 (covariant functors)。

在 Haskell 中，逆变函子形成类型类 **Contravariant**：

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

通常，将函子视为生产者和消费者是方便的。在这个类比中，(协变) 函子是生产者。您可以通过应用 (使用 **fmap**) 一个函数 **a->b** 将 **a** 的生产者转换为 **b** 的生产者。相反，要将 **a** 的消费者转换为 **b** 的消费者，您需要一个反向的函数 **b->a**。

示例：谓词是返回 **True** 或 **False** 的函数：

```
newtype 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)
```

在实践中，唯一非平凡的逆变函子示例是函数对象的变体。

判断给定函数类型在某个类型参数上是协变还是逆变的一种方法是为其定义中使用的类型分配极性。我们说函数的返回类型处于正位置，因此它是协变的；而参数类型处于负位置，因此它是逆变的。但如果你将整个函数对象放在另一个函数的负位置，那么它的极性会被反转。

考虑这个数据类型：

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

它有一个双重负的 `a`，因此处于正位置。这就是为什么它是一个协变的 **Functor**。它充当 `a` 的生产者：

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

注意，括号在这里很重要。一个类似的函数 `a -> Bool -> Bool` 有一个处于负位置的 `a`。这是因为它是一个返回函数 `(Bool -> Bool)` 的 `a` 的函数。等效地，您可以将其解构为一个接受对的函数：`(a, Bool) -> Bool`。无论哪种方式，`a` 最终都处于负位置。

Profunctors

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，只是其中一个函数是反向的。

在范畴论中，这对应于从两个范畴的积到另一个范畴的函子，其中一个是对偶范畴：它是从 $\mathcal{C}^{op} \times \mathcal{C}$ 的函子。从 $\mathcal{C}^{op} \times \mathcal{C}$ 到 **Set** 的函子称为 *Profunctors*。

在 Haskell 中，Profunctors 形成一个类型类：

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

您可以将 Profunctor 视为同时是生产者和消费者的类型。它消费一个类型并生产另一个类型。

函数类型可以写为中缀运算符 `(->)`，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，即函数 $a \rightarrow b$ 消费类型 a 的参数并生产类型 b 的结果。

在编程中，所有非平凡的 **Profunctors** 都是函数类型的变体。

8.4 Hom-函子

任意两个对象之间的箭头构成一个集合。这个集合称为 **hom-集** (**hom-set**)，通常用范畴的名称后跟对象的名称来表示：

$$\mathcal{C}(a, b)$$

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

另一种看待 **hom-集** 的方式是说它们定义了一个映射，该映射为每对对象分配一个集合 $\mathcal{C}(a, b)$ 。集合本身是范畴 **Set** 中的对象。因此，我们有了一个范畴之间的映射。

这个映射是函子性的。为了理解这一点，让我们考虑当我们变换两个对象 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 $\mathcal{C}(a', b')$ 的变换。**Set** 中的箭头是普通函数，因此只需定义它们对集合中单个元素的作用即可。

$\mathcal{C}(a, b)$ 的一个元素是箭头 $h: a \rightarrow b$ ，而 $\mathcal{C}(a', b')$ 的一个元素是箭头 $h': a' \rightarrow b'$ 。我们知道如何将一个变换为另一个：我们需要用箭头 $g': a' \rightarrow a$ 预组合 h ，并用箭头 $g: b \rightarrow b'$ 后组合它。

换句话说，将一对 $\langle a, b \rangle$ 映射到集合 $\mathcal{C}(a, b)$ 的映射是一个 *profunctor* (*profunctor*):

$$\mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathbf{Set}$$

通常，我们只对变化其中一个对象而保持另一个固定感兴趣。当我们固定源对象并变化目标对象时，结果是一个函子，写作：

$$\mathcal{C}(a, -): \mathcal{C} \rightarrow \mathbf{Set}$$

这个函子对箭头 $g: b \rightarrow b'$ 的作用写作：

$$\mathcal{C}(a, g): \mathcal{C}(a, b) \rightarrow \mathcal{C}(a, b')$$

并由后组合给出：

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

变化 b 意味着将焦点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 发出的箭头组合成从 a 的视角看范畴的一个连贯视图。它是“ a 的世界”。

相反，当我们固定目标并变化 **hom-函子** 的源时，我们得到一个逆变函子：

$$\mathcal{C}(-, b): \mathcal{C}^{op} \rightarrow \mathbf{Set}$$

它对箭头 $g' : a' \rightarrow a$ 的作用写作：

$$\mathcal{C}(g', b) : \mathcal{C}(a, b) \rightarrow \mathcal{C}(a', b)$$

并由预组合给出：

$$\mathcal{C}(g', b) = (- \circ g')$$

函子 $\mathcal{C}(-, b)$ 将所有指向 b 的箭头组织成一个连贯的视图。它是 b 的“世界眼中的图像”。

我们现在可以重新表述同构一章中的结果。如果两个对象 a 和 b 是同构的，那么它们的 **hom**-集也是同构的。特别是：

$$\mathcal{C}(a, x) \cong \mathcal{C}(b, x)$$

和

$$\mathcal{C}(x, a) \cong \mathcal{C}(x, b)$$

我们将在下一章讨论自然性条件。

另一种看待 **hom**-函子 $\mathcal{C}(a, -)$ 的方式是将其视为一个预言机，它回答以下问题：“ a 是否与我相连？”如果集合 $\mathcal{C}(a, x)$ 为空，则答案为否定：“ a 与 x 不相连。”否则，集合 $\mathcal{C}(a, x)$ 的每个元素都是这种连接存在的证明。

相反，逆变函子 $\mathcal{C}(-, a)$ 回答以下问题：“我是否与 a 相连？”

总的来说，**profunctor** $\mathcal{C}(x, y)$ 在对象之间建立了一个证明相关的关系。集合 $\mathcal{C}(x, y)$ 的每个元素都是 x 与 y 相连的证明。如果集合为空，则这两个对象不相关。

8.5 函子复合

就像我们可以复合函数一样，我们也可以复合函子。如果两个函子中一个的目标范畴是另一个的源范畴，那么它们就是可复合的。

在对象上， G 在 F 之后的函子复合首先将 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，你只能复合可复合的函子。然而，所有的自函子（**endofunctors**）都是可复合的，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，因此两个函子的复合也是一个参数化的数据类型。在对象上，我们定义：

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

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

或者，你可以通过提供种类签名 (kind signature) 来告诉编译器 `Compose` 的前两个参数是类型构造函数。你还应该导入定义了 `Type` 的 `Data.Kind` 库：

```
import Data.Kind
```

种类签名类似于类型签名，但它可以用来描述操作类型的函数。

常规类型的种类是 `Type`。类型构造函数的种类是 `Type -> Type`，因为它们将类型映射到类型。

`Compose` 接受两个类型构造函数并生成一个类型构造函数，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
```

任何两个类型构造函数都可以这样复合。在这一点上，没有要求它们必须是函子。

然而，如果我们想使用类型构造函数的复合来提升函数，即 `g` 在 `f` 之后，那么它们必须是函子。这一要求在实例声明中被编码为约束：

```
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)` 表示两个类型构造函数都必须是 `Functor` 类的实例。约束后面跟着一个双箭头。

我们正在建立其函子性的类型构造函数是 `Compose f g`，它是 `Compose` 对两个函子的部分应用。

在 `fmap` 的实现中，我们对数据构造函数 `Compose` 进行模式匹配。它的参数 `gfa` 的类型是 `g (f a)`。我们使用一个 `fmap` 来“进入”`g`。然后我们使用 `(fmap h)` 来“进入”`f`。编译器通过分析类型知道使用哪个 `fmap`。

你可以将复合函子视为容器的容器。例如，`[]` 与 `Maybe` 的复合是一个可选值的列表。

Exercise 8.5.1. 定义一个 `Functor` 在 `Contravariant` 之后的复合。提示：你可以重用 `Compose`，但你必须提供一个不同的实例声明。

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，函子是可复合的，并且很容易验证这种复合是结合的。我们还有每个范畴的恒等（自）函子。因此，范畴本身似乎形成了一个范畴，我们称之为 **Cat**。

这就是数学家开始担心“大小”问题的地方。这是说存在潜在悖论的简写。因此，正确的说法是 **Cat** 是一个小范畴的范畴。但只要我们不涉及存在性证明，我们就可以忽略大小问题。

自然变换

我们已经看到，当两个对象 a 和 b 是同构的时，它们会在箭头集合之间产生双射，我们现在可以将其表示为 **hom**-集之间的同构。对于所有 x ，我们有：

$$\mathcal{C}(a, x) \cong \mathcal{C}(b, x)$$

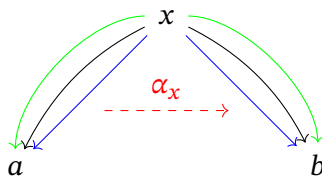
$$\mathcal{C}(x, a) \cong \mathcal{C}(x, b)$$

然而，反过来并不成立。除非满足额外的自然性条件，否则 **hom**-集之间的同构不会导致对象之间的同构。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 Hom-函子之间的自然变换

建立两个对象之间同构的一种方法是直接提供两个箭头——一个作为另一个的逆。但通常更简单的方法是间接地通过定义箭头之间的双射来实现，无论是作用于这两个对象的箭头，还是从这两个对象发出的箭头。

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



换句话说，对于每个 x ，存在一个 **hom**-集的映射：

$$\alpha_x : \mathcal{C}(x, a) \rightarrow \mathcal{C}(x, b)$$

当我们变化 x 时，这两个 **hom**-集成为两个（逆变）函子， $\mathcal{C}(-, a)$ 和 $\mathcal{C}(-, b)$ ，而 α 可以看作它们之间的映射。这样的函子映射，称为变换，实际上是一系列单独的映射 α_x ，每个对象 x 在范畴 \mathcal{C} 中都有一个。

函子 $\mathcal{C}(-, a)$ 描述了世界看待 a 的方式, 而函子 $\mathcal{C}(-, b)$ 描述了世界看待 b 的方式。

变换 α 在这两种视图之间来回切换。 α 的每个组成部分, 即双射 α_x , 表明从 x 看 a 的视图与从 x 看 b 的视图是同构的。

我们之前讨论的自然性条件是:

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

它关联了在不同对象处取的 α 的组成部分。换句话说, 它关联了两个不同观察者 x 和 y 的视图, 这两个观察者通过一个箭头 $g: y \rightarrow x$ 连接。

这个等式的两边都作用于 hom -集 $\mathcal{C}(x, a)$ 。结果在 hom -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 $g: y \rightarrow x$ 的预组合也是 hom -集的映射。事实上, 它是通过逆变 hom -函子提升的 g 。我们可以分别将其写为 $\mathcal{C}(g, a)$ 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:

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

它可以通过以下交换图来说明:

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

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

我们可以对出射箭头进行完全相同的推理。这次我们从一个变换 β 开始, 其组成部分为:

$$\beta_x: \mathcal{C}(a, x) \rightarrow \mathcal{C}(b, x)$$

两个 (协变) 函子 $\mathcal{C}(a, -)$ 和 $\mathcal{C}(b, -)$ 分别描述了从 a 和 b 的角度看待世界的方式。可逆变换 β 告诉我们这两种视图是等价的, 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表现良好。

以下是说明自然性条件的交换图：

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

同样，这样的可逆自然变换 β 建立了 a 和 b 之间的同构。

9.2 函子间的自然变换

上一节中的两个 hom-函子为

$$Fx = \mathcal{C}(a, x)$$

$$Gx = \mathcal{C}(b, x)$$

它们都将范畴 \mathcal{C} 映射到 **Set**，因为 hom-集存在于 **Set** 中。我们可以说它们在 **Set** 内创建了 \mathcal{C} 的两个不同模型。

自然变换是这两个模型之间的结构保持映射。

$$\begin{array}{ccc} & & \mathcal{C}(a, x) \\ & \nearrow \mathcal{C}(a, -) & \downarrow \beta_x \\ x & & \\ & \searrow \mathcal{C}(b, -) & \downarrow \\ & & \mathcal{C}(b, x) \end{array}$$

这一思想自然地扩展到任意一对范畴之间的函子。任意两个函子

$$F: \mathcal{C} \rightarrow \mathcal{D}$$

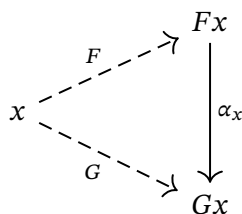
$$G: \mathcal{C} \rightarrow \mathcal{D}$$

都可以看作 \mathcal{C} 在 \mathcal{D} 内的两个不同模型。

为了将一个模型转换为另一个模型，我们使用 \mathcal{D} 中的箭头连接对应的点。对于 \mathcal{C} 中的每个对象 x ，我们选择一个从 Fx 到 Gx 的箭头：

$$\alpha_x: Fx \rightarrow Gx$$

因此，自然变换将对象映射为箭头。



然而，模型的结构不仅与对象有关，还与箭头有关，因此让我们看看箭头会发生什么。对于 \mathcal{C} 中的每个箭头 $f: x \rightarrow y$ ，我们在 \mathcal{D} 中有两个对应的箭头：

$$Ff: Fx \rightarrow Fy$$

$$Gf: Gx \rightarrow Gy$$

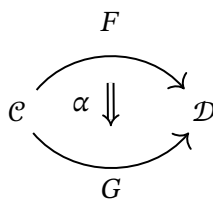
这是 f 的两个提升。我们可以用它们在两个模型的边界内移动。然后还有 α 的分量，它们让我们在模型之间切换。

自然性意味着，无论你是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。这由交换的自然性方块说明：

$$\begin{array}{ccc} Fx & \xrightarrow{Ff} & Fy \\ \downarrow \alpha_x & & \downarrow \alpha_y \\ Gx & \xrightarrow{Gf} & Gy \end{array}$$

满足自然性条件的这样一族箭头 α_x 被称为自然变换。

这是一个展示一对范畴、它们之间的两个函子以及函子之间的自然变换 α 的图表：



由于对于 \mathcal{C} 中的每个箭头都有一个对应的自然性方块，我们可以说自然变换将对象映射为箭头，将箭头映射为交换方块。

如果自然变换的每个分量 α_x 都是同构，则 α 被称为自然同构。

我们现在可以重新表述关于同构的主要结果：两个对象是同构的，当且仅当它们的 **hom**-函子之间存在自然同构（无论是协变还是反变函子——任何一个都可以）。

自然变换提供了一种非常方便的高层次方式来表达各种情况下的交换条件。我们将利用这种能力重新表述代数数据类型的定义。

9.3 编程中的自然变换

自然变换是由对象参数化的一族箭头。在编程中，这对应于由类型参数化的一族函数，即多态函数。

自然变换的参数类型由一个函子描述，返回类型由另一个函子描述。

在 Haskell 中，我们可以定义一个数据类型，它接受两个表示两个函子的类型构造器，并生成自然变换的类型：

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

`forall`量词告诉编译器该函数是多态的——即，它为每个类型`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。

由`forall`定义的类型非常特殊。它们在参数多态的意义上是多态的。这意味着一个公式适用于所有类型。我们已经看到了恒等函数的例子，它可以写成：

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

这个函数的主体非常简单，只是变量`x`。无论`x`是什么类型，公式都保持不变。

这与特设多态形成对比。特设多态函数可能为不同类型使用不同的实现。这种函数的一个例子是`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`有另一个实现，依此类推，逐个案例。

Haskell 中（参数）自然变换的标准定义使用类型同义词：

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

`type`声明为右侧引入了一个别名，一个简写。

事实证明，将自然变换的类型限制为参数多态具有深远的影响。这样的函数自动满足自然性条件。这是参数性产生所谓免费定理的一个例子。

我们无法在 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，编译器会自动确定使用哪个版本的`fmap`和`alpha`的哪些组件。

我们还可以使用更高级的语言选项来明确选择。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`的组件，注释`@f`和`@g`指定了特设多态`fmap`实例化的函子。

这是一个有用的函数的例子，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。)

另一个例子是函数`reverse`，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的想法上的。你可以对容器做两件完全正交的事情：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。或者你可以将数据转移到另一个容器中，而不修改它。这就是自然变换所做的：它是一种在容器之间移动“东西”的过程，而不知道“东西”是什么类型。

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。它以对内容类型不可知的方式进行，这意味着它不能检查、创建或修改内容。它所能做的就是将其移动到新位置，或丢弃它。

自然性条件强制了这两个操作的正交性。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。

这是将复杂问题成功分解为一系列简单问题的另一个例子。不过，请记住，并非所有涉及数据容器的操作都可以以这种方式分解。例如，过滤需要检查数据，以及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或常函子作为源或目标。例如，多态恒等函数可以被视为两个恒等函子之间的自然变换。

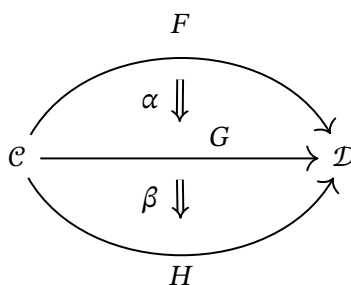
自然变换的垂直复合

自然变换只能在平行函子之间定义，即共享相同源类别和相同目标类别的函子。这样的平行函子形成一个函子类别。两个类别 \mathcal{C} 和 \mathcal{D} 之间的函子类别的标准表示法是 $[\mathcal{C}, \mathcal{D}]$ 。你只需将两个类别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类别，我们必须定义自然变换的复合。如果我们记住自然变换的组件是目标类别中的常规箭头，这就很容易了。这些箭头可以复合。

确实，假设我们有一个在两个函子 F 和 G 之间的自然变换 α 。我们想将其与另一个从 G 到 H 的自然变换 β 复合。



让我们看看这些变换在某个对象 x 处的组件

$$\alpha_x : Fx \rightarrow Gx$$

$$\beta_x : Gx \rightarrow Hx$$

这些只是 \mathcal{D} 中的两个可复合的箭头。所以我们可以定义一个复合自然变换 γ 如下：

$$\gamma : F \rightarrow H$$

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

这被称为自然变换的垂直复合。你会看到它用点 $\gamma = \beta \cdot \alpha$ 或简单的并置 $\gamma = \beta\alpha$ 表示。

γ 的自然性条件可以通过将 α 和 β 的两个自然性方块垂直粘贴在一起来展示：

$$\begin{array}{ccc}
 Fx & \xrightarrow{Ff} & Fy \\
 \downarrow \alpha_x & & \downarrow \alpha_y \\
 Gx & \xrightarrow{Gf} & Gy \\
 \downarrow \beta_x & & \downarrow \beta_y \\
 Hx & \xrightarrow{Hf} & Hy
 \end{array}
 \begin{array}{c}
 \gamma_x \curvearrowright \\
 \gamma_y \curvearrowleft
 \end{array}$$

在 **Haskell** 中，自然变换的垂直复合只是应用于多态函数的常规函数复合。使用自然变换在容器之间移动项目的直觉，垂直复合将两个这样的移动一个接一个地组合起来。

函子范畴

由于自然变换的复合是通过箭头的复合来定义的，因此它自动满足结合律。

对于每个函子 F ，还存在一个恒等自然变换 id_F 。它在 x 处的分量是对象 Fx 上的通常恒等箭头：

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

总结来说，对于每一对范畴 \mathcal{C} 和 \mathcal{D} ，存在一个函子范畴 $[\mathcal{C}, \mathcal{D}]$ ，其中自然变换作为箭头。

该范畴中的 **hom-set** 是两个函子 F 和 G 之间的自然变换的集合。按照标准的符号约定，我们将其写为：

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

其中范畴的名称后跟括号中的两个对象（此处为函子）的名称。

在范畴论中，对象和箭头的表示方式不同。对象用点表示，箭头用带尖的线表示。

在 **Cat**（范畴的范畴）中，函子被表示为箭头。但在函子范畴 $[\mathcal{C}, \mathcal{D}]$ 中，函子被表示为点，而自然变换被表示为箭头。

在一个范畴中的箭头，在另一个范畴中可能成为对象。

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。

自然变换的水平复合

自然变换的第二种复合方式是由函子的复合诱导的。假设我们有一对可复合的函子

$$F: \mathcal{C} \rightarrow \mathcal{D}$$

$$G: \mathcal{D} \rightarrow \mathcal{E}$$

同时，还有另一对可复合的函子：

$$F' : \mathcal{C} \rightarrow \mathcal{D}$$

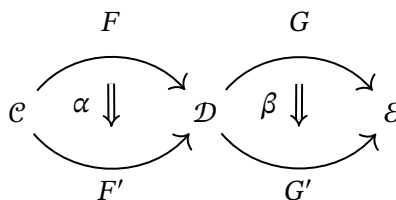
$$G' : \mathcal{D} \rightarrow \mathcal{E}$$

我们还有两个自然变换：

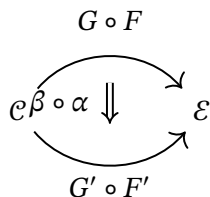
$$\alpha : F \rightarrow F'$$

$$\beta : G \rightarrow G'$$

图示如下：



水平复合 $\beta \circ \alpha$ 将 $G \circ F$ 映射到 $G' \circ F'$ 。



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

$$(\beta \circ \alpha)_x : G(Fx) \rightarrow G'(F'x)$$

我们可以使用 α 将 x 映射到一个箭头

$$\alpha_x : Fx \rightarrow F'x$$

我们可以使用 G 将这个箭头提升

$$G(\alpha_x) : G(Fx) \rightarrow G(F'x)$$

为了从那里到达 $G'(F'x)$ ，我们可以使用 β 的适当分量

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

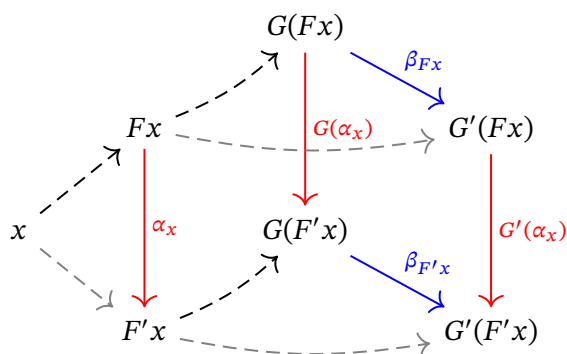
总的来说，我们有

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

但还有另一个同样合理的候选：

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

幸运的是，由于 β 的自然性，它们是相等的。



$\beta \circ \alpha$ 的自然性证明留给热心的读者作为练习。

我们可以直接将其翻译到 **Haskell** 中。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
```

它们的水平复合具有以下类型签名：

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

它有两个等价的实现。第一个是：

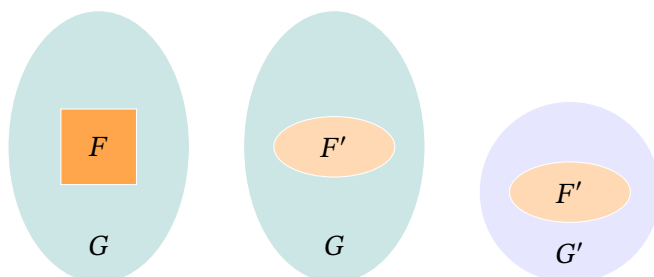
```
beta_alpha = beta . fmap alpha
```

编译器会自动选择 `fmap` 的正确版本，即函子 **G** 的版本。第二个实现是：

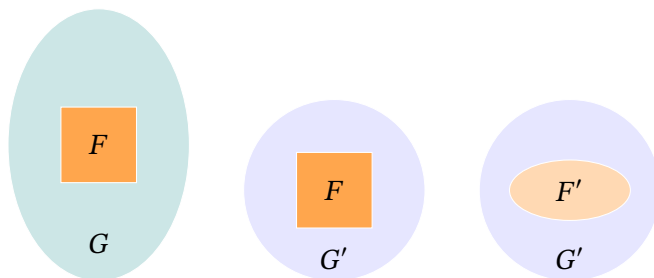
```
beta_alpha = fmap alpha . beta
```

在这里，编译器将选择函子 **G'** 的 `fmap` 版本。

水平复合的直观理解是什么？我们之前看到，自然变换可以看作是在两个容器——函子之间重新包装数据。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，其中填充了由 **F** 描述的每个内部容器。我们有两个自然变换，`alpha` 用于将 **F** 的内容转移到 **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` 重新包装所有内部容器，然后使用 `beta` 重新包装外部容器。



或者我们可以首先使用 `beta` 重新包装外部容器，然后应用 `fmap alpha` 重新包装所有内部容器。最终结果是相同的。



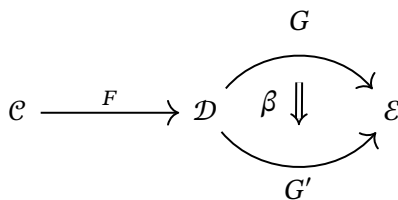
Exercise 9.3.2. 实现 `safeHead` 在 `reverse` 之后的水平复合的两个版本。比较它们在不同参数上的作用结果。

Exercise 9.3.3. 对 `reverse` 在 `safeHead` 之后的水平复合做同样的操作。

Whiskering (whiskering)

在水平组合中，经常会出现其中一个自然变换是恒等变换的情况。对于这种组合，有一种简写表示法。例如， $\beta \circ id_F$ 可以写成 $\beta \circ F$ 。

由于这种组合的图形特征形状，它被称为“whiskering”。



在分量形式中，我们有：

$$(\beta \circ F)_x = \beta_{Fx}$$

让我们考虑如何将其翻译到 Haskell 中。自然变换是一个多态函数。由于参数化，它对于所有类型都由相同的公式定义。因此，右侧的 whiskering 不会改变公式，而是改变函数签名。

例如，如果这是 `beta` 的声明：

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

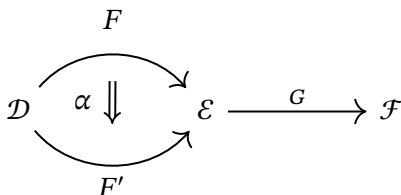
那么它的 whiskered 版本将是：

```
beta_f :: forall x. G (F x) -> G' (F x)
beta_f = beta
```

由于 Haskell 的类型推断，这种转换是隐式的。当调用多态函数时，我们不需要指定执行自然变换的哪个分量——类型检查器通过查看参数的类型来推断它。

在这种情况下，直觉是我们在重新包装外部容器，同时保持内部容器不变。

类似地， $id_G \circ \alpha$ 可以写成 $G \circ \alpha$ 。



在分量形式中：

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

在 Haskell 中， α_x 通过 G 的提升是使用 `fmap` 完成的，因此给定：

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

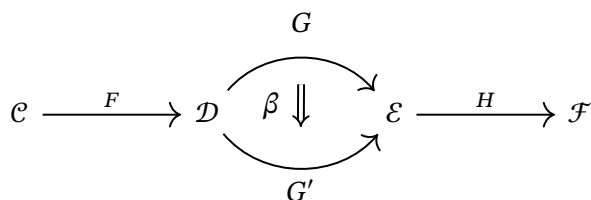
whiskered 版本将是：

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

同样，Haskell 的类型推断引擎会确定使用哪个版本的 `fmap`（在这里，它是 G 的 `Functor` 实例中的那个）。

直觉是我们在重新包装内部容器的内容，同时保持外部容器不变。

最后，在许多应用中，自然变换在两侧都被 whiskered：



在分量形式中，我们有：

$$(H \circ \beta \circ F)x = H(\beta_{Fx})$$

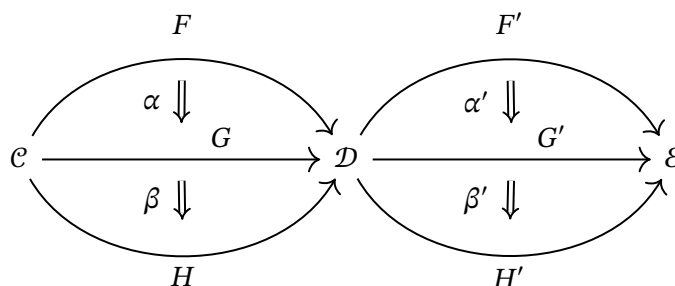
在 Haskell 中：

```
h_beta_f :: forall x. H (G (F x)) -> H (G' (F x))
h_beta_f = fmap beta
```

这里的直觉是，我们有一个三层的容器；我们正在重新排列中间的那一层，同时保持外部容器和所有内部容器不变。

交换律

我们可以将垂直复合与水平复合结合起来，如下图所示：



交换律表明复合的顺序无关紧要：我们可以先进行垂直复合再进行水平复合，或者先进行水平复合再进行垂直复合。

9.4 重新审视万有构造

老子曰：大道至简。

我们已经见过和、积、指数、自然数和列表的定义。

定义这些数据类型的传统方法是探究其内部结构。这是集合论的方式：我们观察新集合的元素是如何从旧集合的元素构造出来的。和的元素要么是第一个集合的元素，要么是第二个集合的元素。积的元素是一对元素。依此类推。我们是从工程的角度来看待对象。

在范畴论中，我们采取相反的方法。我们对对象内部是什么或如何实现不感兴趣。我们感兴趣的是对象的目的、如何使用它以及它如何与其他对象交互。我们是从功利的角度来对待对象。

这两种方法各有优势。范畴论的方法出现较晚，因为你需要研究很多例子才能看到清晰的模式。但一旦你看到了这些模式，你就会发现事物之间意想不到的联系，比如和与积的对偶性。

通过对象之间的联系来定义特定对象，需要考虑可能与它们交互的无限多个对象。

“告诉我你与宇宙的关系，我就能告诉你你是谁。”

通过对象在范畴中所有对象的映射出或映射进来定义对象，称为万有构造。

为什么自然变换如此重要？因为大多数范畴构造都涉及交换图。如果我们可以将这些图重新表述为自然性方块，我们就在抽象的阶梯上提升了一个层次，并获得了新的有价值的见解。

能够将大量事实压缩成简洁优雅公式，有助于我们看到新的模式。例如，我们将看到，**hom**-集之间的自然同构在范畴论中无处不在，并最终引出了伴随的概念。

但首先，我们将更详细地研究几个例子，以理解范畴论的简洁语言。例如，我们将尝试解码以下陈述：两个对象的和或余积由以下自然同构定义：

$$[2, \mathcal{C}](D, \Delta_x) \cong \mathcal{C}(a + b, x)$$

选择对象

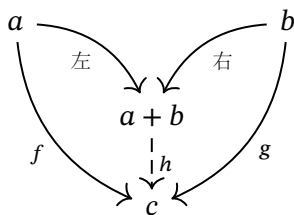
即使在范畴论中，像指向对象这样简单的任务也有特殊的解释。我们已经看到，指向集合中的一个元素等价于从单例集合到它的函数的选择。类似地，在范畴中选择一个对象等价于从单对象范畴中选择一个函子。或者，可以使用来自另一个范畴的常函子来完成。

很多时候，我们希望选择一对对象。这也可以通过从两对象的简笔图范畴中选择一个函子来实现。类似地，选择一个箭头等价于从“行走箭头”范畴中选择一个函子，依此类推。

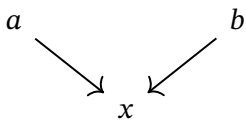
通过明智地选择我们的函子和它们之间的自然变换，我们可以重新表述我们迄今为止见过的所有万有构造。

余跨度作为自然变换

和的定义需要选择两个要相加的对象；以及第三个对象作为映射出的目标。



这个图可以进一步分解为两个更简单的形状，称为余跨度：



要构造一个余跨度，我们首先需要选择一对对象。为此，我们将从一个两对象范畴 **2** 开始。我们将其对象称为 **1** 和 **2**。我们将使用一个函子

$$D: \mathbf{2} \rightarrow \mathcal{C}$$

来选择对象 a 和 b ：

$$D 1 = a$$

$$D 2 = b$$

(D 代表“图”，因为这两个对象在 \mathcal{C} 中形成了一个非常简单的图，由两个点组成。)

我们将使用常函子

$$\Delta_x : \mathbf{2} \rightarrow \mathcal{C}$$

来选择对象 x 。这个函子将 1 和 2 都映射到 x （并将两个恒等箭头映射到 id_x ）。

由于这两个函子都从 $\mathbf{2}$ 到 \mathcal{C} ，我们可以定义它们之间的自然变换 α 。在这种情况下，它只是一对箭头：

$$\alpha_1 : D 1 \rightarrow \Delta_x 1$$

$$\alpha_2 : D 2 \rightarrow \Delta_x 2$$

这些正是余跨度中的两个箭头。

α 的自然性条件是平凡的，因为 $\mathbf{2}$ 中没有箭头（除了恒等箭头）。

可能有许多余跨度共享相同的三个对象——这意味着：可能有许多自然变换在函子 D 和 Δ_x 之间。这些自然变换形成了函子范畴 $[\mathbf{2}, \mathcal{C}]$ 中的一个 hom-集，即：

$$[\mathbf{2}, \mathcal{C}](D, \Delta_x)$$

余跨度的函子性

让我们考虑当我们在余跨度中开始变化对象 x 时会发生什么。我们有一个映射 F ，它将 x 映射到 x 上的余跨度集：

$$Fx = [\mathbf{2}, \mathcal{C}](D, \Delta_x)$$

这个映射在 x 上是函子性的。

要看到这一点，考虑一个箭头 $m : x \rightarrow y$ 。这个箭头的提升是两个自然变换集之间的映射：

$$[\mathbf{2}, \mathcal{C}](D, \Delta_x) \rightarrow [\mathbf{2}, \mathcal{C}](D, \Delta_y)$$

这可能看起来非常抽象，直到你记住自然变换有分量，而这些分量只是常规箭头。左边的一个元素是一个自然变换：

$$\mu : D \rightarrow \Delta_x$$

它有两个分量，对应于 $\mathbf{2}$ 中的两个对象。例如，我们有

$$\mu_1 : D 1 \rightarrow \Delta_x 1$$

或者，使用 D 和 Δ 的定义：

$$\mu_1 : a \rightarrow x$$

这只是我们余跨度的左腿。

类似地，右边的一个元素是一个自然变换：

$$\nu: D \rightarrow \Delta_y$$

它在 1 处的分量是一个箭头

$$\nu_1: a \rightarrow y$$

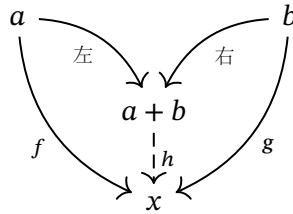
我们可以通过将 μ_1 与 $m: x \rightarrow y$ 后复合来从 μ_1 得到 ν_1 。因此， m 的提升是分量逐个分量的后复合 ($m \circ -$)：

$$\nu_1 = m \circ \mu_1$$

$$\nu_2 = m \circ \mu_2$$

和作为万有余跨度

在你可以构建在 a 和 b 上的所有余跨度中，具有我们称为左和右的箭头汇聚在 $a + b$ 上的那个非常特殊。从它到任何其他余跨度都有一个唯一的映射——一个使两个三角形交换的映射。



我们现在可以将这个条件转化为关于自然变换和 hom-集的陈述。箭头 h 是 hom-集

$$\mathcal{C}(a + b, x)$$

中的一个元素。 x 上的余跨度是一个自然变换，即函子范畴中的 hom-集的一个元素：

$$[\mathbf{2}, \mathcal{C}](D, \Delta_x)$$

两者都是各自范畴中的 hom-集。两者都只是集合，即范畴 **Set** 中的对象。这个范畴在函子范畴 $[\mathbf{2}, \mathcal{C}]$ 和“常规”范畴 \mathcal{C} 之间架起了一座桥梁，尽管在概念上它们似乎处于非常不同的抽象层次。

借用西格蒙德·弗洛伊德的话，“有时候，集合就是集合。”

我们的万有构造是集合的双射或同构：

$$[\mathbf{2}, \mathcal{C}](D, \Delta_x) \cong \mathcal{C}(a + b, x)$$

此外，如果我们变化对象 x ，这两边表现得像从 \mathcal{C} 到 **Set** 的函子。因此，询问这个函子映射是否是自然同构是有意义的。

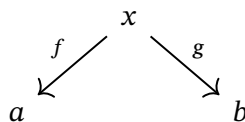
事实上，可以证明这个同构的自然性条件转化为和定义中三角形的交换条件。因此，和的定义可以用一个单一的方程来替代。

积作为万有跨度

关于积的万有构造，可以提出类似的论点。再次，我们从简笔图范畴 **2** 和函子 D 开始。但这次我们使用一个自然变换，方向相反

$$\alpha: \Delta_x \rightarrow D$$

这样的自然变换是一对箭头，形成一个跨度：



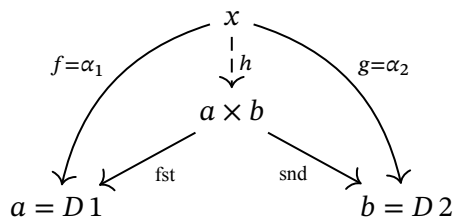
这些自然变换共同形成了函子范畴中的一个 hom-集：

$$[2, \mathcal{C}](\Delta_x, D)$$

这个 hom-集的每个元素都与一个唯一的映射 h 到积 $a \times b$ 一一对应。这样的映射是 hom-集 $\mathcal{C}(x, a \times b)$ 的一个成员。这种对应关系表示为同构：

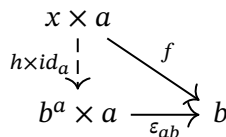
$$[2, \mathcal{C}](\Delta_x, D) \cong \mathcal{C}(x, a \times b)$$

可以证明，这个同构的自然性保证了图中三角形的交换：



指数

指数或函数对象由以下交换图定义：



这里， f 是 hom-集 $\mathcal{C}(x \times a, b)$ 的一个元素， h 是 $\mathcal{C}(x, b^a)$ 的一个元素。

这些集合之间的同构，在 x 上是自然的，定义了指数量对象。

$$\mathcal{C}(x \times a, b) \cong \mathcal{C}(x, b^a)$$

上图中的 f 是左边的一个元素， h 是右边对应的元素。变换 α_x （它也依赖于 a 和 b ）将 f 映射到 h 。

$$\alpha_x: \mathcal{C}(x \times a, b) \rightarrow \mathcal{C}(x, b^a)$$

在 Haskell 中，我们称之为 `curry`。它的逆 α^{-1} 被称为 `uncurry`。

与前面的例子不同，这里两个 hom-集都在同一个范畴中，因此更容易详细分析同构。特别是，我们希望看到交换条件：

$$f = \varepsilon_{ab} \circ (h \times id_a)$$

是如何从自然性中产生的。

标准的 Yoneda 技巧是对 x 进行替换，将其中一个 hom-集简化为自 hom-集，即源与目标相同的 hom-集。这将允许我们选择该 hom-集的一个规范元素，即恒等箭头。

在我们的情况下，将 b^a 替换为 x 将允许我们选择 $h = id_{(b^a)}$ 。

$$\begin{array}{ccc} b^a \times a & & \\ \downarrow id_{(b^a)} \times id_a & \searrow f & \\ b^a \times a & \xrightarrow{\varepsilon_{ab}} & b \end{array}$$

在这种情况下，交换条件告诉我们 $f = \varepsilon_{ab}$ 。换句话说，我们得到了 ε_{ab} 的公式，用 α 表示：

$$\varepsilon_{ab} = \alpha_x^{-1}(id_x)$$

其中 x 等于 b^a 。

由于我们认识到 α^{-1} 是 `uncurry`，而 ε 是函数应用，我们可以在 Haskell 中将其写为：

```
apply :: (a -> b, a) -> b
apply = uncurry id
```

这可能一开始令人惊讶，直到你意识到 $(a \rightarrow b, a) \rightarrow b$ 的柯里化导致 $(a \rightarrow b) \rightarrow (a \rightarrow b)$ 。

我们还可以将主同构的两边编码为 Haskell 函子：

```
data LeftFunctor a b x = LF ((x, a) -> b)

data RightFunctor a b x = RF (x -> (a -> b))
```

它们在类型变量 x 中都是逆变函子。

```
instance Contravariant (LeftFunctor a b) where
  contramap g (LF f) = LF (f . bimap g id)
```

这表示 $g: x \rightarrow y$ 的提升由以下前复合给出：

$$\mathcal{C}(y \times a, b) \xrightarrow{(- \circ (g \times id_a))} \mathcal{C}(x \times a, b)$$

类似地：

```
instance Contravariant (RightFunctor a b) where
  contramap g (RF h) = RF (h . g)
```

翻译为：

$$\mathcal{C}(y, b^a) \xrightarrow{(- \circ g)} \mathcal{C}(x, b^a)$$

自然变换 α 只是 `curry` 的薄封装；它的逆是 `uncurry`：

```
alpha :: forall a b x. LeftFunctor a b x -> RightFunctor a b x
alpha (LF f) = RF (curry f)
```

```
alpha_1 :: forall a b x. RightFunctor a b x -> LeftFunctor a b x
alpha_1 (RF h) = LF (uncurry h)
```

使用 $g: x \rightarrow y$ 的提升的两个公式，这里是自然性方块：

$$\begin{array}{ccc} \mathcal{C}(y \times a, b) & \xrightarrow{(- \circ (g \times id_a))} & \mathcal{C}(x \times a, b) \\ \downarrow \alpha_y & & \downarrow \alpha_x \\ \mathcal{C}(y, b^a) & \xrightarrow{(- \circ g)} & \mathcal{C}(x, b^a) \end{array}$$

现在让我们对其应用 Yoneda 技巧，并将 y 替换为 b^a 。这也允许我们将 g ——现在从 x 到 b^a ——替换为 h 。

$$\begin{array}{ccc} \mathcal{C}(b^a \times a, b) & \xrightarrow{(- \circ (h \times id_a))} & \mathcal{C}(x \times a, b) \\ \downarrow \alpha_{(b^a)} & & \downarrow \alpha_x \\ \mathcal{C}(b^a, b^a) & \xrightarrow{(- \circ h)} & \mathcal{C}(x, b^a) \end{array}$$

我们知道 \mathbf{hom} -集 $\mathcal{C}(b^a, b^a)$ 至少包含恒等箭头，因此我们可以在左下角选择元素 $id_{(b^a)}$ 。

反转左边的箭头，我们知道 α^{-1} 作用于恒等产生 ε_{ab} 在左上角（这是 `uncurry id` 技巧）。

h 的前复合作用于恒等产生 h 在右下角。

α^{-1} 作用于 h 产生 f 在右上角。

$$\begin{array}{ccc} \varepsilon_{ab} & \xrightarrow{(- \circ (h \times id_a))} & f \\ \alpha^{-1} \uparrow & & \uparrow \alpha^{-1} \\ id_{(b^a)} & \xrightarrow{(- \circ h)} & h \end{array}$$

(\mapsto 箭头表示函数对集合元素的作用。)

因此，在左下角选择 $id_{(ba)}$ 固定了其他三个角。特别是，我们可以看到上箭头应用于 ε_{ab} 产生 f ，这正是我们想要推导的交换条件：

$$\varepsilon_{ab} \circ (h \times id_a) = f$$

9.5 极限与余极限

在前一节中，我们使用自然变换定义了和与积。这些变换是在由非常简单的棒图范畴 **2** 定义的图之间进行的，其中一个函子是常函子。

没有什么能阻止我们用更复杂的范畴替换 **2**。例如，我们可以尝试在对象之间具有非平凡箭头的范畴，或者具有无限多个对象的范畴。

围绕这些构造有一整套词汇。

我们使用范畴 **2** 中的对象来索引范畴 \mathcal{C} 中的对象。我们可以用任意的索引范畴 \mathcal{J} 替换 **2**。 \mathcal{C} 中的图仍然定义为函子 $D: \mathcal{J} \rightarrow \mathcal{C}$ 。它选择 \mathcal{C} 中的对象，但也选择它们之间的箭头。

作为第二个函子，我们仍然使用常函子 $\Delta_x: \mathcal{J} \rightarrow \mathcal{C}$ 。

自然变换，即 hom-集

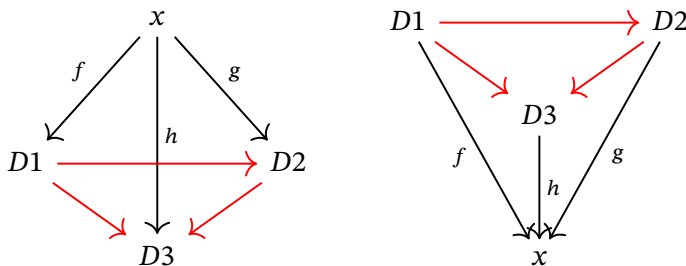
$$[\mathcal{J}, \mathcal{C}](\Delta_x, D)$$

的元素，现在称为一个锥。它的对偶，即

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x)$$

的元素，称为一个余锥。它们分别推广了跨度和余跨度。

在图中，锥和余锥看起来像这样：



由于索引范畴现在可能包含箭头，这些图的自然性条件不再平凡。常函子 Δ_x 将所有顶点收缩为一个，因此自然性方块收缩为三角形。自然性意味着所有以 x 为顶点的三角形现在必须交换。

如果存在，通用锥称为图 D 的极限，并记为 $\text{Lim}D$ 。通用性意味着它满足以下同构，对于 x 是自然的：

$$[\mathcal{J}, \mathcal{C}](\Delta_x, D) \cong \mathcal{C}(x, \text{Lim}D)$$

对于每个以 x 为顶点的锥，存在一个从 x 到极限 $\text{Lim}D$ 的唯一映射。

Set 值函子的极限有一个特别简单的特征。它是一组以单例集为顶点的锥。实际上，极限的元素，即从单例集到它的函数，与这些锥一一对应：

$$[\mathcal{J}, \mathcal{C}](\Delta_1, D) \cong \mathcal{C}(1, \text{Lim} D)$$

对偶地，通用余锥称为余极限，并由以下自然同构描述：

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x) \cong \mathcal{C}(\text{Colim} D, x)$$

我们现在可以说，积是来自索引范畴 **2** 的图的极限（和是余极限）。

极限和余极限提炼了模式的本质。

极限，如积，由其映射入属性定义。余极限，如和，由其映射出属性定义。

有许多有趣的极限和余极限，我们将在讨论代数和余代数时看到一些。

Exercise 9.5.1. 证明“行走箭头”范畴的极限，即具有连接两个对象的箭头的两对象范畴，具有与图中第一个对象相同的元素（“元素”是从终端对象出发的箭头）。

等化子

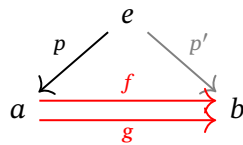
许多高中数学涉及学习如何解方程或方程组。方程将两种不同方式产生的结果等同起来。如果我们允许减去某些东西，我们通常将所有东西移到一边，并将问题简化为计算某个表达式的零点。在几何中，同样的思想表示为两个几何对象的交集。

在范畴论中，所有这些模式都体现在一个称为等化子的单一构造中。等化子是一个图的极限，其模式由具有两个平行箭头的棒图范畴给出：

$$i \rightrightarrows j$$

这两个箭头表示两种产生某物的方式。

从这个范畴的函子选择目标范畴中的一对对象和一对态射。这个图的极限体现了两个结果的交集。它是一个对象 e 和两个箭头 $p: e \rightarrow a$ 和 $p': e \rightarrow b$ 。



我们有两个交换条件：

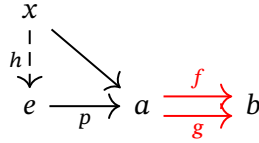
$$p' = f \circ p$$

$$p' = g \circ p$$

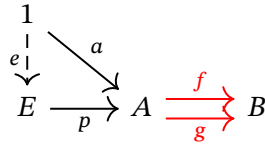
这意味着 p' 由其中一个方程完全确定，而另一个方程转化为约束：

$$f \circ p = g \circ p$$

由于等化子是极限，它是通用的这样一对，如图所示：



为了发展对等化子的直觉，考虑它在集合中的工作方式是有意义的。通常，技巧是将 x 替换为单例集 1 ：

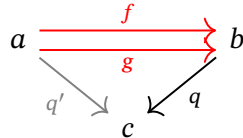


在这种情况下， a 是 A 的一个元素，使得 $fa = ga$ 。这只是说 a 是由一对函数定义的方程的解。通用性意味着存在一个唯一的元素 e 属于 E ，使得 $p \circ e = a$ 。换句话说， E 的元素与方程组的解一一对应。

余等化子

与等同或交集对偶的概念是什么？它是发现共性并将事物组织到桶中的过程。例如，我们可以将整数分配到偶数和奇数桶中。在范畴论中，这种桶化过程由余等化子描述。

余等化子用于定义等化子的相同图的余极限：



这一次，箭头 q' 由 q 完全确定；并且 q 必须满足方程：

$$q \circ f = q \circ g$$

再次，我们可以通过考虑两个函数在集合上的余等化子来获得一些直觉。

$$A \begin{array}{c} \xrightarrow{f} \\ \xrightarrow{g} \end{array} B \xrightarrow{q} C$$

一个 $x \in A$ 被映射到 B 中的两个元素 fx 和 gx ，但然后 q 必须将它们映射回 C 中的一个元素。这个元素代表桶。

通用性意味着 C 是 B 的一个副本，其中从同一个 x 产生的元素已被识别。

考虑一个例子，其中 A 是一对整数的集合 (m, n) ，使得它们要么都是偶数，要么都是奇数。我们想要余等化两个函数，即两个投影 (fst, snd) 。等化子集 C 将有两个元素对应于两个桶。我们将它表示为 **Bool**。等化函数 q 选择桶：

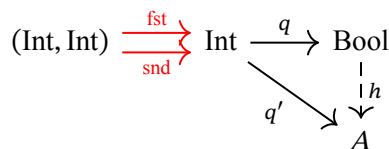
```
q :: Int -> Bool
q n = n `mod` 2 == 0
```

任何函数 q' 如果不能区分我们对的组件，并且只对它们的奇偶性敏感：

$$q' \circ \text{fst} = q' \circ \text{snd}$$

可以通过函数 h 唯一地分解：

```
h :: (Int -> a) -> Bool -> a
h q' True  = q' 0
h q' False = q' 1
```



Exercise 9.5.2. 运行一些测试，显示分解 $(h q') \circ q$ 给出与 q' 相同的结果，其中 q' 由以下定义给出：

```
import Data.Bits

q' :: Int -> Bool
q' x = testBit x 0
```

终端对象的存在性

老子说：伟大的行为由小行为组成。

到目前为止，我们一直在研究小图的极限，即来自简单棒图范畴的函子。然而，没有什么能阻止我们定义极限和余极限，其中模式被取为无限范畴。但无限性有等级之分。当范畴中的对象形成一个真集时，我们称这样的范畴为小范畴。不幸的是，最基本的例子，集合的范畴 **Set**，不是小范畴。我们知道没有所有集合的集合。**Set** 是一个大范畴。但至少 **Set** 中的所有 **hom** 集都是集合。我们说 **Set** 是局部小的。在接下来的内容中，我们将始终处理局部小范畴。

小极限是小图的极限，即来自对象和态射形成集合的范畴的函子。所有小极限存在的范畴称为小完备的，或简称为完备的。特别是，在这样的范畴中，任意集的对象积存在。你也可以等化两个对象之间的任意集箭头。如果这样的范畴是局部小的，这意味着所有等化子都存在。

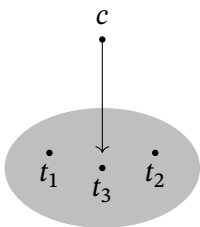
相反，一个（小）余完备范畴具有所有小余极限。特别是，这样的范畴具有所有小余积和余等化子。

范畴 **Set** 既是完备的又是余完备的。

在余完备的局部小范畴中，存在终端对象的一个简单标准：存在弱终端集就足够了。

弱终端对象，就像终端对象一样，有来自任何对象的箭头；只是这样的箭头不一定是唯一的。

弱终端集是由集合 I 索引的对象族 t_i ，使得对于 \mathcal{C} 中的任何对象 c ，存在一个 i 和一个箭头 $c \rightarrow t_i$ 。这样的集也称为解集。



在余完备范畴中，我们总是可以构造一个余积 $\coprod_{i \in I} t_i$ 。这个余积是一个弱终端对象，因为从每个 c 都有一个箭头指向它。这个箭头是从某个 t_i 的箭头后跟注入 $t_i: t_i \rightarrow \coprod_{j \in I} t_j$ 的复合。

给定一个弱终端对象，我们可以构造（强）终端对象。我们首先定义 \mathcal{C} 的一个子范畴 \mathcal{T} ，其对象是 t_i 。 \mathcal{T} 中的态射是 \mathcal{C} 中所有在 \mathcal{T} 的对象之间进行的态射。这称为 \mathcal{C} 的全子范畴。根据我们的构造， \mathcal{T} 是小的。

有一个明显的包含函子 F 将 \mathcal{T} 嵌入 \mathcal{C} 。这个函子在 \mathcal{C} 中定义了一个小图。事实证明，这个图的余极限是 \mathcal{C} 中的终端对象。

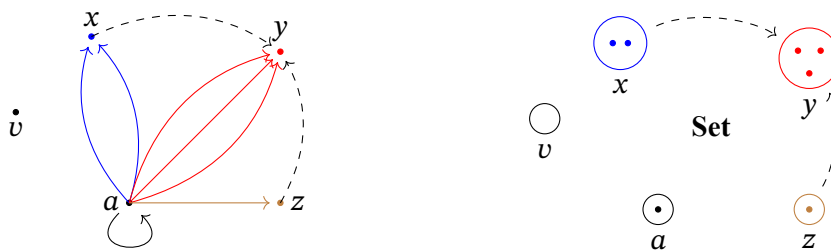
对偶地，类似的构造可以用于将初始对象定义为弱初始集的极限。

解集的这个属性将在证明 Freyd 的伴随函子定理时派上用场。

9.6 米田引理 (Yoneda Lemma)

从某个范畴 \mathcal{C} 到集合范畴的函子可以被视为该范畴在 **Set** 中的模型。一般来说，建模是一个有损的过程：它会丢弃一些信息。一个常值 **Set**-值函子是一个极端的例子：它将整个范畴映射到一个单一的集合及其恒等函数。

同态函子 (hom-functor) 从某个视角生成该范畴的模型。例如，函子 $\mathcal{C}(a, -)$ 提供了从 a 的视角看到的 \mathcal{C} 的全景图。它将所有从 a 出发的箭头组织成整齐的包，这些包通过箭头之间的图像连接起来，所有这些都符合源范畴的原始结构。



有些视角比其他视角更好。例如，从初始对象 (initial object) 的视角看，视图相当稀疏。每个对象 x 都被映射到一个单例集 $\mathcal{C}(0, x)$ ，对应于唯一的映射 $0 \rightarrow x$ 。

从终止对象 (terminal object) 的视角看则更有趣：它将所有对象映射到它们的（全局）元素集 $\mathcal{C}(1, x)$ 。

米田引理可以被视为范畴论中最深刻的陈述之一，或者是最平凡的陈述之一。让我们从深刻的版本开始。

考虑 \mathcal{C} 在 **Set** 中的两个模型。第一个由同态函子 $\mathcal{C}(a, -)$ 给出。这是从 a 的视角看到的 \mathcal{C} 的全景图，非常详细。第二个由某个任意函子 $F: \mathcal{C} \rightarrow \mathbf{Set}$ 给出。它们之间的任何自然变换 (natural transformation) 都将一个模型嵌入到另一个模型中。事实证明，所有这样的自然变换的集合完全由集合 Fa 决定。

由于自然变换的集合是函子范畴 $[\mathcal{C}, \mathbf{Set}]$ 中的同态集，米田引理的形式化陈述如下：

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), F) \cong Fa$$

此外，这个同构在 a 和 F 中都是自然的。

之所以这能成立，是因为该定理中涉及的所有映射都受到保持范畴 \mathcal{C} 结构及其模型结构的约束。特别是，自然性条件对自然变换的组成部分从一个点传播到另一个点的方式施加了大量的约束。

米田引理的证明从一个单一的恒等箭头开始，并让自然性将其传播到整个范畴。

以下是证明的概要。它由两部分组成：首先，给定一个自然变换，我们构造 Fa 的一个元素。其次，给定 Fa 的一个元素，我们构造相应的自然变换。

首先，让我们在米田引理的左侧选择一个任意元素：一个自然变换 α 。它在 x 处的分量是一个函数：

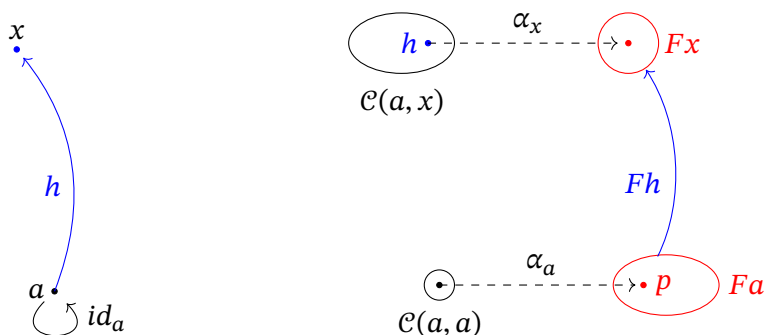
$$\alpha_x: \mathcal{C}(a, x) \rightarrow Fx$$

我们现在可以应用米田技巧：将 a 替换为 x ：

$$\alpha_a: \mathcal{C}(a, a) \rightarrow Fa$$

然后选择恒等 id_a 作为 $\mathcal{C}(a, a)$ 的规范元素。结果是集合 Fa 中的一个元素 $\alpha_a(id_a)$ 。这定义了一个从自然变换到集合 Fa 元素的映射。

现在反过来。给定集合 Fa 的一个元素 p ，我们想要构造一个自然变换 α 。首先，我们将 p 分配为 α_a 在 $id_a \in \mathcal{C}(a, a)$ 上的作用。



现在让我们取一个任意对象 x 和 $\mathcal{C}(a, x)$ 的一个任意元素。后者是一个箭头 $h: a \rightarrow x$ 。我们的自然变换必须将其映射到 Fx 的一个元素。我们可以通过使用 F 提升箭头 h 来实现这一点。我们得到一个函数：

$$Fh: Fa \rightarrow Fx$$

我们可以将这个函数应用于 p 并得到 Fx 的一个元素。我们将这个元素作为 α_x 在 h 上的作用：

$$\alpha_x h = (Fh)p$$

米田引理中的同构在 a 和 F 中都是自然的。后者意味着你可以通过在函子范畴中应用一个箭头（即自然变换）从函子 F “移动”到另一个函子 G 。这是在抽象层次上的一个巨大飞跃，但所有关于函子性和自然性的定义在函子范畴中同样适用，其中对象是函子，箭头是自然变换。

Exercise 9.6.1. 当 Fa 为空时，填补证明中的空白。

Exercise 9.6.2. 证明上述定义的映射

$$\mathcal{C}(a, x) \rightarrow Fx$$

是一个自然变换。提示：使用某个 $f: x \rightarrow y$ 来变化 x 。

Exercise 9.6.3. 证明 α_x 的公式可以从 $\alpha_a(id_a) = p$ 和自然性条件推导出来。提示：同态函子 $\mathcal{C}(a, h)$ 对 h 的提升由后复合给出。

编程中的米田引理

现在来看平凡的部分：米田引理的证明直接翻译为 Haskell 代码。我们从同态函子 $a \rightarrow x$ 和某个函子 f 之间的自然变换类型开始，并证明它等价于 f 作用于 a 的类型。

```
forall x. (a -> x) -> f x.    -- 同构于 (f a)
```

我们使用标准的米田技巧生成类型 $(f\ a)$ 的值

```
yoneda :: Functor f => (forall x. (a -> x) -> f x) -> f a
yoneda g = g id
```

这是逆映射:

```
yoneda_1 :: Functor f => f a -> (forall x. (a -> x) -> f x)
yoneda_1 y = \h -> fmap h y
```

注意, 我们在这里稍微作弊, 混合了类型和集合。当前公式中的米田引理适用于 **Set**-值函子。再次强调, 正确的说法是我们在一个自富集范畴中使用米田引理的富集版本。

米田引理在编程中有一些有趣的应用。例如, 让我们考虑当我们将米田引理应用于恒等函子时会发生什么。我们得到类型 a (恒等函子作用于 a) 与

```
forall x. (a -> x) -> x
```

之间的同构。我们将其解释为: 任何数据类型 a 都可以被一个高阶多态函数替换。这个函数接受另一个函数——称为处理程序、回调或延续——作为参数。

这是标准的延续传递变换 (continuation passing transformation), 在分布式编程中经常使用, 例如当类型 a 的值必须从远程服务器检索时。它也有助于将递归算法转换为尾递归函数的程序变换。

延续传递风格 (continuation-passing style) 难以处理, 因为延续的组合非常复杂, 导致程序员常说的“回调地狱”。幸运的是, 延续形成了一个单子 (monad), 这意味着它们的组合可以隐藏在 `do` 符号后面。

逆变米田引理

通过反转几个箭头, 米田引理也可以应用于逆变函子 (contravariant functor)。它适用于逆变同态函子 $\mathcal{C}(-, a)$ 和逆变函子 F 之间的自然变换:

$$[\mathcal{C}^{op}, \mathbf{Set}](\mathcal{C}(-, a), F) \cong Fa$$

这是 Haskell 中的映射实现:

```
coyoneda :: Contravariant f => (forall x. (x -> a) -> f x) -> f a
coyoneda g = g id
```

这是逆变换:

```
coyonedata_1 :: Contravariant f => f a -> (forall x. (x -> a) -> f x)
coyonedata_1 y = \h -> contramap h y
```

9.7 Yoneda 嵌入

在闭范畴中，我们有指数对象作为 **hom**-集的替代。这显然在 **Set** 中是成立的，因为 **hom**-集本身就是集合，自动成为 **Set** 中的对象。

另一方面，在范畴的范畴 **Cat** 中，**hom**-集是函子的集合，并且它们能否提升为 **Cat** 中的对象——即范畴——并不立即显而易见。但是，正如我们之前所见，它们可以！任意两个范畴之间的函子形成一个函子范畴。

正因为如此，我们可以像对函数进行柯里化一样对函子进行柯里化。来自积范畴的函子可以被视为返回函子的函子。换句话说，**Cat** 是一个闭（对称）幺半范畴。

特别地，我们可以对 **hom**-函子 $\mathcal{C}(a, b)$ 进行柯里化。它是一个 **profunctor**，或者来自积范畴的函子：

$$\mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathbf{Set}$$

但它也是第一个参数 a 上的逆变函子。对于 \mathcal{C}^{op} 中的每个 a ，它产生一个协变函子 $\mathcal{C}(a, -)$ ，这是函子范畴 $[\mathcal{C}, \mathbf{Set}]$ 中的一个对象。我们可以将这个映射写为：

$$\mathcal{C}^{op} \rightarrow [\mathcal{C}, \mathbf{Set}]$$

或者，我们可以固定 b 并产生一个逆变函子 $\mathcal{C}(-, b)$ 。这个映射可以写为：

$$\mathcal{C} \rightarrow [\mathcal{C}^{op}, \mathbf{Set}]$$

这两个映射都是函子性的，这意味着，例如， \mathcal{C} 中的一个箭头被映射到 $[\mathcal{C}^{op}, \mathbf{Set}]$ 中的一个自然变换。

这些 **Set**-值的函子范畴非常常见，以至于它们有特殊的名称。 $[\mathcal{C}^{op}, \mathbf{Set}]$ 中的函子被称为预层，而 $[\mathcal{C}, \mathbf{Set}]$ 中的函子被称为余预层。（这些名称来自代数拓扑。）

让我们将注意力集中在 **hom**-函子的以下解释上：

$$y : \mathcal{C} \rightarrow [\mathcal{C}^{op}, \mathbf{Set}]$$

它取一个对象 x 并将其映射到一个预层：

$$y_x = \mathcal{C}(-, x)$$

这可以可视化从所有可能方向观察 x 的总和。

让我们也回顾一下它对箭头的作用。函子 \mathcal{Y} 将一个箭头 $f: x \rightarrow y$ 提升为预层的映射：

$$\alpha: \mathcal{C}(-, x) \rightarrow \mathcal{C}(-, y)$$

这个自然变换在某个 z 处的分量是 \mathbf{hom} -集之间的函数：

$$\alpha_z: \mathcal{C}(z, x) \rightarrow \mathcal{C}(z, y)$$

它简单地实现为后复合 ($f \circ -$)。注意，这使得 \mathcal{Y}_x 在 x 上是协变的。

函子 \mathcal{Y} 被称为 *Yoneda* 函子。它是一个混合变异的函子（一个 *profunctor*），当我们固定它的第二个参数时，它产生一个预层 $\mathcal{Y}_x: [\mathcal{C}^{op}, \mathbf{Set}]$ 。当我们固定第一个参数时，它产生一个余预层 $\mathcal{Y}^x: [\mathcal{C}, \mathbf{Set}]$ ：

$$\mathcal{Y}^x = \mathcal{C}(x, -)$$

Yoneda 函子 $\mathcal{Y}: \mathcal{C} \rightarrow [\mathcal{C}^{op}, \mathbf{Set}]$ 可以被视为在预层范畴中创建 \mathcal{C} 的模型。但这并不是一个普通的模型——它是一个范畴在另一个范畴中的嵌入。这个特定的嵌入被称为 *Yoneda* 嵌入。

首先， \mathcal{C} 的每个对象都被映射到 $[\mathcal{C}^{op}, \mathbf{Set}]$ 中的一个不同的对象（预层）。我们说它在对象上是单射的。

但这还不是全部： \mathcal{C} 中的每个箭头都被映射到一个不同的箭头。在箭头上是单射的函子被称为忠实的。

如果这还不够， \mathbf{hom} -集的映射也是满射的，这意味着 $[\mathcal{C}^{op}, \mathbf{Set}]$ 中对象之间的每个箭头都来自 \mathcal{C} 中的某个箭头。在箭头上是满射的函子被称为满的。

总的来说，嵌入是完全忠实的，即箭头的映射是一对一的。然而，一般来说，*Yoneda* 嵌入在对象上不是满射的，因此使用“嵌入”这个词。

嵌入是完全忠实的是 *Yoneda* 引理的直接结果。事实上，我们知道，对于任何函子 $F: \mathcal{C}^{op} \rightarrow \mathbf{Set}$ ，我们有一个自然同构：

$$[\mathcal{C}^{op}, \mathbf{Set}](\mathcal{C}(-, x), F) \cong Fx$$

特别地，我们可以用另一个 \mathbf{hom} -函子 $\mathcal{C}(-, y)$ 替换 F 来得到：

$$[\mathcal{C}^{op}, \mathbf{Set}](\mathcal{C}(-, x), \mathcal{C}(-, y)) \cong \mathcal{C}(x, y)$$

左边是预层范畴中的 \mathbf{hom} -集，右边是 \mathcal{C} 中的 \mathbf{hom} -集。它们是同构的，这证明了嵌入是完全忠实的。事实上，*Yoneda* 引理告诉我们，这个同构在 x 和 y 上是自然的。

让我们更仔细地看看这个同构。让我们选取右边集合 $\mathcal{C}(x, y)$ 中的一个元素——一个箭头 $f: x \rightarrow y$ 。同构将其映射到一个自然变换，其在 z 处的分量是一个函数：

$$\mathcal{C}(z, x) \rightarrow \mathcal{C}(z, y)$$

这个映射实现为后复合 ($f \circ -$)。

在 *Haskell* 中，我们会这样写：

```
toNatural :: (x -> y) -> (forall z. (z -> x) -> (z -> y))
toNatural f = \h -> f . h
```

事实上，这种语法也适用：

```
toNatural f = (f . )
```

逆映射是：

```
fromNatural :: (forall z. (z -> x) -> (z -> y)) -> (x -> y)
fromNatural alpha = alpha id
```

(注意再次使用了 Yoneda 技巧。)

这个同构将恒等映射到恒等，将复合映射到复合。这是因为它实现为后复合，而后复合既保留了恒等也保留了复合。我们之前在关于同构的章节中已经展示了这个事实：

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

因为它保留了复合和恒等，这个同构也保留了同构。所以如果 x 与 y 同构，那么预层 $\mathcal{C}(-, x)$ 和 $\mathcal{C}(-, y)$ 是同构的，反之亦然。

这正是我们一直在使用的结果，以证明前面章节中的众多同构。如果 **hom**-集是自然同构的，那么对象是同构的。

Yoneda 嵌入基于这样一个思想：一个对象除了它与其他对象的关系之外别无他物。预层 $\mathcal{C}(-, a)$ 像全息图一样，从整个范畴 \mathcal{C} 的角度编码了 a 的所有视图。Yoneda 嵌入告诉我们，当我们将所有这些单独的全息图组合在一起时，我们得到了整个范畴的完美全息图。

9.8 可表函子

在余预层范畴中，对象是将集合分配给 \mathcal{C} 中对象的函子。其中一些函子通过选择一个参考对象 a ，并将所有对象 x 分配给它们的 **hom** 集 $\mathcal{C}(a, x)$ 来工作：

$$Fx = \mathcal{C}(a, x)$$

这样的函子，以及所有与之同构的函子，被称为可表函子。整个函子由单个对象 a “表示”。

在闭范畴中，将每个对象 x 分配给指数对象 x^a 的元素集合的函子由 a 表示。这是因为 x^a 的元素集合与 $\mathcal{C}(a, x)$ 同构：

$$\mathcal{C}(1, x^a) \cong \mathcal{C}(1 \times a, x) \cong \mathcal{C}(a, x)$$

从这个角度看，表示对象 a 就像函子的对数。

这个类比更深一层：就像乘积的对数是对数的和，乘积数据类型的表示对象是一个和。例如，使用乘积将其参数平方的函子 $Fx = x \times x$ 由 2 表示，即 $1 + 1$ 的和。确实，我们之前已经看到 $x \times x \cong x^2$ 。

可表函子在 **Set** 值函子范畴中扮演着非常特殊的角色。注意，Yoneda 嵌入将 \mathcal{C} 的所有对象映射为可表预层。它将对象 x 映射为由 x 表示的预层：

$$y : x \mapsto \mathcal{C}(-, x)$$

我们可以找到整个范畴 \mathcal{C} ，包括对象和态射，作为可表函子嵌入在预层范畴中。问题是，在预层范畴中“介于”可表函子之间的还有什么？

就像有理数在实数中是稠密的，可表函子在（余）预层中也是“稠密”的。每个实数都可以用有理数来近似。每个预层都是可表函子的余极限（每个余预层是极限）。当我们讨论（余）端时，我们将回到这个话题。

Exercise 9.8.1. 将极限和余极限描述为表示对象。它们表示什么函子？

Exercise 9.8.2. 考虑一个单子函子 $F : \mathcal{C} \rightarrow \mathbf{Set}$ ，它将每个对象 c 分配给一个仅包含该对象的单子集 $\{c\}$ （即每个对象都有一个不同的单子集）。定义 F 在箭头上的作用。证明 F 是可表函子等价于 \mathcal{C} 有一个初始对象。

猜谜游戏

对象可以通过它们与其他对象的交互方式来描述，这一思想有时通过一个假想的猜谜游戏来说明。一位范畴论者选择一个范畴中的秘密对象，另一位必须猜出它是哪个对象（当然，在同构的意义上）。

猜谜者可以指向对象，并将它们用作“探针”来探测秘密对象。对手每次应回答一个集合：从探测对象 a 到秘密对象 x 的箭头集合。这当然是 \mathbf{hom} 集 $\mathcal{C}(a, x)$ 。

只要对手不作弊，这些答案的总和将定义一个预层 $F : \mathcal{C}^{op} \rightarrow \mathbf{Set}$ ，而他们隐藏的对象就是它的表示对象。

但我们怎么知道第二位范畴论者没有作弊呢？为了测试这一点，我们询问关于箭头的问题。对于每个我们选择的箭头，他们应该给我们两个集合之间的函数——他们为箭头的端点给出的集合。然后我们可以检查所有恒等箭头是否映射到恒等函数，以及箭头的复合是否映射到函数的复合。换句话说，我们将能够验证 F 确实是一个函子。

然而，足够聪明的对手仍然可能欺骗我们。他们向我们展示的预层可能描述一个奇异的对象——他们想象中的虚构物——而我们无法分辨。事实证明，这些奇异的生物通常和真实的对象一样有趣。

编程中的可表函子

在 Haskell 中，我们使用两个见证同构的函数来定义一类可表函子：

$$Fx = \mathcal{C}(a, x)$$

第一个函数 `tabulate` 将函数转换为查找表；第二个函数 `index` 使用表示类型 `Key` 对其进行索引。

```
class Representable f where
  type Key f :: Type
  tabulate :: (Key f -> x) -> f x
  index    :: f x -> (Key f -> x)
```

使用和类型的代数数据类型是不可表的（没有公式可以取和的对数）。例如，列表类型被定义为一个和，因此它不可表。然而，无限流是可表的。

从概念上讲，流就像一个无限元组，技术上是一个乘积。这样的流由自然数类型表示。换句话说，无限流等价于从自然数映射出来的东西。

```
data Stream a = Stm a (Stream a)
```

以下是实例定义：

```
instance Representable Stream where
  type Key Stream = Nat
  tabulate g = tab Z
    where
      tab n = Stm (g n) (tab (S n))
  index stm = \n -> ind n stm
    where
      ind Z (Stm a _) = a
      ind (S n) (Stm _ as) = ind n as
```

可表类型对于实现函数的记忆化非常有用。

Exercise 9.8.3. 为 `Pair` 实现 `Representable` 实例：

```
data Pair x = Pair x x
```

Exercise 9.8.4. 将一切映射到终端对象的常函子是可表的吗？提示： I 的对数是什么？

在 *Haskell* 中，这样的函子可以实现为：

```
data Unit a = U
```

为它实现 *Representable* 的实例。

Exercise 9.8.5. 列表函子不可表。但它可以被视为可表函子的和吗？

9.9 2-范畴 Cat

在范畴的范畴 **Cat** 中，hom-集不仅仅是集合。它们中的每一个都可以提升为一个函子范畴，其中自然变换扮演箭头的角色。这种结构被称为 2-范畴。

在 2-范畴的语言中，对象被称为 0-细胞，它们之间的箭头被称为 1-细胞，而箭头之间的箭头被称为 2-细胞。

这种结构的明显推广是拥有在 2-细胞之间的 3-细胞，依此类推。一个 n -范畴拥有直到第 n 层的细胞。

为什么不拥有一直延伸的箭头呢？这就是无穷范畴的引入。 ∞ -范畴远非一种奇观，它们具有实际应用。例如，在代数拓扑中，它们被用来描述点、点之间的路径、路径扫过的曲面、曲面扫过的体积，等等，无限延伸。

9.10 常用公式

- 协变函子的米田引理 (Yoneda lemma):

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), F) \cong Fa$$

- 逆变函子的米田引理 (Yoneda lemma):

$$[\mathcal{C}^{op}, \mathbf{Set}](\mathcal{C}(-, a), F) \cong Fa$$

- 米田引理的推论:

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(x, -), \mathcal{C}(y, -)) \cong \mathcal{C}(y, x)$$

$$[\mathcal{C}^{op}, \mathbf{Set}](\mathcal{C}(-, x), \mathcal{C}(-, y)) \cong \mathcal{C}(x, y)$$

Chapter 10

伴随

雕塑家去除无关的石头，直到雕塑显现。数学家抽象无关的细节，直到模式显现。

我们能够使用它们的映射入和映射出性质来定义许多构造。这些性质又可以简洁地写成 **hom** 集之间的同构。这种 **hom** 集之间的自然同构模式被称为伴随（adjunction），一旦被识别，几乎无处不在。

10.1 柯里化伴随

指数的定义是伴随的经典例子，它关联了映射出和映射入。每个从积中映射出的映射都对应一个唯一的映射入指数的映射：

$$\mathcal{C}(e \times a, b) \cong \mathcal{C}(e, b^a)$$

对象 b 在左侧扮演焦点的角色；对象 e 在右侧成为观察者。

我们可以发现两个函子在起作用。它们都由 a 参数化。在左侧，我们有应用于 e 的积函子 $(- \times a)$ 。在右侧，我们有应用于 b 的指数函子 $(-)^a$ 。

如果我们将这些函子写成：

$$L_a e = e \times a$$

$$R_a b = b^a$$

那么自然同构

$$\mathcal{C}(L_a e, b) \cong \mathcal{C}(e, R_a b)$$

被称为它们之间的伴随。

在分量中，这个同构告诉我们，给定一个映射 $\phi \in \mathcal{C}(L_a e, b)$ ，存在一个唯一的映射 $\phi^T \in \mathcal{C}(e, R_a b)$ ，反之亦然。这些映射有时被称为彼此的转置（transpose）——这个术语来自矩阵代数。

伴随的简写符号是 $L \dashv R$ 。将积函子代入 L ，指数函子代入 R ，我们可以将柯里化伴随简洁地写成：

$$(- \times a) \dashv (-)^a$$

指数对象 b^a 有时被称为内部 *hom* (internal hom)，并写作 $[a, b]$ 。这与外部 *hom* (external hom) 形成对比，后者是集合 $\mathcal{C}(a, b)$ 。外部 *hom* 不是 \mathcal{C} 中的对象（除非 \mathcal{C} 本身是 **Set**）。使用这种符号，柯里化伴随可以写成：

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

这种伴随成立的范畴被称为笛卡尔闭范畴。

由于函数在每个编程语言中扮演核心角色，笛卡尔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 $a \rightarrow b$ 。

这里 e 扮演外部环境的角色——lambda 演算中的 Γ 。 $\mathcal{C}(\Gamma \times a, b)$ 中的态射被解释为在环境 Γ 中扩展了一个类型为 a 的变量的类型为 b 的表达式。因此，函数类型 $a \rightarrow b$ 表示一个可能从其环境中捕获类型为 e 的值的闭包。

顺便提一下，(小) 范畴的范畴 **Cat** 也是笛卡尔闭的，这反映在积范畴和函子范畴之间的伴随中，使用了相同的内部 *hom* 符号：

$$\mathbf{Cat}(\mathcal{A} \times \mathcal{B}, \mathcal{C}) \cong \mathbf{Cat}(\mathcal{A}, [\mathcal{B}, \mathcal{C}])$$

这里，两边都是自然变换的集合。

10.2 和与积的伴随

Currying 伴随关系涉及两个自函子，但伴随关系可以很容易地推广到不同范畴之间的函子。让我们先看一些例子。

对角函子

和类型与积类型是通过双射定义的，其中一边是单个箭头，另一边是一对箭头。一对箭头可以看作是积范畴中的单个箭头。

为了探讨这一想法，我们需要定义对角函子 Δ ，它是从 \mathcal{C} 到 $\mathcal{C} \times \mathcal{C}$ 的特殊映射。它取一个对象 x 并复制它，生成一对对象 $\langle x, x \rangle$ 。它还取一个箭头 f 并复制它 $\langle f, f \rangle$ 。

有趣的是，对角函子与我们之前见过的常函子有关。常函子可以被视为一个双变量函子——它只是忽略第二个变量。我们在 Haskell 定义中见过这一点：

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


为了看到这种联系，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，**Cat** 中的指数对象 \mathcal{C}^2 。实际上，从 **2**（有两个对象的简笔范畴）的函子选择一对对象——这等价于积范畴中的单个对象。

一个函子 $\mathcal{C} \rightarrow [2, \mathcal{C}]$ 可以反 **curry** 化为 $\mathcal{C} \times 2 \rightarrow \mathcal{C}$ 。对角函子忽略第二个参数，即来自 **2** 的参数：无论第二个参数是 **1** 还是 **2**，它都做同样的事情。这正是常函子所做的。这就是为什么我们对两者使用相同的符号 Δ 。

顺便说一下，这个论证可以很容易地推广到任何索引范畴，而不仅仅是 **2**。

和伴随

回想一下，和类型是由其映射出属性定义的。从和 $a + b$ 出来的箭头与分别从 a 和 b 出来的箭头对之间存在一一对应关系。在 **hom** 集的术语中，我们可以写成：

$$\mathcal{C}(a + b, x) \cong \mathcal{C}(a, x) \times \mathcal{C}(b, x)$$

其中右边的积只是集合的笛卡尔积，即对的集合。此外，我们之前已经看到这个双射在 x 上是自然的。

我们知道，一对箭头是积范畴中的单个箭头。因此，我们可以将右边的元素视为 $\mathcal{C} \times \mathcal{C}$ 中从对象 $\langle a, b \rangle$ 到对象 $\langle x, x \rangle$ 的箭头。后者可以通过对角函子 Δ 作用于 x 得到。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 **hom** 集之间的双射。它满足自然性条件，因此是一个自然同构。

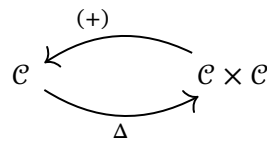
我们也可以在这里发现一对函子。在左边，我们有一个函子，它取一对对象 $\langle a, b \rangle$ 并生成它们的和 $a + b$ ：

$$(+): \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$$

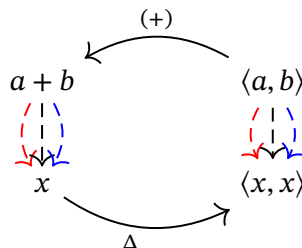
在右边，我们有对角函子 Δ 朝相反的方向：

$$\Delta: \mathcal{C} \rightarrow \mathcal{C} \times \mathcal{C}$$

总的来说，我们有一对范畴之间的一对函子：



以及 **hom** 集之间的同构：



换句话说，我们有伴随关系：

$$(+)\dashv\Delta$$

积伴随

我们可以将相同的推理应用于积的定义。这次我们有一对箭头与映射到积之间的自然同构。

$$\mathcal{C}(x, a) \times \mathcal{C}(x, b) \cong \mathcal{C}(x, a \times b)$$

将箭头对替换为积范畴中的箭头，我们得到：

$$(\mathcal{C} \times \mathcal{C})(\Delta x, \langle a, b \rangle) \cong \mathcal{C}(x, a \times b)$$

这是两个朝相反方向的函子：

$$\begin{array}{ccc} & \Delta & \\ \mathcal{C} \times \mathcal{C} & \xleftarrow{\quad} & \mathcal{C} \\ & (\times) & \end{array}$$

以及 hom 集的同构：

$$\begin{array}{ccc} & \Delta & \\ \langle x, x \rangle & \xleftarrow{\quad} & x \\ \text{---} \text{---} \text{---} & & \text{---} \text{---} \text{---} \\ \langle a, b \rangle & \xrightarrow{\quad} & a \times b \\ & (\times) & \end{array}$$

换句话说，我们有伴随关系：

$$\Delta \dashv (\times)$$

分配律

在双笛卡尔闭范畴中，积对和具有分配性。我们已经使用泛构造看到了证明的一个方向。伴随关系与 Yoneda 引理相结合，为我们提供了更强大的工具来解决这个问题。

我们想要展示自然同构：

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

与其直接证明这个恒等式，不如展示从两边到任意对象 x 的映射是同构的：

$$\mathcal{C}((b + c) \times a, x) \cong \mathcal{C}(b \times a + c \times a, x)$$

左边是积的映射，因此我们可以对其应用 **currying** 伴随关系：

$$\mathcal{C}((b + c) \times a, x) \cong \mathcal{C}(b + c, x^a)$$

这给出了和的映射，根据和伴随关系，它同构于两个映射的积：

$$\mathcal{C}(b + c, x^a) \cong \mathcal{C}(b, x^a) \times \mathcal{C}(c, x^a)$$

我们现在可以对两个分量应用 **currying** 伴随关系的逆：

$$\mathcal{C}(b, x^a) \times \mathcal{C}(c, x^a) \cong \mathcal{C}(b \times a, x) \times \mathcal{C}(c \times a, x)$$

使用和伴随关系的逆，我们得到最终结果：

$$\mathcal{C}(b \times a, x) \times \mathcal{C}(c \times a, x) \cong \mathcal{C}(b \times a + c \times a, x)$$

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。根据 **Yoneda** 引理，分配律左右两边的两个对象因此是同构的。

这个陈述的一个更简短的证明来自我们即将讨论的左伴随的性质。

10.3 函子之间的伴随关系

一般来说，伴随关系描述的是在两个范畴之间反向进行的两个函子之间的关系。左函子

$$L : \mathcal{D} \rightarrow \mathcal{C}$$

和右函子：

$$R : \mathcal{C} \rightarrow \mathcal{D}$$

伴随关系 $L \dashv R$ 被定义为两个 **hom**-集之间的自然同构。

$$\mathcal{C}(Lx, y) \cong \mathcal{D}(x, Ry)$$

换句话说，我们有一族集合之间的可逆函数：

$$\phi_{xy} : \mathcal{C}(Lx, y) \rightarrow \mathcal{D}(x, Ry)$$

在 x 和 y 上都是自然的。例如，在 y 上的自然性意味着，对于任何 $f : y \rightarrow y'$ ，以下图表交换：

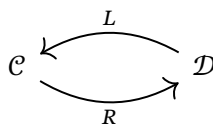
$$\begin{array}{ccc} \mathcal{C}(Lx, y) & \xrightarrow{\mathcal{C}(Lx, f)} & \mathcal{C}(Lx, y') \\ \uparrow \phi_{xy} & & \uparrow \phi_{xy'} \\ \mathcal{D}(x, Ry) & \xrightarrow{\mathcal{D}(x, Rf)} & \mathcal{D}(x, Ry') \end{array}$$

或者，考虑到通过 hom-函子提升箭头与后复合相同：

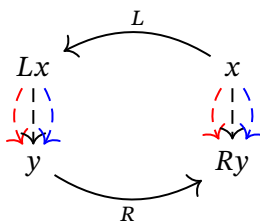
$$\begin{array}{ccc} \mathcal{C}(Lx, y) & \xrightarrow{f \circ -} & \mathcal{C}(Lx, y') \\ \uparrow \phi_{xy} & & \uparrow \phi_{xy'} \\ \mathcal{D}(x, Ry) & \xrightarrow{Rf \circ -} & \mathcal{D}(x, Ry') \end{array}$$

双箭头可以沿任一方向遍历（向上时使用 ϕ_{xy}^{-1} ），因为它们是同构的分量。

图示上，我们有两个函子：



并且，对于任何一对 x 和 y ，有两个同构的 hom-集：



这些 hom-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，这个的简化版本可以编码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)
```

它需要在文件顶部添加以下编译指示：

```
{-# 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点（或者我们可以用现代版本的楔形文字将其印在粘土上）：

$$(\+) \dashv \Delta \dashv (\times)$$

Exercise 10.3.1. 画出见证伴随函数 ϕ_{xy} 在 x 上自然性的交换方块。

Exercise 10.3.2. 伴随公式左侧的 hom-集 $\mathcal{C}(Lx, y)$ 表明 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. 反过来，预层 P 的表示对象 a 定义为：

$$Px \cong \mathcal{D}(x, a)$$

在伴随公式中， Ry 是哪个预层的表示对象。

10.4 极限与余极限作为伴随

极限的定义也涉及到同态集之间的自然同构：

$$[\mathcal{J}, \mathcal{C}](\Delta_x, D) \cong \mathcal{C}(x, \text{Lim} D)$$

左边的同态集位于函子范畴中。它的元素是锥，或者是常函子 Δ_x 与图表函子 D 之间的自然变换。右边的同态集位于 \mathcal{C} 中。

在所有极限存在的范畴中，我们有以下两个函子之间的伴随关系：

$$\Delta_{(-)} : \mathcal{C} \rightarrow [\mathcal{J}, \mathcal{C}]$$

和：

$$\text{Lim}(-) : [\mathcal{J}, \mathcal{C}] \rightarrow \mathcal{C}$$

对偶地，余极限由以下自然同构描述：

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x) \cong \mathcal{C}(\text{Colim} D, x)$$

我们可以用一个简洁的公式来表示这两个伴随关系：

$$\text{Colim} \dashv \Delta \dashv \text{Lim}$$

特别地，由于积范畴 $\mathcal{C} \times \mathcal{C}$ 等价于 \mathcal{C}^2 ，即函子范畴 $[\mathbf{2}, \mathcal{C}]$ ，我们可以将积和余积重写为极限和余极限：

$$[\mathbf{2}, \mathcal{C}](\Delta_x, \langle a, b \rangle) \cong \mathcal{C}(x, a \times b)$$

$$\mathcal{C}(a + b, x) \cong [\mathbf{2}, \mathcal{C}](\langle a, b \rangle, \Delta_x)$$

其中 $\langle a, b \rangle$ 表示一个图表，它是函子 $D : \mathbf{2} \rightarrow \mathcal{C}$ 在 $\mathbf{2}$ 的两个对象上的作用。

10.5 伴随的单位与余单位

我们通过等式来比较箭头，但在比较对象时，我们更倾向于使用同构。

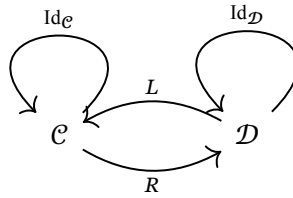
然而，当我们处理函子时，问题就出现了。一方面，它们是函子范畴中的对象，因此同构是合适的选择；另一方面，它们是 **Cat** 中的箭头，所以也许可以用等式来比较它们？

为了阐明这一困境，我们应该问自己为什么我们使用等式来比较箭头。这并不是因为我们喜欢等式，而是因为，在集合中，除了比较元素的等式之外，我们别无他法。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，我们知道，**Cat** 是一个 2-范畴。在这里，同态集本身具有范畴的结构——函子范畴。在 2-范畴中，我们有箭头之间的箭头，因此特别是，我们可以定义箭头之间的同构。在 **Cat** 中，这些将是函子之间的自然同构。

然而，尽管我们可以选择用同构来替换箭头等式，**Cat** 中的范畴法则仍然表示为等式。例如，函子 F 与恒等函子的复合等于 F ，结合性也是如此。在这种法则“严格”满足的 2-范畴中，称为严格 2-范畴，而 **Cat** 就是严格 2-范畴的一个例子。

但在比较范畴时，我们有更多的选择。范畴是 **Cat** 中的对象，因此可以定义范畴的同构为一对函子 L 和 R ：



使得：

$$L \circ R = \text{Id}_D$$

$$\text{Id}_C = R \circ L$$

然而，这个定义涉及函子的等式。更糟糕的是，作用于对象时，它涉及对象的等式：

$$L(Rx) = x$$

$$y = R(Ly)$$

这就是为什么更合适的是讨论范畴的等价这一较弱的概念，其中等式被同构取代：

$$L \circ R \cong \text{Id}_D$$

$$\text{Id}_C \cong R \circ L$$

在对象上，范畴的等价意味着往返操作产生的对象与原始对象同构，而不是相等。在大多数情况下，这正是我们想要的。

伴随也定义为方向相反的一对函子，因此询问往返操作的结果是有意义的。

定义伴随的同构适用于任何对象对 x 和 y

$$\mathcal{C}(Lx, y) \cong \mathcal{D}(x, Ry)$$

因此，特别是，如果我们用 Lx 替换 y ，它也适用

$$\mathcal{C}(Lx, Lx) \cong \mathcal{D}(x, R(Lx))$$

我们现在可以使用 Yoneda 技巧，在左侧选择恒等箭头 id_{Lx} 。同构将其映射到右侧的唯一箭头，我们称之为 η_x ：

$$\eta_x : x \rightarrow R(Lx)$$

这个映射不仅对每个 x 都有定义，而且在 x 上是自然的。自然变换 η 称为伴随的单位。如果我们注意到左侧的 x 是恒等函子作用于 x 的结果，我们可以写成：

$$\eta : Id_{\mathcal{D}} \rightarrow R \circ L$$

例如，让我们评估余积伴随的单位：

$$\mathcal{C}(a + b, x) \cong (\mathcal{C} \times \mathcal{C})(\langle a, b \rangle, \Delta x)$$

通过用 $a + b$ 替换 x 。我们得到：

$$\eta_{\langle a, b \rangle} : \langle a, b \rangle \rightarrow \Delta(a + b)$$

这是一对箭头，正是两个注入 $\langle \text{Left}, \text{Right} \rangle$ 。

我们可以通过用 Ry 替换 x 来做类似的技巧：

$$\mathcal{C}(L(Ry), y) \cong \mathcal{D}(Ry, Ry)$$

对应于右侧的 id_{Ry} ，我们在左侧得到一个箭头：

$$\varepsilon_y : L(Ry) \rightarrow y$$

这些箭头形成另一个自然变换，称为伴随的余单位：

$$\varepsilon : L \circ R \rightarrow Id_{\mathcal{C}}$$

注意，如果这两个自然变换是可逆的，它们将见证范畴的等价。但即使它们不是，这种“半等价”在范畴论的背景下仍然非常有趣。

例如，让我们评估积伴随的余单位：

$$(\mathcal{C} \times \mathcal{C})(\Delta x, \langle a, b \rangle) \cong \mathcal{C}(x, a \times b)$$

通过用 $a \times b$ 替换 x 。我们得到：

$$\varepsilon_{\langle a, b \rangle} : \Delta(a \times b) \rightarrow \langle a, b \rangle$$

这是一对箭头，正是两个投影 $\langle \text{fst}, \text{snd} \rangle$ 。

Exercise 10.5.1. 推导余积伴随的余单位和积伴随的单位。

三角恒等式

我们可以使用单位/余单位对来表述伴随的等价定义。为此，我们从一对自然变换开始：

$$\eta : \text{Id}_{\mathcal{D}} \rightarrow R \circ L$$

$$\varepsilon : L \circ R \rightarrow \text{Id}_{\mathcal{C}}$$

并施加额外的三角恒等式。

这些恒等式可以通过伴随的标准定义推导出来，注意到 η 可以用来用复合 $R \circ L$ 替换恒等函子，从而让我们在任何恒等函子起作用的地方插入 $R \circ L$ 。

类似地， ε 可以用来消除复合 $L \circ R$ （即用恒等替换它）。

因此，例如，从 L 开始：

$$L = L \circ \text{Id}_{\mathcal{D}} \xrightarrow{L \circ \eta} L \circ R \circ L \xrightarrow{\varepsilon \circ L} \text{Id}_{\mathcal{C}} \circ L = L$$

在这里，我们使用了自然变换的水平复合，其中一个变换是恒等变换（也称为 **whiskering**）。

第一个三角恒等式是这一系列变换结果等于恒等自然变换的条件。图示如下：

$$\begin{array}{ccc} L & \xrightarrow{L \circ \eta} & L \circ R \circ L \\ & \searrow \text{id}_L & \downarrow \varepsilon \circ L \\ & & L \end{array}$$

类似地，我们希望以下一系列自然变换也复合为恒等：

$$R = \text{Id}_{\mathcal{D}} \circ R \xrightarrow{\eta \circ R} R \circ L \circ R \xrightarrow{R \circ \varepsilon} R \circ \text{Id}_{\mathcal{C}} = R$$

或图示如下：

$$\begin{array}{ccc} R & \xrightarrow{\eta \circ R} & R \circ L \circ R \\ & \searrow \text{id}_R & \downarrow R \circ \varepsilon \\ & & R \end{array}$$

事实证明，伴随可以等价地用两个自然变换 η 和 ε 来定义，满足三角恒等式：

$$(\varepsilon \circ L) \cdot (L \circ \eta) = \text{id}_L$$

$$(R \circ \varepsilon) \cdot (\eta \circ R) = \text{id}_R$$

从这些恒等式中，可以很容易地恢复同态集的映射。例如，让我们从一个箭头 $f : x \rightarrow Ry$ 开始，它是 $\mathcal{D}(x, Ry)$ 的一个元素。我们可以将其提升为

$$Lf : Lx \rightarrow L(Ry)$$

然后我们可以使用 η 将复合 $L \circ R$ 折叠为恒等。结果是一个箭头 $Lx \rightarrow y$ ，它是 $\mathcal{C}(Lx, y)$ 的一个元素。

使用单位和余单位定义的伴随在某种意义上更通用，因为它可以转化为任意的 2-范畴设置。

Exercise 10.5.2. 给定一个箭头 $g: Lx \rightarrow y$, 使用 ε 和 R 是函子的事实, 实现一个箭头 $x \rightarrow Ry$ 。提示: 从对象 x 开始, 看看如何通过一个中转点从那里到达 Ry 。

柯里化伴随的单位与余单位

让我们计算柯里化伴随的单位与余单位:

$$\mathcal{C}(e \times a, b) \cong \mathcal{C}(e, b^a)$$

如果我们用 $e \times a$ 替换 b , 我们得到

$$\mathcal{C}(e \times a, e \times a) \cong \mathcal{C}(e, (e \times a)^a)$$

对应于左侧的恒等箭头, 我们在右侧得到伴随的单位:

$$\eta: e \rightarrow (e \times a)^a$$

这是积构造器的柯里化版本。在 Haskell 中, 我们写成:

```
unit :: e -> (a -> (e, a))
unit = curry id
```

余单位更有趣。用 b^a 替换 e , 我们得到:

$$\mathcal{C}(b^a \times a, b) \cong \mathcal{C}(b^a, b^a)$$

对应于右侧的恒等箭头, 我们得到:

$$\varepsilon: b^a \times a \rightarrow b$$

这是函数应用箭头。

在 Haskell 中:

```
counit :: (a -> b, a) -> b
counit = uncurry id
```

当伴随在两个自函子之间时, 我们可以使用单位和余单位写出 Haskell 中的替代定义:

```
class (Functor left, Functor right) =>
  Adjunction left right | left -> right, right -> left where
  unit   :: x -> right (left x)
  counit :: left (right x) -> x
```

额外的两个子句`left -> right`和`right -> left`告诉编译器，在使用伴随的实例时，一个函子可以从另一个函子派生出来。这个定义需要以下编译扩展：

```
{-# language MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
```

形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
```

柯里化的`Adjunction`实例是：

```
instance Adjunction (L r) (R r) where
    unit x = R (\r -> L (x, r))
    counit (L (R f, r)) = f r
```

第一个三角恒等式表明以下多态函数：

```
triangle :: L r x -> L r x
triangle = counit . fmap unit
```

是恒等函数，第二个也是如此：

```
triangle' :: R r x -> R r x
triangle' = fmap counit . unit
```

注意，这两个函数需要使用函数依赖才能正确定义。三角恒等式无法在 `Haskell` 中表达，因此伴随的实现者需要证明它们。

Exercise 10.5.3. 测试柯里化伴随的第一个三角恒等式的几个例子。以下是一个例子：

```
triangle (L (2, 'a'))
```

Exercise 10.5.4. 你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，因此你无法显示它，但你可以调用它。

10.6 使用通用箭头的伴随

我们已经看到了使用同态集同构定义的伴随，以及使用单位/余单位对定义的另一种方式。事实证明，只要满足某些通用性条件，我们可以仅使用这对中的一个元素来定义伴随。为了理解这一点，我们将构建一个新范畴，其对象是箭头。

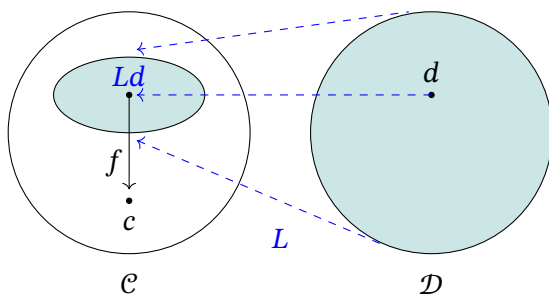
我们之前已经见过这样一个范畴的例子——切片范畴 \mathcal{C}/c ，它收集了所有收敛于 c 的箭头。这样的范畴描述了从 \mathcal{C} 中每个可能的角度观察对象 c 的情况。

逗号范畴

在处理伴随时：

$$\mathcal{C}(Ld, c) \cong \mathcal{D}(d, Rc)$$

我们是从由函子 L 定义的更窄的视角观察对象 c 。将 L 视为在 \mathcal{C} 内部定义 \mathcal{D} 的模型。我们感兴趣的是从这个模型的视角观察 c 的情况。描述这种视角的箭头形成了逗号范畴 L/c 。



在逗号范畴 L/c 中，一个对象是一个对 $\langle d, f \rangle$ ，其中 d 是 \mathcal{D} 的一个对象， $f: Ld \rightarrow c$ 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 $h: d \rightarrow d'$ ，使得左边的图表交换：

$$\begin{array}{ccc} Ld & \xrightarrow{Lh} & Ld' \\ f \searrow & & \swarrow f' \\ & c & \end{array} \qquad d \xrightarrow{h} d'$$

通用箭头

从 L 到 c 的通用箭头定义为逗号范畴 L/c 中的终对象。让我们拆解这个定义。 L/c 中的终对象是一个对 $\langle t, \tau \rangle$ ，具有从任何对象 $\langle d, f \rangle$ 的唯一态射。这样的态射是一个箭头 $h: d \rightarrow t$ ，满足交换条件：

$$\begin{array}{ccc} Ld & \overset{Lh}{\dashrightarrow} & Lt \\ f \searrow & & \swarrow \tau \\ & c & \end{array}$$

换句话说，对于同态集 $\mathcal{C}(Ld, c)$ 中的任何 f ，在同态集 $\mathcal{D}(d, t)$ 中存在唯一元素 h ，使得：

$$f = \tau \circ Lh$$

这种两个同态集元素之间的一一对应关系暗示了底层的伴随。

从伴随中得到的通用箭头

首先让我们确信，当函子 L 有一个右伴随 R 时，对于每个 c ，存在一个从 L 到 c 的通用箭头。实际上，这个箭头由对 $\langle Rc, \varepsilon_c \rangle$ 给出，其中 ε 是伴随的余单位。首先，余单位的分量具有逗号范畴 L/c 中对象的正确签名：

$$\varepsilon_c : L(Rc) \rightarrow c$$

我们希望证明 $\langle Rc, \varepsilon_c \rangle$ 是 L/c 中的终对象。也就是说，对于任何对象 $\langle d, f : Ld \rightarrow c \rangle$ ，存在唯一的 $h : d \rightarrow Rc$ ，使得 $f = \varepsilon_c \circ Lh$ ：

$$\begin{array}{ccc} Ld & \xrightarrow{\quad Lh \quad} & L(Rc) \\ & \searrow f & \swarrow \varepsilon_c \\ & c & \end{array}$$

为了证明这一点，让我们将 ϕ_{dc} 的一个自然性条件写为 d 的函数：

$$\phi_{dc} : \mathcal{C}(Ld, c) \rightarrow \mathcal{D}(d, Rc)$$

对于任何箭头 $h : d \rightarrow d'$ ，以下图表必须交换：

$$\begin{array}{ccc} \mathcal{C}(Ld', c) & \xrightarrow{-\circ Lh} & \mathcal{C}(Ld, c) \\ \updownarrow \phi_{d', c} & & \updownarrow \phi_{d, c} \\ \mathcal{D}(d', Rc) & \xrightarrow{-\circ h} & \mathcal{D}(d, Rc) \end{array}$$

我们可以使用 Yoneda 技巧，将 d' 设为 Rc 。

$$\begin{array}{ccc} \mathcal{C}(L(Rc), c) & \xrightarrow{-\circ Lh} & \mathcal{C}(Ld, c) \\ \updownarrow \phi_{Rc, c} & & \updownarrow \phi_{d, c} \\ \mathcal{D}(Rc, Rc) & \xrightarrow{-\circ h} & \mathcal{D}(d, Rc) \end{array}$$

我们现在可以选择同态集 $\mathcal{D}(Rc, Rc)$ 中的特殊元素，即恒等箭头 id_{Rc} ，并将其传播到图表的其余部分。左上角变为 ε_c ，右下角变为 h ，右上角变为 h 的伴随，我们称之为 f ：

$$\begin{array}{ccc} \varepsilon_c & \xrightarrow{-\circ Lh} & f \\ \updownarrow \phi_{Rc, c} & & \updownarrow \phi_{d, c} \\ id_{Rc} & \xrightarrow{-\circ h} & h \end{array}$$

上箭头则给出了我们寻求的等式 $f = (-\circ Lh)\varepsilon_c = \varepsilon_c \circ Lh$ 。

从通用箭头构造伴随

相反的结果更有趣。如果对于每个 c ，我们有一个从 L 到 c 的通用箭头，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。这个函子在对象上的作用由 $Rc = t_c$ 给出，而族 ε_c 在 c 中自动是自然的，并且它形成了伴随的余单位。

还有一个对偶的陈述：可以从通用箭头族 η_d 开始构造伴随，这些通用箭头形成逗号范畴 d/R 中的始对象。

这些结果将帮助我们证明 Freyd 的伴随函子定理。

10.7 伴随函子的性质

左伴随函子保持余极限

我们将余极限定义为通用的余锥。对于每个余锥——即从图 $D: \mathcal{J} \rightarrow \mathcal{C}$ 到常函子 Δ_x 的自然变换——应该存在一个唯一的因子化态射从余极限 $\text{Colim } D$ 到 x 。这个条件可以写成余锥集合与特定 hom 集合之间的一一对应关系：

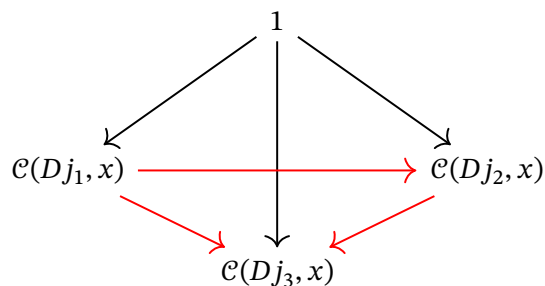
$$[\mathcal{J}, \mathcal{C}](D, \Delta_x) \cong \mathcal{C}(\text{Colim } D, x)$$

因子化条件被编码在这个同构的自然性中。

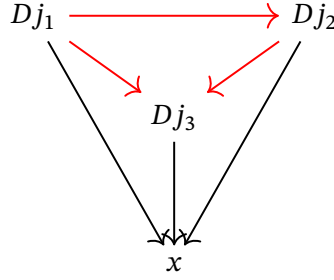
事实证明，余锥集合作为 **Set** 中的一个对象，本身是以下 **Set** 值函子 $F: \mathcal{J} \rightarrow \mathbf{Set}$ 的极限：

$$Fj = \mathcal{C}(Dj, x)$$

为了证明这一点，我们将从 F 的极限开始，最终得到余锥集合。你可能记得，**Set** 值函子的极限等于以 **1**（单元素集合）为顶点的锥集合。在我们的情况下，每个这样的锥描述了从相应的 hom 集合 $\mathcal{C}(Dj, x)$ 中选择的态射：



这些态射的目标都是同一个对象 x ，因此它们形成了以 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 为顶点的余锥的交换条件。但这些正是集合 $[J, \mathcal{C}](D, \Delta_x)$ 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合，得到：

$$\text{Lim } \mathcal{C}(D-, x) \cong \mathcal{C}(\text{Colim } D, x)$$

逆变 **hom** 函子有时记为：

$$h_x = \mathcal{C}(-, x)$$

在这种记法中，我们可以写成：

$$\text{Lim}(h_x \circ D) \cong h_x(\text{Colim } D)$$

作用于图 D 的 **hom** 函子的极限同构于作用于该图的余极限的 **hom** 函子。这通常简化为：**hom** 函子保持余极限。（理解逆变 **hom** 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 **hom** 函子是余连续的。

现在假设我们有伴随 $L \dashv R$ ，其中 $L: \mathcal{C} \rightarrow \mathcal{D}$ ，而 R 方向相反。我们想证明左函子 L 保持余极限，即：

$$L(\text{Colim } D) \cong \text{Colim}(L \circ D)$$

对于任何存在余极限的图 $D: J \rightarrow \mathcal{C}$ 。

我们将使用 Yoneda 引理来证明从两边到任意 x 的映射是同构的：

$$\mathcal{D}(L(\text{Colim } D), x) \cong \mathcal{D}(\text{Colim}(L \circ D), x)$$

我们将伴随应用于左边，得到：

$$\mathcal{D}(L(\text{Colim } D), x) \cong \mathcal{C}(\text{Colim } D, Rx)$$

hom 函子保持余极限的性质给我们：

$$\cong \text{Lim } \mathcal{C}(D-, Rx)$$

再次使用伴随，我们得到：

$$\cong \text{Lim } \mathcal{D}((L \circ D)-, x)$$

第二次应用余极限的保持性质，我们得到了期望的结果：

$$\cong \mathcal{D}((\text{Colim } (L \circ D), x)$$

由于这对任何 x 都成立，我们得到了我们的结果。

我们可以利用这个结果重新表述我们在笛卡尔闭范畴中关于分配性的早期证明。我们利用乘积是指数函子的左伴随这一事实。左伴随保持余极限。余积是余极限，因此：

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

这里，左函子是 $Lx = x \times a$ ，而图 D 选择了一对对象 b 和 c 。

右伴随函子保持极限

使用对偶论证，我们可以证明右伴随函子保持极限，即：

$$R(\text{Lim } D) \cong \text{Lim } (R \circ D)$$

我们首先证明（协变） hom 函子保持极限。

$$\text{Lim } \mathcal{C}(x, D-) \cong \mathcal{C}(x, \text{Lim } D)$$

这源于定义极限的锥集合同构于 **Set** 值函子的极限的论证：

$$Fj = \mathcal{C}(x, Dj)$$

保持极限的函子称为连续函子。

为了证明在伴随 $L \dashv R$ 下，右函子 $R: \mathcal{D} \rightarrow \mathcal{C}$ 保持极限，我们使用 Yoneda 论证：

$$\mathcal{C}(x, R(\text{Lim } D)) \cong \mathcal{C}(x, \text{Lim } (R \circ D))$$

确实，我们有：

$$\mathcal{C}(x, R(\text{Lim } D)) \cong \mathcal{D}(Lx, \text{Lim } D) \cong \text{Lim } \mathcal{D}(Lx, D-) \cong \mathcal{C}(x, \text{Lim } (R \circ D))$$

10.8 Freyd 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”来弥补丢失的信息。如果我们以有组织的方式进行，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它的伴随。

这个问题的答案由 Freyd 伴随函子定理给出。乍一看，这似乎是一个技术性定理，涉及一个非常抽象的构造，称为解集条件（solution set condition）。我们稍后会看到，这个条件直接转化为一种称为去函数化（defunctionalization）的编程技术。

在接下来的内容中，我们将专注于构造函数子 $L: \mathcal{D} \rightarrow \mathcal{C}$ 的右伴随。对偶的推理可以用于解决寻找函子 $R: \mathcal{C} \rightarrow \mathcal{D}$ 的左伴随的逆问题。

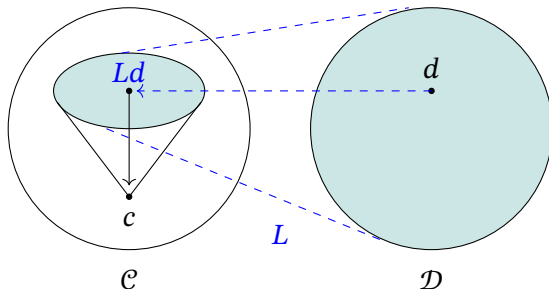
第一个观察是，由于伴随中的左函子保持余极限，我们必须假设我们的函子 L 保持余极限。这给了我们一个提示，即右伴随的构造依赖于在 \mathcal{D} 中构造余极限的能力，并能够使用 L 将它们以某种方式传输回 \mathcal{C} 。

我们可以要求 \mathcal{D} 中存在所有余极限，无论大小，但这个条件太强了。即使是一个具有所有余极限的小范畴也自动成为一个预序——也就是说，它不能在任意两个对象之间有多于一个的态射。

但让我们暂时忽略大小问题，看看如何构造一个保持余极限的函子 L 的右伴随，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）。

预序中的 Freyd 定理

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 的通用箭头。这样的箭头是逗号范畴 L/c 中的终端对象——该范畴中的箭头起源于 L 的像并收敛于对象 c 。

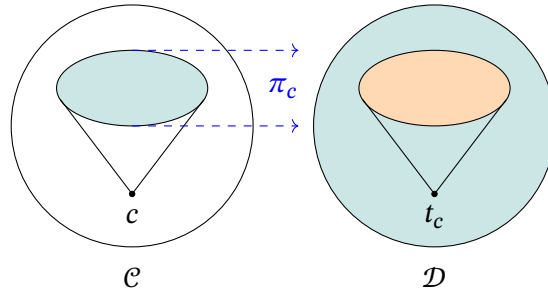


重要的观察是，这个逗号范畴描述了 \mathcal{C} 中的一个余锥。这个余锥的基由那些在 L 的像中且对 c 有畅通视野的对象组成。余锥基中的箭头是 L/c 中的态射。这些正是使余锥的边交换的箭头。

$$\begin{array}{ccc}
 Ld & \xrightarrow{Lh} & Ld' \\
 f \searrow & & \swarrow f' \\
 & c &
 \end{array}
 \qquad
 d \xrightarrow{h} d'$$

然后将这个余锥的基投影回 \mathcal{D} 。有一个投影 π_c ，它将 L/c 中的每对 (d, f) 映射回 d ，从而忘记箭头 f 。它还将 L/c 中的每个态射映射到 \mathcal{D} 中产生它的箭头。这样 π_c 定义了 \mathcal{D} 中的一个图。这个图的余极限存在，因为我们假设 \mathcal{D} 中存在所有余极限。让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 L/c 中的终端对象。我们必须找到一个箭头，让我们称之为 $\varepsilon_c: Lt_c \rightarrow c$ ，使得对 $\langle t_c, \varepsilon_c \rangle$ 是 L/c 中的终端对象。

注意， L 将 π_c 生成的图映射回由 L/c 定义的余锥的基。投影 π_c 所做的不过是忽略这个余锥的边，保持其基不变。

我们现在在 \mathcal{C} 中有两个具有相同基的余锥：原始的以 c 为顶点的余锥和通过将 L 应用于 \mathcal{D} 中的余锥得到的新余锥。由于 L 保持余极限，新余锥的余极限是 Lt_c ——余极限 t_c 的像：

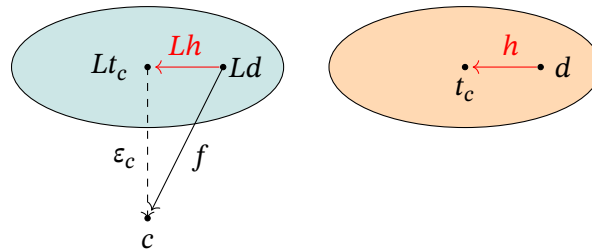
$$\operatorname{colim} (L \circ \pi_c) = L(\operatorname{colim} \pi_c) = Lt_c$$

通过通用构造，我们推断必须存在一个从余极限 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 中的终端对象，即对于任何 $\langle d, f: Ld \rightarrow c \rangle$ ，存在一个唯一的逗号范畴态射 $h: d \rightarrow t_c$ ，使得以下三角形交换：

$$\begin{array}{ccc} Ld & \xrightarrow{\quad Lh \quad} & Lt_c \\ & \searrow f \quad \swarrow \varepsilon_c & \\ & c & \end{array}$$

注意，任何这样的 d 自动成为 π_c 生成的图的一部分（它是 π_c 作用于 $\langle d, f \rangle$ 的结果）。我们知道 t_c 是 π_c 图的极限。因此在极限余锥中必须有一条从 d 到 t_c 的线。我们选择这条线作为我们的 h 。



交换条件随后由 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。

这证明了对于每个 c 都存在一个通用箭头 $\langle t_c, \varepsilon_c \rangle$ ，因此我们有一个函子 R ，定义为 $Rc = t_c$ ，它是 L 的右伴随。

解集条件

先前证明的问题在于，在大多数实际情况下，逗号范畴（comma category）是大的：它们的对象不构成一个集合。但也许我们可以通过选择一个更小但具有代表性的对象和箭头集来近似逗号范畴？

为了选择对象，我们将使用从某个索引集 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 ，我们选择对象以及箭头 $f_i: Ld_i \rightarrow c$ 。

逗号范畴的相关部分被编码在满足交换条件的对象之间的态射中。我们可以尝试将这个条件专门化，使其仅适用于我们的对象族，但这还不够。我们必须找到一种方法来探测逗号范畴的所有其他对象。

为此，我们将交换条件重新解释为通过某个对 $\langle d_i, f_i \rangle$ 分解任意 $f: Ld \rightarrow c$ 的配方：

$$\begin{array}{ccc} Ld & \xrightarrow{Lh} & Ld_i \\ & \searrow f & \swarrow f_i \\ & c & \end{array}$$

一个解集（solution set）是一组由集合 I 索引的对 $\langle d_i, f_i: Ld_i \rightarrow c \rangle$ ，可以用来分解任何对 $\langle d, f: Ld \rightarrow c \rangle$ 。这意味着存在一个索引 $i \in I$ 和一个箭头 $h: d \rightarrow d_i$ ，使得 f 被分解为：

$$f = f_i \circ Lh$$

表达这个性质的另一种方式是，在逗号范畴 L/c 中存在一个弱终集（weakly terminal set）。弱终集的性质是，对于范畴中的任何对象，都存在一个态射到该集合中的至少一个对象。

之前我们已经看到，对于每个 c ，在逗号范畴 L/c 中存在终对象足以定义伴随。事实证明，我们可以使用解集实现相同的目标。

Freyd 伴随函子定理的假设指出，我们有一个保持余极限的函子 $L: \mathcal{D} \rightarrow \mathcal{C}$ ，来自一个小余完备范畴。这两个条件都与小图相关。如果我们能为每个 c 选择一个解集 $\langle d_i, f_i: Ld_i \rightarrow c \rangle$ ，则右伴随 R 存在。不同 c 的解集可能不同。

我们之前已经看到，在余完备范畴中，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 ，我们可以构造从 L 到 c 的通用箭头。而这足以定义整个伴随。

伴随函子定理的对偶版本可以用来构造左伴随。

去函数化 (Defunctionalization)

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（将函数作为参数的函数、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 函数）。事实证明，即使在这样的语言中，也

可以通过称为去函数化的过程来实现高阶函数。该技术基于伴随函子定理 (adjoint functor theorem)。此外，当传递函数不切实际时（例如在分布式系统中），也可以使用去函数化。

去函数化的核心思想是将函数类型定义为积的右伴随。

$$\mathcal{C}(e \times a, b) \cong \mathcal{C}(e, b^a)$$

伴随函子定理可用于近似这个伴随。

一般来说，任何有限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）构成了我们可以用来构造函数类型的解集。在实践中，我们只对那些作为其他函数的参数或返回值的函数子集进行此操作。

高阶函数的一个典型应用是在延续传递风格 (continuation passing style) 中。例如，以下是一个计算列表元素和的函数。但它不是直接返回和，而是调用延续 `k` 并传递结果：

```
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，函数将调用延续并传递零。否则，它递归调用自身，传递两个参数：列表的尾部 `is`，以及一个新的延续：

```
\s -> k (i + s)
```

这个新延续调用之前的延续 `k`，并传递列表头部与其参数 `s`（即累加和）的和。

注意这个 `lambda` 是一个闭包：它是一个单变量 `s` 的函数，但它也可以访问其环境中的 `k` 和 `i`。

为了提取最终的和，我们使用平凡的延续（即恒等函数）调用递归函数：

```
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，如果我们想要提取延续，就必须显式传递环境。

例如，我们可以将第一个 `lambda`：

```
\s -> k (i + s)
```

替换为函数 `more`，但必须显式传递对 `(i, k)` 作为类型为 `(Int, Int -> r)` 的环境：

```
more :: (Int, Int -> r) -> Int -> r
more (i, k) s = k (i + s)
```

另一个 lambda（即恒等函数）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，如果我们只关心计算和，可以将多态类型 `r` 替换为 `Int`，而无需其他更改。

此实现仍然使用高阶函数。为了消除它们，我们必须分析将函数作为参数传递的含义。这样的函数只能以一种方式使用：它可以应用于其参数。函数类型的这一性质表示为柯里化伴随的余单位 (counit)：

$$\varepsilon : b^a \times a \rightarrow b$$

或在 Haskell 中，作为一个高阶函数：

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

这次我们感兴趣的是从基本原理构造余单位。我们已经看到，这可以通过逗号类别 (comma category) 实现。在我们的例子中，积函子 $L_a = (-) \times a$ 的逗号类别的一个对象是一对

$$(e, f : (e \times a) \rightarrow b)$$

或在 Haskell 中：

```
data Comma a b e = Comma e ((e, a) -> b)
```

此类别中 (e, f) 和 (e', f') 之间的态射是一个箭头 $h: e \rightarrow e'$ ，它满足交换条件：

$$f' \circ h = f$$

我们将此态射解释为将环境 e “缩减”到 e' 。箭头 f' 能够使用由 $h(e)$ 给出的潜在更小的环境生成相同类型 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。

$$\begin{array}{ccc} e \times a & \xrightarrow{h \times a} & e' \times a \\ & \searrow f & \swarrow f' \\ & b & \end{array} \qquad e \xrightarrow{h} e'$$

事实上，我们在定义 `more` 和 `done` 时执行了这种缩减。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 定理，我们可以将函数对象 $a \rightarrow b$ 定义为由逗号类别定义的图的余极限 (colimit)。这样的余极限本质上是所有环境的巨大余积 (coproduct)，模由逗号类别态射给出的标识。这种标识将 $a \rightarrow b$ 所需的环境缩减到最低限度。

在我们的例子中，我们感兴趣的延续是函数 `Int -> Int`。事实上，我们并不感兴趣生成通用的函数类型 `Int -> Int`；只是生成能够容纳我们两个函数 `more` 和 `done` 的最小函数类型。我们可以通过创建一个非常小的解集来实现。

在我们的例子中，解集由 $(e_i, f_i: e_i \times a \rightarrow b)$ 对组成，使得任何对 $(e, f: e \times a \rightarrow b)$ 都可以通过其中一个 f_i 进行因式分解。更准确地说，我们感兴趣的唯二环境是 $(\text{Int}, \text{Int} \rightarrow \text{Int})$ （用于 `more`）和空环境 $()$ （用于 `done`）。

原则上，我们的解集应允许对逗号类别的每个对象进行因式分解，即类型为：

```
(e, (e, Int) -> Int)
```

的对，但这里我们只对两个特定函数感兴趣。此外，我们不关心表示的唯一性，因此，我们不会使用余极限（如我们在伴随函子定理中所做的那样），而是仅使用所有感兴趣环境的余积。我们最终得到以下数据类型，它是我们感兴趣的两个环境 $()$ 和 $(\text{Int}, \text{Int} \rightarrow \text{Int})$ 的和。我们最终得到类型：

```
data Kont = Done | More Int Kont
```

注意，我们已将环境的 `Int->Int` 部分递归编码为 `Kont`。因此，我们也消除了将函数作为数据构造器参数的需要。

如果你仔细查看这个定义，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` 都会将另一个整数推入 `Kont` 栈。这种解释与我们的直觉一致，即递归算法需要某种运行时栈。

我们现在准备实现我们对伴随余单位的近似。它由两个函数的主体组成，并理解递归调用也通过 `apply` 进行：

```

apply :: (Kont, Int) -> Int
apply (Done, i) = i
apply (More i k, s) = apply (k, i + s)

```

将其与我们之前的代码进行比较：

```

done i = i
more (i, k) s = k (i + s)

```

现在，主算法可以重写为不包含任何高阶函数或 `lambda` 的形式：

```

sumK'' :: [Int] -> Kont -> Int
sumK'' [] k = apply (k, 0)
sumK'' (i : is) k = sumK'' is (More i k)

sumList'' is = sumK'' is Done

```

去函数化的主要优点是它可以在分布式环境中使用。只要远程函数的参数是数据结构而不是函数，就可以将其序列化并通过网络发送。接收方只需访问 `apply` 即可。

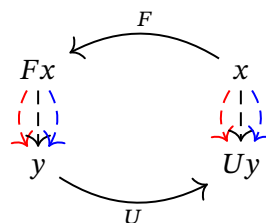
10.9 自由/遗忘伴随

伴随中的两个函子扮演着不同的角色：伴随的图示并不对称。这一点在自由/遗忘伴随的情况下体现得尤为明显。

遗忘函子（forgetful functor）是一种“遗忘”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，被遗忘的结构是显而易见的。通常，目标范畴就是集合范畴，它被认为是无结构的典范。在这种情况下，遗忘函子的结果被称为“底层”集合，而函子本身通常记为 U 。

更准确地说，我们说一个函子遗忘结构，如果 \mathbf{hom} 集的映射不是满射的，也就是说，目标 \mathbf{hom} 集中存在没有对应源 \mathbf{hom} 集中箭头的箭头。直观上，这意味着源中的箭头需要保留某些结构，因此它们的数量较少；而这些结构在目标中是不存在的。

遗忘函子的左伴随被称为自由函子。



自由/遗忘伴随的一个经典例子是自由幺半群的构造。

幺半群范畴

幺半群范畴 \mathcal{C} 中的幺半群形成它们自己的范畴 $\mathbf{Mon}(\mathcal{C})$ 。其对象是幺半群，其箭头是 \mathcal{C} 中保留幺半群结构的箭头。

以下图示解释了 f 作为幺半群态射的含义，从幺半群 (M_1, η_1, μ_1) 到幺半群 (M_2, η_2, μ_2) ：

$$\begin{array}{ccccc}
 & & M_1 & \xleftarrow{\mu_1} & M_1 \otimes M_1 \\
 \eta_1 \nearrow & & \downarrow f & & \downarrow f \otimes f \\
 I & & & & \\
 \eta_2 \searrow & & M_2 & \xleftarrow{\mu_2} & M_2 \otimes M_2
 \end{array}$$

幺半群态射 f 必须将单位元映射到单位元，这意味着：

$$f \circ \eta_1 = \eta_2$$

并且它必须将乘法映射到乘法：

$$f \circ \mu_1 = \mu_2 \circ (f \otimes f)$$

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** 是幺半群的，其笛卡尔积和终端对象提供了幺半群结构。

特别地，**Set** 中的幺半群是具有额外结构的集合。它们形成自己的范畴 $\mathbf{Mon}(\mathbf{Set})$ ，并且存在一个遗忘函子 U ，它简单地将幺半群映射到其元素的集合。当我们说幺半群是一个集合时，我们指的是底层集合。

自由幺半群

我们希望构造自由函子

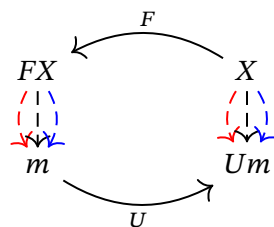
$$F: \mathbf{Set} \rightarrow \mathbf{Mon}(\mathbf{Set})$$

它是遗忘函子 U 的伴随。

我们从任意集合 X 和任意幺半群 m 开始。在伴随的右侧，我们有从 X 到 Um 的函数集合。在左侧，我们有一组高度受限的结构保持的幺半群态射，从 FX 到 m 。这两个 \mathbf{hom} 集如何同构？

在 $\mathbf{Mon}(\mathbf{Set})$ 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，满足额外的约束：保留单位元和乘法。

另一方面，**Set** 中的箭头只是没有额外约束的函数。因此，一般来说，幺半群之间的箭头比它们的底层集合之间的箭头要少。



这里的想法是：如果我们想要在箭头之间有一一对应，我们希望 FX 比 X 大得多。这样，从它到 m 的函数会更多——即使排除了那些不保留结构的函数，我们仍然有足够的函数来匹配每个函数 $f: X \rightarrow Um$ 。

我们将从集合 X 开始构造幺半群 FX ，并逐步添加更多元素。我们将初始集合 X 称为 FX 的生成元。我们将从原始函数 f 开始构造一个幺半群态射 $g: FX \rightarrow m$ ，并将其扩展到更多的元素上。

在生成元上， $x \in X$ ， g 与 f 相同：

$$gx = fx$$

由于 FX 应该是一个幺半群，它必须有一个单位元。我们不能选择一个生成元作为单位元，因为它会对 g 的已经由 f 固定的部分施加约束——它必须将其映射到 m 的单位元 e' 。因此，我们只需在 FX 中添加一个额外的元素 e ，并将其称为单位元。我们将通过说它被映射到 m 的单位元 e' 来定义 g 在它上的作用：

$$ge = e'$$

我们还必须在 FX 中定义幺半群乘法。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，因为这将再次约束 g 的已经由 f 固定的部分——乘积必须被映射到乘积。因此，我们必须使所有生成元的乘积成为 FX 的新元素。同样， g 在这些乘积上的作用是固定的：

$$g(a \cdot b) = ga \cdot gb$$

继续这个构造，任何新的乘法都会产生 FX 的新元素，除非它可以通过应用幺半群定律简化为现有元素。例如，新的单位元 e 乘以生成元 a 必须等于 a 。但我们已经确保 e 被映射到 m 的单位元，因此乘积 $ge \cdot ga$ 自动等于 ga 。

另一种看待这个构造的方式是将集合 X 视为一个字母表。 FX 的元素就是来自这个字母表的字符串。生成元是单字母字符串，如“ a ”、“ b ”等。单位元是空字符串“”。乘法是字符串连接，因此“ a ”乘以“ b ”是一个新字符串“ ab ”。连接自动是结合的和单位的，空字符串作为单位元。

自由函子的直觉是它们“自由地”生成结构，即“没有额外的约束”。它们也以惰性的方式生成：它们不执行操作，而是记录操作。它们创建了可以在以后由特定解释器执行的通用领域特定程序。

自由幺半群“记住稍后执行乘法”。它将乘法的参数存储在字符串中，但不执行乘法。它只能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元的命令。由于结合性，它也可以“跳过括号”。

Exercise 10.9.1. 自由幺半群伴随 $F \dashv U$ 的单位元和余单位元是什么？

编程中的自由幺半群

在 Haskell 中，幺半群使用以下类型类定义：

```
class Monoid m where
  mappend :: m -> m -> m
  mempty  :: m
```

这里，`mappend` 是乘积映射的柯里化形式： $(m, m) \rightarrow m$ 。`mempty` 元素对应于从终端对象（幺半群范畴的单位元）的箭头，或者简单地说是 `m` 的一个元素。

由某个类型 `a` 生成的自由幺半群，作为生成元的集合，由列表类型 `[a]` 表示。空列表作为单位元；幺半群乘法实现为列表连接，传统上以中缀形式书写：

```
(++) :: [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 = (++)
```

为了证明它是一个自由幺半群，我们必须能够从 `a` 的列表构造一个幺半群态射到任意幺半群 `m`，前提是我们有一个（无约束的）从 `a` 到（底层集合）`m` 的映射。我们无法在 Haskell 中表达所有这些，但我们可以定义函数：

```
foldMap :: Monoid m => (a -> m) -> ([a] -> m)
foldMap f = foldr mappend mempty . fmap f
```

这个函数使用 `f` 将列表的元素转换为幺半群值，然后使用 `mappend` 从单位元 `mempty` 开始折叠它们。

很容易看出，空列表被映射到幺半群单位元。不难看出，两个列表的连接被映射到结果的幺半群乘积。因此，`foldMap` 确实是一个幺半群态射。

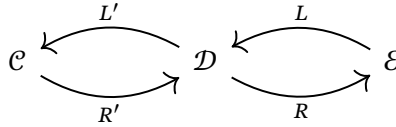
根据自由幺半群作为乘法操作的领域特定程序的直觉，`foldMap` 提供了这个程序的解释器。它执行所有被推迟的乘法。请注意，同一个程序可以根据具体幺半群和函数 `f` 的选择以多种不同的方式解释。

我们将在关于代数的章节中再次讨论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解释它：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 伴随关系的范畴

我们可以通过利用定义伴随关系的函子的复合来定义伴随关系的复合。两个伴随关系 $L \dashv R$ 和 $L' \dashv R'$ 是可复合的，如果它们共享中间的范畴：



通过复合函子，我们得到一个新的伴随关系 $(L' \circ L) \dashv (R \circ R')$ 。

实际上，让我们考虑 **hom**-集：

$$\mathcal{C}(L'(Le), c)$$

利用 $L' \dashv R'$ 伴随关系，我们可以将 L' 转置到右边，它变为 R' ：

$$\mathcal{D}(Le, R'c)$$

然后利用 $L \dashv R$ ，我们可以类似地转置 L ：

$$\mathcal{E}(e, R(R'c))$$

结合这两个同构，我们得到复合的伴随关系：

$$\mathcal{C}((L' \circ L)e, c) \cong \mathcal{E}(e, (R \circ R')c)$$

由于函子的复合是结合的，伴随关系的复合也是结合的。容易看出，一对恒等函子形成一个平凡的伴随关系，它在伴随关系的复合中充当恒等元。因此，我们可以定义一个范畴 **Adj(Cat)**，其中对象是范畴，箭头是伴随关系（按照惯例，指向左伴随的方向）。

伴随关系可以纯粹用函子和自然变换来定义，即 2-范畴 **Cat** 中的 1-细胞和 2-细胞。**Cat** 并没有什么特别之处，事实上，伴随关系可以在任何 2-范畴中定义。此外，伴随关系的范畴本身也是一个 2-范畴。

10.11 抽象层次

范畴论 (Category theory) 是关于知识结构化的理论。特别是，它可以应用于范畴论自身的知识。因此，我们在范畴论中看到了大量抽象层次的混合。我们在一个层次上看到的结构可以被分组为更高层次的结构，这些结构展现出更高层次的结构，依此类推。

在编程中，我们习惯于构建抽象层次结构。值被分组为类型，类型被分组为种类 (kinds)。操作值的函数与操作类型的函数被区别对待。我们经常使用不同的语法来区分抽象层次。但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 (discrete category)。集合的元素是这个范畴的对象，除了必须的恒等态射 (identity morphisms) 外，它们之间没有箭头。

然后，同一个集合可以被视为范畴 **Set** 中的一个对象。这个范畴中的箭头是集合之间的函数。

范畴 **Set** 本身又是范畴 **Cat** 中的一个对象。**Cat** 中的箭头是函子（functors）。

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。这个范畴中的箭头是自然变换（natural transformations）。

我们可以在函子范畴、积范畴（product categories）、对偶范畴（opposite categories）等之间定义函子，如此无限延伸。

完成这个循环，每个范畴中的 hom-集（hom-sets）都是集合。我们可以在它们之间定义映射和同构（isomorphisms），跨越不同的范畴。伴随（adjunctions）之所以可能，是因为我们可以比较存在于不同范畴中的 hom-集。

代数

代数的本质是对表达式进行形式化操作。但什么是表达式，我们又如何操作它们呢？

首先，观察像 $2(x + y)$ 或 $ax^2 + bx + c$ 这样的代数表达式，我们会发现它们有无限多个。虽然构建它们的规则是有限的，但这些规则可以以无限多种组合方式使用。这表明这些规则是递归使用的。

在编程中，表达式几乎等同于（解析）树。考虑以下算术表达式的简单示例：

```
data Expr = Val Int
          | Plus Expr Expr
```

这是一个构建树的配方。我们首先使用 **Val** 构造函数构建小树。然后我们将这些幼苗种植到节点中，依此类推。

```
e2 = Val 2
e3 = Val 3
e5 = Plus e2 e3
e7 = Plus e5 e2
```

这样的递归定义在编程语言中非常有效。问题是，每个新的递归数据结构都需要自己的函数库来操作它。

从类型理论的角度来看，我们已经能够通过提供特定的引入和消除规则来定义递归类型，例如自然数或列表。我们需要的是更一般的东西，一种从更简单的可插拔组件生成任意递归类型的程序。

在处理递归数据结构时，有两个正交的关注点。一个是递归的机制，另一个是可插拔的组件。

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树种植到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。

11.1 自函子的代数

将较小的树种植到节点中的想法要求我们形式化具有“洞”的数据结构的含义——即“容器的容器”。这正是函子的用途。因为我们希望递归地使用这些函子，所以它们必须是自函子。

例如，我们之前示例中的自函子将由以下数据结构定义，其中 `x` 标记了位置：

```
data ExprF x = ValF Int
              | PlusF x x
```

关于所有可能表达式形状的信息被抽象为一个单一的函子。

在定义代数时，另一个重要的信息是评估表达式的配方。这也可以使用相同的自函子进行编码。

递归地思考，假设我们知道如何评估较大表达式的所有子树。那么剩下的步骤就是将这些结果插入到顶层节点并对其进行评估。

例如，假设函子中的 `x` 被替换为整数——子树评估的结果。很明显，在最后一步我们应该做什么。如果树的顶部只是一个叶子 `ValF`（这意味着没有子树需要评估），我们将只返回其中存储的整数。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
```

我们基于常识做出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，我们假设表达式应该评估为一个整数。但同样合理的评估器可以通过将表达式转换为字符串来美化打印它。这个评估器使用连接而不是加法：

```
pretty :: ExprF String -> String
pretty (ValF n)      = show n
pretty (PlusF s t)   = s ++ " + " ++ t
```

事实上，有无限多个评估器，有些合理，有些则不太合理，但我们不应该进行评判。任何目标类型的选择和任何评估器的选择都应该是同样有效的。这导致了以下定义：

自函子 F 的代数是一个对 (c, α) 。对象 c 被称为代数的载体，评估器 $\alpha: Fc \rightarrow c$ 被称为结构映射。

在 Haskell 中，给定函子 `f`，我们定义：

```
type Algebra f c = f c -> c
```

请注意，评估器不是一个多态函数。它是针对特定类型 `c` 的特定函数选择。对于给定的类型，可能有多种载体类型的选择，也可能有许多不同的评估器。它们都定义了不同的代数。

我们之前为 `ExprF` 定义了两个代数。这个以 `Int` 为载体：

```
eval :: Algebra ExprF Int
eval (ValF n)    = n
eval (PlusF m n) = m + n
```

而这个以 `String` 为载体：

```
pretty :: Algebra ExprF String
pretty (ValF n)    = show n
pretty (PlusF s t) = s ++ " + " ++ t
```

11.2 代数范畴

对于一个给定的自函子 F ，其代数构成一个范畴。该范畴中的箭头是代数同态（algebra morphism），即它们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。这正是函子性发挥作用的地方。为了从一个结构映射切换到另一个结构映射，我们必须能够提升它们载体之间的箭头。

给定一个自函子 F ，两个代数 (a, α) 和 (b, β) 之间的代数同态是一个箭头 $f: a \rightarrow b$ ，使得以下图表交换：

$$\begin{array}{ccc} Fa & \xrightarrow{Ff} & Fb \\ \downarrow \alpha & & \downarrow \beta \\ a & \xrightarrow{f} & b \end{array}$$

换句话说，以下等式必须成立：

$$f \circ \alpha = \beta \circ Ff$$

两个代数同态的复合仍然是代数同态，这可以通过将两个这样的图表粘贴在一起来看出（函子将复合映射到复合）。恒等箭头也是代数同态，因为

$$id_a \circ \alpha = \alpha \circ F(id_a)$$

（函子将恒等映射到恒等）。

代数同态定义中的交换条件非常严格。例如，考虑一个将整数映射到字符串的函数。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` 的代数同态。

Exercise 11.2.1. 证明 `show` 不是代数同态。提示：考虑 `PlusF` 节点会发生什么。

初始代数

对于一个给定函子 F 的代数范畴中的初始对象称为初始代数，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) 有一个唯一的代数同态 f 从它到任何其他代数 (a, α) 。用图表表示为：

$$\begin{array}{ccc} Fi & \xrightarrow{Ff} & Fa \\ \downarrow \iota & & \downarrow \alpha \\ i & \xrightarrow{f} & a \end{array}$$

这个唯一的同态称为代数 (a, α) 的 *catamorphism*。它有时使用“香蕉括号”写成 $\langle \alpha \rangle$ 。

Exercise 11.2.2. 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
```

给出一个令人信服的论证，说明 `log`（对数）是这两个代数之间的代数同态。（`Float` 是内置的浮点数类型。）

11.3 Lambek 引理与不动点

Lambek 引理指出，初始代数（initial algebra）的结构映射 ι 是一个同构。

其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) ，结果 $(Fa, F\alpha)$ 也是一个代数，其结构映射为 $F\alpha : F(Fa) \rightarrow Fa$ 。

特别地，如果你提升初始代数 (i, ι) ，你会得到一个以 Fi 为载体、结构映射为 $F\iota: F(Fi) \rightarrow Fi$ 的新代数。由此可知，必然存在一个从初始代数到这个新代数的唯一代数态射：

$$\begin{array}{ccc} Fi & \xrightarrow{Fh} & F(Fi) \\ \downarrow \iota & & \downarrow F\iota \\ i & \xrightarrow{h} & Fi \end{array}$$

这里的 h 就是 ι 的逆。为了验证这一点，让我们考虑复合映射 $\iota \circ h$ 。它是下图底部的箭头：

$$\begin{array}{ccccc} Fi & \xrightarrow{Fh} & F(Fi) & \xrightarrow{F\iota} & Fi \\ \downarrow \iota & & \downarrow F\iota & & \downarrow \iota \\ i & \xrightarrow{h} & Fi & \xrightarrow{\iota} & i \end{array}$$

这是将原始图表与一个显然交换的图表拼接在一起的结果。因此整个矩形是交换的。我们可以将其解释为 $\iota \circ h$ 是从 (i, ι) 到其自身的一个代数态射。但已经存在这样的代数态射——即恒等映射。因此，根据从初始代数出发的映射的唯一性，这两者必须相等：

$$\iota \circ h = id_i$$

知道这一点后，我们现在可以回到之前的图表，它表明：

$$h \circ \iota = F\iota \circ Fh$$

由于 F 是一个函子，它将复合映射映射为复合映射，将恒等映射映射为恒等映射。因此右边等于：

$$F(\iota \circ h) = F(id_i) = id_{Fi}$$

我们由此证明了 h 是 ι 的逆，这意味着 ι 是一个同构。换句话说：

$$Fi \cong i$$

我们将这个等式解释为 i 是 F 的一个不动点（在同构意义下）。 F 在 i 上的作用“不会改变它”。

可能存在许多不动点，但这是最小不动点，因为存在一个从它到任何其他不动点的代数态射。自函子 F 的最小不动点记为 μF ，因此我们写作：

$$i = \mu F$$

Haskell 中的不动点

让我们考虑不动点的定义如何与我们最初由自函子给出的例子一起工作：

```
data ExprF x = ValF Int | PlusF x x
```

它的不动点是一个数据结构，其性质是`ExprF`作用于它时会重现它。如果我们称这个不动点为`Expr`，则不动点方程变为（在伪 Haskell 中）：

```
Expr = ExprF Expr
```

展开`ExprF`我们得到：

```
Expr = ValF Int | PlusF Expr Expr
```

将其与递归定义（实际的 Haskell）进行比较：

```
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（甚至只是一个类型构造器）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。它只适用于那些具有“叶子”成分的函子。

让我们称`Fix f`为函子`f`的不动点。符号上，不动点方程可以写成：

$$f(\text{Fix } f) \cong \text{Fix } f$$

或者，在代码中，

```
data Fix f where
  In :: f (Fix f) -> Fix f
```

数据构造器`In`正是初始代数的结构映射，其载体是`Fix f`。它的逆是：

```
out :: Fix f -> f (Fix f)
out (In x) = x
```

Haskell 标准库包含一个更地道的定义：

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

为了创建类型为`Fix f`的项，我们经常使用“智能构造器”。例如，对于`ExprF`函子，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)
```

并使用它来生成如下的表达式树：

```
e9 :: Fix ExprF
e9 = plus (plus (val 2) (val 3)) (val 4)
```

11.4 Catamorphisms (Catamorphisms)

作为程序员，我们的目标是能够对递归数据结构执行计算——即“折叠”它。现在我们已经具备了所有要素。

数据结构被定义为一个函子的不动点。该函子的代数定义了我们要执行的操作。我们已经在以下图表中看到了不动点和代数的结合：

$$\begin{array}{ccc} Fi & \xrightarrow{Ff} & Fa \\ \downarrow \iota & & \downarrow \alpha \\ i & \dashrightarrow f & a \end{array}$$

该图表定义了代数 (a, α) 的 catamorphism f 。

最后一条信息是 Lambek 引理，它告诉我们 ι 可以被反转，因为它是一个同构。这意味着我们可以将这个图表解读为：

$$f = \alpha \circ Ff \circ \iota^{-1}$$

并将其解释为 f 的递归定义。

让我们使用 Haskell 符号重新绘制这个图表。Catamorphism 依赖于代数，因此对于载体为 `a` 且求值器为 `alg` 的代数，我们将得到 catamorphism `cata alg`。

$$\begin{array}{ccc} f \text{ (Fix f)} & \xrightarrow{\text{fmap (cata alg)}} & f \text{ a} \\ \uparrow \text{out} & & \downarrow \text{alg} \\ \text{Fix f} & \dashrightarrow \text{cata alg} & a \end{array}$$

通过简单地跟随箭头，我们得到了这个递归定义：

```
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

以下是发生的事情：我们将这个定义应用于某个 `Fix f`。每个 `Fix f` 都是通过将 `In` 应用于一个包含 `Fix f` 的函子得到的：

```
data Fix f where
  In :: f (Fix f) -> Fix f
```

函数 `out` “剥离”数据构造函数 `In`：

```
out :: Fix f -> f (Fix f)
out (In x) = x
```

我们现在可以通过在其上 `fmap'ing cata alg` 来评估包含 `Fix f` 的函子。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。当它到达叶子时，递归终止。

在这一步之后，我们剩下一个包含值的函子，我们对其应用求值器 `alg`，以得到最终结果。

这种方法的强大之处在于所有的递归都封装在一个数据类型和一个库函数中：我们有 `Fix` 的定义和 `cata-morphism cata`。库的客户端只提供 非递归的部分：函子和代数。这些部分更容易处理。我们将一个复杂的问题分解为更简单的组件。

示例

我们可以立即将此构造应用于我们之前的示例。你可以验证：

```
cata eval e9
```

计算结果为 9，而

```
cata pretty e9
```

计算结果为字符串 `"2 + 3 + 4"`。

有时我们希望以缩进的方式在多行上显示树。这需要将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) ' '
```

以下代码的结果：

```
cata pretty' e9 0
```

打印出来如下所示：

```

    2
  +
    3
+
    4

```

让我们尝试为其他熟悉的函子定义代数。**Maybe** 函子的不动点：

```
data Maybe x = Nothing | Just x
```

经过一些重命名后，等价于自然数类型：

```
data Nat = Z | S Nat
```

该函子的代数由载体 **a** 的选择和求值器组成：

```
alg :: Maybe a -> a
```

从 **Maybe** 的映射由两件事决定：对应于 **Nothing** 的值和对应于 **Just** 的函数 **a->a**。在我们讨论自然数类型时，我们称这些为 **init** 和 **step**。我们现在可以看到，**Nat** 的消去规则是该代数的 **catamorphism**。

列表作为初始代数

我们之前看到的列表类型等价于以下函子的不动点，该函子由列表内容的类型 **a** 参数化：

```
data ListF a x = NilF | ConsF a x
```

该函子的代数是一个映射：

```
alg :: ListF a c -> c
alg NilF = init
alg (ConsF a c) = step (a, c)
```

它由值 **init** 和函数 **step** 决定：

```
init :: c
step :: (a, c) -> c
```

这种代数的 **catamorphism** 是列表递归器：

```
recList :: c -> ((a, c) -> c) -> (List a -> c)
```

其中 `(List a)` 可以识别为不动点 `Fix (ListF a)`。

我们之前见过一个递归函数，它反转一个列表。它是通过将元素附加到列表的末尾来实现的，这非常低效。使用 `catamorphism` 重写这个函数很容易，但问题仍然存在。

另一方面，前置元素是廉价的。一个更好的算法将遍历列表，将元素累积在一个先进先出的队列中，然后逐个弹出它们并将它们前置到一个新列表中。

队列机制可以使用闭包的组合来实现：每个闭包都是一个记住其环境的函数。以下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，这个代数都会创建一个新函数。这个函数在执行时，会将前一个函数 `f` 应用于一个列表，该列表是将当前元素 `a` 前置到函数的参数 `as` 的结果。生成的闭包记住了当前元素 `a` 和前一个函数 `f`。

该代数的 `catamorphism` 累积了这些闭包的队列。要反转一个列表，我们将该代数的 `catamorphism` 的结果应用于空列表：

```
reverse :: Fix (ListF a) -> [a]
reverse as = (cata revAlg as) []
```

这个技巧是左折叠函数 `foldl` 的核心。使用它时应小心，因为存在栈溢出的风险。

列表非常常见，以至于它们的消去器（称为“折叠”）被包含在标准库中。但存在无限多种可能的递归数据结构，每种数据结构由其自己的函子生成，我们可以在所有这些数据结构上使用相同的 `catamorphism`。

值得一提的是，列表构造在任何具有余积的幺半群范畴中都有效。我们可以用更一般的函子替换列表函子：

$$Fx = I + a \otimes x$$

其中 I 是单位对象， \otimes 是张量积。不动点方程的解：

$$L_a \cong I + a \otimes L_a$$

可以正式写为一个级数：

$$L_a = I + a + a \otimes a + a \otimes a \otimes a + \dots$$

我们将其解释为列表的定义，它可以是空的 I ，单元素 a ，双元素列表 $a \otimes a$ ，依此类推。

顺便说一句，如果你仔细观察，这个解可以通过一系列形式变换得到：

$$\begin{aligned}
 L_a &\cong I + a \otimes L_a \\
 L_a - a \otimes L_a &\cong I \\
 (I - a) \otimes L_a &\cong I \\
 L_a &\cong I / (I - a) \\
 L_a &\cong I + a + a \otimes a + a \otimes a \otimes a + \dots
 \end{aligned}$$

其中最后一步使用了几何级数的求和公式。诚然，中间步骤没有意义，因为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，它可以通过考虑对象链的余极限来严格化。

11.5 从普遍性看初始代数

另一种看待初始代数的方式，至少在 **Set** 中，是将其视为一组 **catamorphisms** (范畴态射)，这些 **catamorphisms** 作为一个整体暗示了底层对象的存在。我们不是将 μF 看作一组树，而是将其看作从代数到其载体的函数集合。

在某种程度上，这只是 **Yoneda** 引理的另一种表现形式：每个数据结构都可以通过映射进入或映射出来描述。在这种情况下，映射进入的是递归数据结构的构造函数，而映射出来的是可以应用于它的所有 **catamorphisms**。

首先，让我们在 **cata** 的定义中显式地表示多态性：

```
cata :: Functor f => forall a. Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

然后交换参数。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** 反转了函数的参数顺序：

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f b a = f a b
```

这给了我们一个从 **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，我们可以重建 `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` 互为逆函数，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)
```

现在我们可以用 `Mu f` 替换所有使用 `Fix f` 的地方。

在 `Mu f` 上进行折叠很容易，因为 `Mu` 本身携带了它自己的一组 catamorphisms：

```
cataMu :: Algebra f a -> (Mu f -> a)
cataMu alg (Mu h) = h alg
```

你可能会想知道如何为列表等类型构造 `Mu f` 的项。这可以通过递归来完成：

```
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，你必须使用语言编译指示：

```
{-# language ScopedTypeVariables #-}
```

这将类型变量 `a` 置于 `where` 子句的作用域内。

Exercise 11.5.1. 编写一个测试，该测试接受一个整数列表，将其转换为 `Mu` 形式，并使用 `cataMu` 计算总和。

11.6 初始代数作为余极限

一般来说，无法保证代数范畴中的初始对象存在。但如果它存在，Lambek 引理告诉我们，它是这些代数的自函子的不动点。这个不动点的构造有些神秘，因为它涉及递归的“打结”。

粗略地说，不动点是在我们无限次应用函子后达到的。然后，再应用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来看，这个想法可以变得精确。为了简单起见，让我们考虑集合范畴中的代数，它具有所有良好的性质。

在我们的例子中，我们已经看到，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。

如果我们将函子 F 应用于初始对象——空集 0 ，我们可以将它们提取出来。由于空集没有元素，类型 $F0$ 的实例只能是叶子。

事实上，类型 `Maybe Void` 的唯一居民是使用 `Nothing` 构造的。类型 `ExprF Void` 的唯一居民是 `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，只有一个叶子。这个函子的叶子类型是一个单例：

```
m1 :: Maybe Void
m1 = Nothing
```

在第二次迭代中，我们将 F 应用于上一步的叶子，得到深度最多为二的树。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
```

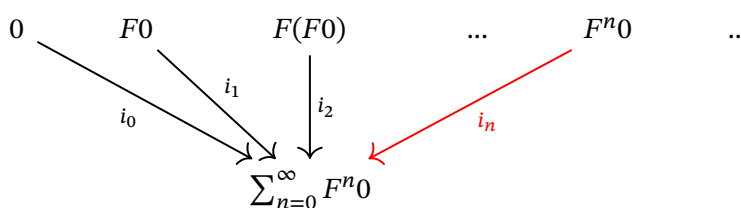
我们可以继续这个过程，在每一步添加越来越深的树。在第 n 次迭代中，类型 F^n0 (F 对初始对象的 n 次应用) 描述了所有深度不超过 n 的树。然而，对于每个 n ，仍然有无限多个深度大于 n 的树没有被覆盖。

如果我们知道如何定义 $F^\infty0$ ，我们将覆盖所有可能的树。我们可以尝试的次优方法是将所有部分树相加，并构造一个无限和类型。就像我们定义了两个类型的和一样，我们可以定义多个类型的和，包括无限多个。

一个无限和（余积）：

$$\sum_{n=0}^{\infty} F^n0$$

就像有限和一样，只是它有无限多个构造器 i_n ：



它具有通用的映射出性质，就像两个类型的和一样，只是有无限多个情况。（显然，我们无法在 Haskell 中表达它。）

要构造一个深度为 n 的树，我们首先从 $F^n 0$ 中选择它，并使用第 n 个构造器 i_n 将其注入到和中。

只有一个问题：相同的树形也可以通过任何 $F^m 0$ 构造，其中 $m > n$ 。

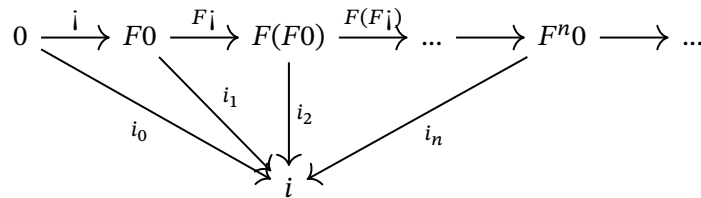
事实上，我们已经看到叶子 **Nothing** 出现在 **Maybe Void** 和 **Maybe (Maybe Void)** 中。事实上，它出现在 **Maybe** 作用于 **Void** 的任何非零幂中。

类似地，**Just Nothing** 出现在所有从二开始的幂中。

Just (Just (Nothing)) 出现在所有从三开始的幂中，依此类推...

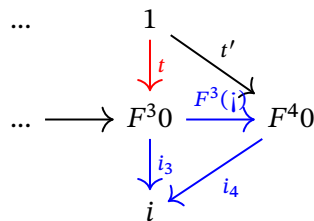
但有一种方法可以消除所有这些重复。诀窍是用余极限代替和。我们可以构造一个链（这样的链称为 ω -链），而不是由离散对象组成的图。让我们称这个链为 Γ ，它的余极限为 i ：

$$i = \text{Colim } \Gamma$$



它几乎与和相同，但在余锥的底部有额外的箭头。这些箭头是从初始对象到 $F0$ 的唯一箭头 i 的累积提升（我们在 Haskell 中称它为 **absurd**）。这些箭头的效果是将同一树的无限多个副本折叠成一个代表。

要看到这一点，考虑一个深度为 3 的树。它首先可以作为 $F^3 0$ 的一个元素找到，也就是说，作为一个箭头 $t: 1 \rightarrow F^3 0$ 。它被注入到余极限 i 中，作为复合 $i_3 \circ t$ 。



相同的树形也可以在 $F^4 0$ 中找到，作为复合 $t' = F^3(i) \circ t$ 。它被注入到余极限中，作为复合 $i_4 \circ t' = i_4 \circ F^3(i) \circ t$ 。

然而，这次我们有一个交换三角形——余锥的面：

$$i_4 \circ F^3(i) = i_3$$

这意味着：

$$i_4 \circ t' = i_4 \circ F^3(i) \circ t = i_3 \circ t$$

两个树的副本在余极限中被识别。你可以相信这个过程消除了所有重复。

证明

我们可以直接证明 $i = \text{Colim } \Gamma$ 是初始代数。然而，我们必须做一个假设：函子 F 必须保持 ω -链的余极限。 $F\Gamma$ 的余极限必须等于 Fi 。

$$\text{Colim}(F\Gamma) \cong Fi$$

幸运的是，这个假设在 **Set** 中成立¹。

以下是证明的概要：为了证明同构，我们首先构造一个箭头 $i \rightarrow Fi$ ，然后构造一个箭头 $\iota: Fi \rightarrow i$ 。我们将跳过它们互为逆的证明。然后，我们通过构造一个到任意代数的 **catamorphism** 来展示 (i, ι) 的通用性。

所有后续的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。根据通用性，必须有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映射 $i \rightarrow Fi$ 。如果我们能构造一个从链 Γ 到 $\text{Colim}(F\Gamma)$ 的余锥，那么根据通用性，必须有一个从 i 到 $\text{Colim}(F\Gamma)$ 的箭头。后者，根据我们的假设 F 保持余极限，同构于 Fi 。因此，我们将有一个映射 $i \rightarrow Fi$ 。

为了构造这个余锥，首先注意到 $\text{Colim}(F\Gamma)$ 根据定义是余锥 $F\Gamma$ 的顶点。

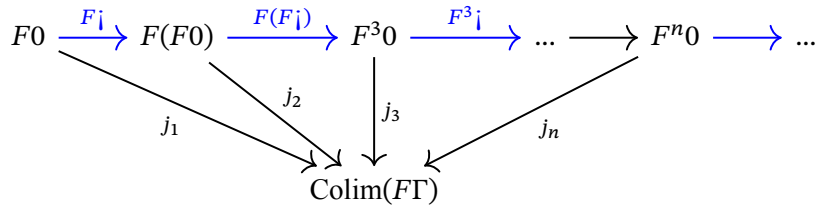
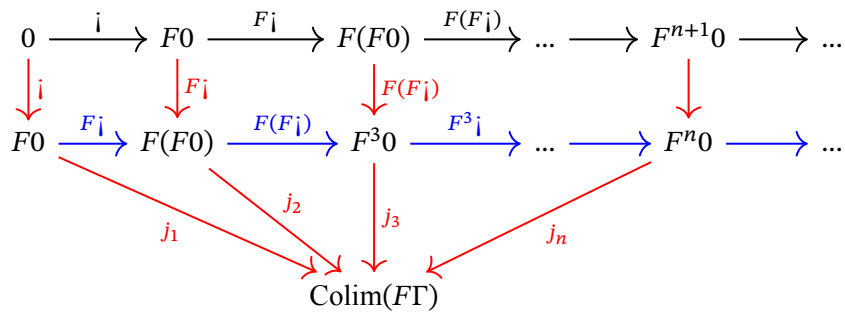


图 $F\Gamma$ 与 Γ 相同，只是它缺少链开头的初始对象。

我们正在寻找的从 Γ 到 $\text{Colim}(F\Gamma)$ 的余锥的辐条在下图中用红色标记：

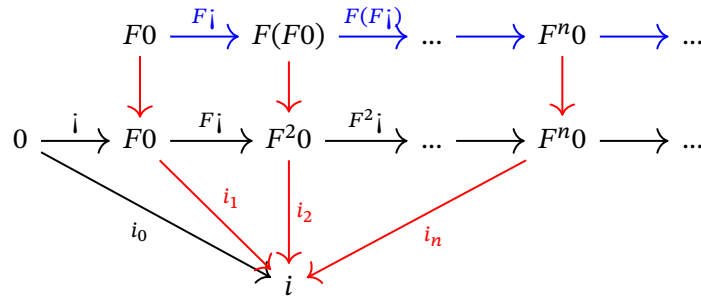


由于 $i = \text{Colim } \Gamma$ 是基于 Γ 的通用余锥的顶点，必须有一个从它到 $\text{Colim}(F\Gamma)$ 的唯一映射，正如我们所说，它等于 Fi 。这就是我们正在寻找的映射：

$$i \rightarrow Fi$$

¹这是 **Set** 中的余极限由集合的不交并构建的结果。

接下来, 注意到链 $F\Gamma$ 是 Γ 的一个子链, 因此可以嵌入其中。这意味着我们可以通过 (Γ 的一个子链) 构造一个从 $F\Gamma$ 到顶点 i 的余锥 (下面的红色箭头)。



根据 $\text{Colim}(F\Gamma)$ 的通用性, 存在一个映射出

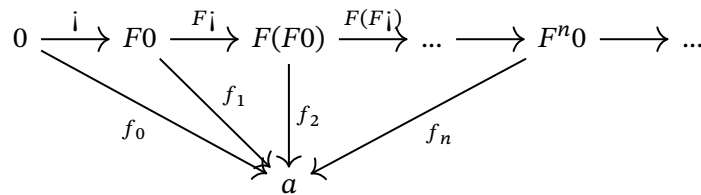
$$\text{Colim}(F\Gamma) \rightarrow i$$

因此, 我们有另一个方向的映射:

$$\iota: Fi \rightarrow i$$

这表明 i 是一个代数的载体。事实上, 可以证明这两个映射互为逆, 正如我们从 Lambek 引理所期望的那样。

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 $(a, \alpha: Fa \rightarrow a)$ 的映射。同样, 我们可以使用通用性, 只要我们能构造一个从 Γ 到 a 的余锥。



这个余锥的第零个辐条从 0 到 a , 所以它只是 $f_0 = i$ 。

第一个辐条, $F0 \rightarrow a$, 是 $f_1 = \alpha \circ Ff_0$, 因为 $Ff_0: F0 \rightarrow Fa$ 和 $\alpha: Fa \rightarrow a$ 。

第三个辐条, $F(F0) \rightarrow a$ 是 $f_2 = \alpha \circ Ff_1$ 。依此类推...

从 i 到 a 的唯一映射就是我们的 **catamorphism**。通过更多的图追踪, 可以证明它确实是一个代数态射。

请注意, 这个构造只有在我们可以通过创建函子的叶子来“启动”过程时才有效。另一方面, 如果 $F0 \cong 0$, 那么没有叶子, 所有进一步的迭代将继续复制 0 。

Chapter 12

余代数

余代数 (Coalgebras) 只是对偶范畴中的代数。本章结束！

好吧，也许还没完... 正如我们之前所见，我们所工作的范畴在对偶性方面并不对称。特别是，如果我们比较终对象和始对象，它们的性质并不对称。我们的始对象没有进入的箭头，而终对象除了有唯一的进入箭头外，还有许多出去的箭头。

由于始代数是始对象开始构造的，我们可能会期望终余代数——它们是始代数的对偶，因此从终对象生成——不仅仅是它们的镜像，而是会添加它们自己有趣的变化。

我们已经看到，代数的主要应用是处理递归数据结构：在折叠它们时。对偶地，余代数的主要应用是生成或展开递归的、树状的数据结构。展开是使用变形 (anamorphism) 完成的。

我们使用折叠 (catamorphism) 来砍树，我们使用变形来种树。

我们不能无中生有地产生信息，因此一般来说，折叠和变形都倾向于减少其输入中包含的信息量。

在你对一个整数列表求和后，无法恢复原始列表。

同样地，如果你使用变形生成一个递归数据结构，种子必须包含最终出现在树中的所有信息。你不会获得新的信息，但优势在于你现在拥有的信息以一种更方便进一步处理的形式存储。

12.1 自函子的余代数

自函子 F 的余代数是一个由载体 a 和结构映射组成的对：一个箭头 $a \rightarrow Fa$ 。

在 Haskell 中，我们定义：

```
type Coalgebra f a = a -> f a
```

我们通常将载体视为种子的类型，从中我们可以生成数据结构，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)
```

我们甚至不必为它定义 `Functor` 的实例——`deriving` 子句告诉编译器为我们生成规范的实例（连同 `Show` 实例，如果我们想显示它，允许转换为 `String`）。

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。这些新种子可以递归地用于生成子树。

这是函子 `TreeF` 的一个余代数，它接受一个整数列表作为种子：

```
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，它生成一个叶子；否则它创建一个新节点。该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（这里是 `(<= n)`，小于或等于 `n`）拆分列表。结果是一对列表：第一个满足谓词；第二个不满足。

你可以确信，递归应用这个余代数会创建一个二叉排序树。我们稍后将使用这个余代数来实现排序。

12.2 余代数的范畴

通过与代数同态的类比，我们可以将余代数同态定义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) ，箭头 $f : a \rightarrow b$ 是一个余代数同态，如果下面的图表交换：

$$\begin{array}{ccc} a & \xrightarrow{f} & b \\ \downarrow \alpha & & \downarrow \beta \\ Fa & \xrightarrow{Ff} & Fb \end{array}$$

其解释是，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 然后使用提升 Ff 将箭头应用于其内容，结果都是相同的。

余代数同态可以组合，且恒等箭头自动是一个余代数同态。很容易看出，余代数与代数一样，形成了一个范畴。

然而，这次我们感兴趣的是这个范畴中的终端对象——终端余代数。如果存在一个终端余代数 (t, τ) ，它满足 `Lambek` 引理的对偶。

Exercise 12.2.1. *Lambek* 引理：证明终端余代数 (t, τ) 的结构映射 τ 是一个同构。提示：证明与初始代数的证明对偶。

作为 *Lambek* 引理的一个结果，终端代数的载体是所讨论的自函子的一个不动点。

$$Ft \cong t$$

其中 τ 和 τ^{-1} 作为这个同构的见证。

同样可以得出 (t, τ^{-1}) 是一个代数；正如 (i, ι^{-1}) 是一个余代数，假设 (i, ι) 是初始代数。

我们之前已经看到，初始代数的载体是一个不动点。原则上，同一个自函子可能有许多不动点。初始代数是最大不动点，而终端余代数是最大不动点。

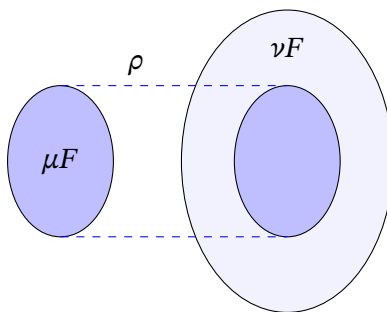
自函子 F 的最大不动点记为 νF ，因此我们有：

$$t = \nu F$$

我们还可以看到，从初始代数到终端余代数必须存在唯一的代数同态（一个 *catamorphism*）。这是因为终端余代数也是一个代数。

类似地，从初始代数（也是一个余代数）到终端余代数存在唯一的余代数同态。事实上，可以证明在这两种情况下，它是相同的底层同态 $\rho: \mu F \rightarrow \nu F$ 。

在集合范畴中，初始代数的载体集是终端余代数的载体集的一个子集，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，情况更加微妙。但是，至少对于具有叶子分量的函子——即它们在初始对象上的作用是非平凡的——*Haskell* 的不动点类型既可以作为初始代数的载体，也可以作为终端余代数的载体。

```
data Fix f where
  In :: f (Fix f) -> Fix f
```

Exercise 12.2.2. 证明，对于 *Set* 中的恒等函子，每个对象都是一个不动点，空集是最小不动点，单例集是最大不动点。提示：最小不动点必须有箭头指向所有其他不动点，而最大不动点必须有箭头来自所有其他不动点。

Exercise 12.2.3. 证明空集是 **Set** 中恒等函子的初始代数的载体。对偶地，证明单例集是这个函子的终端余代数。提示：证明唯一的箭头确实是（余）代数同态。

12.3 变形 (Anamorphisms)

终余代数 (terminal coalgebra) (t, τ) 由其泛性质定义：对于任何余代数 (a, α) ，存在唯一的余代数态射 h 到 (t, τ) 。这个态射被称为变形 (anamorphism)。作为余代数态射，它使得以下图表交换：

$$\begin{array}{ccc} a & \xrightarrow{h} & t \\ \downarrow \alpha & & \downarrow \tau \\ Fa & \xrightarrow{Fh} & Ft \end{array}$$

就像代数的情况一样，我们可以使用 Lambek 引理来“求解” h ：

$$h = \tau^{-1} \circ Fh \circ \alpha$$

这个解被称为变形，有时使用“透镜括号”表示为 $[(\alpha)]$ 。

由于终余代数（就像初始代数）是函子的不动点，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
```

这个公式的解释如下：给定一个类型为 `a` 的种子，我们首先用余代数 `coa` 作用于它。这给我们提供了一组种子。我们通过递归地应用变形来扩展这些种子，使用 `fmap`。然后我们应用构造函数 `In` 来得到最终结果。

作为一个例子，我们可以将变形应用于之前定义的 `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变形 (catamorphism) 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
```

它将左列表与单元素枢轴和右列表连接起来。为了对列表进行排序，我们将变形与变形结合：

```
qsort = cata toList . ana split
```

这为我们提供了一个（非常低效的）快速排序实现。我们将在下一节中再次讨论它。

无限数据结构

在研究代数时，我们依赖于具有叶子组件的数据结构——即作用于初始对象时会产生不同于初始对象的结果的自函子。在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一要求。我们不再需要“手动”构造递归数据结构——我们有变形来为我们完成这项工作。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数据结构。

由于 **Haskell** 的惰性，无限数据结构是可表示的。事物是根据需要来评估的。只有那些明确需要的无限数据结构部分才会被计算；其余部分的评估保持在暂停状态。

在严格的语言中实现无限数据结构，必须将值表示为函数——这是 **Haskell** 在幕后所做的（这些函数被称为 *thunks*）。

让我们看一个简单的例子：一个无限的值流。为了生成它，我们首先定义一个函子，它看起来非常像我们用来生成列表的函子，只是它缺少叶子组件（空列表构造函数）。你可能会认出它是一个积函子，第一个组件固定为流的有效载荷：

```
data StreamF a x = StreamF a x
    deriving Functor
```

无限流是这个函子的不动点。

```
type Stream a = Fix (StreamF a)
```

这里是一个简单的余代数，它使用一个整数 `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载荷，并用 `n + 1` 作为下一个流的种子。

当以零为种子时，这个余代数的变形生成所有自然数的流。

```
allNats :: Stream Int
allNats = ana step 0
```

在非惰性语言中，这个变形将永远运行，但在 **Haskell** 中它是即时的。增量代价仅在我们想要检索某些数据时支付，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
```

12.4 合态射 (Hylomorphisms)

同态射 (anamorphism) 的输出类型是函子的不动点, 这与归约态射 (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, 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 (coalgebra) 与代数 (algebra) 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合态射:

```
qsort = hylo toList split
```

注意, 在合态射的定义中并没有不动点的痕迹。从概念上讲, 余代数用于从种子构建 (展开) 递归数据结构, 而代数用于将其折叠为类型 `b` 的值。但由于 Haskell 的惰性求值特性, 中间数据结构不必完全在内存中具体化。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会被求值, 并且一旦处理完毕, 它们就会被传递给垃圾回收器。

在 Haskell 中, 合态射是递归回溯算法的一个方便替代品, 而在命令式语言中, 这些算法很难正确实现。我们利用了设计数据结构比遵循复杂的控制流和跟踪递归算法中的位置更容易这一事实。

通过这种方式, 数据结构可以用来可视化复杂的控制流。

阻抗不匹配

我们已经看到, 在集合范畴中, 初始代数 (initial algebras) 不一定与终结余代数 (terminal coalgebras) 重合。例如, 恒等函子 (identity functor) 的空集作为初始代数的载体, 而单例集作为其终结余代数的载体。

我们还有其他没有叶组件的函子, 例如流函子 (stream functor)。这种函子的初始代数也是空集。

在 `Set` 中, 初始代数是终结余代数的子集, 合态射只能为这个子集定义。这意味着, 只有当特定余代数的同态射将我们带到这个子集时, 我们才能使用合态射。在这种情况下, 由于初始代数在终结余代数中的嵌入是单射的, 我们可以在初始代数中找到相应的元素, 并对其应用归约态射。

然而, 在 Haskell 中, 我们有一个类型 `Fix f`, 它结合了初始代数和终结余代数。这就是将 Haskell 类型简单地解释为值集合的局限性所在。

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 = hyllo add step 1
```

这是一个完全合法的 Haskell 程序，可以通过类型检查。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。它会陷入无限递归，最终耗尽计算机的资源。

这是现实生活中的计算的一个方面，仅靠集合之间的函数无法建模。某些计算机函数可能永远不会终止。

递归函数在形式化上由域理论 (*domain theory*) 描述为部分定义函数的极限。如果函数未定义某个参数值，则称其返回一个底部值 \perp 。如果我们将底部作为每个类型的特殊元素（这些类型被称为提升 (*lifted*) 类型），我们可以说我们的函数 `sumAllNats` 返回类型 `Int` 的底部。一般来说，无限类型的归约态射不会终止，因此我们可以将它们视为返回底部。

然而，应该注意的是，包含底部会使 Haskell 的范畴解释变得复杂。特别是，许多依赖于映射唯一性的通用构造不再像宣传的那样工作。

“底线”是，Haskell 代码应被视为范畴概念的说明，而不是严格证明的来源。

12.5 从普遍性看终端余代数

anamorphism 的定义可以看作是终端余代数 (terminal coalgebra) 普遍性质的一种表达。以下是其定义，其中普遍量化被显式地表达出来：

```
ana :: Functor f => forall a. Coalgebra f a -> (a -> Fix f)
ana coa = In . fmap (ana coa) . coa
```

它告诉我们的是，给定任何余代数，都存在一个从其载体到终端余代数载体 `Fix f` 的映射。根据 Lambek 引理，我们知道这个映射实际上是一个余代数同态。

让我们对这个定义进行反柯里化 (`uncurry`)：

```
ana :: Functor f => forall a. (a -> f a, a) -> Fix f
ana (coa, x) = In (fmap (curry ana coa) (coa x))
```

我们可以将这个公式作为终端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——我们称之为 `Nu f`。类型签名：

```
forall a. (a -> f a, a) -> Nu f
```

告诉我们，我们可以从一对 `(a -> f a, a)` 构造一个 `Nu f` 的元素。它看起来就像一个数据构造函数，只不过它在 `a` 上是多态的。

具有多态构造函数的数据类型被称为存在类型 (existential types)。在伪代码（不是实际的 Haskell）中，我们可以将 `Nu f` 定义为：

```
data Nu f = Nu (exists a. (Coalgebra f a, a))
```

将其与代数的最小不动点定义进行比较：

```
data Mu f = Mu (forall a. Algebra f a -> a)
```

为了构造一个存在类型的元素，我们可以选择最方便的类型——即我们拥有构造函数所需数据的类型。

例如，我们可以通过选择 `Int` 作为方便的类型，并提供以下对来构造类型 `Nu (StreamF Int)` 的项：

```
nuArgs :: (Int -> StreamF Int Int, Int)
nuArgs = (\n -> StreamF n (n+1) , 0)
```

存在数据类型的客户端不知道在其构造中使用了什么类型。他们只知道这样的类型存在——因此得名。如果他们想使用存在类型，他们必须以不敏感于其构造中所做选择的方式进行。在实践中，这意味着存在类型必须携带隐藏值的生产者和消费者。

在我们的例子中，情况确实如此：生产者只是类型为 `a` 的值，而消费者是函数 `a -> f a`。

简单来说，客户端在不知道类型 `a` 是什么的情况下，唯一能做的就是将函数应用于该值。但如果 `f` 是一个函子，他们可以做更多的事情。他们可以通过将提升后的函数应用于 `f a` 的内容来重复这个过程，依此类推。他们最终会得到包含在无限流中的所有信息。

在 Haskell 中，有几种定义存在数据类型的方式。我们可以直接使用 `anamorphism` 的反柯里化版本作为数据构造函数：

```
data Nu f where
  Nu :: forall a f. (a -> f a, a) -> Nu f
```

请注意，在 Haskell 中，如果我们显式量化一个类型，所有其他类型变量也必须被量化：在这里，它是类型构造函数 `f`（然而，`Nu f` 在 `f` 上不是存在类型，因为它是一个显式参数）。

我们也可以完全省略量化：

```
data Nu f where
  Nu :: (a -> f a, a) -> Nu f
```

这是因为不是类型构造函数参数的类型变量会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)
```

(这个需要量化 `a`。)

在撰写本文时，有一个提议要在 Haskell 中引入关键字 `exists`，以使这个定义生效：

```
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 `coend`。)

`Nu f` 的构造函数字面上就是（反柯里化的）anamorphism：

```
anaNu :: Coalgebra f a -> a -> Nu f
anaNu coa a = Nu (coa, a)
```

如果我们以 `Nu (Stream a)` 的形式给定一个流，我们可以使用访问函数访问其元素。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
```

而这个函数推进流：

```
tail :: Nu (StreamF a) -> Nu (StreamF a)
tail (Nu (unf, s)) =
  let (StreamF _ s') = unf s
  in Nu (unf, s')
```

你可以在一个无限整数流上测试它们：

```
allNats = Nu nuArgs
```

12.6 终余代数作为极限

在范畴论中，我们并不惧怕无穷——我们赋予它们意义。

从表面上看，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是没有意义的。但这个想法非常具有说服力：再应用一次 F 就像在无穷大上加一——它仍然是无穷大。因此，直观上，这是 F 的一个不动点：

$$F(F^\infty 1) \cong F^{\infty+1} 1 \cong F^\infty 1$$

为了将这种粗略的推理转化为严格的证明，我们必须驯服无穷，这意味着我们必须定义某种极限过程。

作为一个例子，让我们考虑积函子：

$$F_a x = a \times x$$

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：

$$F_a 1 = a \times 1 \cong a$$

我们可以将其想象为长度为 1 的流。我们可以继续：

$$F_a(F_a 1) = a \times (a \times 1) \cong a \times a$$

长度为 2 的流，依此类推。

这看起来很有希望，但我们需要的是一个能够结合所有这些近似的对象。我们需要一种将下一个近似与前一个近似粘合起来的方法。

回想一下，在之前的练习中，“行走箭头”图的极限。这个极限与图中的起始对象具有相同的元素。特别地，考虑单箭头图 D_1 的极限：

$$\begin{array}{ccc} & \text{Lim} D_1 & \\ \pi_0 \swarrow & & \searrow \pi_1 \\ 1 & \xleftarrow{F!} & F1 \end{array}$$

($!$ 是目标为终对象 1 的唯一态射)。这个极限与 $F1$ 具有相同的元素。类似地，双箭头图 D_2 的极限：

$$\begin{array}{ccccc} & & \text{Lim} D_2 & & \\ & \pi_0 \swarrow & & \searrow \pi_1 & \\ 1 & \xleftarrow{F!} & F1 & \xleftarrow{F!} & F(F1) \end{array}$$

与 $F(F1)$ 具有相同的元素。

我们可以继续将这个图扩展到无穷。事实证明，这个无限链的极限就是终余代数的不动点载体。

$$\begin{array}{ccccccc} & & t & & & & \\ & \pi_0 \swarrow & & \searrow \pi_1 & & \searrow \pi_n & \\ 1 & \xleftarrow{F!} & F1 & \xleftarrow{F!} & F(F1) & \xleftarrow{F(F!)} & \dots \xleftarrow{F^n!} F^n 1 \xleftarrow{F^n!} \dots \end{array}$$

这个事实的证明可以通过反转箭头从初始代数的类似证明中获得。

Chapter 13

效应

车轮、陶罐和木屋有什么共同点？它们之所以有用，是因为它们中心的空虚。

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？它们中心都有空虚。

13.1 带有副作用的编程

到目前为止，我们一直在讨论基于集合之间的函数（除了非终止情况）来建模的计算。在编程中，这样的函数被称为全函数和纯函数。

全函数为其所有参数值定义。

纯函数仅根据其参数实现，在闭包的情况下，还包括捕获的值——它无法访问外部世界，更不用说修改外部世界的能力。

然而，大多数现实世界的程序必须与外部世界交互：它们读写文件、处理网络数据包、提示用户输入数据等。大多数编程语言通过允许副作用来解决这个问题。副作用是任何破坏函数全函数性或纯函数性的东西。

不幸的是，命令式语言采用的这种散弹枪方法使得程序推理变得极其困难。在组合有副作用的计算时，必须逐个案例仔细推理副作用的组合。更困难的是，大多数副作用不仅隐藏在特定函数的实现中（而不是接口中），还隐藏在它调用的所有函数的实现中，递归地。

像 **Haskell** 这样的纯函数式语言采用的解决方案是将副作用编码在纯函数的返回类型中。令人惊讶的是，这对所有相关效应都是可能的。

其思想是，我们使用函数 $a \rightarrow f\ b$ 来代替具有副作用的 $a \rightarrow b$ 类型的计算，其中类型构造器 f 编码了适当的效应。此时对 f 没有施加任何条件。它甚至不必是 **Functor**，更不用说应用函子或单子了。如果我们愿意

实现一个单一的整体函数来生成值和副作用，我们就完成了。这个函数的调用者只需解包结果并继续愉快地进行。但编程是关于将复杂动作分解为更简单组件的能力。

以下是常见效应及其纯函数版本的列表。我们将在接下来的章节中讨论组合。

部分性

在命令式语言中，部分性通常使用异常来编码。当函数以“错误”的参数值调用时，它会抛出异常。在某些语言中，异常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** 函子中的结果的函数来实现。这样的函数在以“错误”参数调用时返回 **Nothing**，否则将结果包装在 **Just** 构造器中。

如果我们想编码更多关于失败类型的信息，可以使用 **Either** 函子，其中 **Left** 传统上传递错误数据（通常是简单的 **String**）；而 **Right** 封装真正的返回值（如果可用）。

Maybe 值函数的调用者不能轻易忽略异常条件。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的“穷人的 **Maybe**”形成对比，后者使用空指针编码错误条件。

日志记录

有时计算必须将一些数据记录在外部数据结构中。日志记录或审计是一种在并发程序中特别危险的副作用，其中多个线程可能同时尝试访问同一日志。

简单的解决方案是让函数返回计算值与要记录的项目配对。换句话说，类型为 **a -> b** 的日志记录计算可以替换为纯函数：

```
a -> (b, w)
```

然后，此函数的调用者负责提取要记录的值。这是一个常见的技巧：让函数提供所有数据，并让调用者处理副作用。

为了方便起见，我们稍后将定义一个新类型 **Writer w a**，它与 **(a, w)** 同构。这将允许我们将其作为类型类的实例，例如 **Functor**、**Applicative** 和 **Monad**。

请记住，同构类型可以使用记录语法定义，如：

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

我们自动获得一对形成同构的函数。其中之一是数据构造器：

```
Writer :: (a, w) -> Writer w a
```

另一个是它的逆函数：

```
runWriter :: Writer w a -> (a, w)
```

环境

一些计算需要对存储在环境中的某些外部数据进行只读访问。与其让计算秘密访问，不如将只读环境作为附加参数传递给函数。如果我们有一个需要访问某些环境 e 的计算 $a \rightarrow b$ ，我们将其替换为函数 $(a, e) \rightarrow b$ 。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)
```

如果我们使用与 $e \rightarrow a$ 同构的类型`Reader e a`。

```
newtype Reader e a = Reader { runReader :: e -> a }
```

那么我们可以通过将计算 $a \rightarrow b$ 替换为以下函数来编码环境副作用：

```
a -> Reader e b
```

这是一个延迟副作用的例子。我们不想处理副作用，因此我们将此责任委托给调用者。我们的函数生成一个“脚本”，调用者使用`runReader`执行它，传递类型为 e 的合适参数。

状态

最常见的副作用与访问和可能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来源。这在面向对象语言中是一个严重的问题，其中有状态的对象可以在许多客户端之间透明地共享。在 Java 中，这样的对象可能会被提供单独的互斥锁，但代价是性能受损和死锁风险。

在函数式编程中，我们使状态操作显式化：我们将状态作为附加参数传递，并返回修改后的状态与返回值配对。因此，我们将有状态计算 $a \rightarrow b$ 替换为

```
(a, s) -> (b, s)
```

其中 s 是状态的类型。与之前一样，我们可以柯里化这样的函数以使其变为以下形式：

```
a -> (s -> (b, s))
```

这个返回类型可以封装在新类型中：

```
newtype State s a = State { runState :: s -> (a, s) }
```

函数的调用者：

```
a -> State s b
```

被交给一个脚本。然后可以使用`runState`执行此脚本，它接受初始状态并生成修改后的状态与值配对。

非确定性

想象一下进行一个测量电子自旋的量子实验。一半时间自旋向上，一半时间自旋向下。结果是非确定性的。描述它的一种方法是使用多世界解释：当我们进行实验时，宇宙分裂成两个宇宙，每个结果一个。

函数非确定性意味着什么？这意味着每次调用它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。在实践中，我们将满足于一个（可能是无限的）结果列表：

我们将非确定性计算`a -> b`替换为返回单子结果的纯函数——这次是列表单子：

```
a -> [b]
```

同样，由调用者决定如何处理这些结果。

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，为了保持程序纯净，交互必须在程序之外进行。诀窍是让程序生成一个脚本。然后将此脚本传递给运行时以执行。运行时是运行程序的有效虚拟机。

这个脚本本身位于不透明的预定义`IO`单子中。隐藏在此单子中的值对程序不可访问：没有`runIO`函数。相反，程序生成的`IO`值至少在概念上是在程序完成后执行的。

实际上，由于 `Haskell` 的惰性，`I/O` 的执行与程序的其余部分交错进行。构成程序大部分的纯函数是按需评估的——需求由`IO`脚本的执行驱动。如果不是因为 `I/O`，什么都不会被评估。

`Haskell` 程序生成的`IO`对象称为`main`，其类型签名为：

```
main :: IO ()
```

它是包含单位的`IO`单子——意味着：除了输入/输出脚本外，没有有用的值。

稍后我们将讨论如何创建`IO`操作。

延续

我们已经看到，作为 `Yoneda` 引理的结果，我们可以用接受该值处理程序的函数替换类型为`a`的值。这个处理程序称为延续。调用处理程序被认为是计算的副作用。在纯函数方面，我们将其编码为：

```
a -> Cont r b
```

其中`Cont r`是封装提供类型为`a`值的承诺的数据类型：

```
newtype Cont r a = Cont { runCont :: (a -> r) -> r }
```

此函数的调用者负责提供适当的延续，即函数`k :: a -> r`，并检索结果：

```
runCont :: Cont r a -> (a -> r) -> r
runCont (Cont f) k = f k
```

在笛卡尔闭范畴中，延续由自函子生成：

$$K_r a = r^{r^a}$$

Exercise 13.1.1. 为`Cont r a`实现`Functor`实例。注意：这是一个协变函子，因为`a`出现在双重否定位置。

组合有副作用的计算

现在我们知道如何使用纯函数实现有副作用的计算，我们必须解决组合它们的问题。有两种基本策略可以做到这一点：并行和顺序组合。前者使用应用函子完成，后者使用单子完成。

应用函子 (Applicative Functors)

老子会说：“如果你并行执行两件事，所需时间将减半。”

14.1 并行组合

当我们将范畴论的语言翻译成编程语言时，我们将箭头视为计算。但计算需要时间，这引入了一个新的时间维度。这反映在我们使用的语言中。例如，两个箭头的组合结果 $g \circ f$ ，通常被读作 g 在 f 之后。 g 的执行必须跟随 f 的执行，因为我们需要 f 的结果才能调用 g 。我们称之为 **顺序组合**。它反映了计算之间的依赖关系。

然而，有些计算至少原则上可以并行执行，因为它们之间没有依赖关系。这只有在并行计算的结果可以组合时才有意义，这意味着我们需要在幺半群范畴中工作。因此，并行组合的基本构建块是两个箭头的张量积， $f: x \rightarrow a$ 和 $g: y \rightarrow b$ ：

$$x \otimes y \xrightarrow{f \otimes g} a \otimes b$$

在笛卡尔范畴中，我们通常使用笛卡尔积作为张量。在编程中，我们只需运行两个函数，然后组合它们的结果。

幺半群函子

当我们尝试组合有副作用的计算结果时，问题就出现了。为此，我们需要一种方法来组合一对结果 `f a` 和 `f b` 以及它们的副作用，以生成单个结果 `f (a, b)`。当然，每种类型的副作用需要不同的处理方式。

让我们以部分性 (partiality) 副作用为例。我们可以使用以下方式组合两个这样的副作用：

```
combine :: Maybe a -> Maybe b -> Maybe (a, b)
combine (Just a) (Just b) = Just (a, b)
```

```
combine _ _ = Nothing
```

只有当两个计算都成功时，结果才是成功的。

一个计算还应该能够产生一个”忽略我”的结果。这将是一个返回单位值（即相对于笛卡尔积的单位）和”忽略我”副作用的计算。注意，返回 `Nothing` 是不行的，因为它会使另一个计算的结果无效。正确的实现是：

```
ignoreMe :: Maybe ()
ignoreMe = Just ()
```

当与任何其他部分计算 `combine`’d 时，它会被忽略（符合左/右单位律）。

一般来说，如果类型构造器 `f` 是 `Monoidal` 类的实例，它就支持并行组合：

```
class Monoidal f where
  unit   :: f ()
  (>*<) :: f a -> f b -> f (a, b)
```

特别是，`Maybe` 函子是 `Monoidal` 的：

```
instance Monoidal Maybe where
  unit   = Just ()
  Just a >*< Just b = Just (a, b)
  _ >*< _ = Nothing
```

么半群函子必须与笛卡尔范畴的么半群结构兼容，因此它们应满足明显的单位和结合律。

应用函子

尽管 `Monoidal` 类为副作用的并行组合提供了足够的支持，但它并不十分实用。

首先，我们需要能够对组合结果进行操作，因此我们需要 `f` 的 `Functor` 实例。这让我们可以将类型为 `(a, b) -> c` 的函数应用于有副作用的对 `f(a, b)`，而不影响副作用。

一旦我们知道如何并行运行两个计算，我们就可以利用结合律来组合任意数量的计算。例如：

```
run3 :: (Functor f, Monoidal f) =>
  (x -> f a) -> (y -> f b) -> (z -> f c) ->
  (x, y, z) -> f (a, b, c)
run3 f1 f2 f3 (x, y, z) =
```



```
let fab = f1 x >*< f2 y
    fabc = fab >*< f3 z
in fmap reassoc fabc
```

其中我们重新关联结果的三元组：

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

`let` 子句用于引入局部绑定。在这里，局部变量 `fab` 和 `fabc` 被初始化为相应的么半群积。`let/in` 构造是一个表达式，其值由 `in` 子句的内容给出。

但随着计算数量的增加，事情变得越来越笨拙。

幸运的是，有一种更符合人体工程学的方法。在大多数应用中，并行计算的结果被输入到一个函数中，该函数在生成最终结果之前收集它们。我们需要的是提升这种多参数函数的方法——这是 **Functor** 执行的单参数函数提升的推广。

例如，要组合两个并行计算的结果，我们可以使用函数：

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

不幸的是，我们需要无限多个这样的函数，以适应所有可能的参数数量。诀窍是用一个利用柯里化的函数来替换它们。

一个有两个参数的函数是一个返回函数的单参数函数。因此，假设 `f` 是一个函子，我们可以先 `fmap`’ping：

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

在第一个参数 (`f a`) 上得到：

```
f (b -> c)
```

现在我们需要将 `f (b -> c)` 应用于第二个参数 (`f b`)。如果函子是 **Monoidal**，我们可以使用操作符 `>*<` 将两者组合，得到类型为：

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

然后我们可以 `fmap` 函数应用 `apply` 到它上面：

```
fmap apply (ff >*< fa)
```

其中：

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

或者，我们可以定义中缀操作符 `<*>`，它让我们在组合副作用的同时将函数应用于参数：

```
(<*>) :: f (a -> b) -> f a -> f b
fs <*> as = fmap apply (fs >*< as)
```

这个操作符有时被称为“splat”。

相反，我们可以用 `splat` 来实现么半群操作符：

```
fa >*< fb = fmap (,) fa <*> fb
```

其中我们使用了对构造器 `(,)` 作为双参数函数。

使用 `<*>` 而不是 `>*<` 的优势在于，它让我们可以通过重复剥离柯里化层次来应用任意数量参数的函数。

例如，我们可以将三个参数的函数 `g :: a -> b -> c -> d` 应用于三个值 `fa :: f a`，`fb :: f b`，和 `fc :: f c`，使用：

```
fmap g fa <*> fb <*> fc
```

我们甚至可以使用 `fmap` 的中缀版本 `<$>` 使其看起来更像函数应用：

```
g <$> fa <*> fb <*> fc
```

其中：

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

这两个操作符都向左绑定，因此我们可以将此符号视为函数应用的直接推广：

```
g a b c
```

除了它还累积了三个计算的副作用。

为了完善这个图景，我们还需要定义应用一个有副作用的零参数函数的含义。它只是意味着在一个单一值上附加一个“忽略我”的副作用。我们可以使用么半群的 `unit` 来实现这一点：

```
pure :: a -> f a
pure a = fmap (const a) unit
```

相反，`unit` 可以用 `pure` 来实现：

```
unit = pure ()
```

因此，在笛卡尔闭范畴中，我们可以使用等价的但更方便的 **Applicative** 来代替 **Monoidal**：

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

14.2 应用函子实例

最常见的应用函子同时也是单子（最著名的反例是**ZipList**）。通常选择使用**Applicative**还是**Monad**语法是基于便利性，尽管在某些情况下存在性能差异——尤其是在涉及并行执行时。

部分性

一个**Maybe**函数只有在两者都是**Just**时才能应用于**Maybe**参数：

```
instance Applicative Maybe where
  pure = Just
  mf <*> ma =
    case (mf, ma) of
      (Just f, Just a) -> Just (f a)
      _ -> Nothing
```

我们可以通过定义**Either**函子的版本来添加关于失败原因的信息：

```
data Validation e a = Failure e | Success a
```

Validation的**Applicative**实例使用**mappend**累积错误：

```
instance Monoid e => Applicative (Validation e) where
  pure = Success
  Failure e1 <*> Failure e2 = Failure (mappend e1 e2)
  Failure e <*> Success _ = Failure e
  Success _ <*> Failure e = Failure e
  Success f <*> Success x = Success (f x)
```

日志记录

```
newtype Writer w a = Writer { runWriter :: (a, w) }
    deriving Functor
```

为了使`Writer`函子支持组合，我们必须能够组合记录的值，并且有一个“忽略我”的元素。这意味着日志必须是一个么半群：

```
instance Monoid w => Applicative (Writer w) where
    pure a = Writer (a, mempty)
    wf <*> wa = let (f, w)  = runWriter wf
                  (a, w') = runWriter wa
                in Writer (f a, mappend w w')
```

环境

```
newtype Reader e a = Reader { runReader :: e -> a }
    deriving Functor
```

由于环境是不可变的，我们并行传递给两个计算。无效果的`pure`忽略环境。

```
instance Applicative (Reader e) where
    pure a = Reader (const a)
    rf <*> ra = Reader (\e -> (runReader rf e) (runReader ra e))
```

状态

```
newtype State s a = State { runState :: s -> (a, s) }
    deriving Functor
```

`State`应用函子展示了并行和顺序组合。两个动作是并行创建的，但它们是顺序执行的。第二个动作使用第一个动作修改后的状态：

```
instance Applicative (State s) where
    pure a = State (\s -> (a, s))
    sf <*> sa = State (\s ->
        let (f, s') = runState sf s
```

```
(a, s'') = runState sa s'
in (f a, s'')
```

非确定性

列表函子有两个独立的`Applicative`实例。它们对应于两种不同的列表组合方式。第一种是逐个元素压缩两个列表；第二种产生所有可能的组合。为了对同一数据类型有两个实例，我们必须将其中一个封装在`newtype`中：

```
newtype ZipList a = ZipList { unZip :: [a] }
deriving Functor
```

`splat` 操作符将每个函数应用于其对应的参数。它在较短列表的末尾停止。有趣的是，如果我们希望`pure`符合单位律，它必须产生一个无限列表：

```
repeat :: a -> [a]
repeat a = a : repeat a
```

这样，当与任何有限列表压缩时，它不会截断结果。

```
instance Applicative ZipList where
  pure a = ZipList (repeat a)
  zf <*> za = ZipList (zipWith ($) (unZip zf) (unZip za))
```

两个（可能无限的）列表通过应用函数应用操作符`$`组合。

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f [] _ = []
zipWith f _ [] = []
zipWith f (a : as) (b : bs) = f a b : zipWith f as bs
```

列表的第二个`Applicative`实例将所有函数应用于所有参数。为了实现 `splat` 操作符，我们使用 Haskell 的列表推导语法：

```
instance Applicative [] where
  pure a = [a]
  fs <*> as = [ f a | f <- fs, a <- as ]
```

`[f a | f <- fs, a <- as]`的含义是：创建一个`(f a)`的列表，其中`f`取自列表`fs`，`a`取自列表`as`。

延续

```
newtype Cont r a = Cont { runCont :: (a -> r) -> r }
    deriving Functor
```

让我们为延续实现`<*>`操作符。作用于一对延续

```
kf :: Cont r (a -> b)
ka :: Cont r a
```

它应该产生一个延续：

```
Cont r b
```

后者接受一个处理程序 `k :: b -> r`，并应该用函数应用 `(f a)` 的结果调用它。为了进行最后一次调用，我们需要提取函数和参数。为了提取 `a`，我们必须运行延续 `ka` 并传递 `a` 的消费者：

```
runCont ka (\a -> k (f a)))
```

为了提取 `f`，我们必须运行延续 `kf` 并传递 `f` 的消费者：

```
runCont kf (\f -> runCont ka (\a -> k (f a))))
```

总的来说，我们得到：

```
instance Applicative (Cont r) where
    pure a = Cont (\k -> k a)
    kf <*> ka = Cont (\k ->
        runCont kf (\f ->
            runCont ka (\a -> k (f a))))
```

输入/输出

输入/输出操作通常作为单子组合，但也可以使用应用语法。注意，尽管应用函子是并行组合的，但副作用是序列化的。这很重要，例如，如果你希望在获取用户输入之前打印提示：

```
prompt :: String -> IO String
prompt str = putStrLn str *> getLine
```

这里我们组合了 `putStrLn`（将其参数打印到终端）和 `getLine`（等待用户输入）。半 `splat` 操作符忽略第一个参数产生的值（在这种情况下是单位 `()`），但保留副作用（这里，打印字符串）：

```
(*>) :: Applicative f => f a -> f b -> f b
u *> v = (\ _ x -> x) <$> u <*> v
```

我们现在可以应用一个双参数函数：

```
greeting :: String -> String -> String
greeting s1 s2 = "Hi " ++ s1 ++ " " ++ s2 ++ "!"
```

到一对IO参数：

```
getNamesAndGreet :: IO String
getNamesAndGreet =
    greeting <$> prompt "First name: " <*> prompt "Last name: "
```

解析器

解析是一种高度可分解的活动。有许多领域特定语言可以使用应用解析器（而不是更强大的单子解析器）进行解析。

解析器可以描述为接受字符（或标记）字符串的函数。解析可能失败，因此结果是`Maybe`。成功的解析器返回解析值以及未消耗的输入字符串部分：

```
newtype Parser a =
    Parser { parse :: String -> Maybe (a, String) }
```

例如，这是一个检测数字的简单解析器：

```
digit :: Parser Char
digit = Parser (\s -> case s of
    (c:cs) | isDigit c -> Just (c, cs)
    _                  -> Nothing)
```

解析器的`Applicative`实例将部分性与状态结合起来：

```
instance Applicative Parser where
    pure a = Parser (\s -> Just (a, s))
    pf <*> pa = Parser (\s ->
        case parse pf s of
            Nothing      -> Nothing
```

```
Just (f, s') -> case parse pa s' of
  Nothing      -> Nothing
  Just (a, s'') -> Just (f a, s'')
```

大多数语法包含替代项，例如，标识符或数字；if 语句或循环等。这可以通过 **Applicative** 的以下子类捕获：

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

这表明，无论类型 `a` 是什么，类型 `(f a)` 都是一个幺半群。

Parser 的 **Alternative** 实例尝试第一个解析器，如果失败，则尝试第二个。幺半群单位是一个总是失败的解析器。

```
instance Alternative Parser where
  empty = Parser (\s -> Nothing)
  pa <|> pb = Parser (\s ->
    case parse pa s of
      Just as -> Just as
      Nothing -> parse pb s)
```

如果我们使用 **Maybe** 的 **Alternative** 实例，这可以简化：

```
pa <|> pb = Parser (\s -> parse pa s <|> parse pb s)
```

拥有 **Applicative** 超类允许我们链式替代项。我们可以使用 **some** 来解析一个或多个项目：

```
some :: f a -> f [a]
some pa = (:) <$> pa <*> many pa
```

和 **many** 来解析零个或多个项目：

```
many :: f a -> f [a]
many pa = some pa <|> pure []
```

Exercise 14.2.1. 实现 **Maybe** 的 **Alternative** 实例。提示：如果第一个参数是失败，尝试第二个。

并发和并行

启动线程是一个 I/O 操作，因此任何并发或并行执行本质上都是有副作用的。计算是并行还是串行执行取决于它们的组合方式。并行性是通过应用组合实现的。

Concurrently 函子是一个用于并行操作的 **Applicative** 示例。它接受一个 **IO** 操作并在单独的线程中开始执行它。然后使用应用组合来组合结果——这里，通过添加两个整数：

```
f :: Concurrently Int
f = (+) <$> Concurrently (fileChars "1-Types.hs")
    <*> Concurrently (fileChars "2-Composition.hs")
    where fileChars path = length <$> readFile path
```

Do Notation

在范畴论中，我们以图的方式组合箭头。我们可以将它们串联或并联，并且不需要为中间对象的元素命名。这样的图可以直接翻译为编程，产生用于串联组合的无点符号和用于并行组合的应用符号。然而，命名中间结果通常很方便。这可以使用 **do** 符号完成。例如，并发示例可以重写为：

```
{-# language ApplicativeDo #-}
f :: Concurrently Int
f = do
    m <- Concurrently (fileChars "1-Types.hs")
    n <- Concurrently (fileChars "2-Composition.hs")
    pure (m + n)
    where fileChars path = length <$> readFile path
```

我们将在关于单子的章节中更多地讨论 **do** 符号。现在，重要的是要知道，当你使用语言编译指示 **ApplicativeDo** 时，编译器默认会尝试在 **do** 块的翻译中使用应用组合。如果不可能，它将回退到单子组合。

应用函子的组合

应用函子的一个很好的特性是它们的函子组合再次是一个应用函子：

```
instance (Applicative f, Applicative g) =>
    Applicative (Compose g f) where
    pure :: a -> Compose g f a
    pure x = Compose (pure (pure x))
```

```

(<*>) :: Compose g f (a -> b) -> Compose g f a -> Compose g f b
Compose gff <*> Compose gfa = Compose (fmap (<*>) gff <*> gfa)

```

我们稍后会看到，单子并非如此：两个单子的组合通常不是单子。

14.3 范畴论中的么半函子

我们已经看到了几个么半范畴 (monoidal category) 的例子。这类范畴配备了某种二元运算，例如笛卡尔积、和、复合（在自函子范畴中）等。它们还有一个特殊对象，作为该二元运算的单位元。单位元和结合律要么严格满足（在严格么半范畴中），要么在同构意义下满足。

每当我们有多个某种结构的实例时，我们可能会问自己一个问题：是否存在一个由这些事物构成的完整范畴？在这种情况下：么半范畴能否形成它们自己的范畴？为此，我们必须定义么半范畴之间的箭头。

一个从么半范畴 $(\mathcal{C}, \otimes, i)$ 到另一个么半范畴 (\mathcal{D}, \oplus, j) 的么半函子 (monoidal functor) F 将张量积映射到张量积，单位元映射到单位元——所有这些都在同构意义下成立：

$$Fa \oplus Fb \cong F(a \otimes b)$$

$$j \cong Fi$$

这里，左边是目标范畴中的张量积和单位元，右边是源范畴中的对应物。

如果所讨论的两个么半范畴不是严格的，即单位元和结合律仅在同构意义下满足，那么还需要额外的相干性条件，以确保单位元映射到单位元，结合子映射到结合子。

以么半函子为箭头的么半范畴的范畴称为 **MonCat**。实际上，它是一个 2-范畴，因为可以定义么半函子之间的结构保持自然变换。

松弛么半函子

么半范畴的一个优点是可以帮助我们定义么半群 (monoid)。你可以很容易地确信，么半函子将么半群映射到么半群。事实证明，你不需要么半函子的全部功能来实现这一点。让我们考虑一个函子将么半群映射到么半群所需的最小条件。

让我们从么半范畴 $(\mathcal{C}, \otimes, i)$ 中的一个么半群 (m, μ, η) 开始。考虑一个将 m 映射到 Fm 的函子 F 。我们希望 Fm 成为目标么半范畴 (\mathcal{D}, \oplus, j) 中的一个么半群。为此，我们需要找到两个映射：

$$\eta' : j \rightarrow Fm$$

$$\mu' : Fm \oplus Fm \rightarrow Fm$$

满足么半群定律。

由于 m 是一个么半群，我们确实可以利用原始映射的提升：

$$F\eta: Fi \rightarrow Fm$$

$$F\mu: F(m \otimes m) \rightarrow Fm$$

为了实现 η' 和 μ' ，我们缺少的是两个额外的箭头：

$$j \rightarrow Fi$$

$$Fm \oplus Fm \rightarrow F(m \otimes m)$$

一个么半函子会提供这样的箭头。然而，对于我们试图实现的目标，我们不需要这些箭头是可逆的。

一个松弛么半函子 (lax monoidal functor) 是一个配备了一个态射 ϕ_i 和一个自然变换 ϕ_{ab} 的函子：

$$\phi_i: j \rightarrow Fi$$

$$\phi_{ab}: Fa \oplus Fb \rightarrow F(a \otimes b)$$

满足适当的单位性和结合性条件。

这样的函子将么半群 (m, μ, η) 映射到么半群 (Fm, μ', η') ，其中：

$$\eta' = F\eta \circ \phi_i$$

$$\mu' = F\mu \circ \phi_{ab}$$

在 Haskell 中，我们将 **Monoidal** 函子视为一个松弛么半自函子的例子，它保留了笛卡尔积。

函子强度

函子与么半结构交互的另一种方式，在我们编程时隐藏得很明显。我们理所当然地认为函数可以访问环境。这样的函数称为闭包 (closure)。

例如，这里有一个函数从环境中捕获变量 **a** 并将其与其参数配对：

```
\x -> (a, x)
```

这个定义单独来看没有意义，但当环境中包含变量 **a** 时，它就有意义了，例如：

```
pairWith :: Int -> (String -> (Int, String))
pairWith a = \x -> (a, x)
```

调用 `pairWith 5` 返回的函数从其环境中“捕获”了 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，如果这不起作用，你会非常惊讶。但我们在这里做的事情非常不平凡：我们正在将环境“偷运”到列表函子下。根据我们的 `lambda` 演算模型，闭包是从环境和函数参数的乘积到结果的态射。这里的 `lambda`，实际上是 `(Int, String)` 的函数，定义在列表函子内部，但它捕获了在列表外部定义的 `a`。

让我们将环境偷运到函子下的属性称为函子强度 (functorial strength) 或张量强度 (tensorial strength)，可以在 `Haskell` 中实现为：

```
strength :: Functor f => (e, f a) -> f (e, a)
strength (e, as) = fmap (e, ) as
```

符号 `(e,)` 称为元组部分 (tuple section)，等价于对构造函数的部分应用：`(,) e`。

在范畴论中，自函子 F 的强度被定义为一个将张量积偷运到函子中的自然变换：

$$\sigma : a \otimes F(b) \rightarrow F(a \otimes b)$$

还有一些额外的条件，确保它与所讨论的幺半范畴的单位元和结合子良好地配合。

我们能够为任意函子实现 `strength` 的事实意味着，在 `Haskell` 中，每个函子都是强的。这就是为什么我们不必担心从函子内部访问环境的原因。

然而，在范畴论中，并非幺半范畴中的每个自函子都是强的。目前，我们的魔法咒语是我们正在使用的范畴是自丰富的 (self-enriched)，并且在 `Haskell` 中定义的每个自函子都是丰富的。当我们讨论丰富范畴时，我们会回到这一点。在 `Haskell` 中，强度归结为我们总是可以 `fmap` 一个部分应用的元组构造函数 `(a,)`。

闭函子

如果你眯着眼睛看 `splat` 操作符的定义：

```
(<*>) :: f (a -> b) -> (f a -> f b)
```

你可能会将其视为将函数对象映射到函数对象。

如果你考虑两个闭范畴之间的函子，这一点会变得更加清晰。你可以从源范畴中的函数对象 b^a 开始，并对其应用函子 F ：

$$F(b^a)$$

或者，你可以映射两个对象 a 和 b ，并在目标范畴中构造它们之间的函数对象：

$$(Fb)^{Fa}$$

如果我们要求这两种方式同构，我们就得到了严格闭函子（**closed functor**）的定义。但是，就像么半函子的情况一样，我们更感兴趣的是松弛版本，它配备了一个单向自然变换：

$$F(b^a) \rightarrow (Fb)^{Fa}$$

如果 F 是一个自函子，这直接转化为 **splat** 操作符的定义。

松弛闭函子的完整定义包括么半单位元的映射和一些相干性条件。综上所述，应用函子（**applicative functor**）就是一个松弛闭函子。

单子 (Monads)

当在编程语境中解释单子时，很难在关注函子 (functors) 的同时看到共同模式。要理解单子，你必须深入函子内部，并解读代码字里行间的含义。有时人们说单子让我们重载了分号。这是因为在许多命令式语言中，分号用于操作序列化。在函数式语言中，单子扮演着序列化带副作用计算的角色。

15.1 副作用的顺序组合

并行组合作为顺序组合的特例。

—

在命令式语言中，带副作用计算的组合方式是使用常规的函数组合来处理值，而让副作用随意地自行组合。

当我们将带副作用计算表示为纯函数时，我们面临组合两个形式如下的函数的问题：

```
g :: a -> f b
h :: b -> f c
```

在所有感兴趣的情况下，类型构造器 `f` 恰好是一个 `Functor`，因此我们在接下来的讨论中假设这一点。

天真的方法是将第一个函数的结果解包，将值传递给下一个函数，然后在侧面组合两个函数的副作用，并将它们与第二个函数的结果结合。这并不总是可能的，即使对于我们迄今为止研究的情况也是如此，更不用说对于任意类型构造器了。

为了论证起见，看看我们如何为 `Maybe` 函子做到这一点是有启发性的。如果第一个函数返回 `Just`，我们对其进行模式匹配以提取内容，并用它调用下一个函数。

但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，但这意味着通过基于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然。我们如何描述这些“好”的函子？

对于一个函子来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)
```

这与常规的函数组合非常相似：

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

因此很自然地会问是否存在一个范畴，其中前者定义了箭头的组合。让我们看看构建这样一个范畴还需要什么。

在这个新范畴中，对象与之前的 `Haskell` 类型相同。但一个箭头，我们将其写为 $a \Rightarrow b$ ，是作为一个 `Haskell` 函数实现的：

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

我们的 `composeWithEffects` 然后可以用于实现这些箭头的组合。

要拥有一个范畴，我们要求这种组合是结合的。我们还需要每个对象 `a` 的一个单位箭头。这是一个箭头 $a \Rightarrow a$ ，因此它对应于一个 `Haskell` 函数：

```
idWithEffects :: a -> f a
```

它必须相对于 `composeWithEffects` 表现得像单位元。

看待这个箭头的另一种方式是，它允许你为任何类型 `a` 的值添加一个平凡的副作用。这是一个与任何其他副作用结合时对其不做任何事的副作用。我们之前在 `Applicative` 函子的 `pure` 定义中见过这一点。

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
```

中缀运算符 `<=<` 替换了函数 `composeWithEffects`。 `return` 函数是我们新范畴中的单位箭头。（这不是你在 `Haskell` 的 `Prelude` 中找到的单子定义，但正如我们很快将看到的，它与之等价。）

作为练习，让我们为 `Maybe` 定义 `Monad` 实例。“鱼”运算符 `<=<` 组合了两个函数：

```
f :: a -> Maybe b
g :: b -> Maybe c
```


成为一个类型为 `a -> Maybe c` 的函数。这个组合的单位 `return`，将值封装在 `Just` 构造器中。

```
instance Monad Maybe where
  g <=< f = \a -> case f a of
    Nothing -> Nothing
    Just b -> g b
  return = Just
```

你可以很容易地让自己相信范畴律是满足的。特别是组合 `return <=< g` 与 `g` 相同，`f <=< return` 与 `f` 相同。结合性的证明也相当直接：如果任何函数返回 `Nothing`，结果就是 `Nothing`；否则它只是一个直接的函数组合，这是结合的。

我们刚刚定义的范畴称为单子 `m` 的 *Kleisli* 范畴。函数 `a -> m b` 称为 *Kleisli* 箭头。它们使用 `<=<` 进行组合，单位箭头称为 `return`。

在副作用部分列出的所有效果都是 `Monad` 实例。如果你将它们视为类型构造器，很难看出它们之间的任何相似之处。每种效果都不同。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：组合是发生在事物之间的事情。当我们将注意力集中在事物上时，我们常常忽略了间隙中的东西。

15.2 其他定义

使用 *Kleisli* 箭头定义单子 (`monad`) 的优势在于，单子法则就是 *Kleisli* 范畴的结合律和单位律。单子还有另外两个等价的定义，一个受数学家青睐，另一个则更受程序员欢迎。

首先，我们注意到鱼操作符 (*fish operator*) 接受两个函数作为参数。函数的唯一用途就是应用于某个参数。当我们应用第一个函数 `f :: a -> m b` 时，会得到一个类型为 `m b` 的值。此时如果没有 `m` 是函子 (*functor*) 这一事实，我们会陷入困境。函子性允许我们将第二个函数 `g :: b -> m c` 应用于 `m b`。实际上，通过 `m` 提升 `g` 后的类型为：

```
m b -> m (m c)
```

这几乎就是我们想要的结果。如果我们能把 `m(m c)` 展平为 `m c` 就好了。这种展平操作，如果可能的话，称为 `join`。换句话说，如果我们有：

```
join :: m (m a) -> m a
```

我们就可以实现 `<=<`：

```
g <=< f = \a -> join (fmap g (f a))
```

或者，使用无点 (point-free) 表示法：

```
g <=< f = join . fmap g . f
```

反过来，我们也可以用`<=<`来实现`join`：

```
join = id <=< id
```

后一个定义可能不会立即显而易见，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)
```

重新解释为 Kleisli 范畴中的箭头 $m(ma) \rightarrow ma$ 。类似地，函数：

```
m a -> m a
```

实现了一个 Kleisli 箭头 $ma \rightarrow a$ 。它们的 Kleisli 组合产生了一个 Kleisli 箭头 $m(ma) \rightarrow a$ 或一个 Haskell 函数：

```
m (m a) -> m a
```

这引导我们得到了用`join`和`return`定义的等价单子定义：

```
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 `Prelude` 中找到的定义。由于鱼操作符是点操作符的泛化，使用它就相当于无点编程。它让我们可以在不命名中间值的情况下组合箭头。尽管有些人认为无点程序更优雅，但大多数程序员发现它们难以理解。

从程序的角度来看，顺序函数组合实际上分两步完成：我们先应用第一个函数，然后将第二个函数应用于结果。显式命名中间结果通常有助于理解发生了什么。

要在 Kleisli 箭头上实现同样的效果，我们必须知道如何将第二个 Kleisli 箭头应用于一个命名的单子值——第一个 Kleisli 箭头的结果。实现这一功能的函数称为 *bind*，并写为中缀运算符：

```
(>=>) :: 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，bind 也可以用 Kleisli 箭头来实现：

```
ma >>= k = (k <=< id) ma
```

这引导我们得到了以下定义：

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

这几乎就是你在Prelude中找到的定义，除了额外的Applicative约束。我们接下来会讨论它。

我们也可以用 bind 来实现join：

```
join :: (Monad m) => m (m a) -> m a
join mma = mma >>= id
```

Haskell 函数id从m a到m a，或者作为 Kleisli 箭头， $ma \rightarrow a$ 。

有趣的是，使用 bind 定义的Monad自动成为函子。它的提升函数称为liftM

```
liftM :: Monad m => (a -> b) -> (m a -> m b)
liftM f ma = ma >>= (return . f)
```

作为应用函子的单子

在笛卡尔范畴中，每个单子都是松弛幺半群的。事实上，如果你知道如何顺序组合有副作用的计算，你就知道如何并行组合它们。如果第一个计算的结果没有作为第二个计算的输入使用，它们就可以并行完成。另一方面，副作用总是顺序组合的。

在 Haskell 中，我们可以为任何Monad定义Monoidal实例：

```
instance Monad m => Monoidal m where
  unit = return ()
  ma >*< mb = ma >>=
    (\a -> mb >>=
      \b -> return (a, b))
```

你会注意到，最后的`return`需要访问`a`和`b`，它们是在外部环境中定义的。如果单子不强，这将是是不可能的。

正如我们之前讨论的，每个 Haskell 函子都是强的，所以 Haskell 中的每个单子由于是函子而都是强的。这很重要，因为我们希望单子代码能够访问环境。

在笛卡尔闭范畴中，每个单子¹，作为`Monoidal`，自动成为`Applicative`。我们可以通过实现`ap`直接展示这一点，它具有与 `splat` 运算符相同的类型签名：

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap fs as = fs >=>
  (\f -> as >=>
    \a -> return (f a))
```

我们也可以使用单子的`return`作为应用函子的`pure`。

单子与应用函子之间的这种关系在 Haskell 中通过使`Applicative`成为`Monad`的超类来表达：

```
class Applicative m => Monad m where
  (>=>)      :: forall a b. m a -> (a -> m b) -> m b
  return     :: a -> m a
  return     = pure
```

反之则不成立：并非每个`Applicative`都是`Monad`。标准的反例是封装在`ZipList`中的列表函子的`Applicative`实例。

当为 Haskell 类型构造器实例化单子时，我们将定义拆分到三个类型类中：`Functor`、`Applicative`和`Monad`。在大多数情况下，`Functor`实例可以自动派生。如果我们手头还没有完整的`Applicative`实例，我们使用`ap`来定义 `splat` 运算符。`Applicative`和`Monad`实例的定义顺序无关紧要。

让我们通过一个简单的例子来说明这一点：

```
data Id a = MakeId { getId :: a }
  deriving Functor
```

`Applicative`实例定义了`pure`和`<*>`：

```
instance Applicative Id where
  pure = MakeId
  (<*>) = ap
```

后者使用了在`Control.Monad`中定义的`ap`函数：

¹再次强调，正确的说法是“每个富集单子”

`let`子句不仅可以引入局部绑定，还可以进行模式匹配。

`pure`在`Applicative`实例中实现：

```
pure a = Writer (a, mempty)
```

环境

读取器单子 (reader monad) 是对从环境到返回类型的函数的简单封装：

```
newtype Reader e a = Reader { runReader :: e -> a }
```

以下是`Monad`实例：

```
instance Monad (Reader e) where
  ma >=> k = Reader (\e -> let a = runReader ma e
                           in runReader (k a) e)
```

`pure`在`Applicative`实例中实现：

```
pure a = Reader (\e -> a)
```

读取器单子的绑定实现创建了一个以环境为参数的函数。这个环境被使用了两次，首先用于运行`ma`以获取`a`的值，然后用于评估`k a`生成的值。

`pure`的实现忽略了环境。

Exercise 15.3.1. 为以下数据类型定义`Functor`和`Monad`实例：

```
newtype E e a = E { runE :: e -> Maybe a }
```

状态

与读取器类似，状态单子 (state monad) 是一个函数类型：

```
newtype State s a = State { runState :: s -> (a, s) }
```

它的绑定实现类似，只是`k`作用于`a`的结果必须使用修改后的状态`s'`来运行。

```
instance Monad (State s) where
  st >=> k = State (\s -> let (a, s') = runState st s
                           in runState (k a) s')
```

`pure`在`Applicative`实例中实现:

```
pure a = State (\s -> (a, s))
```

将绑定应用于恒等函数 (`identity`) 可以得到`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, 我们本质上是第一个`runState`的结果传递给第二个`runState`, 只是我们必须对第二个函数进行反柯里化 (`uncurry`), 以便它可以接受一个二元组:

```
join mma = State (\s -> (uncurry runState) (runState mma s))
```

在这种形式下, 很容易将其转换为无点表示法 (`point-free notation`):

```
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`()`), 我们可以用它们来构造任意的状态计算。第一个用于获取当前状态:

```
get :: State s s
get = State (\s -> (s, s))
```

第二个用于修改状态:

```
set :: s -> State s ()
set s = State (\_ -> ((), s))
```

许多单子都带有自己的预定义基本 Kleisli 箭头库。

非确定性

对于列表单子, 让我们考虑如何实现`join`。它必须将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实现。从那里, 我们可以推导出绑定的实现。

```
instance Monad [] where
  as >>= k = concat (fmap k as)
```

`pure`构造一个单例列表。(因此, 非确定性的一个简单版本是确定性。)

```
pure a = [a]
```

命令式语言使用嵌套循环完成的事情，我们可以在 Haskell 中使用列单子来完成。将绑定中的 `as` 视为聚合内层循环运行的结果，而 `k` 视为在外层循环中运行的代码。

在许多方面，Haskell 的列表更像命令式语言中所谓的迭代器或生成器。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 列表概念化为指向头部的指针以及一个将其向前推进到尾部的配方。或者你可以将列表视为一个协程，按需生成序列中的元素。

延续

延续单子 (continuation monad) 的绑定实现：

```
newtype Cont r a = Cont { runCont :: (a -> r) -> r }
```

由于控制反转的固有特性——“不要调用我们，我们会调用你”的原则——需要一些逆向思维。

绑定的结果是 `Cont r b` 类型。要构造它，我们需要一个以延续 `k :: b -> r` 为参数的函数：

```
ma >>= fk = Cont (\k -> ...)
```

我们有两个可用的成分：

```
ma :: Cont r a
fk :: a -> Cont r b
```

我们想要运行 `ma`，为此我们需要一个接受 `a` 的延续。

```
runCont ma (\a -> ...)
```

在其中我们可以执行 `fk`。结果是 `Cont r b` 类型，因此我们可以使用延续 `k :: b -> r` 来运行它。

```
runCont (fk a) k
```

综合起来，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))
```

`pure` 来自 `Applicative` 实例：

```
pure a = Cont (\k -> k a)
```

如前所述，组合延续不适合胆小的人。然而，它只需要实现一次——在延续单子的定义中。从那里开始，`do` 表示法将使其余部分相对容易。

输入/输出

IO单子的实现是嵌入在语言中的。基本的 I/O 原语通过库提供。它们要么是 **Kleisli** 箭头的形式，要么是 **IO** 对象（概念上是从终端对象 **()** 出发的 **Kleisli** 箭头）。

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；但程序可以通过一系列进一步的 **Kleisli** 箭头来处理它。

IO单子是终极的拖延者：其 **Kleisli** 箭头的组合堆积了一个又一个任务，稍后由 **Haskell** 运行时执行。

要输出一个字符串并换行，你可以使用这个 **Kleisli** 箭头：

```
putStrLn :: String -> IO ()
```

将两者结合，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。

与应用函子 (**applicative**) 不同，单子允许我们对中间值进行分支或模式匹配。在这个例子中，我们对 **IO** 对象的内容进行分支（我们接下来会讨论 **do** 表示法）：

```
main :: IO ()
main = do
  s <- getLine
  if s == "yes"
  then putStrLn "Thank you!"
  else putStrLn "Next time."
```

当然，实际的值检查会推迟到运行时对生成的 **IO** 对象运行解释器时。

15.4 Do 表示法

值得再次强调的是，在编程中单子 (**monad**) 的唯一目的是让我们能够将一个大的 **Kleisli** 箭头分解成多个较小的 **Kleisli** 箭头。

这可以通过两种方式实现：一种是直接使用无点 (**point-free**) 风格，通过 **Kleisli** 组合 **<=<**；另一种是通过命名中间值，并使用 **>>=** 将它们绑定到 **Kleisli** 箭头上。

一些 Kleisli 箭头在库中定义，其他一些足够可重用，值得单独实现，但在实践中，大多数只是一次性的内联 lambda 表达式。

这里有一个简单的例子：

```
main :: IO ()
main =
  getLine >=> \s1 ->
    getLine >=> \s2 ->
      putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

它使用了一个临时定义的 Kleisli 箭头，类型为 `String->IO ()`，由 lambda 表达式定义：

```
\s1 ->
  getLine >=> \s2 ->
    putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

这个 lambda 的主体进一步通过另一个临时定义的 Kleisli 箭头分解：

```
\s2 -> putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

这种构造非常常见，因此有一种特殊的语法称为 `do` 表示法，可以大大减少样板代码。例如，上述代码可以写成：

```
main = do
  s1 <- getLine
  s2 <- getLine
  putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

编译器会自动将其转换为一组嵌套的 lambda 表达式。行 `s1<-getLine` 通常读作：“s1 获取 `getLine` 的结果。”

这是另一个例子：一个使用列单子生成两个列表中所有可能元素对的函数。

```
pairs :: [a] -> [b] -> [(a, b)]
pairs as bs = do
  a <- as
  b <- bs
  pure (a, b)
```

注意，`do` 块中的最后一行必须产生一个单子值——在这里这是通过 `pure` 实现的。

大多数命令式语言缺乏抽象能力来通用地定义单子，而是试图硬编码一些更常见的单子。例如，它们将异常实现为 `Either` 单子的替代品，或将并发任务实现为延续单子的替代品。一些语言，如 C++，引入了协程来模仿 Haskell 的 `do` 表示法。

Exercise 15.4.1. 实现：

```
ap :: Monad m => m (a -> b) -> m a -> m b
```

使用 `do` 表示法。

Exercise 15.4.2. 使用绑定操作符和 `lambda` 表达式重写 `pairs` 函数。

15.5 延续传递风格

我之前提到过，`do` 语法糖使得使用延续（continuation）更加自然。延续最重要的应用之一是将程序转换为使用 CPS（延续传递风格，Continuation Passing Style）。CPS 转换在编译器构造中很常见。CPS 的另一个非常重要的应用是将递归转换为迭代。

深度递归程序的一个常见问题是它们可能会耗尽运行时栈。函数调用通常从将函数参数、局部变量和返回地址压入栈开始。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）运行时栈，导致运行时错误。这就是为什么命令式语言更喜欢循环而不是递归，以及为什么大多数程序员在学习递归之前先学习循环的主要原因。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，递归算法更加自然。

使用循环的问题在于它们需要可变状态。通常会有某种计数器或指针在每次循环迭代时前进和检查。这就是为什么避免可变状态的纯函数式语言必须使用递归来代替循环。但由于循环更高效且不会消耗运行时栈，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。

尾递归与 CPS

尾递归意味着递归调用发生在函数的最后。函数不会对尾调用的结果执行任何额外的操作。例如，以下程序不是尾递归的，因为它必须将 `i` 加到递归调用的结果上：

```
sum1 :: [Int] -> Int
sum1 [] = 0
sum1 (i : is) = i + sum1 is
```

相比之下，以下实现是尾递归的，因为对 `go` 的递归调用的结果直接返回，没有进一步修改：

```
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，而是将第一个参数 `n` 的值覆盖为 `n + i`，将指向列表头部的指针覆盖为指向其尾部的指针，然后跳转到函数的开头。

但请注意，这并不意味着 **Haskell** 编译器无法巧妙地优化第一个实现。这只是意味着第二个实现（尾递归的）保证会被转换为循环。

事实上，通过执行 CPS 转换，总是可以将递归转换为尾递归。这是因为延续封装了剩余的`计算`，所以它总是函数中的最后一个调用。

为了了解它在实践中如何工作，考虑一个简单的树遍历。让我们定义一个在节点和叶子中都存储字符串的树：

```
data Tree = Leaf String
          | Node Tree String Tree
```

为了连接所有这些字符串，我们使用先递归到左子树，然后递归到右子树的遍历：

```
show :: Tree -> String
show (Node lft s rgt) =
  let ls = show lft
      rs = show rgt
  in ls ++ s ++ rs
```

这绝对不是一个尾递归函数，而且如何将其转换为尾递归并不明显。然而，我们可以几乎机械地使用延续单子重写它：

```
showk :: Tree -> Cont r String
showk (Leaf s) = pure s
showk (Node lft s rgt) = do
  ls <- showk lft
  rs <- showk rgt
  pure (ls ++ s ++ rs)
```

然后我们可以使用平凡的延续 `id` 运行结果函数：

```
show :: Tree -> String
show t = runCont (showk t) id
```

这个实现自动是尾递归的。我们可以通过去糖化 `do` 语法清楚地看到这一点：

```
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)))
```

让我们分析这段代码。函数调用自身，传递左子树 `lft` 和以下延续：

```
\ls ->
  showk rgt (\rs ->
    k (ls ++ s ++ rs))
```

这个 `lambda` 又调用 `showk`，传递右子树 `rgt` 和另一个延续：

```
\rs -> k (ls ++ s ++ rs)
```

这个最内层的 `lambda` 可以访问所有三个字符串：左、中和右。它将它们连接起来，并使用结果调用最外层的延续 `k`。

在每种情况下，对 `showk` 的递归调用都是最后一个调用，其结果立即返回。此外，结果的类型是泛型 `r`，这本身保证了我们无法对其执行任何操作，即使我们想这样做。

当我们最终运行 `showk` 的结果时，我们传递给它（针对 `String` 类型实例化的）恒等函数：

```
show :: Tree -> String
show t = runCont' (showk t) id
  where runCont' cont k = cont k
```

使用命名函数

但假设我们的编程语言不支持匿名函数。是否可以用命名函数替换 `lambda`？我们在讨论伴随函子定理时已经这样做过。我们注意到，延续单子生成的 `lambda` 是闭包——它们从环境中捕获了一些值。如果我们想用命名函数替换它们，我们必须显式传递环境。

让我们用名为 `next` 的函数调用替换第一个 `lambda`，并以三元组 `(s, rgt, k)` 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))
```

这三个值是树当前节点的字符串、右子树和外部延续。

函数 `next` 对 `showk` 进行递归调用，传递右子树和名为 `conc` 的命名延续：

```
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延续的环境。它执行连接并使用结果调用外部延续：

```
conc :: (String, String, String -> r) -> String -> r
conc (ls, s, k) rs = k (ls ++ s ++ rs)
```

最后，我们定义平凡的恒等延续：

```
done :: String -> String
done s = s
```

我们用它来提取最终结果：

```
show t = showk t done
```

去函数化

延续传递风格需要使用高阶函数。如果这是一个问题，例如在实现分布式系统时，我们总是可以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
```

注意，这个递归数据结构可以重新解释为列表或栈。

```
data Kont = Done | Cons Sum Kont
```

其中：

```
data Sum = Next' String Tree | Conc' String String
```

这个列表是我们实现递归算法所需的运行时栈的版本。

由于我们只关心生成字符串作为最终结果，我们将近似 `String -> String` 函数类型。这是定义它的伴随的近似余单位（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，`showk` 函数可以在不依赖高阶函数的情况下实现：

```
showk :: Tree -> Kont -> String
showk (Leaf s) k = apply (k, s)
showk (Node lft s rgt) k = showk lft (Next s rgt k)
```

为了提取结果，我们使用 `Done` 调用它：

```
showTree t = showk t Done
```

15.6 范畴论中的单子 (Monad)

在范畴论中，单子最初产生于代数的研究。特别是，绑定运算符 (bind operator) 可以用来实现非常重要的替换操作。

替换

考虑这个简单的表达式类型。它由类型 `x` 参数化，我们可以用这个类型来命名变量：

```
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)
```

其中：

```
pure x = Var x
```

现在假设你想通过将变量 a 替换为 $x_1 + 2$ ，将 b 替换为 x_2 来进行替换（为简单起见，我们不考虑字母表中的其他字母）。这个替换由 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`。

当我们把这个 Kleisli 箭头绑定到 `ex` 时：

```
ex' :: Ex String
ex' = ex >>= sub
```

我们得到了预期的树，对应于 $(2 + (x_1 + 2)) + x_2$ 。

单子作为幺半群

让我们分析使用 `join` 的单子定义：

```
class Functor m => Monad m where
  join :: m (m a) -> m a
  pure :: a -> m a
```


我们有一个自函子 m 和两个多态函数。

在范畴论中，定义单子的函子传统上记为 T （可能是因为单子最初被称为“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”（ T 与自身的复合）映射到 T ：

$$\mu: T \circ T \rightarrow T$$

（当然，只有自函子才能以这种方式平方。）

第二个对应于 `pure`，将恒等函子映射到 T ：

$$\eta: \text{Id} \rightarrow T$$

将其与我们之前在幺半范畴中定义的幺半群进行比较：

$$\mu: m \otimes m \rightarrow m$$

$$\eta: I \rightarrow m$$

相似之处非常明显。这就是为什么我们经常将自然变换 μ 称为单子乘法 (monadic multiplication)。但在什么范畴中，函子的复合可以被视为张量积？

进入自函子范畴。这个范畴中的对象是自函子，箭头是自然变换。

但这个范畴还有更多的结构。我们知道任何两个自函子都可以复合。如果我们想把自函子视为对象，如何解释这种复合？一个操作将两个对象并产生第三个对象，看起来像张量积。张量积的唯一条件是它在两个参数上都是函子性的。也就是说，给定一对箭头：

$$\alpha: T \rightarrow T'$$

$$\beta: S \rightarrow S'$$

我们可以将其提升到张量积的映射：

$$\alpha \otimes \beta: T \otimes S \rightarrow T' \otimes S'$$

在自函子范畴中，箭头是自然变换，因此如果我们将 \otimes 替换为 \circ ，提升就是映射：

$$\alpha \circ \beta: T \circ T' \rightarrow S \circ S'$$

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）。

这个幺半范畴中的单位对象是恒等自函子，单位律“严格”满足，即

$$\text{Id} \circ T = T = T \circ \text{Id}$$

我们不需要任何单位子 (unitor)。我们也不需要任何结合子 (associator)，因为函子复合自动满足结合律。

单位子和结合子都是恒等态射的么半范畴称为严格么半范畴。

然而，请注意，复合不是对称的，所以这不是一个对称么半范畴。

综上所述，单子是自函子么半范畴中的么半群。

单子 (T, η, μ) 由自函子范畴中的一个对象——即自函子 T ；和两个箭头——即自然变换组成：

$$\begin{aligned}\eta &: \text{Id} \rightarrow T \\ \mu &: T \circ T \rightarrow T\end{aligned}$$

要成为么半群，这些箭头必须满足么半律。以下是单位律（单位子被严格等式替换）：

$$\begin{array}{ccccc}\text{Id} \circ T & \xrightarrow{\eta \circ T} & T \circ T & \xleftarrow{T \circ \eta} & T \circ \text{Id} \\ & \searrow = & \downarrow \mu & \swarrow = & \\ & & T & & \end{array}$$

这是结合律：

$$\begin{array}{ccc}(T \circ T) \circ T & \xrightarrow{=} & T \circ (T \circ T) \\ \downarrow \mu \circ T & & \downarrow T \circ \mu \\ T \circ T & \xrightarrow{\mu} & T \circ T \\ & \searrow \mu \quad \swarrow \mu & \\ & T & \end{array}$$

我们使用了水平复合的 whiskering 表示法 $\mu \circ T$ 和 $T \circ \mu$ 。

这些是用 μ 和 η 表示的单子律。它们可以直接转换为 [join](#) 和 [pure](#) 的定律。它们也等价于由箭头 $a \rightarrow Tb$ 构建的 Kleisli 范畴的定律。

15.7 自由单子 (Free Monads)

单子 (monad) 允许我们指定一系列可能产生副作用的操作。这样的序列既告诉计算机要做什么，也告诉它如何去做。但有时需要更大的灵活性：我们希望将“做什么”与“如何做”分离开来。自由单子 (free monad) 允许我们生成操作序列，而无需承诺使用特定的单子来执行它。这与定义自由么半群 (free monoid，即列表) 类似，后者允许我们推迟选择应用于它的代数；或者与在编译为可执行代码之前创建抽象语法树 (AST) 类似。

自由构造被定义为遗忘函子 (forgetful functor) 的左伴随。因此，我们首先需要定义“遗忘单子结构”的含义。由于单子是一个配备了额外结构的自函子 (endofunctor)，我们希望遗忘这个结构。我们取一个单子 (T, η, μ) 并只保留 T 。但为了将这种映射定义为一个函子，我们首先需要定义单子的范畴。

单子的范畴

单子范畴 $\mathbf{Mon}(\mathcal{C})$ 中的对象是单子 (T, η, μ) 。我们可以将两个单子 (T, η, μ) 和 (T', η', μ') 之间的箭头定义为两个自函子之间的自然变换：

$$\lambda: T \rightarrow T'$$

然而，由于单子是带有结构的自函子，我们希望这些自然变换能够保持结构。单位元的保持意味着以下图表必须交换：

$$\begin{array}{ccc} & \text{Id} & \\ \eta \swarrow & & \searrow \eta' \\ T & \xrightarrow{\lambda} & T' \end{array}$$

乘法的保持意味着以下图表必须交换：

$$\begin{array}{ccc} T \circ T & \xrightarrow{\lambda \circ \lambda} & T' \circ T' \\ \downarrow \mu & & \downarrow \mu' \\ T & \xrightarrow{\lambda} & T' \end{array}$$

另一种看待 $\mathbf{Mon}(\mathcal{C})$ 的方式是，它是么半范畴 $([\mathcal{C}, \mathcal{C}], \circ, \text{Id})$ 中的么半群范畴。

自由单子

现在有了单子的范畴，我们可以定义遗忘函子：

$$U: \mathbf{Mon}(\mathcal{C}) \rightarrow [\mathcal{C}, \mathcal{C}]$$

它将每个三元组 (T, η, μ) 映射到 T ，并将每个单子态射映射到底层的自然变换。

我们希望自由单子由这个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，这与大小问题有关：单子往往会“膨胀”事物。结论是，自由单子对某些自函子存在，但并非对所有自函子都存在。因此，我们不能通过伴随来定义自由单子。幸运的是，在大多数感兴趣的情况下，自由单子可以定义为代数的固定点。

这种构造类似于我们如何将自由么半群定义为列表函子的初始代数：

```
data ListF a x = NilF | ConsF a x
```

或更一般的函子：

$$F_a x = I + a \otimes x$$

在么半范畴 $(\mathcal{C}, \otimes, I)$ 中。

然而，这一次，单子被定义为么半群的么半范畴是自函子范畴 $([\mathcal{C}, \mathcal{C}], \circ, \text{Id})$ 。这个范畴中的自由么半群是映射函子到函子的高阶“列表”函子的初始代数：

$$\Phi_F G = \text{Id} + F \circ G$$

这里，两个函子的余积（coproduct）是逐点定义的。在对象上：

$$(F + G)a = Fa + Ga$$

在箭头上：

$$(F + G)f = Ff + Gf$$

（我们利用余积的函子性来形成两个态射的余积。我们假设 \mathcal{C} 是余笛卡尔的，即所有余积都存在。）

初始代数是这个算子的（最小）固定点，或递归方程的解：

$$L_F \cong \text{Id} + F \circ L_F$$

这个公式建立了两个函子之间的自然同构。

Haskell 中的自由单子

在范畴论中，函子只是两个范畴之间的映射。在 Haskell 中，如果我们想实现比简单的 Haskell 自函子更高级的东西，我们必须引入一些新的类。Haskell 函子是一个类型构造器——即类型到类型的映射：

```
f :: Type -> Type
```

以及 `fmap` 的实现——即函数到函数的映射。

一个高阶函子（higher order functor）将函子映射到函子。因此，它是一个类型构造器，接受一个类型构造器并生成另一个类型构造器：

```
hf :: (Type -> Type) -> (Type -> Type)
```

它还必须将自然变换映射到自然变换。为了在 Haskell 中实现它，我们定义一个新的类型类：

```
class HFunctor (hf :: (Type -> Type) -> Type -> Type) where
  hmap :: (Functor f, Functor g) =>
    Natural f g -> Natural (hf f) (hf g)
```

我们的高阶列表函子：

$$\Phi_F G = \text{Id} + F \circ G$$

就是这样一个 `HFunctor`，它还依赖于另一个类型构造器 `f`。由于它是一个和类型，它将有二个构造器，对应于二个注入：

```
data Phi f g a where
  IdF :: a -> Phi f g a
  CompF :: f (g a) -> Phi f g a
```

给定一个自然变换 `alpha :: Nat g h`，它生成另一个自然变换 `Nat (Phi f g) (Phi f h)`：

```
instance Functor f => HFunctor (Phi f) where
  hmap :: (Functor f, Functor g, Functor h) =>
    Natural g h -> Natural (Phi f g) (Phi f h)
  hmap alpha (IdF a) = IdF a
  hmap alpha (CompF fga) = CompF (fmap alpha fga)
```

这只是余积的函子性和函子组合的应用。

我们必须分别断言将 `Phi f` 应用于函子 `g` 的结果的函子性：

```
instance (Functor f, Functor g) => Functor (Phi f g) where
  fmap h (IdF a) = IdF (h a)
  fmap h (CompF fga) = CompF (fmap (fmap h) fga)
```

生成自由单子的同构：

$$L_F \cong \text{Id} + F \circ L_F$$

可以看作是一个递归数据类型的定义。从右到左，我们有自然变换，我们将其识别为初始代数的结构映射：

$$\iota : \text{Id} + F \circ L_F \rightarrow L_F$$

它是从和类型中映射出来的，因此它等价于一对自然变换：

$$\eta : \text{Id} \rightarrow L_F$$

$$\varphi : F \circ L_F \rightarrow L_F$$

当将其翻译到 Haskell 时，这些变换的组件成为两个构造器。它们定义了以下由函子 `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，构造器 `Free` 接受一个装满 `(FreeMonad f a)` 的函子并将其存储起来。因此，类型为 `FreeMonad f a` 的值是一棵树，其中每个节点都是一个装满分支的函子，每个叶子包含一个类型为 `a` 的值。

`FreeMonad` 是一个高阶函子：

```
instance HFunctor FreeMonad where
  hmap :: (Functor f, Functor g) =>
    Natural f g -> Natural (FreeMonad f) (FreeMonad g)
  hmap _ (Pure a) = Pure a
  hmap alpha (Free ffa) = Free (alpha (fmap (hmap alpha) ffa))
```

它的结果是一个 `Functor`：

```
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，外部的 `fmap` 使用 `f` 的 `Functor` 实例，而内部的 `fmap g` 递归到分支中。

通过构造，`FreeMonad` 是一个 `Monad`。单子单位 `eta` 只是恒等函子的薄封装：

```
eta :: a -> FreeMonad f a
eta a = Pure a
```

单子乘法，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
  m >>= k = mu (fmap k m)
```

其中：

```
pure a = eta a
```

我们也可以直接定义绑定：

```
(Pure a)    >>= k = k a
(Free ffa) >>= k = Free (fmap (>>= k) ffa)
```

自由单子将单子操作累积在树状结构中，而无需承诺任何特定的求值策略。这棵树可以使用代数进行“解释”。但这一次，它是在自函子范畴中的代数，因此它的载体是一个自函子 G ，结构映射 α 是一个自然变换 $\Phi_F G \rightarrow G$ ：

$$\alpha: \text{Id} + F \circ G \rightarrow G$$

这个自然变换，作为从和类型中映射出来的，等价于一对自然变换：

$$\lambda: \text{Id} \rightarrow G$$

$$\rho: F \circ G \rightarrow G$$

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)
```

由于自由单子是初始代数，存在一个唯一的映射，即 **catamorphism**，从它到任何其他代数。回想我们如何为常规代数定义 **catamorphism**：

```
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

类似的图表定义了自由单子的 **catamorphism**：

$$\begin{array}{ccc} \text{Id} + F \circ L_F & \xrightarrow{\text{hmap}(\text{mcata } \alpha)} & \text{Id} + F \circ G \\ \downarrow \iota = \langle \eta, \varphi \rangle & & \downarrow \alpha = \langle \lambda, \rho \rangle \\ L_F & \xrightarrow{\text{mcata } \alpha} & G \end{array}$$

在 Haskell 中，我们通过对自由单子的两个构造器进行模式匹配来实现它（这对应于反转初始代数 ι ）。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 ρ 应用于结果：

```
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)
```

许多树状单子实际上是简单函子的自由单子。另一方面，列名单子不是自由的，因为它的 **join** 不可逆地将列表压在一起。

Exercise 15.7.1. 一个（非空）玫瑰树定义为：

```
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 15.7.2. 实现非空二叉树和 `FreeMonad Bin a` 之间的转换，其中：

```
data Bin a = Bin a a
```

栈计算器示例

作为一个例子，让我们考虑一个作为嵌入式领域特定语言（EDSL）实现的栈计算器。我们将使用自由单子来累积用这种语言编写的简单命令。

命令由函子 `StackF` 定义。将参数 `k` 视为延续。

```
data StackF k = Push Int k
              | Top (Int -> k)
              | Pop k
              | Add k
    deriving Functor
```

例如，`Push` 应该将一个整数压入栈中，然后调用延续 `k`。

这个函子的自由单子可以被视为一棵树，大多数分支只有一个子节点，从而形成列表。例外是 `Top` 节点，它有许多子节点，每个 `Int` 值对应一个子节点。

这是这个函子的自由单子：

```
type FreeStack = FreeMonad StackF
```

为了创建领域特定的程序，我们将定义一些辅助函数。有一个通用的函数，它将一个装满值的函子提升为自由单子：

```
liftF :: (Functor f) => f r -> FreeMonad f r
liftF fr = Free (fmap Pure fr)
```

我们还需要一系列“智能构造器”，它们是我们自由单子的 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 ())

```

由于自由单子是一个单子，我们可以方便地使用 `do` 表示法组合 Kleisli 箭头。例如，这里是一个将两个数字相加并返回它们的和的玩具程序：

```

calc :: FreeStack Int
calc = do
  push 3
  push 4
  add
  x <- top
  pop
  pure x

```

为了执行这个程序，我们需要定义一个代数，其载体是一个自函子。由于我们想实现一个基于栈的计算器，我们将使用状态函子的一个版本。它的状态是一个栈——一个整数列表。状态函子被定义为一个函数类型；这里它是一个接受一个列表并返回一个新列表与类型参数 `k` 耦合的函数：

```

newtype StackAction k = St ([Int] -> ([Int], k))
  deriving Functor

```

为了运行这个动作，我们将函数应用于栈：

```

runAction :: StackAction k -> [Int] -> ([Int], k)
runAction (St act) ns = act ns

```

我们将代数定义为一对多态函数，对应于自由单子的两个构造器，**Pure** 和 **Free**：

```
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))
```

第二个函数根据命令的类型进行模式匹配。每个命令都带有一个延续。这个延续必须在一个（可能被修改的）栈上运行。每个命令以不同的方式修改栈：

```
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** 丢弃栈顶。**Top** 从栈顶取一个整数并使用它来选择要执行的分支。它通过将函数 **ik** 应用于整数来实现。**Add** 将栈顶的两个数字相加并压入结果。

注意，我们定义的代数不涉及递归。将递归与操作分离是自由单子方法的一个优势。递归被一次性编码在 **catamorphism** 中。

以下是可用于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。一个格式错误的程序将导致运行时错误。一个更健壮的实现将使用允许错误传播的代数。

使用自由单子的另一个优势是，相同的程序可以使用不同的代数进行解释。

Exercise 15.7.3. 实现一个“漂亮打印机”，显示使用我们的自由单子构建的程序。提示：实现使用 **Const** 函子作为载体的代数：

```
showAlg :: MAlg StackF (Const String) a
```

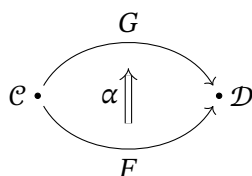
单子与伴随

16.1 弦图

一条线将平面分割。我们可以将其视为将平面分割，或者将平面的两半连接起来。

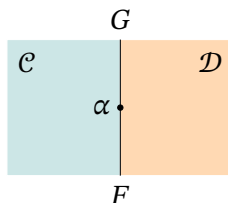
一个点将一条线分割。我们可以将其视为将两条半线分开，或者将它们连接在一起。

这是一个图表，其中两个类别表示为点，两个函子表示为箭头，自然变换表示为双箭头。



但同样的想法可以通过将类别表示为平面的区域，函子表示为区域之间的线，自然变换表示为连接线段的点来表示。

这个想法是，函子总是在一对类别之间进行，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进行，因此可以将其绘制为连接两条线段的点。



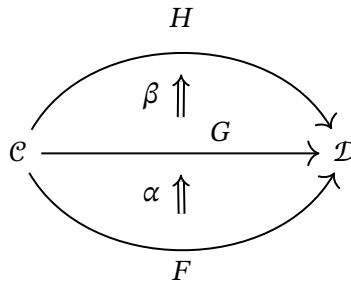
这是一个弦图的例子。你从下到上、从左到右阅读这样的图表（想象 (x,y) 坐标系）。

图表的底部显示了从 C 到 D 的函子 F 。图表的顶部显示了在同一对类别之间的函子 G 。转换发生在中间，自然变换 α 将 F 映射到 G 。

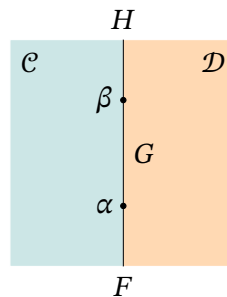
在 Haskell 中，这个图表被解释为两个自函子之间的多态函数：

```
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。但让我们将其应用到更有趣的东西上：自然变换的垂直组合：



相应的弦图显示了两个类别和它们之间的三个函子，由两个自然变换连接。



如你所见，你可以通过从下到上扫描弦图来重建原始图表。

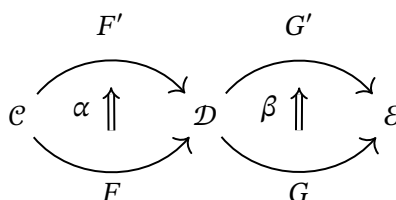
再次，在 Haskell 中，我们将处理三个自函子，以及 `beta` 在 `alpha` 之后的垂直组合：

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

使用常规函数组合实现：

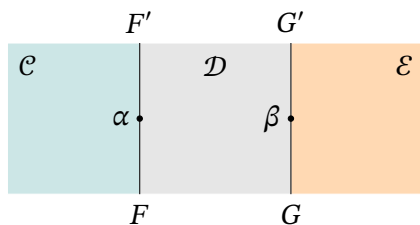
```
beta_alpha :: forall x. F x -> H x
beta_alpha = beta . alpha
```

让我们继续自然变换的水平组合：



这次我们有三个类别，因此我们将有三个区域。

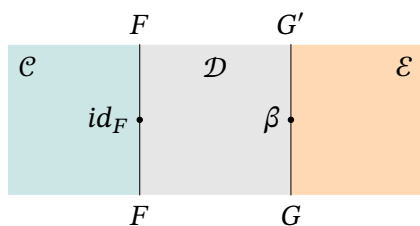
弦图的底部对应于函子 $G \circ F$ 的组合（按此顺序）。顶部对应于 $G' \circ F'$ 。一个自然变换 α 将 F 连接到 F' ；另一个 β 将 G 连接到 G' 。



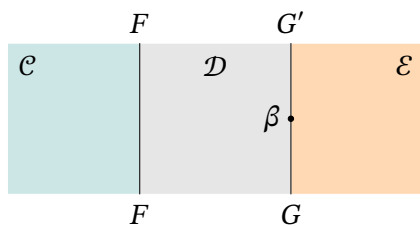
在这个新系统中，平行的垂直线对应于函子组合。

你可以将自然变换的水平组合视为沿着图表中间的假想水平线发生。但如果有人在绘制图表时粗心，其中一个点比另一个点高一点呢？事实证明，由于交换律，点的确切位置并不重要。

但首先，让我们说明 **whiskering**：其中一个自然变换是恒等变换的水平组合。我们可以这样绘制：



但实际上，恒等变换可以插入到垂直线上的任何点，因此我们甚至不需要绘制它。以下图表表示 $\beta \circ F$ 的 whiskering。



在 Haskell 中，其中 `beta` 是一个多态函数：

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

我们这样阅读这个图表：

```
beta_f :: forall x. G (F x) -> G' (F x)
beta_f = beta
```

理解类型检查器为正确类型实例化多态函数 `beta`。

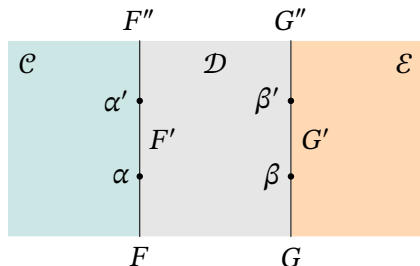
同样，你可以轻松想象 $G \circ \alpha$ 的图表及其 Haskell 实现：

```
g_alpha :: forall x. G (F x) -> G (F' x)
beta_f = fmap alpha
```

其中:

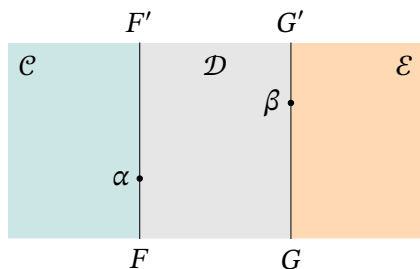
```
alpha :: forall x. F x -> F' x
```

这是对应于交换律的弦图:



这个图表故意模糊。我们是应该先进行自然变换的垂直组合, 然后进行水平组合? 还是应该将 $\beta \circ \alpha$ 和 $\beta' \circ \alpha'$ 水平组合, 然后将结果垂直组合? 交换律说这无关紧要: 结果是相同的。

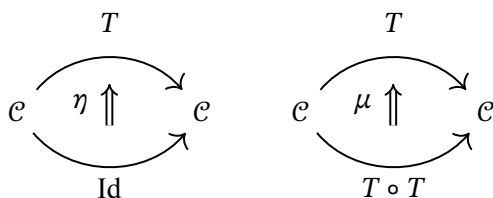
现在尝试在这个图表中用恒等变换替换一对自然变换。如果你替换 α' 和 β' , 你将得到 $\beta \circ \alpha$ 的水平组合。如果你用恒等自然变换替换 α' 和 β , 并将 β' 重命名为 β , 你将得到 α 相对于 β 下移的图表, 依此类推。



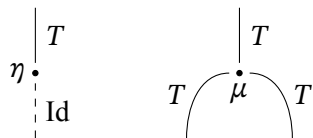
交换律告诉我们所有这些图表都是相等的。我们可以自由地滑动自然变换, 就像在弦上滑动珠子一样。

单子的弦图

单子被定义为一个配备了两个自然变换的自函子, 如下面的图表所示:



由于我们只处理一个类别，当将这些图表转换为弦图时，我们可以省略类别的命名（和阴影），只绘制弦。

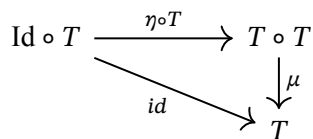


在第一个图表中，通常省略对应于恒等函子的虚线。 η 点可以自由地将 T 线注入图表中。两条 T 线可以通过 μ 点连接。

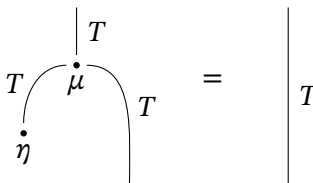
弦图在表达单子定律时特别有用。例如，我们有左单位律：

$$\mu \circ (\eta \circ T) = id$$

可以可视化为一个交换图表：

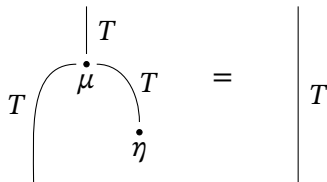


相应的弦图表示通过这个图表的两个路径的相等性：

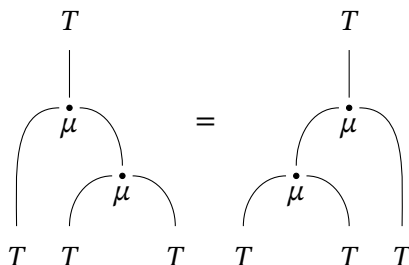


你可以将这种相等性视为拉紧顶部和底部弦的结果，导致 η 附加部分被缩回直线。

有一个对称的右单位律：



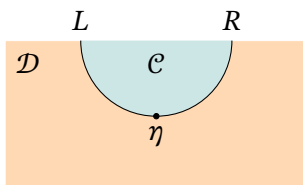
最后，这是弦图形式的结合律：



伴随的弦图

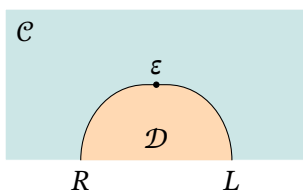
正如我们之前讨论的，伴随（adjunction）是一对函子之间的关系，即 $L: \mathcal{D} \rightarrow \mathcal{C}$ 和 $R: \mathcal{C} \rightarrow \mathcal{D}$ 。它可以通过一对自然变换来定义，即单位（unit） η 和余单位（counit） ε ，它们满足三角恒等式。

伴随的单位可以用一个“杯”形图来表示：



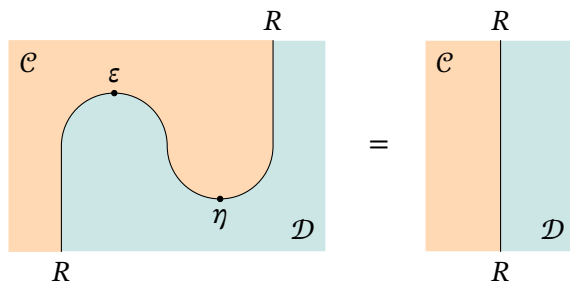
图中底部的恒等函子被省略了。 η 点将其下方的恒等函子转换为其上方的复合函子 $R \circ L$ 。

类似地，余单位可以可视化为一个“帽”形弦图，其中顶部的恒等函子是隐含的：



三角恒等式可以很容易地用弦图表示。它们也具有直观的意义，因为你可以想象从两侧拉弦以拉直曲线。

例如，这是第一个三角恒等式，有时称为之字形恒等式：



从左图自下而上读取，产生一系列映射：

$$Id_{\mathcal{D}} \circ R \xrightarrow{\eta \circ R} R \circ L \circ R \xrightarrow{R \circ \varepsilon} R \circ Id_{\mathcal{C}}$$

这必须等于右侧，可以将其解释为 R 上的（隐式的）恒等自然变换。

在 R 是自函子的情况下，我们可以直接将第一个图转换为 Haskell。伴随的单位 η 通过 R 的“whiskering”导致多态函数 `unit` 在 $\mathbf{R} \ x$ 处实例化。 ε 的“whiskering”导致 `counit` 通过函子 R 的提升。垂直复合转换为函数复合：

```
triangle :: forall x. R x -> R x
triangle = fmap counit . unit
```

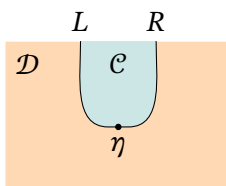
Exercise 16.1.1. 绘制第二个三角恒等式的弦图并将其转换为 *Haskell*。

16.2 从伴随函子导出的单子

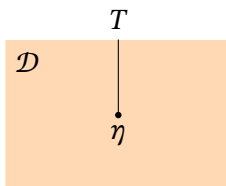
你可能已经注意到，相同的符号 η 被用于伴随函子的单位（`unit`）和单子的单位。这并非巧合。

乍一看，我们似乎是在比较苹果和橘子：伴随函子是由两个范畴之间的两个函子定义的，而单子是由作用于单个范畴的一个自函子（`endofunctor`）定义的。然而，两个方向相反的函子的复合是一个自函子，而伴随函子的单位将恒等自函子映射到自函子 $R \circ L$ 。

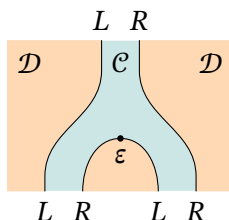
比较以下图表：



与定义单子单位的图表：



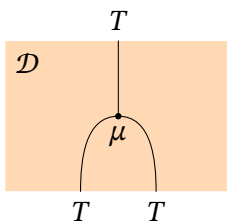
事实证明，对于任何伴随函子 $L \dashv R$ ，自函子 $T = R \circ L$ 都是一个单子，其乘法 μ 由以下图表定义：



从下到上阅读此图表，我们得到以下变换（想象在点处水平切割）：

$$R \circ L \circ R \circ L \xrightarrow{R \circ \varepsilon \circ L} R \circ L$$

将其与单子 μ 的定义进行比较：



我们得到单子 $R \circ L$ 的 μ 定义为 ε 的双重 whiskering：

$$\mu = R \circ \varepsilon \circ L$$

在 Haskell 中，用 ε 定义 μ 的字符串图表的翻译总是可能的。单子的乘法，即 `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` 定义的提升，该自函子定义为 $R \circ L$ 的复合。注意，在这种情况下， \mathcal{D} 是 Haskell 的类型和函数的范畴，但 \mathcal{C} 可以是一个外部范畴。

为了完善这一图景，我们可以使用字符串图表来推导单子定律，利用三角恒等式。技巧是将单子定律中的所有字符串替换为平行的字符串对，然后根据规则重新排列它们。

总结一下，每个伴随函子 $L \dashv R$ 与单位 η 和余单位 ε 一起定义了一个单子 $(R \circ L, \eta, R \circ \varepsilon \circ L)$ 。

我们稍后会看到，对偶地，另一种复合 $L \circ R$ 定义了一个余单子（comonad）。

Exercise 16.2.1. 绘制字符串图表来说明从伴随函子导出的单子的单子定律（单位和结合性）。

16.3 伴随函子生成的单子示例

我们将通过几个伴随函子的例子，这些伴随函子生成了我们在编程中使用的一些单子。当我们讨论单子变换器时，我们会进一步扩展这些例子。

大多数例子涉及离开 Haskell 类型和函数范畴的函子，尽管生成单子的往返最终是一个自函子。这就是为什么在 Haskell 中通常无法表达这样的伴随函子。

此外，由于与数据构造函数的显式命名相关的许多簿记工作，这可能会使底层公式的简洁性变得模糊，而这些簿记工作对于类型推断是必要的。

自由幺半群与列表单子

列表单子是由我们之前见过的自由幺半群伴随函子生成的。这个伴随函子的单位 $\eta_X : X \rightarrow U(FX)$ 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。单位 η_X 将 X 的元素映射到这样的单元素列表：

```
return x = [x]
```

为了实现余单位 $\epsilon_M : F(UM) \rightarrow M$ ，我们取一个幺半群 M ，忘记其乘法，并使用其元素集合作为新自由幺半群的生成元。 M 处的余单位分量是从自由幺半群回到 M 的幺半群态射，或者在 Haskell 中为 $[m] \rightarrow m$ 。事实证明，这个幺半群态射是 **catamorphism** 的一个特例。

首先，回忆一下 Haskell 中一般列表 **catamorphism** 的实现：

```
foldMap :: Monoid m => (a -> m) -> ([a] -> m)
foldMap f = foldr mappend mempty . fmap f
```

在这里，我们将 $(a \rightarrow m)$ 解释为从 a 到幺半群 m 的底层集合的常规函数。结果被解释为从由 a 生成的自由幺半群（即 a 的列表）到 m 的幺半群态射。这只是伴随函子的一个方向：

$$\mathbf{Set}(a, Um) \cong \mathbf{Mon}(Fa, m)$$

为了将余单位作为幺半群态射 $[m] \rightarrow m$ ，我们将 **foldMap** 应用于恒等函数。结果是 **(foldMap id)**，或者用 **foldr** 表示：

```
epsilon = foldr mappend mempty
```

它是一个幺半群态射，因为它将空列表映射到幺半群单位，并将连接映射到幺半群积。

单子乘法，或 **join**，由余单位的 **whiskering** 给出：

$$\mu = U \circ \epsilon \circ F$$

你可以很容易地相信，这里的左 **whiskering** 并没有做太多事情，因为它只是通过遗忘函子提升了一个幺半群态射（它保留了函数，同时忘记了其保持结构的特殊性质）。

由 F 进行的右 **whiskering** 更有趣。这意味着分量 μ_X 对应于 FX 处的 ϵ 分量，即由集合 X 生成的自由幺半群。这个自由幺半群定义为：

```
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。

柯里化伴随函子与状态单子

状态单子是由我们用来定义指数对象的柯里化伴随函子生成的。左函子由与某个固定对象 s 的积定义：

$$L_s a = a \times s$$

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，由相同的对象 s 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：

$$(R_s \circ L_s) a = (a \times s)^s$$

在 Haskell 中，我们可以将其写为：

```
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))
```

伴随函子 $L_s \dashv R_s$ 的单位是一个映射：

$$\eta_a : a \rightarrow (a \times s)^s$$

可以在 Haskell 中实现为：

```
unit :: a -> R s (L s a)
unit a = R (\s -> L (a, s))
```

你可能会在其中识别出状态单子的`return`的薄封装版本：

```
return :: a -> State s a
return a = State (\s -> (a, s))
```

这是 c 处这个伴随函子的余单位分量：

$$\varepsilon_c : c^s \times s \rightarrow c$$

可以在 Haskell 中实现为：

```
counit :: L s (R s a) -> a
counit (L ((R f), s)) = f s
```

在剥离数据构造函数后，这等同于`apply`，或`runState`的非柯里化版本。

单子乘法 μ 由 ε 的 whiskering 给出：

$$\mu = R_s \circ \varepsilon \circ L_s$$

这是它在 Haskell 中的翻译：

```
mu :: R s (L s (R s (L s a))) -> R s (L s a)
mu = fmap counit
```

右 whiskering 除了选择自然变换的分量外，没有做任何事情。这是由 Haskell 的类型推断引擎自动完成的。左 whiskering 通过提升自然变换的分量来完成。再次，类型推断选择了正确的`fmap`实现——在这里，它等同于预组合。

将其与`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，其类型签名变为：

```
uncurry runState :: (State s a, s) -> (a, s)
```

这等同于`counit`的类型签名。

当部分应用时，`runState`只是剥离数据构造函数，暴露底层函数类型：

```
runState st :: s -> (a, s)
```

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。这个幺半群用于累积日志条目。我们将使用的伴随函子涉及该幺半群的 M -集范畴。

一个 M -集是一个集合 S ，我们在其上定义了一个幺半群 M 的作用。这样的作用是一个映射：

$$a : M \times S \rightarrow S$$

我们经常使用作用的柯里化版本，将幺半群元素放在下标位置。因此 a_m 成为一个函数 $S \rightarrow S$ 。

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它们的幺半群积的作用：

$$a_{m_1} \circ a_{m_2} = a_{m_1 \cdot m_2}$$

这种乘法顺序的选择定义了所谓的左作用。（右作用将两个幺半群元素在右侧交换。）

M -集形成一个范畴 \mathbf{MSet} 。对象是 $(S, a : M \times S \rightarrow S)$ ，箭头是等变映射，即保持作用的集合之间的函数。

一个函数 $f : S \rightarrow R$ 是从 (S, a) 到 (R, b) 的等变映射，如果对于每个 $m \in M$ ，以下图表交换：

$$\begin{array}{ccc} S & \xrightarrow{f} & R \\ \downarrow a_m & & \downarrow b_m \\ S & \xrightarrow{f} & R \end{array}$$

换句话说，无论我们是先做作用 a_m ，然后映射集合；还是先映射集合，然后做相应的作用 b_m ，结果都是一样的。

有一个遗忘函子 U 从 \mathbf{MSet} 到 \mathbf{Set} ，它将集合 S 分配给对 (S, a) ，从而忘记作用。

与之对应的是一个自由函子 F 。它对集合 S 的作用产生一个 M -集。它是一个集合，是 S 和 M 的笛卡尔积，其中 M 被视为元素集合（换句话说，遗忘函子对幺半群作用的结果）。这个 M -集的一个元素是 $(x \in S, m \in M)$ ，自由作用定义为：

$$\phi_n : (x, m) \mapsto (x, n \cdot m)$$

保持元素 x 不变，只乘以 m 分量。

为了证明 F 是 U 的左伴随函子，我们必须构造以下自然同构：

$$\mathbf{MSet}(FS, Q) \cong \mathbf{Set}(S, UQ)$$

对于任何集合 S 和任何 \mathbf{M} -集 Q 。如果我们将 Q 表示为 (R, b) ，伴随函子右侧的元素是一个普通函数 $u: S \rightarrow R$ 。我们可以使用这个函数在左侧构造一个等变映射。

这里的技巧是注意到这样的等变映射 $f: FS \rightarrow Q$ 完全由其在形式为 $(x, 1) \in FS$ 的元素上的作用决定，其中 1 是么半群单位。

实际上，从等变条件可以得出：

$$\begin{array}{ccc} (x, 1) & \xrightarrow{f} & r \\ \downarrow \phi_m & & \downarrow b_m \\ (x, m \cdot 1) & \xrightarrow{f} & r' \end{array}$$

或者：

$$f(\phi_m(x, 1)) = f(x, m) = b_m(f(x, 1))$$

因此，每个函数 $u: S \rightarrow R$ 唯一地定义了一个等变映射 $f: FS \rightarrow Q$ ，由下式给出：

$$f(x, m) = b_m(ux)$$

这个伴随函子的单位 $\eta_S: S \rightarrow U(FS)$ 将一个元素 x 映射到对 $(x, 1)$ 。将其与 `Writer` 单子的 `return` 定义进行比较：

```
return a = Writer (a, mempty)
```

余单位由一个等变映射给出：

$$\varepsilon_Q: F(UQ) \rightarrow Q$$

左侧是通过取 Q 的底层集合并将其与 M 的底层集合的乘积构造的 \mathbf{M} -集。 Q 的原始作用被遗忘，并被自由作用取代。余单位的明显选择是：

$$\varepsilon_Q: (x, m) \mapsto a_m x$$

其中 x 是 Q 的（底层集合的）元素， a 是 Q 中定义的作用。

单子乘法 μ 由余单位的 `whiskering` 给出。

$$\mu = U \circ \varepsilon \circ F$$

这意味着在 ε_Q 的定义中用自由 \mathbf{M} -集替换 Q ，其作用是自由作用。换句话说，我们用 (x, m) 替换 x ，用 ϕ_n 替换 a_n 。（与 U 的 `whiskering` 没有改变任何东西。）

$$\mu_S: ((x, m), n) \mapsto \phi_n(x, m) = (x, n \cdot m)$$

将其与 `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)
```

点对象与 `Maybe` 单子

点对象是具有指定元素的对象。由于选择元素是使用从终端对象的箭头完成的，点对象的范畴使用对 $(a, p : 1 \rightarrow a)$ 定义，其中 a 是 \mathcal{C} 中的对象。

这些对之间的态射是 \mathcal{C} 中保持点的箭头。因此，从 $(a, p : 1 \rightarrow a)$ 到 $(b, q : 1 \rightarrow b)$ 的态射是一个箭头 $f : a \rightarrow b$ ，使得 $q = f \circ p$ 。这个范畴也被称为余切片范畴，并写为 $1/\mathcal{C}$ 。

有一个明显的遗忘函子 $U : 1/\mathcal{C} \rightarrow \mathcal{C}$ ，它忘记了点。它的左伴随函子是一个自由函子 F ，它将对象 a 映射到对 $(1 + a, \text{Left})$ 。换句话说， F 使用余积自由地向对象添加一个点。

`Either` 单子类似地通过用固定对象 e 替换 1 来构造。

Exercise 16.3.1. 证明 $U \circ F$ 是 `Maybe` 单子。

延续单子

延续单子是根据集合范畴中的一对逆变函子定义的。我们不需要修改伴随函子的定义来处理逆变函子。只需为其中一个端点选择相反范畴即可。

我们将左函子定义为：

$$L_Z : \mathbf{Set}^{op} \rightarrow \mathbf{Set}$$

它将集合 X 映射到 \mathbf{Set} 中的 hom -集：

$$L_Z X = \mathbf{Set}(X, Z)$$

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：

$$R_Z : \mathbf{Set} \rightarrow \mathbf{Set}^{op}$$

$$R_Z X = \mathbf{Set}^{op}(Z, X) = \mathbf{Set}(X, Z)$$

组合 $R \circ L$ 可以在 Haskell 中写为 `((x -> r) -> r)`，这与定义延续单子的（协变）自函子相同。

16.4 单子变换器 (Monad Transformers)

假设你想组合多个效应，例如状态与可能的失败。一种选择是从头定义你自己的单子。你定义一个函子：


```
newtype MaybeState s a = MS (s -> Maybe (a, s))
deriving Functor
```

同时定义提取结果（或报告失败）的函数：

```
runMaybeState :: MaybeState s a -> s -> Maybe (a, s)
runMaybeState (MS h) s = h s
```

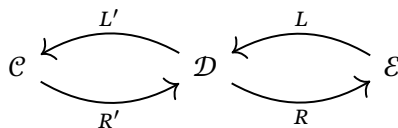
你为它定义单子实例：

```
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，检查它是否满足单子定律。

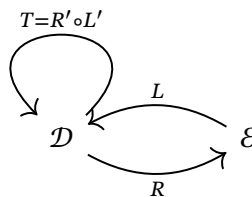
没有通用的方法来组合单子。从这个意义上说，单子是不可组合的。然而，我们知道伴随 (adjunctions) 是可组合的。我们也已经看到如何从伴随中获取单子，并且，正如我们将要看到的，每个单子都可以通过这种方式获得。因此，如果我们能匹配伴随，它们生成的单子将自动组合。

考虑两个可组合的伴随：



这幅图中有三个单子。有“内部”单子 $R' \circ L'$ 和“外部”单子 $R \circ L$ 以及组合 $R \circ R' \circ L' \circ L$ 。

如果我们称内部单子为 $T = R' \circ L'$ ，那么 $R \circ T \circ L$ 就是组合单子，称为单子变换器，因为它将单子 T 转换为一个新的单子。



在我们的例子中，我们可以将 `Maybe` 视为内部单子：

$$Ta = 1 + a$$

它使用生成状态单子的外部伴随 $L_s \dashv R_s$ 进行变换：

$$L_s a = a \times s$$

$$R_s c = c^s$$

结果是：

$$(R_s \circ T \circ L_s) a = (1 + a \times s)^s$$

或者，在 Haskell 中：

```
s -> Maybe (a, s)
```

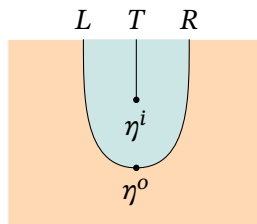
这与我们的 **MaybeState** 单子的定义一致。

一般来说，内部单子 T 由其单位 η^i 和乘法 μ^i 定义（上标 i 表示“内部”）。外部伴随由其单位 η^o 和余单位 ϵ^o 定义。

组合单子的单位是自然变换：

$$\eta : Id \rightarrow R \circ T \circ L$$

由字符串图给出：



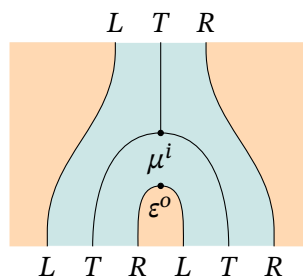
它是内部单位 $R \circ \eta^i \circ L$ 和外部单位 η^o 的垂直组合。在分量中：

$$\eta_a = R(\eta_{La}^i) \circ \eta_a^o$$

组合单子的乘法是一个自然变换：

$$\mu : R \circ T \circ L \circ R \circ T \circ L \rightarrow R \circ T \circ L$$

由字符串图给出：



它是外部余单位的多次 whiskered 垂直组合：

$$R \circ T \circ \epsilon^o \circ T \circ L$$

后跟内部乘法的 whiskered $R \circ \mu^i \circ L$ 。在分量中：

$$\mu_c = R(\mu_{Lc}^i) \circ (R \circ T)(\epsilon_{(T \circ L)c}^o)$$

状态单子变换器

让我们为状态单子变换器的情况展开这些方程。状态单子由柯里化伴随生成。左函子 L_s 是积函子 (a, s) ，右函子 R_s 是指数函子，也称为读者函子 $(s \rightarrow a)$ 。

正如我们之前所见，外部余单位 ϵ_a^o 是函数应用：

```
counit :: (s -> a, s) -> a
counit (f, x) = f x
```

单位 η_a^o 是柯里化的对构造函数：

```
unit :: a -> s -> (a, s)
unit x = \s -> (x, s)
```

我们将保持内部单子 (T, η^i, μ^i) 为任意。在 Haskell 中，我们将这个三元组称为 `m`、`return` 和 `join`。

通过将状态单子变换器应用于单子 T 得到的组合单子是组合 $R \circ T \circ L$ ，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
```

单子变换器的单位是 η^o 和 $R \circ \eta^i \circ L$ 的垂直组合。在分量中：

$$\eta_a = R(\eta_{La}^i) \circ \eta_a^o$$

这个公式中有很多移动的部分，所以让我们逐步分析它。我们从右边开始：我们有伴随单位的 a 分量，它是一个从 a 到 $R(La)$ 的箭头。在 Haskell 中，它是函数 `unit`。

```
unit :: a -> s -> (a, s)
```

让我们在某个 $x :: a$ 处评估这个函数。结果是另一个函数 $s \rightarrow (a, s)$ 。我们将这个函数作为参数传递给 $R(\eta_{La}^i)$ 。

η_{La}^i 是内部单子的 `return` 在 La 处的分量。这里, La 是类型 (a, s) 。因此, 我们将多态函数 `return :: a -> m a` 实例化为函数 $(a, s) \rightarrow m (a, s)$ 。(类型推断器会自动为我们完成此操作。)

接下来, 我们使用 R 提升 `return` 的这个分量。这里, R 是指数 $(-)^s$, 因此它通过后组合提升函数。它将 `return` 后组合到传递给它的任何函数。在我们的例子中, 这是由 `unit` 生成的函数。注意类型匹配: 我们将 $(a, s) \rightarrow m (a, s)$ 后组合到 $s \rightarrow (a, s)$ 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。这是组合单子的 `return`, 用内部单子的 `return` 表示。

同样的推理可以应用于组合 μ 在某个 a 处的分量的公式:

$$\mu_a = R(\mu_{La}^i) \circ (R \circ T)(\epsilon_{(T \circ L)a}^o)$$

内部 μ^i 是单子 `m` 的 `join`。应用 R 将其转换为后组合。

外部 ϵ^o 是在 $T(La)$ 或 `m (a, s)` 处取函数应用。它是一个类型为:

```
(s -> m (a, s), s) -> m (a, s)
```

的函数, 插入适当的数据构造函数后, 可以写为 `uncurry runStateT`:

```
uncurry runStateT :: (StateT s m a, s) -> m (a, s)
```

$(R \circ T)$ 的应用使用函子 R 和 T 的组合提升 ϵ 的这个分量。前者实现为后组合, 后者是单子 `m` 的 `fmap`。

将所有这些放在一起, 我们得到了状态单子变换器的 `join` 的无点公式:

```
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` 示例现在可以使用单子变换器重写：

```
type MaybeState s a = StateT s Maybe a
```

通过将 `StateT` 单子变换器应用于恒等函子，可以恢复原始的 `State` 单子，该函子在库中定义了 `Monad` 实例（注意在此定义中跳过了最后一个类型变量 `a`）：

```
type State s = StateT s Identity
```

其他单子变换器遵循相同的模式。它们在单子变换器库 `MTL` 中定义。

16.5 单子代数

每个伴随都会生成一个单子，到目前为止，我们已经能够为我们感兴趣的所有单子定义伴随。但是，是否每个单子都是由伴随生成的呢？答案是肯定的，并且通常每个单子都有许多伴随——实际上是一个完整的伴随范畴。

为单子找到伴随类似于因式分解。我们希望将一个函子表示为另外两个函子的复合，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范畴。我们将通过研究单子的代数来找到这样的范畴。

单子由一个自函子定义，我们知道可以为自函子定义代数。数学家通常将单子视为生成表达式的工具，而代数则是评估这些表达式的工具。然而，由单子生成的表达式对这些代数施加了一些兼容性条件。

例如，你可能会注意到单子单位 $\eta_a : a \rightarrow Ta$ 的类型签名看起来像代数结构映射 $\alpha : Ta \rightarrow a$ 的逆。当然， η 是一个为每个类型定义的自然变换，而代数有一个固定的载体类型。尽管如此，我们可能会合理地期望其中一个可以抵消另一个的作用。

考虑之前的表达式单子 `Ex` 的例子。这个单子的代数是一个载体类型的选择，比如 `Char` 和一个箭头：

```
alg :: Ex Char -> Char
```

由于 `Ex` 是一个单子，它定义了一个单位，即 `return`，这是一个多态函数，可以用来从值生成简单的表达式。`Ex` 的单位是：

```
return x = Var x
```

我们可以为任意类型实例化单位，特别是为我们的代数的载体类型。要求评估 `Var c`（其中 `c` 是一个字符）应该返回相同的 `c` 是合理的。换句话说，我们希望：

```
alg . return = id
```

这个条件将立即排除许多代数，例如：

```
alg (Var c) = 'a' -- 与单子 Ex 不兼容
```

我们希望施加的第二个条件是，与单子兼容的代数尊重替换。单子允许我们使用 `join` 来展平嵌套表达式。代数允许我们评估这些表达式。

有两种方法可以做到这一点：我们可以将代数应用于展平的表达式，或者我们可以先将其应用于内部表达式（使用 `fmap`），然后评估结果表达式。

```
alg (join mma) = alg (fmap alg mma)
```

其中 `mma` 是嵌套类型 `Ex (Ex Char)`。

在范畴论中，这两个条件定义了一个单子代数。

我们说 $(a, \alpha: Ta \rightarrow a)$ 是单子 (T, μ, η) 的 **单子代数**，如果以下图表交换：

$$\begin{array}{ccc} a & \xrightarrow{\eta_a} & Ta \\ & \searrow id_a & \downarrow \alpha \\ & & a \end{array} \quad \begin{array}{ccc} T(Ta) & \xrightarrow{T\alpha} & Ta \\ \downarrow \mu_a & & \downarrow \alpha \\ Ta & \xrightarrow{\alpha} & a \end{array}$$

这些定律有时被称为单子代数的单位律和乘法律。

由于单子代数只是特殊类型的代数，它们形成了代数的子范畴。回想一下，代数态射是满足以下条件的箭头：

$$\begin{array}{ccc} Ta & \xrightarrow{Tf} & Tb \\ \downarrow \alpha & & \downarrow \beta \\ a & \xrightarrow{f} & b \end{array}$$

根据这个定义，我们可以重新解释第二个单子代数图表，断言单子代数的结构映射 α （底部的箭头）也是从 (Ta, μ_a) 到 (a, α) 的代数态射。这将在接下来的内容中派上用场。

Eilenberg-Moore 范畴

给定单子 T 在范畴 \mathcal{C} 上的单子代数范畴称为 Eilenberg-Moore 范畴，记作 \mathcal{C}^T 。事实证明，它是一个很好的中间范畴选择，允许我们将单子 T 分解为一对伴随函子的复合。

过程如下：我们定义一对函子，证明它们形成一个伴随，然后证明由这个伴随生成的单子是原始单子。

首先，有一个明显的遗忘函子，我们称之为 U^T ，从 \mathcal{C}^T 到 \mathcal{C} 。它将代数 (a, α) 映射到其载体 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 F^T ，它是 U^T 的左伴随。

$$\begin{array}{ccc} & F^T & \\ \mathcal{C}^T & \xleftarrow{\quad} & \mathcal{C} \\ & U^T & \end{array}$$

在对象上， F^T 将 \mathcal{C} 中的对象 a 映射到一个单子代数，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 $T(Ta) \rightarrow Ta$ ，我们选择单子乘法的分量 $\mu_a: T(Ta) \rightarrow Ta$ 。

很容易验证这个代数 (Ta, μ_a) 确实是一个单子代数——必要的交换条件来自单子定律。实际上，将代数 (Ta, μ_a) 代入单子代数图表，我们得到（代数部分用红色绘制）：

$$\begin{array}{ccc} Ta & \xrightarrow{\eta_{Ta}} & T(Ta) \\ & \searrow id_{Ta} & \downarrow \mu_a \\ & & Ta \end{array} \quad \begin{array}{ccc} T(T(Ta)) & \xrightarrow{T\mu_a} & T(Ta) \\ & \downarrow \mu_{Ta} & \downarrow \mu_a \\ T(Ta) & \xrightarrow{\mu_a} & Ta \end{array}$$

第一个图表只是左单子单位律的分量形式。 η_{Ta} 箭头对应于 $\eta \circ T$ 的 whiskering。第二个图表是 μ 的结合律，其中两个 whiskering $\mu \circ T$ 和 $T \circ \mu$ 以分量形式表示。

为了证明我们有一个伴随，我们将定义两个自然变换作为伴随的单位和余单位。

对于伴随的单位，我们选择单子 T 的单子单位 η 。它们具有相同的签名——在分量中， $\eta_a: a \rightarrow U^T(F^T a)$ 。

余单位是一个自然变换：

$$\varepsilon: F^T \circ U^T \rightarrow Id$$

ε 在 (a, α) 处的分量是一个代数态射，从由 a 生成的自由代数 (Ta, μ_a) 回到 (a, α) 。正如我们之前所见， α 本身就是这样一种态射。因此，我们可以选择 $\varepsilon_{(a, \alpha)} = \alpha$ 。

对于这些 η 和 ε 的定义，三角恒等式来自单子和单子代数的单位律。

与所有伴随一样，复合 $U^T \circ F^T$ 是一个单子。我们将证明这与我们开始的单子相同。实际上，在对象上，复合 $U^T(F^T a)$ 首先将 a 映射到一个自由单子代数 (Ta, μ) ，然后忘记结构映射。最终结果是将 a 映射到 Ta ，这正是原始单子所做的。

在箭头上，它使用 T 提升一个箭头 $f: a \rightarrow b$ 。 Tf 是从 (Ta, μ_a) 到 (Tb, μ_b) 的代数态射这一事实来自 μ 的自然性：

$$\begin{array}{ccc} T(Ta) & \xrightarrow{T(Tf)} & T(Tb) \\ \downarrow \mu_a & & \downarrow \mu_b \\ Ta & \xrightarrow{Tf} & Tb \end{array}$$

最后，我们必须证明单子 $U^T \circ F^T$ 的单位和余单位与原始单子的单位和余单位相同。

单位在构造上是相同的。

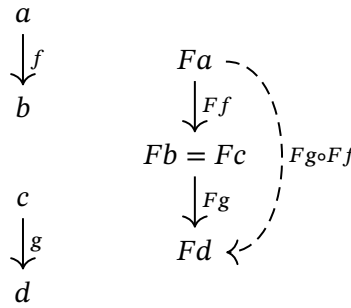
$U^T \circ F^T$ 的单子乘法由伴随单位的 whiskering $U^T \circ \varepsilon \circ F^T$ 给出。在分量中，这意味着在 (Ta, μ_a) 处实例化 ε ，这给了我们 μ_a （ U^T 在箭头上的作用是平凡的）。这确实是原始的单子乘法。

因此，我们已经证明，对于任何单子 T ，我们可以定义 Eilenberg-Moore 范畴和一对伴随函子来分解这个单子。

Kleisli 范畴

在每个 Eilenberg-Moore 范畴内部，都有一个较小的 Kleisli 范畴试图挣脱出来。这个较小的范畴是我们在上一节中构造的自由函子的像。

尽管看起来如此，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，将复合映射到复合。问题可能出现在源范畴中不可复合的两个箭头在目标范畴中变得可复合的情况下。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，则可能会发生这种情况。在下面的例子中， Ff 和 Fg 是可复合的，但它们的复合 $Fg \circ Ff$ 可能不在第一个范畴的像中。

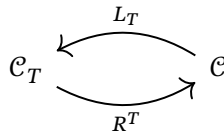


然而，自由函子 F^T 将不同的对象映射到不同的自由代数，因此它的像确实是 \mathcal{C}^T 的一个子范畴。

我们之前遇到过 Kleisli 范畴。有许多方法可以构造相同的范畴，最简单的方法是用 Kleisli 箭头来描述 Kleisli 范畴。

单子 (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 的箭头由 \mathcal{C} 中从 a 到 Tb 的箭头表示。你可能会认出它是我们之前定义的 Kleisli 箭头 $a \multimap b$ 。由于 T 是一个单子，这些 Kleisli 箭头可以使用“鱼”操作符 \leq 进行复合。

为了建立伴随：



我们定义左函子 $L_T: \mathcal{C} \rightarrow \mathcal{C}_T$ 在对象上为恒等。我们仍然需要定义它对箭头的作用。它应该将常规箭头 $f: a \rightarrow b$ 映射到从 a 到 b 的 Kleisli 箭头。这个 Kleisli 箭头 $a \multimap b$ 由 \mathcal{C} 中的箭头 $a \rightarrow Tb$ 表示。这样的箭头总是存在，作为复合 $\eta_b \circ f$ ：

$$L_T f: a \xrightarrow{f} b \xrightarrow{\eta_b} Tb$$

右函子 $R_T: \mathcal{C}_T \rightarrow \mathcal{C}$ 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 $a \multimap b$ ，它由箭头 $g: a \rightarrow Tb$ 表示， R_T 会将其映射到箭头 $R_T a \rightarrow R_T b$ ，即 \mathcal{C} 中的箭头 $Ta \rightarrow Tb$ 。我们

将这个箭头取为复合 $\mu_b \circ Tg$:

$$Ta \xrightarrow{Tg} T(Tb) \xrightarrow{\mu_b} Tb$$

为了建立伴随，我们将证明 \mathbf{hom} -集的同构:

$$\mathcal{C}_T(L_T a, b) \cong \mathcal{C}(a, R_T b)$$

左边的元素是一个 Kleisli 箭头 $a \rightarrow b$ ，它由 $f: a \rightarrow Tb$ 表示。我们可以在右边找到相同的箭头，因为 $R_T b$ 是 Tb 。因此，同构是在 \mathcal{C}^T 中的 Kleisli 箭头和表示它们的 \mathcal{C} 中的箭头之间。

复合 $R_T \circ L_T$ 等于 T ，并且确实可以证明这个伴随生成了原始单子。

一般来说，可能有多个伴随生成相同的单子。伴随本身形成一个 2-范畴，因此可以使用伴随态射（2-范畴中的 1-细胞）来比较伴随。事实证明，Kleisli 伴随是所有生成给定单子的伴随中的初始对象。对偶地，Eilenberg-Moore 伴随是终对象。

余单子 (Comonads)

如果这个词容易发音，我们或许应该将副作用称为“ntext”，因为副作用的对偶是“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。

让我们从熟悉的例子开始，将环境作为上下文。我们之前通过柯里化箭头构造了一个读取器单子 (reader monad)：

```
(a, e) -> b
```

然而，这次我们将它视为一个 co-Kleisli 箭头，它是一个从“上下文”参数出发的箭头。

与单子的情况类似，我们希望能够组合这样的箭头。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)
```

实现一个关于这种组合的恒等箭头也很直接：

```
idWithEnv :: (a, e) -> a
idWithEnv (a, e) = a
```

这强烈暗示了一个想法，即存在一个范畴，其中 co-Kleisli 箭头作为态射。

Exercise 17.0.1. 证明使用 `composeWithEnv` 组合 co-Kleisli 箭头是结合的。

17.1 编程中的余单子

一个函子 `w`（可以将其视为一个风格化的倒置 `m`）如果支持 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
```

在这里，组合以中缀运算符的形式书写。组合的单位称为`extract`，因为它从上下文中提取一个值。

让我们用我们的例子来尝试一下。将环境作为对的第一部分传递是很方便的。余单子由对构造器`((,) e)`的部分应用给出的函子给出。

```
instance Comonad ((,) e) where
  g =<= f = \ea -> g (fst ea, f ea)
  extract = snd
```

与单子一样，co-Kleisli 组合可以用于无点编程风格。但我们也可以使用`join`的对偶`duplicate`：

```
duplicate :: w a -> w (w a)
```

或者称为`extend`的 `bind` 的对偶：

```
extend :: (w a -> b) -> w a -> w b
```

以下是我们如何用`duplicate`和`fmap`实现 co-Kleisli 组合：

```
g =<= f = g . fmap f . duplicate
```

Exercise 17.1.1. 用 `extend` 实现 `duplicate`，反之亦然。

Stream 余单子

余单子的有趣例子涉及更大的，有时是无限的上下文。这里有一个无限流：

```
data Stream a = Cons a (Stream a)
  deriving Functor
```

如果我们认为这样的流是类型`a`在无限尾部的上下文中的值，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)
```

在这里，`extract`返回流的头部，`duplicate`将流转换为流的流，其中每个连续的流是前一个流的尾部。

直觉是`duplicate`为迭代设置了舞台，但它以一种非常通用的方式完成。每个子流的头部可以解释为原始流中未来的”当前位置”。

很容易执行一个遍历这些流的头部元素的计算。但这并不是余单子的力量所在。它让我们执行需要任意前瞻的计算。这样的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`所做的：它将给定的 co-Kleisli 箭头`f`应用于`duplicate`生成的所有流：

```
extend f (Cons a as) = Cons (f (Cons a as)) (extend f as)
```

这是一个 co-Kleisli 箭头的例子，它计算流的前五个元素的平均值：

```
avg :: Stream Double -> Double
avg = (/5). sum . stmTake 5
```

它使用一个辅助函数提取前`n`项：

```
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`以平滑局部波动。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。它生成原始流的运行平均值。

```
smooth :: Stream Double -> Stream Double
smooth = extend avg
```

余单子对于在空间或时间上扩展的数据结构中结构化计算非常有用。这些计算足够局部以定义”当前位置”，但需要从相邻位置收集信息。信号处理或图像处理是很好的例子。模拟也是如此，其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，仅举几例。康威的生命游戏也是测试余单子方法的好地方。

有时，在连续数据流上进行计算，直到最后一步才进行采样是很方便的。这是一个时间函数（由`Double`表示）的信号示例

```
data Signal a = Sig (Double -> a) Double
```

第一个组件是作为时间函数实现的`a`的连续流。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
```

在这里，`extend`将滤波器

```
g :: Signal a -> a
```

卷积到整个流上。

Exercise 17.1.2. 为双向流实现`Comonad`实例：

```
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（按相反顺序）；第二个列表的头部视为现在，其尾部视为未来。

Exercise 17.1.3. 为上一个练习中的`BiStream`实现一个低通滤波器。它对三个值进行平均：当前值、紧邻过去的值和紧邻未来的值。对于电气工程师：实现一个高斯滤波器。

17.2 余单子的范畴定义

我们可以通过反转单子定义中的箭头来得到余单子的定义。我们的`duplicate`对应于反转的`join`，而`extract`则是反转的`return`。

因此，余单子是一个自函子 W ，配备有两个自然变换：

$$\delta : W \rightarrow W \circ W$$

$$\varepsilon : W \rightarrow \text{Id}$$

这些变换（分别对应于`duplicate`和`extract`）必须满足与单子相同的恒等式，只是箭头方向相反。

以下是余单位律：

$$\begin{array}{ccccc}
 \text{Id} \circ W & \xleftarrow{\varepsilon \circ W} & W \circ W & \xrightarrow{W \circ \varepsilon} & W \circ \text{Id} \\
 & \searrow = & \uparrow \delta & \swarrow = & \\
 & & W & &
 \end{array}$$

以及结合律:

$$\begin{array}{ccc}
 (W \circ W) \circ W & \xrightarrow{=} & W \circ (W \circ W) \\
 \delta \circ W \uparrow & & \uparrow W \circ \delta \\
 W \circ W & \xleftarrow{\delta} & W \circ W \\
 & W & \\
 & \xrightarrow{\delta} &
 \end{array}$$

余幺半群

我们已经看到单子定律如何从幺半群定律中得出。我们可以预期余单子定律应该从幺半群的对偶版本中得出。

事实上，余幺半群是幺半群范畴 $(\mathcal{C}, \otimes, I)$ 中的一个对象 w ，配备有两个称为余乘法和余单位的态射：

$$\delta : w \rightarrow w \otimes w$$

$$\varepsilon : w \rightarrow I$$

我们可以将张量积替换为自函子的复合，将单位对象替换为恒等函子，从而将余单子定义为自函子范畴中的余幺半群。

在 Haskell 中，我们可以为笛卡尔积定义一个 **Comonoid** 类型类：

```
class Comonoid w where
    split    :: w -> (w, w)
    destroy  :: w -> ()
```

余幺半群比其兄弟幺半群讨论得少，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映射 $\Delta_a : a \rightarrow a \times a$ 作为余乘法，以及到终端对象的唯一箭头作为余单位，来变成一个余幺半群。

在编程中，这是我们不假思索就会做的事情。余乘法意味着能够复制一个值，而余单位意味着能够丢弃一个值。

在 Haskell 中，我们可以轻松地任何类型实现 **Comonoid** 实例：

```
instance Comonoid w where
    split w    = (w, w)
    destroy w = ()
```

事实上，我们不会多想就使用一个函数的参数两次，或者根本不使用它。但是，如果我们想要明确表达，像这样的函数：

```
f x = x + x
g y = 42
```

可以写成:

```
f x = let (x1, x2) = split x
      in x1 + x2
g y = let () = destroy y
      in 42
```

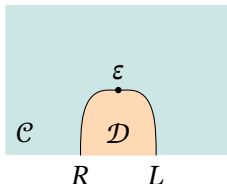
然而,在某些情况下,复制或丢弃一个变量是不可取的。这种情况通常发生在参数是外部资源时,比如文件句柄、网络端口或堆上分配的一块内存。这些资源在被分配和释放之间应该有明确的生命周期。跟踪可以轻松复制或丢弃的对象的生命周期非常困难,并且是编程错误的常见来源。

基于笛卡尔范畴的编程模型总是会有这个问题。解决方案是使用不支持对象复制或销毁的幺半群(闭)范畴。这样的范畴是线性类型的自然设置。线性类型的元素在 Rust 中使用,并且在撰写本文时,正在 Haskell 中进行尝试。在 C++ 中,有一些构造模仿线性性,比如 `unique_ptr` 和移动语义。

17.3 伴随函子导出的余单子

我们已经看到,两个函子 $L: \mathcal{D} \rightarrow \mathcal{C}$ 和 $R: \mathcal{C} \rightarrow \mathcal{D}$ 之间的伴随关系 $L \dashv R$ 会导出一个单子 $R \circ L: \mathcal{D} \rightarrow \mathcal{D}$ 。另一种组合 $L \circ R$ 是 \mathcal{C} 上的自函子,它实际上是一个余单子。

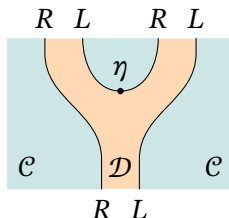
伴随关系的余单位 (counit) 充当了余单子的余单位。这可以通过以下字符串图来说明:



余乘法 (comultiplication) 由 η 的 whiskering 给出:

$$\delta = L \circ \eta \circ R$$

如下字符串图所示:



与之前一样，余单子法则可以从三角形恒等式中推导出来。

余状态余单子

我们已经看到，状态单子可以从积和指数之间的柯里化伴随关系中生成。左函子被定义为与某个固定对象 s 的积：

$$L_s a = a \times s$$

右函子是指数化，由同一个对象 s 参数化：

$$R_s c = c^s$$

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 $s \rightarrow c$ 分配给 c ，左函子将 c 与 s 配对。组合的结果是自函子：

```
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
```

函子实例将函数后组合到 **Store** 的第一个组件：

```
instance Functor (Store s) where
  fmap g (St f s) = St (g . f) s
```

这个伴随关系的余单位，即余单子的 **extract**，是函数应用：

```
extract :: Store s c -> c
extract (St f s) = f s
```

这个伴随关系的单位是一个自然变换 $\eta: \text{Id} \rightarrow R_s \circ L_s$ 。我们曾将其用作状态单子的 **return**。这是它在 c 处的组件：

```
unit :: c -> (s -> (c, s))
unit c = \s -> (c, s)
```

为了得到 **duplicate**，我们需要在两个函子之间对 η 进行 whiskering：

$$\delta = L_s \circ \eta \circ R_s$$

在右侧进行 whiskering 意味着取 η 在对象 $R_s c$ 处的组件，在左侧进行 whiskering 意味着使用 L_s 提升这个组件。由于 Haskell 中的 whiskering 翻译是一个复杂的过程，让我们逐步分析它。

为了简单起见，让我们将类型 s 固定为 `Int`。我们将左函子封装到一个 `newtype` 中：

```
newtype Pair c = P (c, Int)
deriving Functor
```

并将右函子保持为类型同义词：

```
type Fun c = Int -> c
```

伴随关系的单位可以使用显式的 `forall` 写成自然变换：

```
eta :: forall c. c -> Fun (Pair c)
eta c = \s -> P (c, s)
```

我们现在可以将余乘法实现为 `eta` 的 whiskering。右侧的 whiskering 通过使用 `eta` 在 `Fun c` 处的组件编码在类型签名中。左侧的 whiskering 通过使用为 `Pair` 函子定义的 `fmap` 提升 `eta` 来完成。我们使用语言扩展 `TypeApplications` 来明确使用哪个 `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 是 `Int` 的情况。

我们将 `Store Int c` 的第一个组件，即函数 `f :: Int -> c`，解释为对假想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。

在这种解释下，`duplicate` 生成一个无限流的流，每个流以不同的偏移量移动，而 `extend` 在这个流上执行卷积。当然，惰性拯救了这一天：只有我们明确要求的值才会被计算。

还要注意，我们之前的 `Signal` 余单子示例可以通过 `Store Double` 重现。

Exercise 17.3.1. 可以使用存储余单子实现元胞自动机。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或死的：

```
data Cell = L | D
  deriving Show
```

运行几代这个自动机。提示：使用 *Prelude* 中的函数 `iterate`。

余单子余代数

与单子代数对偶的是余单子余代数。给定一个余单子 (W, ϵ, δ) ，我们可以构造一个余代数，它由一个载体对象 a 和一个箭头 $\phi: a \rightarrow Wa$ 组成。为了使这个余代数与余单子良好地组合，我们要求能够提取使用 ϕ 注入的值；并且 ϕ 的提升在作用于 ϕ 的结果时，等同于复制：

$$\begin{array}{ccc}
 a & \xleftarrow{\epsilon_a} & Wa \\
 & \nwarrow id_a & \uparrow \phi \\
 & & a
 \end{array}
 \qquad
 \begin{array}{ccccc}
 & & W(Wa) & \xleftarrow{W\phi} & Wa \\
 & \uparrow \delta_a & & & \uparrow \phi \\
 Wa & \xleftarrow{\phi} & a & &
 \end{array}$$

与单子代数一样，余单子余代数形成一个范畴。给定 \mathcal{C} 中的一个余单子 (W, ϵ, δ) ，其余单子余代数形成一个称为 Eilenberg-Moore 范畴（有时前缀为 *co-*） \mathcal{C}^W 的范畴。

\mathcal{C}^W 有一个 *co-Kleisli* 子范畴，记为 \mathcal{C}_W 。

给定一个余单子 W ，我们可以使用 \mathcal{C}^W 或 \mathcal{C}_W 构造一个伴随关系，重现余单子 W 。构造与单子的构造完全类似。

透镜

Store 余单子的余代数特别有趣。我们首先进行一些重命名。让我们将载体称为 **s**，状态称为 **a**。

```
data Store a s = St (a -> s) a
```

余代数由一个函数给出：

```
phi :: s -> Store a s
```

这等价于一对函数：

```
set :: s -> a -> s
get :: s -> a
```

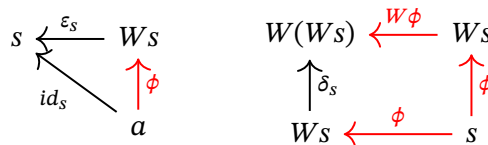
这样的一对称为透镜：**s** 称为源，**a** 是焦点。

在这种解释下，**get** 让我们提取焦点，**set** 用新值替换焦点以生成一个新的 **s**。

透镜最初被引入来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中找到了应用。透镜将读写访问较大对象的一部分的想法对象化。例如，透镜可以聚焦于一对中的一个组件或记录的特定组件。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法则应用于透镜。为了简单起见，让我们从方程中省略数据构造函数。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)
```



第一个法则告诉我们，将 **set** 的结果应用于 **get** 的结果会得到恒等：

```
set s (get s) = s
```

这称为透镜的 **set/get** 法则。当你用相同的焦点替换焦点时，不应有任何变化。

第二个法则要求将 **fmap phi** 应用于 **phi** 的结果：

```
fmap phi (set s, get s) = (phi . set s, get s)
```

这应该等于 **delta** 的应用：

```
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，因此我们将它们应用于某个 `a'` 并得到 `set/set` 透镜法则:

```
set (set s a) a' = set s a'
```

将焦点设置为 `a`，然后用 `a'` 覆盖它，与直接将焦点设置为 `a'` 相同。

第二个组件给出了 `get/set` 法则:

```
get (set s a) = a
```

在我们将焦点设置为 `a` 后，`get` 的结果是 `a`。

满足这些法则的透镜称为合法透镜。它们是存储余单子的余单子余代数。

端与余端

18.1 Profunctors (函子)

在范畴论的高深领域中，我们遇到了一些模式，这些模式与它们的起源相距甚远，以至于我们难以形象化它们。更糟糕的是，模式越抽象，其具体示例之间的差异就越大。

从 a 到 b 的箭头相对容易形象化。我们有一个非常熟悉的模型：一个消耗 a 的元素并生成 b 的元素的函数。**Hom-set**（同态集）是这些箭头的集合。

函子是范畴之间的箭头。它消耗一个范畴中的对象和箭头，并生成另一个范畴中的对象和箭头。我们可以将其视为从源范畴提供的材料构建这些对象（和箭头）的配方。特别是，我们通常将自函子视为建筑材料的容器。

Profunctor（函子）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：

$$\langle f: s \rightarrow a, g: b \rightarrow t \rangle$$

映射到一个函数：

$$P\langle f, g \rangle: P\langle a, b \rangle \rightarrow P\langle s, t \rangle$$

Profunctor 是一种抽象，它结合了许多其他抽象的元素。由于它是一个函子 $\mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathbf{Set}$ ，我们可以将其视为从一对对象构建一个集合，并从一对箭头（其中一个箭头方向相反）构建一个函数。然而，这并没有帮助我们的想象力。

幸运的是，我们有一个很好的 **Profunctor** 模型：**hom-functor**（同态函子）。当对象变化时，两个对象之间的箭头集合表现得像一个 **Profunctor**。同样，变化 **hom-set** 的源和目标之间的差异也是有意义的。

因此，我们可以将任意 **Profunctor** 视为 **hom-functor** 的推广。**Profunctor** 在已有的 **hom-sets** 之上提供了对象之间的额外桥梁。

然而，hom-set $\mathcal{C}(a, b)$ 的元素和集合 $P\langle a, b \rangle$ 的元素之间有一个很大的区别。hom-set 的元素是箭头，而箭头可以组合。如何组合 Profunctors 并不立即显而易见。

诚然，Profunctor 对箭头的提升可以视为组合的推广——只是不是在 Profunctors 之间，而是在 hom-sets 和 Profunctors 之间。例如，我们可以用箭头 $f: s \rightarrow a$ “预组合” $P\langle a, b \rangle$ ，得到 $P\langle s, b \rangle$ ：

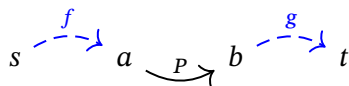
$$P\langle f, id_b \rangle: P\langle a, b \rangle \rightarrow P\langle s, b \rangle$$

同样，我们可以用 $g: b \rightarrow t$ “后组合”它：

$$P\langle id_a, g \rangle: P\langle a, b \rangle \rightarrow P\langle a, t \rangle$$

这种异质组合接受一个由箭头和 Profunctor 元素组成的可组合对，并生成一个 Profunctor 元素。

Profunctor 可以通过提升一对箭头在两侧进行扩展：



Collages (拼贴)

没有理由将 Profunctor 限制在单一范畴中。我们可以轻松地将两个范畴之间的 Profunctor 定义为函子 $P: \mathcal{C}^{op} \times \mathcal{D} \rightarrow \mathbf{Set}$ 。这样的 Profunctor 可以通过生成从 \mathcal{C} 中的对象到 \mathcal{D} 中的对象的缺失的 hom-sets 来将两个范畴粘合在一起。

两个范畴 \mathcal{C} 和 \mathcal{D} 的拼贴（或余图）是一个范畴，其对象来自两个范畴（一个不相交的并集）。两个对象 x 和 y 之间的 hom-set 要么是 \mathcal{C} 中的 hom-set，如果两个对象都在 \mathcal{C} 中；要么是 \mathcal{D} 中的 hom-set，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} 中且 y 在 \mathcal{D} 中。否则，hom-set 为空。

态射的组合是通常的组合，除非其中一个态射是 $P\langle x, y \rangle$ 的元素。在这种情况下，我们提升我们试图预组合或后组合的态射。

很容易看出，拼贴确实是一个范畴。拼贴两侧之间的新态射有时被称为异态射。它们只能从 \mathcal{C} 到 \mathcal{D} ，而不能反过来。

这样看来，Profunctor $\mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathbf{Set}$ 应该真正被称为自-Profunctor。它定义了 \mathcal{C} 与自身的拼贴。

Exercise 18.1.1. 证明存在一个从两个范畴的拼贴到“行走箭头”范畴的函子，该范畴有两个对象和一个箭头（以及两个恒等箭头）。

Exercise 18.1.2. 证明如果存在一个从 \mathcal{C} 到行走箭头范畴的函子，那么 \mathcal{C} 可以被拆分为两个范畴的拼贴。

Profunctors 作为关系

在显微镜下，Profunctor 看起来像 hom-functor，集合 $P\langle a, b \rangle$ 的元素看起来像单个箭头。但当我们放大时，我们可以将 Profunctor 视为对象之间的关系。这些不是通常的关系；它们是证明相关的关系。

为了更好地理解这个概念，让我们考虑一个常规函子 $F: \mathcal{C} \rightarrow \mathbf{Set}$ （换句话说，一个协预层）。一种解释它的方式是，它定义了 \mathcal{C} 对象的一个子集，即那些被映射到非空集合的对象。 Fa 的每个元素都被视为 a 是这个子集成员的证明。另一方面，如果 Fa 是一个空集，那么 a 不是这个子集的成员。

我们可以将相同的解释应用于 Profunctors。如果集合 $P\langle a, b \rangle$ 为空，我们说 b 与 a 无关。如果它不为空，我们说集合 $P\langle a, b \rangle$ 的每个元素都代表 b 与 a 相关的证明。然后，我们可以将 Profunctor 视为一个证明相关的关系。

请注意，我们不对这个关系做任何假设。它不必是自反的，因为 $P\langle a, a \rangle$ 可能为空（事实上， $P\langle a, a \rangle$ 仅对自 Profunctors 有意义）。它也不必是对称的。

由于 hom-functor 是（自）Profunctor 的一个例子，这种解释让我们以新的视角看待 hom-functor：作为范畴中对象之间内置的证明相关关系。如果两个对象之间存在箭头，则它们是相关的。请注意，这种关系是自反的，因为 $\mathcal{C}\langle a, a \rangle$ 永远不会为空：至少它包含恒等态射。

此外，正如我们之前所见，hom-functors 与 Profunctors 相互作用。如果 a 通过 P 与 b 相关，并且 hom-sets $\mathcal{C}\langle s, a \rangle$ 和 $\mathcal{D}\langle b, t \rangle$ 非空，那么自动地 s 通过 P 与 t 相关。因此，Profunctors 是与它们操作的范畴结构兼容的证明相关关系。

我们知道如何将 Profunctor 与 hom-functors 组合，但如何组合两个 Profunctors 呢？我们可以从关系的组合中得到线索。

假设你想给手机充电，但没有充电器。为了将你连接到充电器，只要有一个拥有充电器的朋友就足够了。任何朋友都可以。你将拥有朋友的关系与拥有充电器的人的关系组合起来，得到能够给手机充电的关系。你能给手机充电的证明是一对证明，一个是友谊的证明，另一个是拥有充电器的证明。

一般来说，我们说两个对象通过复合关系相关，如果存在一个中间对象与它们两者都相关。

Haskell 中的 Profunctor 组合

关系的组合可以转化为 Haskell 中的 Profunctor 组合。让我们首先回顾 Profunctor 的定义：

```
class Profunctor p where
  dimap :: (s -> a) -> (b -> t) -> (p a b -> p s t)
```

理解 Profunctor 组合的关键在于它需要中间对象的存在。对于对象 b 通过复合 $P \diamond Q$ 与对象 a 相关，必

须存在一个对象 x 来弥合差距：

$$a \xrightarrow{Q} x \xrightarrow{P} b$$

这可以在 Haskell 中使用存在类型进行编码。给定两个 Profunctors p 和 q ，它们的组合是一个新的 Profunctor `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。这对证明构成了 b 与 a 相关的证明。

存在类型可以看作和类型（sum type）的推广。我们对所有可能的类型 x 求和。就像有限和可以通过注入其中一个替代项来构造（想想 `Either` 的两个构造函数），存在类型可以通过为 x 选择特定类型并将其注入 `Procompose` 的定义来构造。

正如从和类型映射出来需要一对函数，每个替代项一个；从存在类型映射出来需要一个函数族，每个类型一个。例如，从 `Procompose` 映射出来由一个多态函数给出：

```
mapOut :: Procompose p q a b -> (forall x. q a x -> p x b -> c) -> c
mapOut (Procompose qax pxb) f = (f qax pxb)
```

Profunctors 的组合再次是一个 Profunctor，可以从以下实例中看出：

```
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 Profunctor 向左扩展，第二个 Profunctor 向右扩展来扩展复合 Profunctor。

这个 Profunctor 组合的定义在 Haskell 中有效是由于参数性。语言以某种方式限制了 Profunctors 的类型，使得这个定义有效。然而，一般来说，对中间对象进行简单的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。

18.2 余端 (Coends)

在朴素定义 profunctors（profunctor，也称为分布函子）的复合时，过度计数的问题发生在两个中间对象候选通过一个态射连接时：

$$a \xrightarrow{Q} x \xrightarrow{f} y \xrightarrow{P} b$$

我们可以通过在右侧扩展 Q ，通过提升 $Q(id, f)$ ，并使用 y 作为中间对象；或者我们可以在左侧扩展 P ，通过提升 $P(f, id)$ ，并使用 x 作为中介。

为了避免双重计数，我们必须在应用于 **profunctors** 时调整我们对和类型的定义。由此产生的构造称为余端 (coend)。

首先，让我们重新表述这个问题。我们试图在乘积中对所有对象 x 求和：

$$P(a, x) \times Q(x, b)$$

双重计数的发生是因为我们可以在两个 **profunctors** 之间打开一个间隙，只要有一个态射可以插入其中。因此，我们实际上是在看一个更一般的乘积：

$$P(a, x) \times Q(y, b)$$

重要的观察是，如果我们固定端点 a 和 b ，这个乘积是一个关于 $\langle y, x \rangle$ 的 **profunctor**。通过一些重排（在同构意义上）可以很容易地看到这一点：

$$Q(y, b) \times P(a, x)$$

我们感兴趣的是这个 **profunctor** 的对角线部分的和，即当 x 等于 y 时。

因此，让我们看看如何定义一般 **profunctor** P 的所有对角线项的和。事实上，这个构造适用于任何函子 $P: \mathcal{C}^{op} \times \mathcal{C} \rightarrow D$ ，而不仅仅是 **Set** 值的 **profunctors**。

对角线对象的和由注入定义；在这种情况下，每个 \mathcal{C} 中的对象都有一个注入。这里我们只展示其中两个，以及代表所有其他注入的虚线：

$$\begin{array}{ccc} P(y, y) & \text{-----} & P(x, x) \\ & \searrow \quad \swarrow & \\ & i_y \quad i_x & \\ & \searrow \quad \swarrow & \\ & d & \end{array}$$

如果我们在定义一个和，我们会使其成为一个具有这些注入的通用对象。但由于我们处理的是两个变量的函子，我们希望通过“扩展”某个共同祖先来识别相关的注入——在这里， $P(y, x)$ 。我们希望以下图表在任何连接态射 $f: x \rightarrow y$ 存在时交换：

$$\begin{array}{ccccc} & & P(y, x) & & \\ & \swarrow P(id, f) & & \searrow P(f, id) & \\ P(y, y) & & & & P(x, x) \\ & \searrow i_y & & \swarrow i_x & \\ & d & & & \end{array}$$

这个图表被称为余楔 (co-wedge)，其交换条件称为余楔条件。对于每个 $f: x \rightarrow y$ ，我们要求：

$$i_x \circ P(f, id_y) = i_y \circ P(id_x, f)$$

通用余楔称为余端 (coend)。

由于余端将和推广到可能无限的域，我们使用积分符号来表示它，并将“积分变量”放在顶部：

$$\int^{x: \mathcal{C}} P\langle x, x \rangle$$

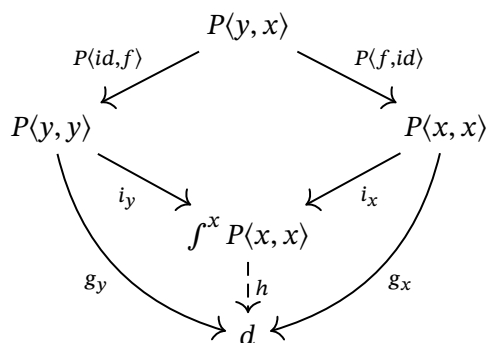
通用性意味着，每当 \mathcal{D} 中有一个对象 d 配备了一族满足余楔条件的箭头 $g_x: P\langle x, x \rangle \rightarrow d$ 时，存在一个从余端到 d 的唯一映射：

$$h: \int^{x: \mathcal{C}} P\langle x, x \rangle \rightarrow d$$

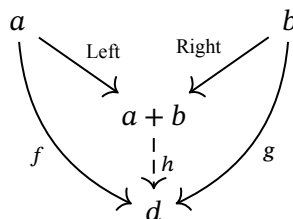
它通过注入 i_x 分解每个 g_x ：

$$g_x = h \circ i_x$$

图示如下：



将其与两个对象的和的定义进行比较：



正如和被定义为通用余楔，余端被定义为通用余楔。

特别是，如果你要构造一个 **Set** 值的 profunctor 的余端，你会从所有集合 $P\langle x, x \rangle$ 的和（一个区分并集）开始。然后你会识别满足余楔条件的这个和的所有元素。每当存在一个元素 $c \in P\langle y, x \rangle$ 和一个态射 $f: x \rightarrow y$ 时，你会将元素 $a \in P\langle x, x \rangle$ 与元素 $b \in P\langle y, y \rangle$ 识别为：

$$P\langle id, f \rangle(c) = b$$

和

$$P\langle f, id \rangle(c) = a$$

注意，在离散范畴（即没有箭头连接的对象集合）中，余楔条件是平凡的（除了恒等态射外没有其他 f ），因此余端只是对角线对象 $P\langle x, x \rangle$ 的简单和（余积）。

外自然变换 (Extranatural transformations)

目标范畴中由源范畴对象参数化的一族箭头通常可以组合成两个函子之间的单个自然变换。

我们在余楔定义中的注入形成了一族由对象参数化的函数，但它们并不完全符合自然变换的定义。

$$\begin{array}{ccc}
 P\langle y, y \rangle & \text{-----} & P\langle x, x \rangle \\
 & \searrow i_y \quad \swarrow i_x & \\
 & d &
 \end{array}$$

问题在于函子 $P: \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{D}$ 在第一个参数中是反变的，在第二个参数中是协变的；因此其对角线部分，在对象上定义为 $x \mapsto P\langle x, x \rangle$ ，既不是反变的也不是协变的。

我们手头最接近的自然性类比是余楔条件：

$$\begin{array}{ccccc}
 & & P\langle y, x \rangle & & \\
 & \swarrow P\langle id, f \rangle & & \searrow P\langle f, id \rangle & \\
 P\langle y, y \rangle & & & & P\langle x, x \rangle \\
 & \searrow i_y & & \swarrow i_x & \\
 & & d & &
 \end{array}$$

确实，与自然性方块一样，它涉及态射 $f: x \rightarrow y$ 的提升（在这里以两种不同的方式）与变换 i 的分量之间的交互。

当然，标准的自然性条件处理的是成对的函子。在这里，变换的目标是一个固定对象 d 。但我们总是可以将其重新解释为常数 profunctor $\Delta_d: \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{D}$ 的输出。因此，为了推广自然性，我们用任意 profunctor Q 替换 Δ_d 。

我们将余楔条件重新解释为更一般的外自然变换的特殊情况。外自然变换是一族箭头：

$$\alpha_{cd}: P\langle c, c \rangle \rightarrow Q\langle d, d \rangle$$

在两个形式如下的函子之间：

$$P: \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{E}$$

$$Q: \mathcal{D}^{op} \times \mathcal{D} \rightarrow \mathcal{E}$$

在 \mathcal{C} 中的外自然性意味着对于任何态射 $f: c \rightarrow c'$ ，以下图表交换：

$$\begin{array}{ccccc}
 & & P\langle c', c \rangle & & \\
 & \swarrow P\langle id, f \rangle & & \searrow P\langle f, id \rangle & \\
 P\langle c', c' \rangle & & & & P\langle c, c \rangle \\
 & \searrow \alpha_{c'd} & & \swarrow \alpha_{cd} & \\
 & & Q\langle d, d \rangle & &
 \end{array}$$

在 d 中的外自然性意味着对于任何态射 $g: d \rightarrow d'$ ，以下图表交换：

$$\begin{array}{ccc}
 & P\langle c, c \rangle & \\
 \alpha_{cd} \swarrow & & \searrow \alpha_{cd'} \\
 Q\langle d, d \rangle & & Q\langle d', d' \rangle \\
 Q\langle id, g \rangle \searrow & & \swarrow Q\langle g, id \rangle \\
 & Q\langle d, d' \rangle &
 \end{array}$$

根据这个定义，我们得到余楔条件作为 profunctor P 与常数 profunctor Δ_d 之间的外自然性。

我们现在可以将余端的定义重新表述为一对 (c, i) ，其中 c 是配备有外自然变换 $i: P \rightarrow \Delta_c$ 的对象，该变换在所有这样的对中是通用的。

通用性意味着对于任何配备有外自然变换 $\alpha: P \rightarrow \Delta_d$ 的对象 d ，存在一个唯一态射 $h: c \rightarrow d$ ，它通过 i 的分量分解 α 的所有分量：

$$\alpha_x = h \circ i_x$$

我们称这个对象 c 为余端，并将其写为：

$$c = \int^x P\langle x, x \rangle$$

Exercise 18.2.1. 验证对于外自然变换 $P \rightarrow \Delta_d$ ，第一个外自然性菱形等价于余楔条件，第二个是平凡满足的。

使用余端的 Profunctor 复合

有了余端的定义，我们现在可以正式定义两个 profunctors 的复合：

$$(P \diamond Q)\langle a, b \rangle = \int^{x: c} Q\langle a, x \rangle \times P\langle x, b \rangle$$

将其与以下 Haskell 代码进行比较：

```
data Procompose p q a b where
  Procompose :: q a x -> p x b -> Procompose p q a b
```

在 Haskell 中我们不必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。余端是使用一族注入定义的；在 Haskell 中，所有这些注入都是由一个单一的多态函数定义的：

```
data Coend p where
  Coend :: p x x -> Coend p
```

参数性 (Parametricity) 然后强制执行余楔条件。

余端在处理 `profunctors` 时引入了一个新的抽象层次。使用余端的计算通常利用它们的映射出属性。要定义一个从余端到某个对象 d 的映射：

$$\int^x P\langle x, x \rangle \rightarrow d$$

只需定义一族从函子的对角线项到 d 的函数：

$$g_x : P\langle x, x \rangle \rightarrow d$$

满足余楔条件。你可以从这个技巧中获得很多好处，特别是与 Yoneda 引理结合时。我们将在接下来的内容中看到这方面的例子。

Exercise 18.2.2. 为以下 *profunctor* 对定义 **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 18.2.3. *Profunctor* 复合可以使用余端表示：

```
newtype CoEndCompose p q a b = CoEndCompose (Coend (ProPair q p a b))
```

为 **CoEndCompose** 定义 **Profunctor** 实例。

余极限作为余端

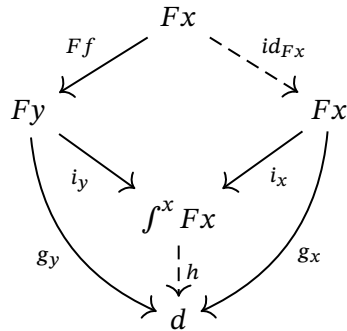
忽略其中一个变量的双变量函数等价于单变量函数。类似地，忽略其中一个变量的 `profunctor` 等价于一个函子。反之，给定一个函子 F ，我们可以构造一个平凡的 `profunctor`。其在对象上的作用由下式给出：

$$P\langle x, y \rangle = Fy$$

其在一对箭头上的作用忽略其中一个箭头：

$$P\langle f, g \rangle = Fg$$

对于任何 $f: x \rightarrow y$ ，我们对此类 **profunctor** 的余端定义简化为以下图表：



在缩小恒等箭头后，原始的余楔变为余锥，通用条件变为余极限的定义。这证明了使用余端符号表示余极限的合理性：

$$\int^x Fx = \operatorname{colim} F$$

函子 F 定义了目标范畴中的一个图表。模式是整个源范畴。

如果我们考虑离散范畴，我们可以获得有用的直觉，其中 **profunctor** 是一个（可能是无限的）矩阵，余端是其对角线元素的和（余积）。沿一个轴为常数的 **profunctor** 对应于一个行相同的矩阵（每行由“向量” Fx 给出）。此类矩阵的对角线元素的和等于向量 Fx 的所有分量的和。

$$\begin{pmatrix} \textcolor{red}{Fa} & Fb & Fc & \dots \\ Fa & \textcolor{red}{Fb} & Fc & \dots \\ Fa & Fb & \textcolor{red}{Fc} & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

在非离散范畴中，这个和推广为余极限。

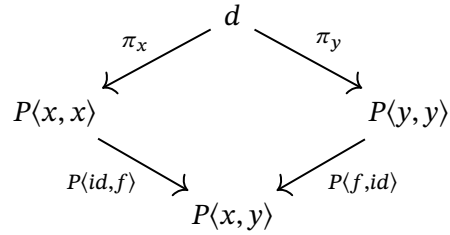
18.3 Ends

正如余 **end** (**coend**) 推广了 **profunctor** 对角元素的和——它的对偶，**end** 推广了积。积由其投影定义，**end** 也是如此。

我们在积的定义中使用的 **span** 的推广将是一个对象 d ，带有一族投影，每个对象 x 对应一个：

$$\pi_x: d \rightarrow P\langle x, x \rangle$$

余楔 (co-wedge) 的对偶称为楔 (wedge):



对于每个箭头 $f: x \rightarrow y$, 我们要求:

$$P\langle f, id_y \rangle \circ \pi_y = P\langle id_x, f \rangle \circ \pi_x$$

end 是一个泛楔 (universal wedge)。我们也用积分符号表示它, 这次将“积分变量”放在底部。

$$\int_{x: c} P\langle x, x \rangle$$

你可能会好奇, 为什么在微积分中, 基于乘法而不是加法的积分很少使用。这是因为我们总是可以使用对数将乘法替换为加法。在范畴论中我们没有这种便利, 因此 **end** 和 **coend** 同样重要。

总结一下, **end** 是一个配备有一族态射 (投影) 的对象:

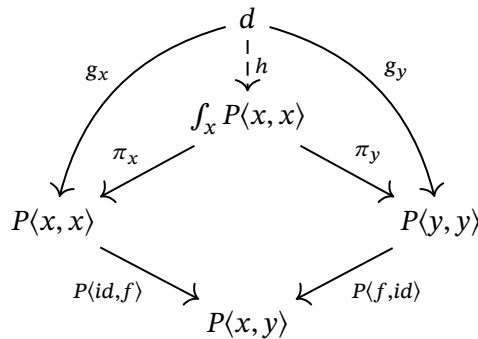
$$\pi_a: \left(\int_x P\langle x, x \rangle \right) \rightarrow P\langle a, a \rangle$$

满足楔条件。

它在这些对象中是泛的; 也就是说, 对于任何其他配备有一族箭头 g_x 并满足楔条件的对象 d , 存在一个唯一的态射 h , 使得族 g_x 通过族 π_x 分解:

$$g_x = \pi_x \circ h$$

图示如下:



等价地, 我们可以说 **end** 是一个由对象 $e = \int_x P\langle x, x \rangle$ 和一个外自然变换 (extranatural transformation) $\pi: \Delta_d \rightarrow e$ 组成的对 (e, π) , 它在这些对中是泛的。楔条件实际上是外自然性条件的一个特例。

如果你要构造一个 **Set** 值 profunctor 的 **end**，你会从范畴 \mathcal{C} 中所有对象的 $P\langle x, x \rangle$ 的巨型积开始，然后剔除不满足楔条件的元组。

特别地，想象用单元素集 **1** 代替 d 。族 g_x 会从每个集合 $P\langle x, x \rangle$ 中选择一个元素。这将给你一个巨型元组。你会剔除大部分这些元组，只留下满足楔条件的那些。

再次，在 Haskell 中，由于参数性 (parametricity)，楔条件自动满足，profunctor p 的 **end** 的定义简化为：

```
type End p = forall x. p x x
```

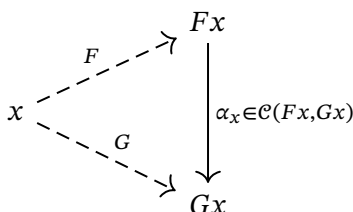
Haskell 中 **End** 的实现并没有展示它是 **Coend** 的对偶。这是因为，在撰写本文时，Haskell 没有内置的存在类型语法。如果有，**Coend** 将实现为：

```
type Coend p = exists x. p x x
```

Coend 和 **End** 之间的存在/全称对偶意味着构造一个 **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**，你必须提供一族值 $p \ x \ x$ ，每个类型 x 对应一个。换句话说，你需要一个由 x 参数化的多态公式。多态函数的定义就是这种公式的典型例子。

自然变换作为 end

end 最有趣的应用是简洁地定义自然变换的集合。考虑两个函子 F 和 G ，它们在两个范畴 \mathcal{B} 和 \mathcal{C} 之间。它们之间的自然变换是 \mathcal{C} 中的一族箭头 α_x 。你可以将其视为从每个 **hom** 集 $\mathcal{C}(Fx, Gx)$ 中选取一个元素 α_x 。



我们知道映射 $\langle a, b \rangle \rightarrow \mathcal{C}(a, b)$ 定义了一个 profunctor。事实证明，对于任何一对函子，映射 $\langle a, b \rangle \rightarrow \mathcal{C}(Fa, Gb)$ 也表现得像一个 profunctor。这个 profunctor 在一对箭头 $\langle f: s \rightarrow a, g: b \rightarrow t \rangle$ 上的作用是一个函数：

$$\mathcal{C}(Fa, Gb) \rightarrow \mathcal{C}(Fs, Gt)$$

由以下复合给出：

$$Fs \xrightarrow{Ff} Fa \xrightarrow{h} Gb \xrightarrow{Gg} Gt$$

其中 h 是 $\mathcal{C}(Fa, Gb)$ 中的一个元素。

这个 profunctor 的对角部分非常适合作为自然变换的分量。事实上，**end**：

$$\int_{x: \mathcal{B}} \mathcal{C}(Fx, Gx)$$

定义了从 F 到 G 的自然变换的集合。

为了证明这一点，让我们检查楔条件。代入我们的 **profunctor**，我们得到：

$$\begin{array}{ccc}
 & \int_x \mathcal{C}(Fx, Gx) & \\
 \pi_a \swarrow & & \searrow \pi_b \\
 \mathcal{C}(Fa, Ga) & & \mathcal{C}(Fb, Gb) \\
 (Ff \circ -) \searrow & & \swarrow (- \circ Gf) \\
 & \mathcal{C}(Fa, Gb) &
 \end{array}$$

我们可以通过实例化单元素集的泛条件来选取集合 $\int_x \mathcal{C}(Fx, Gx)$ 中的一个元素：

$$\begin{array}{ccc}
 & 1 & \\
 \alpha_a \swarrow & \downarrow \alpha & \searrow \alpha_b \\
 & \int_x \mathcal{C}(Fx, Gx) & \\
 \pi_a \swarrow & & \searrow \pi_b \\
 \mathcal{C}(Fa, Ga) & & \mathcal{C}(Fb, Gb) \\
 (Ff \circ -) \searrow & & \swarrow (- \circ Gf) \\
 & \mathcal{C}(Fa, Gb) &
 \end{array}$$

我们从 **hom** 集 $\mathcal{C}(Fa, Ga)$ 中选取分量 α_a ，从 $\mathcal{C}(Fb, Gb)$ 中选取分量 α_b 。楔条件简化为：

$$Ff \circ \alpha_a = \alpha_b \circ Gf$$

对于任何 $f: a \rightarrow b$ 。这正是自然性条件。因此，这个 **end** 的任何元素 α 自动是一个自然变换。

因此，自然变换的集合，或函子范畴中的 **hom** 集，由 **end** 给出：

$$[\mathcal{C}, \mathcal{D}](F, G) \cong \int_{x: \mathcal{B}} \mathcal{C}(Fx, Gx)$$

在 **Haskell** 中，这与我们之前的定义一致：

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

正如我们之前讨论的，要构造一个 **End**，我们必须给它一族由类型参数化的值。在这里，这些值是多态函数的分量。

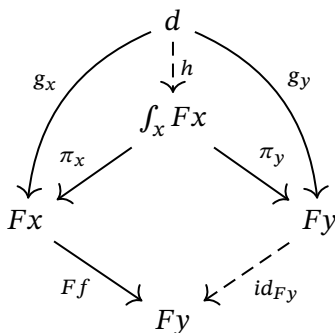
极限作为 ends

正如我们能够将余极限（colimits）表示为 **coends**，我们也可以将极限（limits）表示为 **ends**。和之前一样，我们定义一个忽略其第一个参数的平凡 **profunctor**：

$$P\langle x, y \rangle = Fy$$

$$P\langle f, g \rangle = Fg$$

定义 **end** 的泛条件变成了泛锥（universal cone）的定义：



因此，我们可以使用 **end** 符号表示极限：

$$\int_x Fx = \lim F$$

Exercise 18.3.1. 积是一个从二对象范畴 **2** 出发的函子的极限。证明它可以定义为一个 **end**。提示：**2** 中没有非恒等态射。

18.4 Hom-函子的连续性

在范畴论中，一个函子 F 被称为连续的，如果它保持极限（并且如果它保持余极限，则称为余连续的）。这意味着，如果你在源范畴中有一个图表，那么先使用 F 映射图表然后在目标范畴中取极限，或者先在源范畴中取极限然后使用 F 映射这个极限，结果是相同的。

Hom-函子是一个在其第二个参数上连续的函子的例子。由于积是极限的最简单例子，这意味着特别地：

$$\mathcal{C}(x, a \times b) \cong \mathcal{C}(x, a) \times \mathcal{C}(x, b)$$

左边将 **Hom**-函子应用于积（一个跨度的极限）。右边映射图表，这里只是一对对象，并在目标范畴中取积（极限）。**Hom**-函子的目标范畴是 **Set**，所以这只是一个笛卡尔积。两边通过积的普遍性质是同构的：映射到积由映射到两个对象的一对映射定义。

Hom-函子在第一个参数上的连续性则是相反的：它将余极限映射到极限。同样，余极限的最简单例子是和，所以我们有：

$$\mathcal{C}(a + b, x) \cong \mathcal{C}(a, x) \times \mathcal{C}(b, x)$$

这源于和的普遍性：从和映射出去由从两个对象映射出去的一对映射定义。

可以证明，一个端可以表示为一个极限，而余端可以表示为一个余极限。因此，通过 **Hom**-函子的连续性，我们总是可以将积分号从 **Hom**-集中拉出来。类比于积，我们有一个端的映射入公式：

$$\mathcal{D}\left(d, \int_a P\langle a, a \rangle\right) \cong \int_a \mathcal{D}(d, P\langle a, a \rangle)$$

类比于和，我们有一个余端的映射出公式：

$$\mathcal{D}\left(\int^a P\langle a, a \rangle, d\right) \cong \int_a \mathcal{D}(P\langle a, a \rangle, d)$$

注意，在这两种情况下，右边都是一个端。

18.5 Fubini 规则

微积分中的 Fubini 规则（Fubini rule）说明了在什么条件下我们可以交换二重积分的积分顺序。事实证明，我们同样可以交换双端（double ends）和双余端（coends）的顺序。对于端（ends）的 Fubini 规则适用于形如 $P: \mathcal{C} \times \mathcal{C}^{op} \times \mathcal{D} \times \mathcal{D}^{op} \rightarrow \mathcal{E}$ 的函子。以下表达式，只要它们存在，就是同构的：

$$\int_{c: \mathcal{C}} \int_{d: \mathcal{D}} P\langle c, c \rangle \langle d, d \rangle \cong \int_{d: \mathcal{D}} \int_{c: \mathcal{C}} P\langle c, c \rangle \langle d, d \rangle \cong \int_{\langle c, d \rangle: \mathcal{C} \times \mathcal{D}} P\langle c, c \rangle \langle d, d \rangle$$

在最后一个端中，函子 P 被重新解释为 $P: (\mathcal{C} \times \mathcal{D})^{op} \times (\mathcal{C} \times \mathcal{D}) \rightarrow \mathcal{E}$ 。

类似的规则也适用于余端（coends）。

18.6 忍者米田引理

在将自然变换集表示为 **end** 之后，我们现在可以重写米田引理。这是原始表述：

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), F) \cong Fa$$

这里， F 是一个从 \mathcal{C} 到 **Set** 的（协变）函子（一个余预层），同态函子 $\mathcal{C}(a, -)$ 也是如此。将自然变换集表示为 **end**，我们得到：

$$\int_{x: \mathcal{C}} \mathbf{Set}(\mathcal{C}(a, x), Fx) \cong Fa$$

类似地，我们有一个逆变函子（预层） G 的米田引理：

$$\int_{x: \mathcal{C}} \mathbf{Set}(\mathcal{C}(x, a), Gx) \cong Ga$$

这些用 **end** 表示的米田引理版本，常常被半开玩笑地称为忍者米田引理。由于“积分变量”是显式的，这使得它们在复杂公式中更容易使用。

还有一组对偶的忍者余米田引理，它们使用 **coend** 代替。对于协变函子，我们有：

$$\int^{x: \mathcal{C}} \mathcal{C}(x, a) \times Fx \cong Fa$$

对于逆变函子，我们有：

$$\int_{x: \mathcal{C}} \mathcal{C}(a, x) \times Gx \cong Ga$$

物理学家可能会注意到这些公式与涉及狄拉克 δ 函数（严格来说，是一个分布）的积分的相似性。这就是为什么 **profunctor** 有时被称为 **distributors**，遵循“**distributors** 之于函子，如同 **distributions** 之于函数”的说法。工程师可能会注意到同态函子与脉冲函数的相似性。

这种直觉通常通过说我们可以在这个公式中“对 x 进行积分”来表达，结果是在被积函数 Gx 中用 a 替换 x 。

如果 \mathcal{C} 是一个离散范畴，**coend** 简化为和（余积），同态函子简化为单位矩阵（克罗内克 δ ）。余米田引理则变为：

$$\sum_j \delta_i^j v_j = v_i$$

事实上，许多线性代数直接转化为 **Set** 值函子的理论。你通常可以将这些函子视为向量空间中的向量，其中同态函子形成基。**Profunctor** 成为矩阵，**coend** 用于乘这些矩阵、计算它们的迹，或将向量乘以矩阵。

Profunctor 的另一个名称，尤其是在澳大利亚，是“双模”。这是因为 **profunctor** 对态射的提升与集合上的左右作用有些相似。

我们现在将继续证明余米田引理，这非常有启发性，因为它使用了一些常见的技巧。最重要的是，我们依赖于米田引理的推论，该推论说，如果从两个对象到任意对象的所有映射都是同构的，那么这两个对象本身也是同构的。在我们的情况下，我们将考虑映射到任意集合 S ，并证明：

$$\mathbf{Set} \left(\int^{x: \mathcal{C}} \mathcal{C}(x, a) \times Fx, S \right) \cong \mathbf{Set}(Fa, S)$$

利用同态函子的余连续性，我们可以将积分符号拉出，用 **end** 替换 **coend**：

$$\int_{x: \mathcal{C}} \mathbf{Set}(\mathcal{C}(x, a) \times Fx, S)$$

由于集合范畴是笛卡尔闭的，我们可以对积进行柯里化：

$$\int_{x: \mathcal{C}} \mathbf{Set}(\mathcal{C}(x, a), S^{Fx})$$

我们现在可以使用米田引理“对 x 进行积分”。结果是 S^{Fa} 。最后，在 **Set** 中，指数对象与同态集同构：

$$S^{Fa} \cong \mathbf{Set}(Fa, S)$$

由于 S 是任意的，我们得出结论：

$$\int^{x: \mathcal{C}} \mathcal{C}(x, a) \times Fx \cong Fa$$

Exercise 18.6.1. 证明余米田引理的逆变版本。

Haskell 中的米田引理

我们已经在 Haskell 中看到了米田引理的实现。我们现在可以用 `end` 来重写它。我们首先定义一个将在 `end` 下使用的 `profunctor`。它的类型构造函数接受一个函子 `f` 和一个类型 `a`，并生成一个在 `x` 中逆变、在 `y` 中协变的 `profunctor`：

```
data Yo f a x y = Yo ((a -> x) -> f y)
```

米田引理建立了这个 `profunctor` 上的 `end` 与通过函子 `f` 作用于 `a` 得到的类型之间的同构。这个同构由一对函数见证：

```
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，余米田引理使用以下 `profunctor` 上的 `coend`：

```
data CoY f a x y = CoY (x -> a) (f y)
```

同构由一对函数见证。第一个函数说，如果你有一个函数 `x -> a` 和一个 `x` 的函子，你可以使用 `fmap` 制作一个 `a` 的函子：

```
coyoneda :: Functor f => Coend (CoY f a) -> f a
coyoneda (Coend (CoY g fa)) = fmap g fa
```

你可以在不知道存在类型 `x` 的情况下做到这一点。

第二个函数说，如果你有一个 `a` 的函子，你可以通过将其（与恒等函数一起）注入存在类型来创建一个 `coend`：

```
coyoneda_1 :: Functor f => f a -> Coend (CoY f a)
coyoneda_1 fa = Coend (CoY id fa)
```

18.7 Day 卷积

电气工程师对卷积的概念非常熟悉。我们可以通过对其中一个流进行移位并将其与另一个流的乘积求和来对两个流进行卷积：

$$(f \star g)(x) = \int_{-\infty}^{\infty} f(y)g(x-y)dy$$

这个公式几乎可以逐字翻译到范畴论中。我们可以首先将积分替换为余积（coend）。问题在于，我们不知道如何对对象进行减法运算。然而，在余笛卡尔范畴中，我们知道如何对它们进行加法运算。

注意到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“脉冲函数” $\delta(a+b-x)$ 来强制执行这个条件。在范畴论中，我们使用 hom-函子 $\mathcal{C}(a+b, x)$ 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：

$$(F \star G)x = \int^{a,b} \mathcal{C}(a+b, x) \times Fa \times Gb$$

非正式地，如果我们可以将减法定义为余积的右伴随，我们可以写成：

$$\int^{a,b} \mathcal{C}(a+b, x) \times Fa \times Gb \cong \int^{a,b} \mathcal{C}(a, b-x) \times Fa \times Gb \cong \int^b F(b-x) \times Gb$$

余积并没有什么特别之处，因此一般来说，Day 卷积可以在任何具有张量积的幺半范畴中定义：

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \otimes b, x) \times Fa \times Gb$$

事实上，对于幺半范畴 $(\mathcal{C}, \otimes, I)$ 的 Day 卷积赋予了余预层范畴 $[\mathcal{C}, \mathbf{Set}]$ 一个幺半结构。简单地说，如果你可以在 \mathcal{C} 中乘对象，你就可以在 \mathcal{C} 上乘集合值函子。

很容易验证 Day 卷积是结合的（在同构意义下），并且 $\mathcal{C}(I, -)$ 作为单位对象。例如，我们有：

$$(\mathcal{C}(I, -) \star G)x = \int^{a,b} \mathcal{C}(a \otimes b, x) \times \mathcal{C}(I, a) \times Gb \cong \int^b \mathcal{C}(I \otimes b, x) \times Gb \cong Gx$$

因此，Day 卷积的单位是幺半单位处的 Yoneda 函子，这引出了一个字谜口号：“ONE of DAY is the YONEDA of ONE。”

如果张量积是对称的，那么相应的 Day 卷积也是对称的（在同构意义下）。

在笛卡尔闭范畴的特殊情况下，我们可以使用柯里化伴随来简化公式：

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \times b, x) \times Fa \times Gb \cong \int^{a,b} \mathcal{C}(a, x^b) \times Fa \times Gb \cong \int^b F(x^b) \times Gb$$

在 Haskell 中，基于积的 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——给定一个将两个不同值组合成一个的函数。

Exercise 18.7.1. 为 `Day` 定义 `Functor` 实例。

Exercise 18.7.2. 实现 `Day` 的结合子。

```
assoc :: Day f (Day g h) x -> Day (Day f g) h x
```

提示：在构造结果时，你可以自由选择适合的存在类型，例如，它可以是配对类型。

应用函子作为么半群

我们之前已经看到应用函子作为松弛么半函子的定义。事实证明，就像单子一样，应用函子也可以定义为么半群。

回想一下，么半群是么半范畴中的一个对象。我们感兴趣的范畴是余预层范畴 $[\mathcal{C}, \mathbf{Set}]$ 。如果 \mathcal{C} 是笛卡尔的，那么余预层范畴关于 Day 卷积是么半的，单位对象为 $\mathcal{C}(I, -)$ 。这个范畴中的么半群是一个函子 F ，配备了两个作为单位和乘法的自然变换：

$$\eta : \mathcal{C}(I, -) \rightarrow F$$

$$\mu : F \star F \rightarrow F$$

特别是在单位是终端对象的笛卡尔闭范畴中， $\mathcal{C}(1, a)$ 同构于 a ，单位在 a 处的分量是：

$$\eta_a : a \rightarrow Fa$$

你可能会认出这个函数是 `Applicative` 定义中的 `pure`。

```
pure :: a -> f a
```

让我们考虑从中选择 μ 的自然变换集合。我们将其写为一个 `end`：

$$\mu \in \int_x \mathbf{Set}((F \star F)x, Fx)$$

代入 Day 卷积的定义，我们得到：

$$\int_x \mathbf{Set}\left(\int^{a,b} \mathcal{C}(a \times b, x) \times Fa \times Fb, Fx\right)$$

我们可以利用 `hom`-函子的余连续性将余积拉出：

$$\int_{x,a,b} \mathbf{Set}(\mathcal{C}(a \times b, x) \times Fa \times Fb, Fx)$$

然后我们可以在 **Set** 中使用柯里化伴随来得到：

$$\int_{x,a,b} \mathbf{Set}(\mathcal{C}(a \times b, x), \mathbf{Set}(Fa \times Fb, Fx))$$

最后，我们应用 Yoneda 引理来对 x 进行积分：

$$\int_{a,b} \mathbf{Set}(Fa \times Fb, F(a \times b))$$

结果是从中选择松弛幺半函子第二部分的自然变换集合：

```
(>*<) :: f a -> f b -> f (a, b)
```

自由应用函子

我们刚刚了解到应用函子是幺半范畴中的幺半群：

$$([\mathcal{C}, \mathbf{Set}], \mathcal{C}(I, -), \star)$$

很自然地会问这个范畴中的自由幺半群是什么。

就像我们对自由单子所做的那样，我们将自由应用函子构造为初始代数，或列表函子的最小不动点。回想一下，列表函子定义为：

$$\Phi_a x = 1 + a \otimes x$$

在我们的情况下，它变为：

$$\Phi_F G = \mathcal{C}(I, -) + F \star G$$

它的不动点由递归公式给出：

$$A_F \cong \mathcal{C}(I, -) + F \star A_F$$

在将其翻译到 Haskell 时，我们观察到来自单位 $() \rightarrow a$ 的函数同构于 a 的元素。

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
```

我内联了 Day 卷积的定义：

```
data Day f g x where
  Day :: ((a, b) -> x) -> f a -> g b -> Day f g x
```

展示 `FreeA f` 是应用函子的最简单方法是通过 `Monoidal`:

```
class Monoidal f where
  unit  :: f ()
  (>*<) :: f a -> f b -> f (a, b)
```

由于 `FreeA f` 是列表的推广，自由应用函子的 `Monoidal` 实例推广了列表连接的概念。我们对第一个列表进行模式匹配，结果有两种情况。

在第一种情况下，我们有一个 `DoneA x` 而不是空列表。将其前置到第二个参数不会改变列表的长度，但会修改其中存储的值的类型。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` 是一个包含 `a`'s 的函子，尾部 `frb` 的类型为 `FreeA f b`。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，我们使用递归调用 `>*<` 连接两个尾部，并将 `fa` 前置到它。为了将这个头部与新的尾部粘合，我们必须提供一个重新关联对的函数：

```
reassoc :: ((a, b) -> x) -> (a, (b, y)) -> (x, y)
reassoc abx (a, (b, y)) = (abx (a, b), y)
```

因此，完整的实例是：

```
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，生成 `Applicative` 实例就很简单了：

```
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 18.7.3. 为自由应用函子定义 **Functor** 实例。

18.8 Profunctor 的双范畴

既然我们已经知道如何使用 **coends** 来组合 **profunctor**，那么问题来了：是否存在一个范畴，其中 **profunctor** 作为态射？答案是肯定的，只要我们稍微放宽一些规则。问题在于 **profunctor** 组合的范畴律并不是“严格”满足的，而是只在同构的意义下成立。

例如，我们可以尝试证明 **profunctor** 组合的结合律。我们从：

$$((P \diamond Q) \diamond R)\langle s, t \rangle = \int^b \left(\int^a P\langle s, a \rangle \times Q\langle a, b \rangle \right) \times R\langle b, t \rangle$$

开始，经过一些变换后，得到：

$$(P \diamond (Q \diamond R))\langle s, t \rangle = \int^a P\langle s, a \rangle \times \left(\int^b Q\langle a, b \rangle \times R\langle b, t \rangle \right)$$

我们使用了积的结合性以及可以通过 Fubini 定理交换 **coends** 的顺序这一事实。这两者都只在同构的意义下成立。我们并没有得到“严格”的结合律。

单位 **profunctor** 实际上是 **hom** 函子，可以符号化地表示为 $\mathcal{C}(-, =)$ ，其中两个参数都是占位符。例如：

$$(\mathcal{C}(-, =) \diamond P)\langle s, t \rangle = \int^a \mathcal{C}(s, a) \times P\langle a, t \rangle \cong P\langle s, t \rangle$$

这是（反变的）**ninja co-Yoneda** 引理的结果，它也是一个同构，而不是等式。

范畴律在同构的意义下满足的范畴称为双范畴。注意，这样的范畴必须配备 2-细胞——即态射之间的态射，我们已经在 2-范畴的定义中见过。我们需要这些 2-细胞以便能够定义 1-细胞之间的同构。

双范畴 **Prof** 以（小）范畴为对象，**profunctor** 为 1-细胞，自然变换为 2-细胞。

由于 **profunctor** 是函子 $\mathcal{C}^{op} \times \mathcal{D} \rightarrow \mathbf{Set}$ ，因此它们之间的自然变换的标准定义适用。它是一个由 $\mathcal{C}^{op} \times \mathcal{D}$ 的对象参数化的函数族，这些对象本身是对象的对。

两个 **profunctor** P 和 Q 之间的变换 $\alpha_{\langle a, b \rangle}$ 的自然性条件形式为：

$$\begin{array}{ccc} & P\langle a, b \rangle & \\ \alpha_{\langle a, b \rangle} \swarrow & & \searrow P\langle f, g \rangle \\ Q\langle a, b \rangle & & P\langle s, t \rangle \\ Q\langle f, g \rangle \searrow & & \swarrow \alpha_{\langle s, t \rangle} \\ & Q\langle s, t \rangle & \end{array}$$

对于每一对箭头：

$$\langle f : s \rightarrow a, g : b \rightarrow t \rangle$$

双范畴中的 Monad

我们之前已经看到，范畴、函子和自然变换形成了一个 2-范畴 **Cat**。让我们关注一个对象，即范畴 \mathcal{C} ，它是 **Cat** 中的一个 0-细胞。从该对象开始并结束的 1-细胞形成一个常规范畴，在这种情况下，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 endo-1-细胞。它们之间的箭头是外部 2-范畴的 2-细胞。

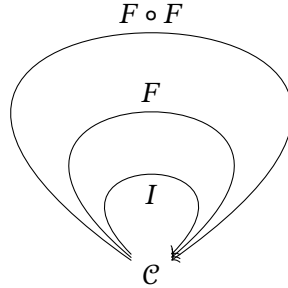
这个 endo-1-细胞范畴自动配备了幺半结构。我们简单地将张量积定义为 1-细胞的组合——所有具有相同源和目标的 1-细胞都可以组合。幺半单位对象是恒等 1-细胞 I 。在 $[\mathcal{C}, \mathcal{C}]$ 中，这个积是自函子的组合，单位是恒等函子。

如果我们现在只关注一个 endo-1-细胞 F ，我们可以“平方”它，即使用幺半积将其与自身相乘。换句话说，使用 1-细胞组合来创建 $F \circ F$ 。我们说 F 是一个 monad，如果我们可以找到 2-细胞：

$$\mu: F \circ F \rightarrow F$$

$$\eta: I \rightarrow F$$

它们的行为类似于乘法和单位，即它们使结合性和单位图交换。



事实上，monad 可以在任意双范畴中定义，而不仅仅是 2-范畴 **Cat**。

Prof 中的 Prearrow 作为 Monad

由于 **Prof** 是一个双范畴，我们可以在其中定义一个 monad。它是一个 endo-profunctor (1-细胞)：

$$P: \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathbf{Set}$$

配备两个自然变换 (2-细胞)：

$$\mu: P \diamond P \rightarrow P$$

$$\eta: \mathcal{C}(-, =) \rightarrow P$$

它们满足结合性和单位条件。

让我们将这些自然变换看作 `ends` 的元素。例如：

$$\mu \in \int_{\langle a, b \rangle} \mathbf{Set} \left(\int^x P\langle a, x \rangle \times P\langle x, b \rangle, P\langle a, b \rangle \right)$$

通过 `co` 连续性，这等价于：

$$\int_{\langle a, b \rangle, x} \mathbf{Set} (P\langle a, x \rangle \times P\langle x, b \rangle, P\langle a, b \rangle)$$

类似地，单位自然变换是：

$$\eta \in \int_{\langle a, b \rangle} \mathbf{Set} (\mathcal{C}(a, b), P\langle a, b \rangle)$$

在 Haskell 中，这样的 profunctor monad 被称为 `prearrow`：

```
class Profunctor p => PreArrow p where
  (>>>) :: p a x -> p x b -> p a b
  arr    :: (a -> b) -> p a b
```

一个 `Arrow` 是一个同时也是 Tambara 模块的 `PreArrow`。我们将在下一章讨论 Tambara 模块。

18.9 存在性透镜

范畴论俱乐部的第一条规则是：不要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，只能使用箭头。

Haskell 中的存在性透镜

一个对象是复合的——即具有部分——意味着什么？至少，你应该能够检索出这样一个对象的一部分。如果你能用一个新的部分替换它，那就更好了。这几乎定义了一个透镜：

```
get :: s -> a
set :: s -> a -> s
```

在这里，`get` 从整体 `s` 中提取部分 `a`，而 `set` 用一个新的 `a` 替换该部分。透镜定律有助于强化这一图景。而这一切都是用箭头来完成的。

描述复合对象的另一种方式是，它可以被拆分为焦点和残差。关键在于，虽然我们想知道焦点的类型，但我们不关心残差的类型。我们只需要知道残差可以与焦点结合以重新创建整个对象。

在 Haskell 中，我们可以使用 `存在类型` 来表达这一思想：

```
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 可以被拆分为并重新构建为积 (c, a) 。



`get/set` 版本的透镜可以从这种存在形式中推导出来。

```
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，我们不需要知道残差的类型。我们利用了存在性透镜同时包含 c 的生产者和消费者这一事实，我们只是在两者之间进行调解。

无法提取“裸”残差，这由以下代码无法编译的事实证明：

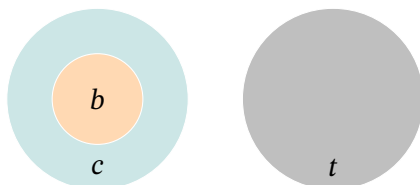
```
getResidue :: LensE s a -> c
getResidue (LensE (l, r)) = fst . l
```

范畴论中的存在性透镜

我们可以通过将存在类型表示为余积，轻松地将透镜的新定义翻译到范畴论中：

$$\int^c \mathcal{C}(s, c \times a) \times \mathcal{C}(c \times a, s)$$

事实上，我们可以将其推广到类型变化的透镜，其中焦点 a 可以被替换为不同类型 b 的新焦点。将 a 替换为 b 将产生一个新的复合对象 t ：



透镜现在由两对对象参数化： $\langle s, t \rangle$ 用于外部对象， $\langle a, b \rangle$ 用于内部对象。存在性残差 c 仍然隐藏：

$$\mathcal{L}\langle s, t \rangle \langle a, b \rangle = \int^c \mathcal{C}(s, c \times a) \times \mathcal{C}(c \times b, t)$$

余积下的积是 *profunctor* 的对角部分，它在 y 中是协变的，在 x 中是逆变的：

$$\mathcal{C}(s, y \times a) \times \mathcal{C}(x \times b, t)$$

Exercise 18.9.1. 证明：

$$\mathcal{C}(s, y \times a) \times \mathcal{C}(x \times b, t)$$

是 $\langle x, y \rangle$ 中的 *profunctor*。

Haskell 中的类型变化透镜

在 Haskell 中，我们可以将类型变化透镜定义为以下存在类型：

```
data LensE s t a b where
  LensE :: (s -> (c, a)) -> ((c, b) -> t) -> LensE s t a b
```

与之前一样，我们可以使用存在性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， $s \rightarrow (c, a)$ 和 $(c, b) \rightarrow t$ 通常被称为 前向和 后向传递。前向传递可用于提取焦点 a 。后向传递回答了以下问题：如果我们希望前向传递的结果是某个其他 b ，我们应该传递什么 t 给它？

有时我们只是在问：如果我们希望通过 b 改变焦点，我们应该对输入进行什么改变 t 。后一种观点在使用透镜描述神经网络时特别有用。

透镜的最简单示例作用于积。它可以提取或替换积的一个组件，将另一个组件视为残差。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，整体的类型是积 (c, a) 。当我们用 b 替换 a 时，我们最终得到目标类型 (c, b) 。由于源和目标已经是积，存在性透镜定义中的两个函数只是恒等函数。

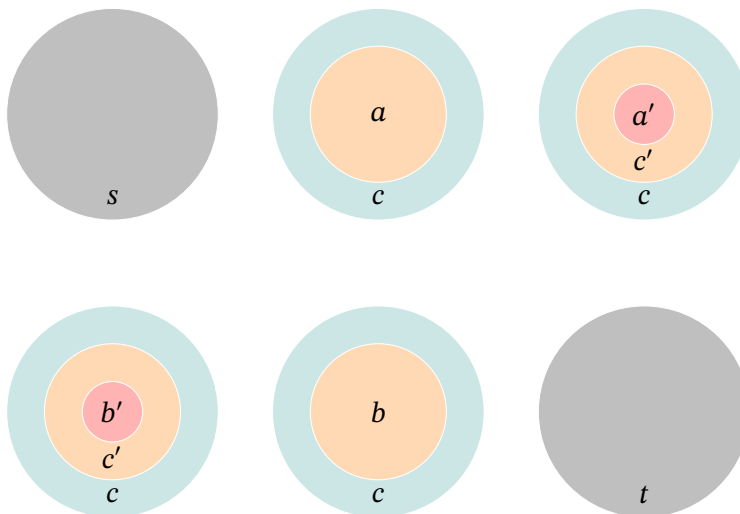
透镜组合

使用透镜的主要优势在于它们可以组合。两个透镜的组合让我们可以放大组件的子组件。

假设我们从一个透镜开始，它允许我们访问焦点 `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` 内部的焦点 `a'`，并用 `b'` 替换它，以给我们一个新的 `b`。

我们现在可以构造一个复合透镜，它可以在 `s` 和 `t` 内部访问 `a'` 和 `b'`。关键在于我们意识到我们可以将两个残差的积作为新的残差：



```
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复合给出：

$$s \xrightarrow{l_1} (c, a) \xrightarrow{(id, l_2)} (c, (c', a')) \xrightarrow{assoc'} ((c, c'), a')$$

右映射由以下复合给出：

$$((c, c'), b') \xrightarrow{assoc} (c, (c', b')) \xrightarrow{(id, r_2)} (c, b) \xrightarrow{r_1} t$$

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)

```

作为一个例子，让我们组合两个积透镜：

```

l3 :: LensE (c, (c', a')) (c, (c', b')) a' b'
l3 = compLens prodLens prodLens

```

并将其应用于嵌套积：

```

x :: (String, (Bool, Int))
x = ("Outer", (True, 42))

```

我们的复合透镜不仅允许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

由于透镜可以组合，你可能会想知道是否存在一个范畴，其中透镜是 `hom`-集。

确实，存在一个范畴 **Lens**，其对象是 \mathcal{C} 中的对象对，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。

存在性透镜的组合公式过于复杂，无法在实践中使用。在下一章中，我们将看到使用 **Tambara** 模块的透镜的另一种表示，其中组合只是函数的组合。

18.10 透镜与纤维化

使用纤维丛的语言，我们可以从另一个角度来理解透镜。定义纤维化的投影 p 可以被视为将丛 E “分解”为纤维。

在这种视角下， p 扮演了 `get` 的角色：

$$p: E \rightarrow B$$

其中基空间 B 表示焦点的类型，而 E 表示可以从中提取该焦点的复合类型。

透镜的另一部分，`set`，是一个映射：

$$q: E \times B \rightarrow E$$

让我们看看如何用纤维化来解释它。

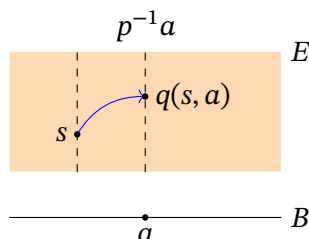
传输律

我们将 q 解释为将丛 E 中的一个元素”传输”到一个新的纤维。新的纤维由 B 中的一个元素指定。

这种传输的性质由 `get/set` 透镜律，或称传输律表达，它表示”你设置的就是你得到的”：

```
get (set s a) = a
```

我们说 $q(s, a)$ 将 s 传输到 a 上的新纤维：

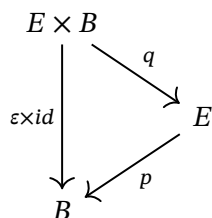


我们可以用 p 和 q 重写这个定律：

$$p \circ q = \pi_2$$

其中 π_2 是积的第二个投影。

等价地，我们可以将其表示为交换图：



这里，我没有使用投影 π_2 ，而是使用了余么半群的余单位 ε ：

$$\varepsilon: E \rightarrow 1$$

然后遵循积的单位律。使用余么半群使得将这个构造推广到么半范畴中的张量积更加容易。

恒等律

这是 `set/get` 律，或称恒等律。它表示”如果你设置你得到的，那么什么都不会改变”：

```
set s (get s) = s
```

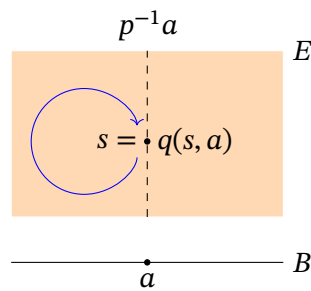
我们可以用余么半群的余乘法来表示它：

$$\delta : E \rightarrow E \times E$$

`set/get` 律要求以下复合是恒等的：

$$E \xrightarrow{\delta} E \times E \xrightarrow{id \times p} E \times B \xrightarrow{q} E$$

这是该定律在丛中的图示：

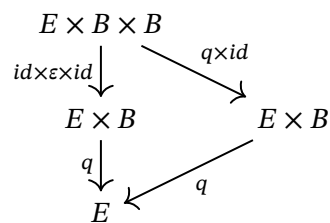


复合律

最后，这是 `set/set` 律，或称复合律。它表示”最后的设置胜出”：

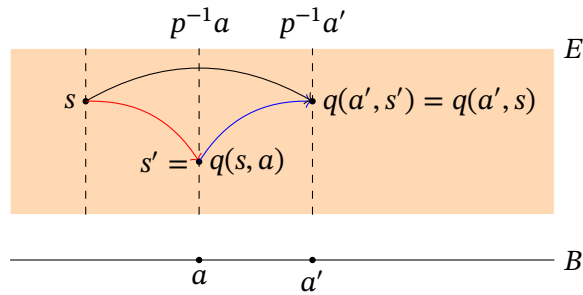
```
set (set s a) a' = set s a'
```

以及相应的交换图：



同样，为了去掉中间的 B ，我使用了余单位而不是积的投影。

这是 `set/set` 律在丛中的图示：

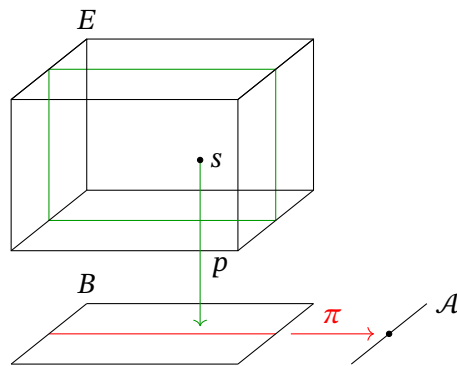


类型转换透镜

类型转换透镜将传输推广到在丛之间进行。我们必须定义一整个丛族。我们从范畴 \mathcal{A} 开始，其对象定义了我们用于透镜焦点的类型。

我们将集合 B 构造为所有焦点类型的所有元素的并集。 B 在 \mathcal{A} 上纤维化——投影 π 将 B 中的一个元素发送到其对应的类型。你可以将 B 视为余切片范畴 $1/\mathcal{A}$ 的对象集。

丛的丛 E 是一个在 B 上纤维化的集合，投影为 p 。由于 B 本身在 \mathcal{A} 上纤维化， E 传递地在 \mathcal{A} 上纤维化，复合投影为 $\pi \circ p$ 。正是这种较粗的纤维化将 E 分解为一系列丛。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 取一个元素 $s \in E$ 并生成一个元素 $b \in B$ ，其类型由 πb 给出。这是 `get` 的推广。

传输 q ，对应于 `set`，取一个元素 $s \in E$ 和一个元素 $b \in B$ 并生成一个新元素 $t \in E$ 。重要的观察是 s 和 t 可能属于 E 的不同子丛。

传输满足以下定律：

`get/set` 律（传输）：

$$p(q(b, s)) = b$$

set/get 律 (恒等):

$$q(p(s), s) = s$$

set/set 律 (复合):

$$q(c, q(b, s)) = q(c, s)$$

18.11 重要公式

这是一个方便的 (余) 端演算速查表。

- 同态函子的连续性:

$$\mathcal{D}\left(d, \int_a P\langle a, a \rangle\right) \cong \int_a \mathcal{D}(d, P\langle a, a \rangle)$$

- 同态函子的余连续性:

$$\mathcal{D}\left(\int_a P\langle a, a \rangle, d\right) \cong \int_a \mathcal{D}(P\langle a, a \rangle, d)$$

- 忍者米田引理:

$$\int_x \mathbf{Set}(\mathcal{C}(a, x), Fx) \cong Fa$$

- 忍者余米田引理:

$$\int^x \mathcal{C}(x, a) \times Fx \cong Fa$$

- 逆变函子 (预层) 的忍者米田引理:

$$\int_x \mathbf{Set}(\mathcal{C}(x, a), Gx) \cong Ga$$

- 逆变函子的忍者余米田引理:

$$\int^x \mathcal{C}(a, x) \times Gx \cong Ga$$

- Day 卷积:

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \otimes b, x) \times Fa \times Gb$$

Tambara 模

在编程中，范畴论的一个晦涩角落突然变得重要并不常见。Tambara 模在应用于 profunctor 光学 (profunctor optics) 时获得了新生。它们为光学组合问题提供了一个巧妙的解决方案。我们已经看到，在透镜 (lens) 的情况下，getter 可以使用函数组合很好地组合，但 setter 的组合涉及一些技巧。存在表示 (existential representation) 并没有多大帮助。另一方面，profunctor 表示使组合变得轻而易举。

这种情况有点类似于图形编程中几何变换的组合问题。例如，如果你尝试围绕两个不同的轴组合两个旋转，新轴和角度的公式会相当复杂。但如果你将旋转表示为矩阵，你可以使用矩阵乘法；或者，如果你将它们表示为四元数，你可以使用四元数乘法。Profunctor 表示允许你使用直接的函数组合来组合光学。

19.1 Tannakian 重构

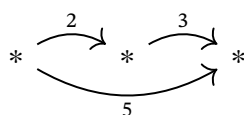
么半群及其表示

表示理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。我们将考虑么半群 (monoid) 而不是群。么半群可以定义为么半范畴 (monoidal category) 中的一个特殊对象，但它也可以被视为一个单对象范畴 \mathcal{M} 。我们称其对象为 $*$ ，唯一的 hom-集 $\mathcal{M}(*, *)$ 包含了我们需要的所有信息。

我们在么半群中称为“乘积”的东西被态射的组合所取代。根据范畴的定律，它是结合的和有单位的——恒等态射扮演了么半单位的角色。

从这个意义上说，每个单对象范畴自动是一个么半群，所有么半群都可以变成单对象范畴。

例如，带有加法的整数么半群可以被认为是一个具有单个抽象对象 $*$ 和每个数字的不同态射的范畴。要组合两个这样的态射，你可以将它们的数字相加，如下例所示：



对应于零的态射自动成为恒等态射。

我们可以将么半群表示为集合的变换。这样的表示由一个函子 $F: \mathcal{M} \rightarrow \mathbf{Set}$ 给出。它将单个对象 $*$ 映射到某个集合 S ，并将 \mathbf{hom} -集 $\mathcal{M}(*, *)$ 映射到一组函数 $S \rightarrow S$ 。根据函子定律，它将恒等映射到恒等，将组合映射到组合，因此它保留了么半群的结构。

如果函子是完全忠实的，它的图像包含与么半群完全相同的信息，没有更多。但一般来说，函子往往会“作弊”。 \mathbf{hom} -集 $\mathbf{Set}(S, S)$ 可能包含一些不在 $\mathcal{M}(*, *)$ 图像中的其他函数；并且 \mathcal{M} 中的多个态射可能被映射到单个函数。

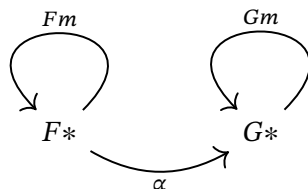
在极端情况下，整个 \mathbf{hom} -集 $\mathcal{M}(*, *)$ 可能被映射到恒等函数 id_S 。因此，仅通过查看集合 S ——函子 F 下 $*$ 的图像——我们无法梦想重建原始么半群。

不过，如果我们被允许同时查看给定么半群的所有表示，那么并非一切都失去了。这样的表示形成一个范畴——函子范畴 $[\mathcal{M}, \mathbf{Set}]$ ，也称为 \mathcal{M} 上的共预层范畴 (co-presheaf category)。这个范畴中的箭头是自然变换。

由于源范畴 \mathcal{M} 只包含一个对象，自然性条件采取特别简单的形式。自然变换 $\alpha: F \rightarrow G$ 只有一个分量，即函数 $\alpha: F* \rightarrow G*$ 。给定一个态射 $m: * \rightarrow *$ ，自然性方块如下：

$$\begin{array}{ccc} F* & \xrightarrow{\alpha} & G* \\ \downarrow Fm & & \downarrow Gm \\ F* & \xrightarrow{\alpha} & G* \end{array}$$

这是作用于两个集合的三个函数之间的关系：



自然性条件告诉我们：

$$\alpha \circ (Fm) = (Gm) \circ \alpha$$

换句话说，如果你在集合 $F*$ 中选取任何元素 x ，你可以使用 α 将其映射到 $G*$ ，然后应用对应于 m 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 m 在集合 $F*$ 上的作用。等变函数通过预组合或后组合将一个集合上的作用连接到另一个集合上的相应作用。

Cayley 定理

在群论中，Cayley 定理指出，每个群都同构于一个（子群的）置换群。一个群 (group) 就是一个么半群，其中每个元素 g 都有一个逆元 g^{-1} 。置换是双射函数，将一个集合映射到自身。它们置换 (permute) 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。

单对象解释与更传统的“元素集合”解释之间的关联很容易建立。我们通过构造函数 $F: \mathcal{M} \rightarrow \mathbf{Set}$ 来实现这一点，该函子将 $*$ 映射到特殊的集合 S ，该集合等于同态集： $S = \mathcal{M}(*, *)$ 。该集合的元素与 \mathcal{M} 中的态射一一对应。我们将 F 在态射上的作用定义为后复合：

$$(Fm)n = m \circ n$$

这里 m 是 \mathcal{M} 中的一个态射， n 是 S 中的一个元素，而 S 中的元素恰好也是 \mathcal{M} 中的态射。

我们可以将这种特定的表示视为幺半群在幺半范畴 \mathbf{Set} 中的另一种定义。我们只需要实现单位元和乘法：

$$\eta: 1 \rightarrow S$$

$$\mu: S \times S \rightarrow S$$

单位元选择 S 中对应于 $\mathcal{M}(*, *)$ 中 id_* 的元素。两个元素 m 和 n 的乘法由对应于 $m \circ n$ 的元素给出。

同时，我们可以将 S 视为 $F: \mathcal{M} \rightarrow \mathbf{Set}$ 的像，在这种情况下，形成幺半群表示的是函数 $S \rightarrow S$ 。这就是 Cayley 定理的本质：每个幺半群都可以由一组自函数表示。

在编程中，应用 Cayley 定理的最佳例子是列表反转的高效实现。回想一下反转的朴素递归实现：

```
reverse :: [a] -> [a]
reverse [] = []
reverse (a : as) = reverse as ++ [a]
```

它将列表分为头部和尾部，反转尾部，并将头部构成的单例列表附加到结果中。问题在于每次附加操作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），将该列表 `as` 前置到其参数 `xs` 中：

```
rep :: [a] -> DList a
rep as = \xs -> as ++ xs
```

这种表示称为差异列表（difference list）。

要将函数转换回列表，只需将其应用于空列表：

```
unRep :: DList a -> [a]
unRep f = f []
```

很容易验证空列表的表示是恒等函数，而两个列表连接的表示是表示的复合：

```
rep [] = id
rep (xs ++ ys) = rep xs . rep ys
```

因此，这正是列表么半群的 Cayley 表示。

我们现在可以将反转算法转换为生成这种新表示：

```
rev :: [a] -> DList a
rev [] = rep []
rev (a : as) = rev as . rep [a]
```

并将其转换回列表：

```
fastReverse :: [a] -> [a]
fastReverse = unRep . rev
```

乍一看，似乎我们除了在递归算法之上添加了一层转换之外，并没有做太多事情。然而，新算法的性能是 $O(N)$ 而不是 $O(N^2)$ 。为了理解这一点，考虑反转一个简单的列表 `[1, 2, 3]`。函数 `rev` 将这个列表转换为函数的复合：

```
rep [3] . rep [2] . rep [1]
```

它以线性时间完成。函数 `unRep` 执行这个复合作用于空列表。但请注意，每个 `rep` 都将其参数前置到累积结果中。特别是，最后的 `rep [3]` 执行：

```
[3] ++ [2, 1]
```

与附加不同，前置是一个常数时间操作，因此整个算法需要 $O(N)$ 时间。

另一种理解方式是意识到 `rev` 按照列表元素的顺序从头部开始将操作排队。但函数的队列是按照先进先出（FIFO）的顺序执行的。

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）。然后根据需要以 FIFO 顺序执行它们。

么半群的 Tannakian 重构

我们需要多少信息才能从么半群的表示中重构出该么半群？仅仅观察集合肯定是不够的，因为任何么半群都可以在任何集合上表示。但如果我们包含这些集合之间的结构保持函数，我们或许有机会。

范畴论的通常方法是着眼于全局，而不是专注于个别情况。因此，我们不是专注于单个函子，而是考虑给定么半群 \mathcal{M} 的所有表示的总和——函子范畴 $[\mathcal{M}, \mathbf{Set}]$ 。

我们可以定义一个称为纤维函子的遗忘函子， $\text{fib} : [\mathcal{M}, \mathbf{Set}] \rightarrow \mathbf{Set}$ 。它在对象（这里是函子）上的作用提取出底层集合：

$$\text{fib } F = F^*$$

它在箭头（这里是自然变换）上的作用产生相应的等变函数。作用于 $\alpha : F \rightarrow G$ 时，我们得到：

$$\text{fib } \alpha : F^* \rightarrow G^*$$

函子 fib 是函子范畴 $[[\mathcal{M}, \mathbf{Set}], \mathbf{Set}]$ 的一个成员。在这个范畴中，箭头是自然变换，它们是由函子参数化的 \mathbf{Set} 中的函数族。对于 Tannakian 重构，我们将专注于从 fib 到其自身的自然变换的集合。我们可以将其写为函子范畴 $[\mathcal{M}, \mathbf{Set}]$ 上的一个端：

$$\int_F \mathbf{Set}(\text{fib } F, \text{fib } F)$$

或者，展开 fib 的定义：

$$\int_F \mathbf{Set}(F^*, F^*)$$

以下是一些细节。端下的 *profunctor* 是一个形式如下的函子：

$$P : [\mathcal{M}, \mathbf{Set}]^{op} \times [\mathcal{M}, \mathbf{Set}] \rightarrow \mathbf{Set}$$

它在对象（从 \mathcal{M} 到 \mathbf{Set} 的函子对）上的作用是：

$$P\langle G, H \rangle = \mathbf{Set}(G^*, H^*)$$

让我们定义它在态射（即自然变换对）上的作用。给定任何一对：

$$\alpha: G' \rightarrow G$$

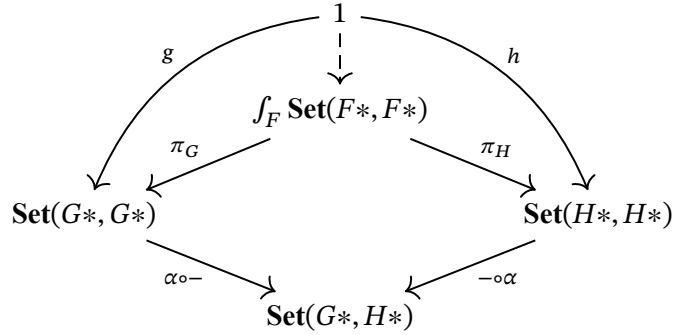
$$\beta: H \rightarrow H'$$

它们的提升是一个函数：

$$P\langle\alpha, \beta\rangle: \mathbf{Set}(G^*, H^*) \rightarrow \mathbf{Set}(G'^*, H'^*)$$

这是通过预组合 α 和后组合 β 来实现的。

端是一个巨大的积，即从同态集 $\mathbf{Set}(F^*, F^*)$ 中选取的函数元组，每个函子对应一个。这些元组进一步受到楔形条件的约束。我们可以通过实例化单例集（终对象）的普遍条件来提取这个约束。这里我们选取 g 作为 $\mathbf{Set}(G^*, G^*)$ 的一个元素， h 作为 $\mathbf{Set}(H^*, H^*)$ 的一个元素，



我们得到以下条件：

$$\alpha \circ g = h \circ \alpha$$

对于任何满足条件的等变 $\alpha: G^* \rightarrow H^*$ ：

$$\alpha \circ (Gm) = (Hm) \circ \alpha$$

在这种情况下，Tannakian 重构定理告诉我们：

$$\int_F \mathbf{Set}(F^*, F^*) \cong \mathcal{M}(*, *)$$

换句话说，我们可以从其表示中恢复么半群。

我们将在更一般的陈述的背景下看到这个定理的证明，但这里是一般思路。在所有可能的表示中，有一个是满且忠实的。它的底层集合是同态集 $\mathcal{M}(*, *)$ ，么半群的作用是后组合 ($m \circ -$)。唯一满足楔形条件的从这个集合到自身的函数是么半群作用本身，它们与 $\mathcal{M}(*, *)$ 的元素一一对应。

Tannakian 重构的证明

么半群重构是一个更一般定理的特殊情况，其中我们使用一个常规范畴（regular category）而不是单对象范畴。与么半群的情况类似，我们将重构同态集（hom-set），只是这次它将是两个对象之间的常规同态集。我们将证明以下公式：

$$\int_{F: [\mathcal{C}, \mathbf{Set}]} \mathbf{Set}(Fa, Fb) \cong \mathcal{C}(a, b)$$

技巧在于使用 Yoneda 引理来表示 F 的作用：

$$Fa \cong [\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), F)$$

对于 Fb 也是如此。我们得到：

$$\int_{F: [\mathcal{C}, \mathbf{Set}]} \mathbf{Set}([\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), F), [\mathcal{C}, \mathbf{Set}](\mathcal{C}(b, -), F))$$

注意，这里的两个自然变换集是 $[\mathcal{C}, \mathbf{Set}]$ 中的同态集。

回忆一下 Yoneda 引理的推论，它适用于任何范畴 \mathcal{A} ：

$$[\mathcal{A}, \mathbf{Set}](\mathcal{A}(x, -), \mathcal{A}(y, -)) \cong \mathcal{A}(y, x)$$

我们可以用端（end）来表示它：

$$\int_{z: \mathcal{C}} \mathbf{Set}(\mathcal{A}(x, z), \mathcal{A}(y, z)) \cong \mathcal{A}(y, x)$$

特别地，我们可以将 \mathcal{A} 替换为函子范畴 $[\mathcal{C}, \mathbf{Set}]$ 。我们得到：

$$\int_{F: [\mathcal{C}, \mathbf{Set}]} \mathbf{Set}([\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), F), [\mathcal{C}, \mathbf{Set}](\mathcal{C}(b, -), F)) \cong [\mathcal{C}, \mathbf{Set}](\mathcal{C}(b, -), \mathcal{C}(a, -))$$

然后我们可以再次对右边应用 Yoneda 引理，得到：

$$\mathcal{C}(a, b)$$

这正是我们想要的结果。

重要的是要理解函子范畴的结构如何通过楔条件（wedge condition）进入端。它通过自然变换来实现这一点。每当我们有两个函子之间的自然变换 $\alpha: G \rightarrow H$ 时，以下图表必须交换：

$$\begin{array}{ccc} & \int_F \mathbf{Set}(Fa, Fb) & \\ \pi_G \swarrow & & \searrow \pi_H \\ \mathbf{Set}(Ga, Gb) & & \mathbf{Set}(Ha, Hb) \\ \searrow \mathbf{Set}(id, \alpha) & & \swarrow \mathbf{Set}(\alpha, id) \\ & \mathbf{Set}(Ga, Hb) & \end{array}$$

为了获得一些关于 Tannakian 重构的直觉，你可以回忆一下 **Set** 值函子可以解释为证明相关子集 (proof-relevant subset)。一个函子 $F: \mathcal{C} \rightarrow \mathbf{Set}$ (一个共预层, co-presheaf) 定义了 (一个小范畴) \mathcal{C} 的对象的一个子集。我们说一个对象 a 在该子集中，当且仅当 Fa 非空。 Fa 的每个元素都可以解释为这一点的证明。

但是，除非所讨论的范畴是离散的，否则并非所有子集都对应于共预层。特别是，每当有一个箭头 $f: a \rightarrow b$ 时，也存在一个函数 $Ff: Fa \rightarrow Fb$ 。根据我们的解释，这样的函数将每个证明 a 在由 F 定义的子集中的证明映射到 b 在该子集中的证明。因此，共预层定义了与范畴结构兼容的证明相关子集。

让我们以同样的精神重新解释 Tannakian 重构。

$$\int_{F: [\mathcal{C}, \mathbf{Set}]} \mathbf{Set}(Fa, Fb) \cong \mathcal{C}(a, b)$$

左边的一个元素是一个证明，表明对于每个与 \mathcal{C} 结构兼容的子集，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 的箭头时才可能。

Exercise 19.1.1. 将 Tannakian 重构的证明应用到上一节中的单对象范畴 \mathcal{M} 。

Haskell 中的 Tannakian 重构

我们可以立即将这个结果翻译到 Haskell 中。我们将 `end` 替换为 `forall`。左侧变为：

```
forall f. Functor f => f a -> f b
```

而右侧是函数类型 `a->b`。

我们之前见过多态函数：它们是针对所有类型定义的函数，有时是针对某些类型类定义的。这里我们有一个针对所有函子定义的函数。它表示：给我一个包含 `a` 的函子，我将生成一个包含 `b` 的函子——无论你使用什么函子。实现这一点的唯一方法（使用参数多态）是，如果这个函数已经秘密捕获了一个类型为 `a->b` 的函数，并使用 `fmap` 应用它。

事实上，同构的一个方向正是这样：捕获一个函数并在参数上使用 `fmap`：

```
toTannaka :: (a -> b) -> (forall f. Functor f => f a -> f b)
toTannaka g fa = fmap g fa
```

另一个方向使用 Yoneda 技巧：

```
fromTannaka :: (forall f. Functor f => f a -> f b) -> (a -> b)
fromTannaka g a = runIdentity (g (Identity a))
```

其中恒等函子定义为：

```
data Identity a = Identity a

runIdentity :: Identity a -> a
runIdentity (Identity a) = a

instance Functor Identity where
  fmap g (Identity a) = Identity (g a)
```

这种重构可能看起来微不足道且毫无意义。为什么有人会想要用更复杂的类型替换函数类型 $a \rightarrow b$ ：

```
type Getter a b = forall f. Functor f => f a -> f b
```

不过，将 $a \rightarrow b$ 视为所有光学（optics）的前身是有启发性的。它是一个聚焦于 a 的 b 部分的透镜。它告诉我们， a 以某种形式包含了足够的信息来构造一个 b 。它是一个“获取器”或“访问器”。

显然，函数可以组合。有趣的是，函子表示也可以组合，并且它们使用简单的函数组合进行组合，如下示例所示：

```
boolToStrGetter :: Getter Bool String
boolToStrGetter = toTannaka (show) . toTannaka (bool (-1) 1)
```

其中：

```
bool :: a -> a -> Bool -> a
bool f _ False = f
bool _ t True  = t
```

其他光学（optics）并不那么容易组合，但它们的函子（和 profunctor）表示可以。

伴随下的 Tannakian 重构

推广 Tannakian 重构的技巧在于在某些专门的函子范畴 \mathcal{T} 上定义 end（我们稍后会将其应用于 Tambara 范畴）。

假设我们在两个函子范畴 \mathcal{T} 和 $[\mathcal{C}, \mathbf{Set}]$ 之间有自由/遗忘伴随 $F \dashv U$ ：

$$\mathcal{T}(FQ, P) \cong [\mathcal{C}, \mathbf{Set}](Q, UP)$$

其中 Q 是 $[\mathcal{C}, \mathbf{Set}]$ 中的函子， P 是 \mathcal{T} 中的函子。

我们 Tannakian 重构的起点是以下 end:

$$\int_{P: \mathcal{T}} \mathbf{Set}((UP)a, (UP)s)$$

由对象 a 参数化的映射 $\mathcal{T} \rightarrow \mathbf{Set}$, 由公式给出:

$$P \mapsto (UP)a$$

这是纤维函子, 因此 end 公式可以解释为两个纤维函子之间的自然变换集合。

从概念上讲, 纤维函子描述了一个对象的“无穷小邻域”。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。这些函数探测了对象所处的环境。特别是, \mathcal{T} 中的自然变换参与了定义 end 的楔条件。(在微积分中, 层的茎起着非常相似的作用。)

正如我们之前所做的那样, 我们首先将 Yoneda 引理应用于协预层 (UP) :

$$\int_{P: \mathcal{T}} \mathbf{Set}([\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), UP), [\mathcal{C}, \mathbf{Set}](\mathcal{C}(s, -), UP))$$

我们现在可以使用伴随:

$$\int_{P: \mathcal{T}} \mathbf{Set}(\mathcal{T}(FC(a, -), P), \mathcal{T}(FC(s, -), P))$$

我们最终得到了函子范畴 \mathcal{T} 中两个自然变换之间的映射。我们可以使用 Yoneda 引理的推论来执行“积分”, 得到:

$$\mathcal{T}(FC(s, -), FC(a, -))$$

我们可以再次应用伴随:

$$\mathbf{Set}(\mathcal{C}(s, -), (U \circ F)\mathcal{C}(a, -))$$

并再次应用 Yoneda 引理:

$$((U \circ F)\mathcal{C}(a, -))s$$

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单子。让我们称这个单子为 Φ 。结果是一个恒等式, 它将作为 profunctor 光学的基础:

$$\int_{P: \mathcal{T}} \mathbf{Set}((UP)a, (UP)s) \cong (\Phi\mathcal{C}(a, -))s$$

右边是单子 $\Phi = U \circ F$ 在可表函子 $\mathcal{C}(a, -)$ 上的作用, 然后在 s 处求值。使用 Yoneda 函子符号, 这可以写成 $(\Phi\mathcal{Y}^a)s$ 。

将其与之前的 Tannakian 重构公式进行比较, 特别是如果我们将其重写为以下形式:

$$\int_{F: [\mathcal{C}, \mathbf{Set}]} \mathbf{Set}(Fa, Fs) \cong \mathcal{C}(a, -)s$$

请记住, 在推导光学时, 我们将用 $\mathcal{C}^{op} \times \mathcal{C}$ 中的对象对 $\langle a, b \rangle$ 和 $\langle s, t \rangle$ 替换 a 和 s 。我们的函子将变成 profunctor。

19.2 Profunctor 透镜

我们的目标是找到光学的函子表示。我们之前看到，例如，类型变化的透镜可以看作是 **Lens** 范畴中的 **hom** 集。**Lens** 中的对象是来自某个笛卡尔范畴 \mathcal{C} 的对象对，从一个这样的对 $\langle s, t \rangle$ 到另一个 $\langle a, b \rangle$ 的 **hom** 集由 **coend** 公式给出：

$$\mathcal{L}\langle s, t \rangle \langle a, b \rangle = \int^{\mathcal{C}} \mathcal{C}(s, c \times a) \times \mathcal{C}(c \times b, t)$$

注意，这个公式中的 **hom** 集对可以看作是积范畴 $\mathcal{C}^{op} \times \mathcal{C}$ 中的单个 **hom** 集：

$$\mathcal{L}\langle s, t \rangle \langle a, b \rangle = \int^{\mathcal{C}} (\mathcal{C}^{op} \times \mathcal{C})(c \bullet \langle a, b \rangle, \langle s, t \rangle)$$

其中我们定义 c 在 $\langle a, b \rangle$ 上的作用为：

$$c \bullet \langle a, b \rangle = \langle c \times a, c \times b \rangle$$

这是 $\mathcal{C}^{op} \times \mathcal{C}$ 在自身上的更一般作用的对角线部分的简写符号，由下式给出：

$$\langle c, c' \rangle \bullet \langle a, b \rangle = \langle c \times a, c' \times b \rangle$$

这表明，为了表示这样的光学，我们应该考虑在范畴 $\mathcal{C}^{op} \times \mathcal{C}$ 上的协预层，即我们应该考虑 **profunctor** 表示。

同构 (Iso)

作为这个想法的快速测试，让我们将重建公式应用于简单情况 $\mathcal{T} = [\mathcal{C}^{op} \times \mathcal{C}, \mathbf{Set}]$ ，其中没有额外的结构。在这种情况下，我们不需要使用遗忘函子 (forgetful functors) 或单子 (monad) Φ ，我们只需直接应用 Tannakian 重建：

$$\mathcal{O}\langle s, t \rangle \langle a, b \rangle = \int_{P: \mathcal{T}} \mathbf{Set}(P\langle a, b \rangle, P\langle s, t \rangle) \cong ((\mathcal{C}^{op} \times \mathcal{C})(\langle a, b \rangle, -))\langle s, t \rangle$$

右侧的表达式为：

$$(\mathcal{C}^{op} \times \mathcal{C})(\langle a, b \rangle, \langle s, t \rangle) = \mathcal{C}(s, a) \times \mathcal{C}(b, t)$$

这种光学结构在 Haskell 中被称为 **Iso**（或适配器）：

```
type Iso s t a b = (s -> a, b -> t)
```

并且它确实有一个对应于以下端 (end) 的 **profunctor** 表示：

```
type IsoP s t a b = forall p. Profunctor p => p a b -> p s t
```

给定一对函数，很容易构造这个 **profunctor** 多态函数：

```
toIsoP :: (s -> a, b -> t) -> IsoP s t a b
toIsoP (f, g) = dimap f g
```

这简单地说明，任何 **profunctor** 都可以用来提升一对函数。

相反，我们可以提出一个问题：一个单一的多态函数如何将集合 $P(a, b)$ 映射到集合 $P(s, t)$ ，对于每一个可以想象的 **profunctor**？这个函数对 **profunctor** 的唯一了解是它可以提升一对函数。因此，它必须是一个闭包，要么包含要么能够生成一对函数 $(s \rightarrow a, b \rightarrow t)$ 。

Exercise 19.2.1. 实现以下函数：

```
fromIsoP :: IsoP s t a b -> (s -> a, b -> t)
```

提示：证明这是一个 *profunctor*：

```
newtype Adapter a b s t = Ad (s -> a, b -> t)
```

并使用一对恒等函数构造 `Adapter a a s s`。

Profunctors 和 lenses (Profunctors 和透镜)

让我们尝试将相同的逻辑应用于透镜 (lenses)。我们需要找到一类 **profunctors** 来插入到我们的 **profunctor** 表示中。假设遗忘函子 (forgetful functor) U 仅遗忘额外的结构但不改变集合，因此集合 $P(a, b)$ 与集合 $(UP)(a, b)$ 相同。

让我们从存在表示 (existential representation) 开始。我们有一个对象 c 和一对函数：

$$\langle f, g \rangle : \mathcal{C}(s, c \times a) \times \mathcal{C}(c \times b, t)$$

我们希望构建一个 **profunctor** 表示，因此我们需要能够将集合 $P(a, b)$ 映射到集合 $P(s, t)$ 。我们可以通过提升这两个函数来得到 $P(s, t)$ ，但前提是从 $P(c \times a, c \times b)$ 开始。实际上：

$$P\langle f, g \rangle : P\langle c \times a, c \times b \rangle \rightarrow P\langle s, t \rangle$$

我们所缺少的是映射：

$$P\langle a, b \rangle \rightarrow P\langle c \times a, c \times b \rangle$$

而这正是我们需要从我们的 **profunctor** 类中要求的额外结构。

Tambara 模

配备以下变换族的函子 P 被称为 *Tambara 模*：

$$\alpha_{\langle a, b \rangle, c} : P\langle a, b \rangle \rightarrow P\langle c \times a, c \times b \rangle$$

我们希望这个变换族在 a 和 b 上是自然的，但对于 c 应该要求什么？ c 的问题在于它出现了两次，一次在逆变位置，一次在协变位置。因此，如果我们希望与像 $h : c \rightarrow c'$ 这样的箭头良好地交互，我们必须修改自然性条件。我们可以考虑一个更一般的函子 $P\langle c' \times a, c \times b \rangle$ ，并将 α 视为生成其对角线元素，即 c' 与 c 相同的元素。

如果对于任意 $f : c \rightarrow c'$ ，以下图表交换，则称两个函子 P 和 Q 的对角部分之间的变换 α 为 *对角自然变换* (*diagonally natural*)：

$$\begin{array}{ccccc}
 & & P\langle c', c \rangle & & \\
 & \swarrow P\langle f, c \rangle & & \searrow P\langle c', f \rangle & \\
 P\langle c, c \rangle & & & & P\langle c', c' \rangle \\
 \downarrow \alpha_c & & & & \downarrow \alpha_{c'} \\
 Q\langle c, c \rangle & & & & Q\langle c', c' \rangle \\
 & \searrow P\langle c, f \rangle & & \swarrow P\langle f, c \rangle & \\
 & & Q\langle c, c' \rangle & &
 \end{array}$$

(我使用了常见的简写 $P\langle f, c \rangle$ ，类似于 whiskering，表示 $P\langle f, id_c \rangle$ 。)

在我们的情况下，对角自然性条件简化为：

$$\begin{array}{ccccc}
 & & P\langle a, b \rangle & & \\
 & \swarrow \alpha_{\langle a, b \rangle, c} & & \searrow \alpha_{\langle a, b \rangle, c'} & \\
 P\langle c \times a, c \times b \rangle & & & & P\langle c' \times a, c' \times b \rangle \\
 & \searrow P\langle c \times a, f \times b \rangle & & \swarrow P\langle f \times b, c \times b \rangle & \\
 & & P\langle c \times a, c' \times b \rangle & &
 \end{array}$$

(这里， $P\langle f \times b, c \times b \rangle$ 表示 $P\langle f \times id_b, id_{c \times b} \rangle$ 。)

Tambara 模还有一个一致性条件：它们必须保持幺半结构。在笛卡尔范畴中，乘以 c 的操作是有意义的：我们必须为任何对象对提供一个积，并且我们希望有一个终端对象作为乘法的单位。Tambara 模必须尊重单位并保持乘法。对于单位（终端对象），我们施加以下条件：

$$\alpha_{\langle a, b \rangle, 1} = id_{P\langle a, b \rangle}$$

对于乘法，我们有：

$$\alpha_{\langle a, b \rangle, c' \times c} \cong \alpha_{\langle c \times a, c \times b \rangle, c'} \circ \alpha_{\langle a, b \rangle, c}$$

或者，图示为：

$$\begin{array}{ccc}
 P\langle a, b \rangle & \xrightarrow{\alpha_{\langle a, b \rangle, c' \times c}} & P\langle c' \times c \times a, c' \times c \times b \rangle \\
 & \searrow \alpha_{\langle a, b \rangle, c} & \nearrow \alpha_{\langle c \times a, c \times b \rangle, c'} \\
 & P\langle c \times a, c \times b \rangle &
 \end{array}$$

(注意，积在同构意义下是结合的，因此图中有一个隐藏的结合子。)

由于我们希望 Tambara 模形成一个范畴，我们必须定义它们之间的态射。这些是保持额外结构的自然变换。假设我们有两个 Tambara 模之间的自然变换 ρ ，即 $\rho: (P, \alpha) \rightarrow (Q, \beta)$ 。我们可以先应用 α 再应用 ρ ，或者先应用 ρ 再应用 β 。我们希望结果相同：

$$\begin{array}{ccc}
 P\langle a, b \rangle & \xrightarrow{\alpha_{\langle a, b \rangle, c}} & P\langle c \times a, c \times b \rangle \\
 \rho_{\langle a, b \rangle} \downarrow & & \downarrow \rho_{\langle c \times a, c \times b \rangle} \\
 Q\langle a, b \rangle & \xrightarrow{\beta_{\langle a, b \rangle, c}} & Q\langle c \times a, c \times b \rangle
 \end{array}$$

请记住，Tambara 范畴的结构编码在这些自然变换中。它们将通过楔条件确定进入函子透镜定义的端的状态。

Profunctor 透镜

现在我们对 Tambara 模与透镜的关系有了一些直观理解，让我们回到主要公式：

$$\mathcal{L}\langle s, t \rangle \langle a, b \rangle = \int_{P: \mathcal{T}} \mathbf{Set}((UP)\langle a, b \rangle, (UP)\langle s, t \rangle) \cong (\Phi(\mathcal{C}^{op} \times \mathcal{C})(\langle a, b \rangle, -))\langle s, t \rangle$$

这次我们是在 Tambara 范畴上取 end。唯一缺失的部分是单子 $\Phi = U \circ F$ 或自由生成 Tambara 模的函子 F 。

事实证明，与其猜测单子，不如猜测余单子更容易。在 profunctor 范畴中存在一个余单子，它取一个 profunctor P 并生成另一个 profunctor ΘP 。公式如下：

$$(\Theta P)\langle a, b \rangle = \int_c P\langle c \times a, c \times b \rangle$$

你可以通过实现 ε 和 δ (**extract** 和 **duplicate**) 来验证这确实是一个余单子。例如， ε 使用终端对象（笛卡尔积的单位）的投影 π_1 将 $\Theta P \rightarrow P$ 映射。

这个余单子的有趣之处在于它的余代数是 Tambara 模。同样，这些是将 profunctor 映射到 profunctor 的余代数。它们是自然变换 $P \rightarrow \Theta P$ 。我们可以将这样的自然变换写为 end 的一个元素：

$$\int_{a, b} \mathbf{Set}(P\langle a, b \rangle, (\Theta P)\langle a, b \rangle) = \int_{a, b} \int_c \mathbf{Set}(P\langle a, b \rangle, P\langle c \times a, c \times b \rangle)$$

我使用了 hom-函子的连续性来取出 c 上的 end。得到的 end 编码了一组自然（在 c 上是双自然的）变换，这些变换定义了一个 Tambara 模：

$$\alpha_{\langle a, b \rangle, c}: P\langle a, b \rangle \rightarrow P\langle c \times a, c \times b \rangle$$

事实上，这些余代数是余单子余代数，即它们与余单子 Θ 兼容。换句话说，Tambara 模形成了余单子 Θ 的 Eilenberg-Moore 余代数范畴。

Θ 的左伴随是一个单子 Φ ，其公式为：

$$(\Phi P)\langle s, t \rangle = \int^{u, v, c} (\mathcal{C}^{op} \times \mathcal{C})(c \bullet \langle u, v \rangle, \langle s, t \rangle) \times P\langle u, v \rangle$$

其中我使用了简写符号：

$$(\mathcal{C}^{op} \times \mathcal{C})(c \bullet \langle u, v \rangle, \langle s, t \rangle) = \mathcal{C}(s, c \times u) \times \mathcal{C}(c \times v, t)$$

这个伴随关系可以通过一些 `end/coend` 操作轻松验证：从 ΦP 到某个 profunctor Q 的映射可以写为一个 `end`。然后可以使用 `hom`-函子的共连续性取出 Φ 中的 `coend`。最后，应用 `ninja-Yoneda` 引理生成到 ΘQ 的映射。我们得到：

$$[(\mathcal{C}^{op} \times \mathcal{C}, \mathbf{Set})(P\Phi, Q) \cong [(\mathcal{C}^{op} \times \mathcal{C}, \mathbf{Set})(P, \Theta Q)$$

将 Q 替换为 P ，我们立即看到 Φ 的代数的集合与 Θ 的余代数的集合同构。事实上，它们是 Φ 的单子代数。这意味着单子 Φ 的 Eilenberg-Moore 范畴与 Tambara 范畴相同。

回想一下，Eilenberg-Moore 构造将单子分解为自由/遗忘伴随。这正是我们在推导 profunctor 光学公式时寻找的伴随关系。

剩下的就是评估 Φ 在可表函子上的作用：

$$(\Phi(\mathcal{C}^{op} \times \mathcal{C})(\langle a, b \rangle, -))\langle s, t \rangle = \int^{u, v, c} (\mathcal{C}^{op} \times \mathcal{C})(c \bullet \langle u, v \rangle, \langle s, t \rangle) \times (\mathcal{C}^{op} \times \mathcal{C})(\langle a, b \rangle, \langle u, v \rangle)$$

应用 `co-Yoneda` 引理，我们得到：

$$\int^c (\mathcal{C}^{op} \times \mathcal{C})(c \bullet \langle a, b \rangle, \langle s, t \rangle) = \int^c \mathcal{C}(s, c \times a) \times \mathcal{C}(c \times b, t)$$

这正是透镜的存在表示。

Haskell 中的 Profunctor 透镜

为了在 Haskell 中定义 profunctor 表示，我们引入了一类关于笛卡尔积的 Tambara 模块的 profunctor（稍后我们会看到更一般的 Tambara 模块）。在 Haskell 库中，这个类被称为 **Strong**。它在文献中也以 **Cartesian** 出现：

```
class Profunctor p => Cartesian p where
  alpha :: p a b -> p (c, a) (c, b)
```

多态函数 `alpha` 具有由参数多态性保证的所有相关自然性属性。

`profunctor` 透镜只是对在 `Cartesian` `profunctor` 中多态的函数类型的类型同义词：

```
type LensP s t a b = forall p. Cartesian p => p a b -> p s t
```

实现这样的函数的最简单方法是从透镜的存在表示开始，并将 `alpha` 和 `dimap` 应用于 `profunctor` 参数：

```
toLensP :: LensE s t a b -> LensP s t a b
toLensP (LensE from to) = dimap from to . alpha
```

因为 `profunctor` 透镜只是函数，所以我们可以像这样组合它们：

```
lens1 :: LensP s t x y
-- p s t -> p x y
lens2 :: LensP x y a b
-- p x y -> p a b
lens3 :: LensP s t a b
-- p s t -> p a b
lens3 = lens2 . lens1
```

从 `profunctor` 表示到透镜的 `get/set` 表示的逆向映射也是可能的。为此，我们需要猜测可以输入到 `LensP` 中的 `profunctor`。事实证明，当我们固定类型对 `a` 和 `b` 时，透镜的 `get/set` 表示就是这样的 `profunctor`。我们定义：

```
data FlipLens a b s t = FlipLens (s -> a) (s -> b -> t)
```

很容易证明它确实是一个 `profunctor`：

```
instance Profunctor (FlipLens a b) where
  dimap f g (FlipLens get set) = FlipLens (get . f) (fmap g . set . f)
```

不仅如此，它还是一个 `Cartesian` `profunctor`：

```
instance Cartesian (FlipLens a b) where
  alpha(FlipLens get set) = FlipLens get' set'
  where get' = get . snd
        set' = \ (x, s) b -> (x, set s b)
```

我们现在可以用一个简单的 `getter` 和 `setter` 对初始化 `FlipLens`，并将其输入到我们的 `profunctor` 表示中：

```

fromLensP :: LensP s t a b -> (s -> a, s -> b -> t)
fromLensP pp = (get', set')
  where FlipLens get' set' = pp (FlipLens id (\s b -> b))

```

19.3 一般光学

Tambara 模最初是为任意具有张量积 \otimes 和单位对象 I 的么半范畴定义的¹。它们的结构映射具有以下形式：

$$\alpha_{\langle a, b \rangle, c} : P\langle a, b \rangle \rightarrow P\langle c \otimes a, c \otimes b \rangle$$

你可以很容易地验证，所有的相干律都直接适用于这种情况，并且 profunctor 光学的推导过程无需改变。

棱镜

从编程的角度来看，有两个明显的么半结构值得探索：积和和。我们已经看到积产生了透镜（lenses）。和，或者说余积，则产生了棱镜（prisms）。

我们只需在透镜的定义中将积替换为和，就可以得到棱镜的存在表示：

$$\mathcal{P}\langle s, t \rangle\langle a, b \rangle = \int^c \mathcal{C}(s, c + a) \times \mathcal{C}(c + b, t)$$

为了简化这个表达式，注意到从和映射出去等价于映射的积：

$$\int^c \mathcal{C}(s, c + a) \times \mathcal{C}(c + b, t) \cong \int^c \mathcal{C}(s, c + a) \times \mathcal{C}(c, t) \times \mathcal{C}(b, t)$$

利用 co-Yoneda 引理，我们可以去掉 coend 得到：

$$\mathcal{C}(s, t + a) \times \mathcal{C}(b, t)$$

在 Haskell 中，这定义了一对函数：

```

match :: s -> Either t a
build :: b -> t

```

为了理解这一点，让我们首先翻译棱镜的存在形式：

```

data Prism s t a b where
  Prism :: (s -> Either c a) -> (Either c b -> t) -> Prism s t a b

```

¹事实上，Tambara 模最初是为在向量空间上富集的范畴定义的

这里 `s` 要么包含焦点 `a`，要么包含余项 `c`。相反，`t` 可以从新的焦点 `b` 或余项 `c` 构建。

这种逻辑反映在转换函数中：

```
toMatch :: Prism s t a b -> (s -> Either t a)
toMatch (Prism from to) s =
  case from s of
    Left  c -> Left  (to (Left  c))
    Right a -> Right a
```

```
toBuild :: Prism s t a b -> (b -> t)
toBuild (Prism from to) b = to (Right b)
```

```
toPrism :: (s -> Either t a) -> (b -> t) -> Prism s t a b
toPrism match build = Prism from to
  where
    from = match
    to (Left  c) = c
    to (Right b) = build b
```

棱镜的 `profunctor` 表示与透镜的几乎相同，只是将积替换为和。

在 Haskell 库中，和类型的 Tambara 模类称为 `Choice`，在文献中称为 `Cocartesian`：

```
class Profunctor p => Cocartesian p where
  alpha' :: p a b -> p (Either c a) (Either c b)
```

`profunctor` 表示是一个多态函数类型：

```
type PrismP s t a b = forall p. Cocartesian p => p a b -> p s t
```

从存在棱镜的转换与透镜的转换几乎相同：

```
toPrismP :: Prism s t a b -> PrismP s t a b
toPrismP (Prism from to) = dimap from to . alpha'
```

同样，`profunctor` 棱镜使用函数组合进行组合。

遍历

遍历 (traversal) 是一种可以同时聚焦多个焦点的光学。例如，想象你有一棵树，它可以有零个或多个类型为 a 的叶子。遍历应该能够获取这些节点的列表。它还应该允许你用一个新的列表替换这些节点。这里的问题是：提供替换的列表的长度必须与节点的数量匹配，否则会发生不好的事情。

类型安全的遍历实现需要我们跟踪列表的大小。换句话说，它需要依赖类型。

在 Haskell 中，(非类型安全的) 遍历通常写为：

```
type Traversal s t a b = s -> ([b] -> t, [a])
```

并且理解这两个列表的大小由 s 决定，并且必须相同。

当将遍历翻译为范畴论语言时，我们将使用列表大小的和来表达这个条件。大小为 n 的计数列表是一个 n 元组，或者 a^n 的一个元素，因此我们可以写：

$$\mathbf{Tr}\langle s, t \rangle \langle a, b \rangle = \mathbf{Set}(s, \sum_n (\mathbf{Set}(b^n, t) \times a^n))$$

我们将遍历解释为一个函数，给定一个源 s ，它产生一个隐藏 n 的存在类型。它表示存在一个 n 和一个由函数 $b^n \rightarrow t$ 和 n 元组 a^n 组成的对。

遍历的存在形式必须考虑到不同 n 的余项原则上会有不同的类型。例如，你可以将一棵树分解为一个 n 元组的叶子 a^n 和带有 n 个洞的余项 c_n 。因此，遍历的正确存在表示必须涉及对所有由自然数索引的序列 c_n 的 coend：

$$\mathbf{Tr}\langle s, t \rangle \langle a, b \rangle = \int^{c_n} \mathcal{C}(s, \sum_m c_m \times a^m) \times \mathcal{C}(\sum_k c_k \times b^k, t)$$

这里的和是 \mathcal{C} 中的余积。

看待序列 c_n 的一种方式是将它们解释为纤维化。例如，在 \mathbf{Set} 中，我们将从一个集合 C 和一个投影 $p: C \rightarrow \mathbb{N}$ 开始，其中 \mathbb{N} 是自然数的集合。类似地， a^n 可以解释为 a 上的自由幺半群 (a 的列表集合) 的纤维化，投影提取列表的长度。

或者我们可以将 c_n 视为从自然数集到 \mathcal{C} 的映射。事实上，我们可以将自然数视为一个离散范畴 \mathcal{N} ，在这种情况下， c_n 是函子 $\mathcal{N} \rightarrow \mathcal{C}$ 。

$$\mathbf{Tr}\langle s, t \rangle \langle a, b \rangle = \int^{c: [\mathcal{N}, \mathcal{C}]} \mathcal{C}(s, \sum_m c_m \times a^m) \times \mathcal{C}(\sum_k c_k \times b^k, t)$$

为了展示这两种表示的等价性，我们首先将和映射重写为映射的积：

$$\int^{c: [\mathcal{N}, \mathcal{C}]} \mathcal{C}(s, \sum_m c_m \times a^m) \times \prod_k \mathcal{C}(c_k \times b^k, t)$$

然后使用柯里化伴随：

$$\int^{c: [\mathcal{N}, \mathcal{C}]} \mathcal{C}(s, \sum_m c_m \times a^m) \times \prod_k \mathcal{C}(c_k, [b^k, t])$$

这里， $[b^k, t]$ 是内部 \mathbf{hom} ，它是指对象 t^{b^k} 的另一种表示。

下一步是认识到这个公式中的积表示 $[\mathcal{N}, \mathcal{C}]$ 中的一组自然变换。事实上，我们可以将其写为一个 \mathbf{end} ：

$$\prod_k \mathcal{C}(c_k, [b^k, t]) \cong \int_{k: \mathcal{N}} \mathcal{C}(c_k, [b^k, t])$$

这是因为在离散范畴上的 \mathbf{end} 只是一个积。或者，我们可以将其写为函子范畴中的 \mathbf{hom} 集：

$$[\mathcal{N}, \mathcal{C}](c_-, [b^-, t])$$

用占位符替换两个函子的参数：

$$k \mapsto c_k$$

$$k \mapsto [b^k, t]$$

我们现在可以在函子范畴 $[\mathcal{N}, \mathcal{C}]$ 中使用 $\mathbf{co}\text{-Yoneda}$ 引理：

$$\int^{c: [\mathcal{N}, \mathcal{C}]} \mathcal{C}(s, \sum_m c_m \times a^m) \times [\mathcal{N}, \mathcal{C}](c_-, [b^-, t]) \cong \mathcal{C}(s, \sum_m [b^m, t] \times a^m)$$

这个结果比我们最初的公式更一般，但当限制在集合范畴时，它会变成原来的公式。

为了推导遍历的 $\mathbf{profunctor}$ 表示，我们应该更仔细地观察所涉及的变换类型。我们定义函子 $c: [\mathcal{N}, \mathcal{C}]$ 在 a 上的作用为：

$$c \cdot a = \sum_m c_m \times a^m$$

这些作用可以通过使用分配律展开公式来组合：

$$c \cdot (c' \cdot a) = \sum_m c_m \times (\sum_n c'_n \times a^n)^m$$

如果目标范畴是 \mathbf{Set} ，这等价于以下 Day 卷积（对于非 \mathbf{Set} 范畴，可以使用 Day 卷积的富集版本）：

$$(c \star c')_k = \int^{m, n} \mathcal{N}(m+n, k) \times c_m \times c'_n$$

这为范畴 $[\mathcal{N}, \mathcal{C}]$ 提供了么半结构。

遍历的存在表示可以用这个么半范畴在 \mathcal{C} 上的作用来写：

$$\mathbf{Tr}\langle s, t \rangle \langle a, b \rangle = \int^{c: [\mathcal{N}, \mathcal{C}]} \mathcal{C}(s, c \cdot a) \times \mathcal{C}(c \cdot b, t)$$

为了推导遍历的 **profunctor** 表示，我们必须将 Tambara 模推广到么半范畴的作用：

$$\alpha_{\langle a, b \rangle, c} : P\langle a, b \rangle \rightarrow P\langle c \bullet a, c \bullet b \rangle$$

事实证明，**profunctor** 光学的原始推导仍然适用于这些广义的 Tambara 模，并且遍历可以写为多态函数：

$$\mathbf{Tr}\langle s, t \rangle \langle a, b \rangle = \int_{P: \mathcal{T}} \mathbf{Set}((UP)\langle a, b \rangle, (UP)\langle s, t \rangle)$$

其中 **end** 是在广义 Tambara 模上取的。

19.4 混合光学

每当我们有一个么半范畴（monoidal category） \mathcal{M} 作用于范畴 \mathcal{C} 时，我们可以定义相应的光学（optic）。具有这种作用的范畴被称为作用范畴（actegory）。我们可以更进一步，考虑两个独立的作用。假设 \mathcal{M} 可以同时作用于 \mathcal{C} 和 \mathcal{D} 。我们将使用相同的符号表示这两个作用：

$$\bullet : \mathcal{M} \times \mathcal{C} \rightarrow \mathcal{C}$$

$$\bullet : \mathcal{M} \times \mathcal{D} \rightarrow \mathcal{D}$$

然后我们可以定义混合光学（mixed optics）为：

$$\mathcal{O}\langle s, t \rangle \langle a, b \rangle = \int^{m: \mathcal{M}} \mathcal{C}(s, m \bullet a) \times \mathcal{D}(m \bullet b, t)$$

这些混合光学在 **profunctor** 的表示下可以表示为：

$$P : \mathcal{C}^{op} \times \mathcal{D} \rightarrow \mathbf{Set}$$

以及使用两个独立作用的相应 Tambara 模：

$$\alpha_{\langle a, b \rangle, m} : P\langle a, b \rangle \rightarrow P\langle m \bullet a, m \bullet b \rangle$$

其中 a 是 \mathcal{C} 的对象， b 是 \mathcal{D} 的对象， m 是 \mathcal{M} 的对象。

Exercise 19.4.1. 当其中一个范畴是终端范畴时，笛卡尔积作用的混合光学是什么？如果第一个范畴是 $\mathcal{C}^{op} \times \mathcal{C}$ 而第二个是终端范畴，情况又如何？

Chapter 20

Kan 扩张

如果范畴论不断提高抽象层次，那是因为它致力于发现模式。一旦模式被发现，就该研究这些模式形成的模式，依此类推。

我们已经在越来越高的抽象层次上看到了相同的反复出现的概念，并且描述得越来越简洁。

例如，我们首先使用泛构造定义了积。然后我们看到积定义中的跨度是自然变换。这导致将积解释为极限。接着我们看到可以使用伴随来定义它。我们能够将其与余积结合在一个简洁的公式中：

$$(\+) \dashv \Delta \dashv (\times)$$

老子说：“欲缩之，必固张之。”

Kan 扩张将抽象层次提得更高。Mac Lane 说：“所有概念都是 Kan 扩张。”

20.1 闭么半范畴

我们已经看到如何将函数对象定义为范畴积的右伴随：

$$\mathcal{C}(a \times b, c) \cong \mathcal{C}(a, [b, c])$$

这里我使用了替代符号 $[b, c]$ 表示内部 hom——即指数 c^b 。

两个函子之间的伴随关系可以被认为是一个是另一个的伪逆。它们不组合成恒等函子，但它们的组合通过单位和余单位与恒等函子相关。例如，如果你仔细观察，柯里化伴随的余单位：

$$\epsilon_{bc} : [b, c] \times b \rightarrow c$$

暗示着 $[b, c]$ 在某种意义上体现了乘法的逆。它在以下公式中扮演着与 c/b 类似的角色：

$$c/b \times b = c$$

以典型的范畴论方式，我们可能会问：如果我们用其他东西替换积会怎样？显然，用余积替换它不起作用（因此我们没有减法的类比）。但也许还有其他表现良好的二元运算具有右伴随。

推广积的自然设置是具有张量积 \otimes 和单位对象 I 的么半范畴。如果我们有一个伴随：

$$\mathcal{C}(a \otimes b, c) \cong \mathcal{C}(a, [b, c])$$

我们将该范畴称为闭么半范畴。在典型的范畴论符号滥用中，除非引起混淆，我们将使用与笛卡尔 hom 相同的符号（一对方括号）来表示么半内部 hom 。张量积的右伴随还有一种替代的棒棒糖符号：

$$\mathcal{C}(a \otimes b, c) \cong \mathcal{C}(a, b \multimap c)$$

它经常在线性类型的上下文中使用。

内部 hom 的定义在对称么半范畴中表现良好。如果张量积不是对称的，伴随关系定义了一个左闭么半范畴。左内部 hom 是“后乘”函子 $(- \otimes b)$ 的伴随。右闭结构定义为“前乘”函子 $(b \otimes -)$ 的右伴随。如果两者都定义了，则该范畴称为双闭。

Day 卷积的内部 hom

作为一个例子，考虑在余预层范畴中具有 Day 卷积的对称么半结构：

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \otimes b, x) \times Fa \times Gb$$

我们寻找伴随：

$$[\mathcal{C}, \mathbf{Set}](F \star G, H) \cong [\mathcal{C}, \mathbf{Set}](F, [G, H]_{\text{Day}})$$

左侧的自然变换可以写成关于 x 的 end ：

$$\int_x \mathbf{Set}\left(\int^{a,b} \mathcal{C}(a \otimes b, x) \times Fa \times Gb, Hx\right)$$

我们可以使用共连续性将 coends 拉出：

$$\int_{x,a,b} \mathbf{Set}(\mathcal{C}(a \otimes b, x) \times Fa \times Gb, Hx)$$

然后我们可以使用 \mathbf{Set} 中的柯里化伴随（方括号表示 \mathbf{Set} 中的内部 hom ）：

$$\int_{x,a,b} \mathbf{Set}(Fa, [\mathcal{C}(a \otimes b, x) \times Gb, Hx])$$

最后，我们使用 hom-set 的连续性将两个 ends 移到 hom-set 内部：

$$\int_a \mathbf{Set}\left(Fa, \int_{x,b} [\mathcal{C}(a \otimes b, x) \times Gb, Hx]\right)$$

我们发现 **Day** 卷积的右伴随由下式给出：

$$([G, H]_{\text{Day}})a = \int_{x,b} [\mathcal{C}(a \otimes b, x), [Gb, Hx]] \cong \int_b [Gb, H(a \otimes b)]$$

最后的变换是 **Set** 中 Yoneda 引理的应用。

Exercise 20.1.1. 在 *Haskell* 中实现 *Day* 卷积的内部 *hom*。提示：使用类型别名。

Exercise 20.1.2. 实现伴随的见证：

```
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)
```

幂与余幂

在集合范畴中，内部 *hom*（函数对象或指数）与外部 *hom*（两个对象之间的态射集）同构：

$$C^B \cong \text{Set}(B, C)$$

因此，我们可以将 **Set** 中定义内部 *hom* 的柯里化伴随重写为：

$$\text{Set}(A \times B, C) \cong \text{Set}(A, \text{Set}(B, C))$$

我们可以将这种伴随关系推广到 *B* 和 *C* 不是集合而是某个范畴 *C* 中的对象的情况。任何范畴中的外部 *hom* 始终是一个集合。但左侧不再由积定义。相反，它定义了集合 *A* 在对象 *b* 上的作用：

$$\mathcal{C}(A \cdot b, c) \cong \text{Set}(A, \mathcal{C}(b, c))$$

这被称为余幂。

你可以将这种作用视为将 *b* 的 *A* 个副本相加（取余积）。例如，如果 *A* 是一个二元集 **2**，我们得到：

$$\mathcal{C}(2 \cdot b, c) \cong \text{Set}(2, \mathcal{C}(b, c)) \cong \mathcal{C}(b, c) \times \mathcal{C}(b, c) \cong \mathcal{C}(b + b, c)$$

换句话说，

$$2 \cdot b \cong b + b$$

在这个意义上，余幂用迭代加法定义了乘法，就像我们在学校学到的那样。

如果我们将 *b* 乘以 *hom-set* $\mathcal{C}(b, c)$ 并对 *b* 取 *coend*，结果与 *c* 同构：

$$\int^b \mathcal{C}(b, c) \cdot b \cong c$$

确实，由于 Yoneda 引理，从任意 x 到两侧的映射是同构的：

$$\mathcal{C}\left(\int^b \mathcal{C}(b, c) \cdot b, x\right) \cong \int_b \mathbf{Set}(\mathcal{C}(b, c), \mathcal{C}(b, x)) \cong \mathcal{C}(c, x)$$

正如预期的那样，在 **Set** 中，余幂退化为笛卡尔积。

$$\mathbf{Set}(A \cdot B, C) \cong \mathbf{Set}(A, \mathbf{Set}(B, C)) \cong \mathbf{Set}(A \times B, C)$$

同样，我们可以将幂表示为迭代乘法。我们使用相同的右侧，但这次我们使用映射入来定义幂：

$$\mathcal{C}(b, A \pitchfork c) \cong \mathbf{Set}(A, \mathcal{C}(b, c))$$

你可以将幂视为将 c 的 A 个副本相乘。确实，将 A 替换为 **2** 会得到：

$$\mathcal{C}(b, \mathbf{2} \pitchfork c) \cong \mathbf{Set}(\mathbf{2}, \mathcal{C}(b, c)) \cong \mathcal{C}(b, c) \times \mathcal{C}(b, c) \cong \mathcal{C}(b, c \times c)$$

换句话说：

$$\mathbf{2} \pitchfork c \cong c \times c$$

这是 c^2 的一种花哨写法。

如果我们将 c 乘以 hom-set $\mathcal{C}(c', c)$ 并对所有 c 取 end，结果与 c' 同构：

$$\int_c \mathcal{C}(c', c) \pitchfork c \cong c'$$

这由 Yoneda 引理得出。确实，从任意 x 到两侧的映射是同构的：

$$\mathcal{C}\left(x, \int_c \mathcal{C}(c', c) \pitchfork c\right) \cong \int_c \mathbf{Set}(\mathcal{C}(c', c), \mathcal{C}(x, c)) \cong \mathcal{C}(x, c')$$

在 **Set** 中，幂退化为指数，它与 hom-set 同构：

$$A \pitchfork C \cong C^A \cong \mathbf{Set}(A, C)$$

这是积对称性的结果。

$$\mathbf{Set}(B, A \pitchfork C) \cong \mathbf{Set}(A, \mathbf{Set}(B, C)) \cong \mathbf{Set}(A \times B, C)$$

$$\cong \mathbf{Set}(B \times A, C) \cong \mathbf{Set}(B, \mathbf{Set}(A, C))$$

20.2 函子的逆

范畴论的一个方面是通过执行有损变换来丢弃信息；另一个方面是恢复丢失的信息。我们已经看到了用自由函子（遗忘函子的伴随函子）来弥补丢失数据的例子。**Kan** 扩展是另一个例子。两者都弥补了由不可逆函子丢失的数据。

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。另一个原因是它的像可能不覆盖整个目标范畴。换句话说，它在对象或箭头上不是满射的。

例如，考虑一个伴随 $L \dashv R$ 。假设 R 不是单射的，并且它将两个对象 c 和 c' 坍缩为单个对象 d

$$Rc = d$$

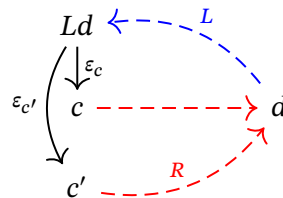
$$Rc' = d$$

L 无法撤销这一点。它不能同时将 d 映射到 c 和 c' 。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' 。这些箭头是定义伴随的余单位分量所必需的：

$$\varepsilon_c : Ld \rightarrow c$$

$$\varepsilon_{c'} : Ld \rightarrow c'$$

其中 Ld 既是 $L(Rc)$ 也是 $L(Rc')$



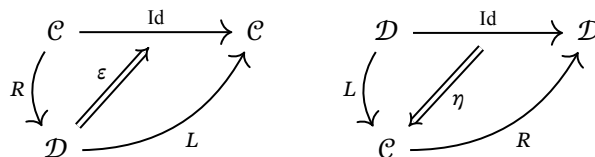
此外，如果 R 在对象上不是满射的，函子 L 必须以某种方式在那些不在 R 的像中的 \mathcal{D} 的对象上定义。同样，单位和余单位的自然性将限制可能的选择，只要这些对象与 R 的像之间有箭头连接。

显然，所有这些约束意味着伴随只能在非常特殊的情况下定义。

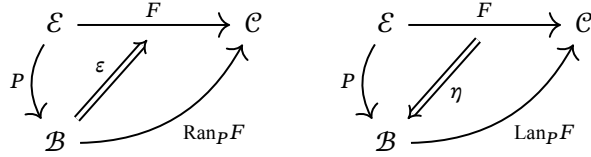
Kan 扩展甚至比伴随更弱。

如果伴随函子像逆一样工作，那么 **Kan** 扩展就像分数一样工作。

如果我们重新绘制定义伴随的余单位和单位的图表，这一点最为明显。在第一个图表中， L 似乎扮演了 $1/R$ 的角色。在第二个图表中， R 假装是 $1/L$ 。



右 Kan 扩展 $\text{Ran}_P F$ 和左 Kan 扩展 $\text{Lan}_P F$ 通过将恒等函子替换为某个函子 $F: \mathcal{E} \rightarrow \mathcal{C}$ 来推广这些。Kan 扩展然后扮演分数 F/P 的角色。从概念上讲，它们撤销 P 的动作，然后执行 F 的动作。



就像伴随一样，“撤销”并不完全。复合 $\text{Ran}_P F \circ P$ 不会重现 F ；相反，它通过称为余单位的自然变换 ε 与 F 相关。类似地，复合 $\text{Lan}_P F \circ P$ 通过单位 η 与 F 相关。

注意， F 丢弃的信息越多，Kan 扩展“逆”函子 P 就越容易。在某种意义上，它只需要“模 F ”地逆 P 。

以下是 Kan 扩展的直觉。我们从一个函子 F 开始：

$$\mathcal{E} \xrightarrow{F} \mathcal{C}$$

有第二个函子 P 将 \mathcal{E} 压缩到另一个范畴 \mathcal{B} 中。这可能是一个有损且非满射的嵌入。我们的任务是以某种方式扩展 F 的定义以覆盖整个 \mathcal{B} 。

在理想情况下，我们希望以下图表交换：

$$\begin{array}{ccc} \mathcal{E} & \xrightarrow{F} & \mathcal{C} \\ P \downarrow & \nearrow \text{Kan}_P F & \\ \mathcal{B} & & \end{array}$$

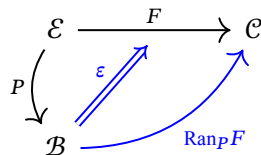
但这将涉及函子的相等性，这是我们尽量避免的。

次好的选择是要求通过此图表的两个路径之间存在自然同构。但即便如此，似乎要求过高。因此，我们最终要求一个路径可以变形为另一个路径，这意味着它们之间存在一个单向的自然变换。这个变换的方向区分了右 Kan 扩展和左 Kan 扩展。

20.3 右 Kan 扩展

右 Kan 扩展是一个函子 $\text{Ran}_P F$ ，配备一个自然变换 ε ，称为 Kan 扩展的余单位，定义为：

$$\varepsilon: (\text{Ran}_P F) \circ P \rightarrow F$$



对 $(\text{Ran}_P F, \varepsilon)$ 是普遍 (universal) 的, 即在所有这样的对 (G, α) 中, 其中 G 是一个函子 $G: \mathcal{B} \rightarrow \mathcal{C}$, 而 α 是一个自然变换:

$$\alpha: G \circ P \rightarrow F$$

A commutative triangle diagram. At the top is \mathcal{E} , at the bottom left is \mathcal{B} , and at the bottom right is \mathcal{C} . A horizontal arrow labeled F points from \mathcal{E} to \mathcal{C} . A curved arrow labeled P points from \mathcal{E} down to \mathcal{B} . A curved arrow labeled G points from \mathcal{B} up to \mathcal{C} . A red arrow labeled α points from \mathcal{B} to \mathcal{C} , representing the natural transformation $\alpha: G \circ P \rightarrow F$.

普遍性意味着对于任何这样的 (G, α) , 存在唯一的自然变换 $\sigma: G \rightarrow \text{Ran}_P F$

A commutative triangle diagram similar to the one above. It shows \mathcal{E} , \mathcal{B} , and \mathcal{C} with arrows F , P , and G . A blue curved arrow labeled $\text{Ran}_P F$ points from \mathcal{B} to \mathcal{C} . A red arrow labeled α points from \mathcal{B} to \mathcal{C} . A double arrow labeled σ points from G to $\text{Ran}_P F$, indicating the unique natural transformation $\sigma: G \rightarrow \text{Ran}_P F$.

它将 α 分解, 即:

$$\alpha = \varepsilon \cdot (\sigma \circ P)$$

这是自然变换的垂直和水平组合, 其中 $\sigma \circ P$ 是 σ 的“whiskering”。以下是相同的方程用弦图表示:

String diagrams representing the equation $\alpha = \varepsilon \cdot (\sigma \circ P)$. The left diagram shows a vertical line from \mathcal{E} to \mathcal{B} labeled P , and a vertical line from \mathcal{B} to \mathcal{C} labeled G . A semi-circular arc labeled α connects the two lines. The right diagram shows the same setup, but the arc is decomposed into a top part labeled ε and a bottom part labeled σ . The two diagrams are separated by an equals sign.

如果对于每个函子 F , 沿 P 的右 Kan 扩展都存在, 那么普遍构造可以推广为一个伴随 (adjunction) —— 这次是两个函子范畴之间的伴随:

$$[\mathcal{E}, \mathcal{C}](G \circ P, F) \cong [\mathcal{B}, \mathcal{C}](G, \text{Ran}_P F)$$

对于每个属于左侧的 α , 存在一个唯一的属于右侧的 σ 。

换句话说, 如果对于每个 F , 右 Kan 扩展都存在, 那么它是函子预组合的右伴随:

$$(- \circ P) \dashv \text{Ran}_P$$

这个伴随的余单位在 F 处的分量是 ε 。

这让人联想到柯里化 (currying) 伴随:

$$\mathcal{E}(a \times b, c) \cong \mathcal{E}(a, [b, c])$$

其中乘积被函子组合所取代。(这个类比并不完美, 因为组合只有在自函子范畴中才能被视为张量积。)

右 Kan 扩展作为 end

回忆一下“忍者”Yoneda 引理:

$$Fb \cong \int_e \mathbf{Set}(\mathcal{B}(b, e), Fe)$$

这里, F 是一个余预层 (co-presheaf), 即从 \mathcal{B} 到 \mathbf{Set} 的函子。 F 沿 P 的右 Kan 扩展推广了这个公式:

$$(\mathbf{Ran}_P F)b \cong \int_e \mathbf{Set}(\mathcal{B}(b, Pe), Fe)$$

这适用于余预层。通常我们感兴趣的是 $F: \mathcal{E} \rightarrow \mathcal{C}$, 所以我们需要用 幂 (power) 替换 \mathbf{Set} 中的 hom-集。因此, 右 Kan 扩展由以下 end 给出 (如果存在):

$$(\mathbf{Ran}_P F)b \cong \int_e \mathcal{B}(b, Pe) \pitchfork Fe$$

证明基本上是自明的: 在每一步中只有一件事可做。我们从伴随开始:

$$[\mathcal{E}, \mathcal{C}](G \circ P, F) \cong [\mathcal{B}, \mathcal{C}](G, \mathbf{Ran}_P F)$$

并用 end 重写它:

$$\int_e \mathcal{C}(G(Pe), Fe) \cong \int_b \mathcal{C}(Gb, (\mathbf{Ran}_P F)b)$$

我们代入公式得到:

$$\cong \int_b \mathcal{C}(Gb, \int_e \mathcal{B}(b, Pe) \pitchfork Fe)$$

我们利用 hom-函子的连续性将 end 提到前面:

$$\cong \int_b \int_e \mathcal{C}(Gb, \mathcal{B}(b, Pe) \pitchfork Fe)$$

然后我们使用幂的定义:

$$\int_b \int_e \mathbf{Set}(\mathcal{B}(b, Pe), \mathcal{C}(Gb, Fe))$$

并应用 Yoneda 引理:

$$\int_e \mathcal{C}(G(Pe), Fe)$$

这个结果确实是伴随的左侧。

如果 F 是一个余预层，右 Kan 扩展公式中的幂退化为指数/hom-集：

$$(\text{Ran}_P F)b \cong \int_e \mathbf{Set}(\mathcal{B}(b, Pe), Fe)$$

还要注意，如果 P 有一个左伴随，我们称之为 P^{-1} ，即：

$$\mathcal{B}(b, Pe) \cong \mathcal{E}(P^{-1}b, e)$$

我们可以使用“忍者”Yoneda 引理来评估 end：

$$(\text{Ran}_P F)b \cong \int_e \mathbf{Set}(\mathcal{B}(b, Pe), Fe) \cong \int_e \mathbf{Set}(\mathcal{E}(P^{-1}b, e), Fe) \cong F(P^{-1}b)$$

得到：

$$\text{Ran}_P F \cong F \circ P^{-1}$$

由于伴随是逆概念的弱化，这个结果与 Kan 扩展“反转” P 并随后应用 F 的直觉是一致的。

Haskell 中的右 Kan 扩展

右 Kan 扩展的 end 公式可以立即翻译为 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 $(\text{Ran } p \ f)$ 与 p 的复合到 f 的自然变换：

```
counit :: forall p f e'. Ran p f (p e') -> f e'
```

为了实现它，我们需要在给定一个多态函数的情况下生成类型为 $(f \ c')$ 的值：

```
h :: forall e. (p e' -> p e) -> f e
```

我们通过在类型 $e = e'$ 处实例化该函数，并使用 $(p \ e')$ 上的恒等函数调用它来实现：

```
counit (Ran h) = h id
```

右 Kan 扩展的计算能力来自于其泛性质。我们从一个配备有自然变换的函子 G 开始：

$$\alpha: G \circ P \rightarrow F$$

这可以表示为 Haskell 数据类型：

```
type Alpha p f g = forall e. g (p e) -> f e
```

泛性质告诉我们，存在一个唯一的自然变换 σ ，从该函子到相应的右 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)
```

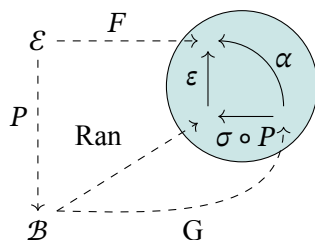
该变换通过余单位 ε 分解 α ：

$$\alpha = \varepsilon \cdot (\sigma \circ P)$$

回想一下，whiskering 意味着我们在 $b = p \ c$ 处实例化 `sigma`。然后紧接着调用 `counit`。因此， α 的分解由以下代码给出：

```
factorize' :: Functor g => Alpha p f g -> forall e. g (p e) -> f e
factorize' alpha gpc = alpha gpc
```

这三个自然变换的分量都是目标范畴 \mathcal{C} 中的态射：



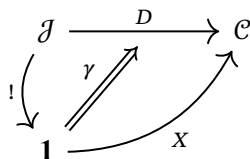
Exercise 20.3.1. 为 `Ran` 实现 `Functor` 实例。

作为 Kan 扩张的极限

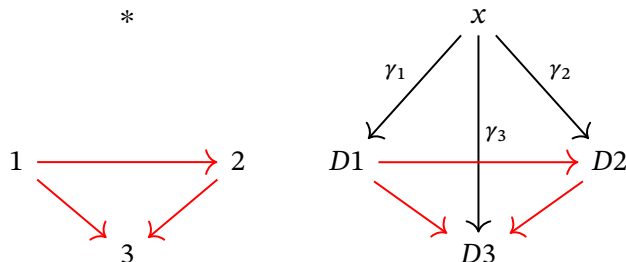
我们之前将极限定义为通用锥体。锥体的定义涉及两个范畴：定义图表形状的索引范畴 \mathcal{J} ，以及目标范畴 \mathcal{C} 。图表是一个将形状嵌入目标范畴的函子 $D: \mathcal{J} \rightarrow \mathcal{C}$ 。

我们可以引入第三个范畴 **1**：这是一个包含单个对象和单个恒等箭头的终端范畴。然后，我们可以使用从该范畴到 \mathcal{C} 的函子 X 来选取锥体的顶点 x 。由于 **1** 在 **Cat** 中是终端的，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 的极限是图表 D 沿 **!** 的右 Kan 扩张。首先，我们观察到复合 $X \circ !$ 将形状 \mathcal{J} 映射到单个对象 x ，因此它起到了常函子 Δ_x 的作用。因此，它选取了锥体的顶点。以 x 为顶点的锥体是一个自然变换 γ ：

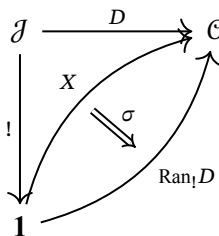


以下图表说明了这一点。在左边，我们有两个范畴： $\mathbf{1}$ 包含单个对象 $*$ ，而 \mathcal{J} 包含三个对象，形成图表的形状。在右边，我们有 D 的像和 $X \circ !$ 的像，即顶点 x 。 γ 的三个分量将顶点 x 连接到图表。 γ 的自然性确保了形成锥体侧面的三角形交换。



右 Kan 扩张 $(\text{Ran}_! D, \varepsilon)$ 是通用的这种锥体。 $\text{Ran}_! D$ 是一个从 $\mathbf{1}$ 到 \mathcal{C} 的函子，因此它在 \mathcal{C} 中选择一个对象。这确实是通用锥体的顶点， $\text{Lim} D$ 。

通用性意味着对于任何对 (X, γ) ，存在一个自然变换 $\sigma: X \rightarrow \text{Ran}_! D$



它将 γ 分解。

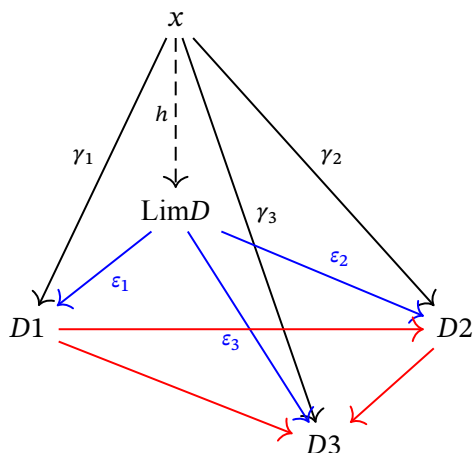
变换 σ 只有一个分量 σ_* ，它是一个连接顶点 x 到顶点 $\text{Lim} D$ 的箭头 h 。分解：

$$\gamma = \varepsilon \cdot (\sigma \circ !)$$

在分量中表示为：

$$\gamma_i = \varepsilon_i \circ h$$

它使得以下图表中的三角形交换：



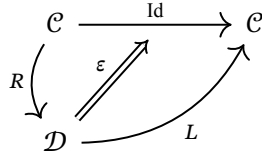
这个通用条件使得 $\text{Lim} D$ 成为图表 D 的极限。

左伴随作为右 Kan 延拓

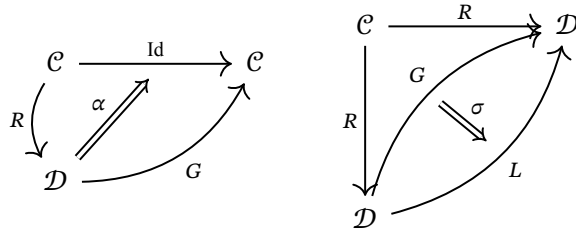
我们首先将 Kan 延拓描述为伴随关系的推广。从图示来看，如果我们有一对伴随函子 $L \dashv R$ ，我们期望左函子是恒等函子沿右函子的右 Kan 延拓。

$$L \cong \text{Ran}_R \text{Id}$$

当然，Kan 扩展的余单位 (counit) 与伴随 (adjunction) 的余单位是相同的：



我们还需要证明其具有泛性 (universality)：



为此，我们可以利用伴随关系的单位：

$$\eta: \text{Id} \rightarrow R \circ L$$

我们构造 σ 为复合：

$$G \rightarrow G \circ \text{Id} \xrightarrow{G \circ \eta} G \circ R \circ L \xrightarrow{\alpha \circ L} \text{Id} \circ L \rightarrow L$$

换句话说，我们定义 σ 为：

$$\sigma = (\alpha \circ L) \cdot (G \circ \eta)$$

我们可以提出一个相反的问题：如果 $\text{Ran}_R \text{Id}$ 存在，它是否自动成为 R 的左伴随？事实证明，我们还需要一个额外的条件：Kan 延拓必须被 R 保持，即：

$$R \circ \text{Ran}_R \text{Id} \cong \text{Ran}_R R$$

我们将在下一节中看到，这个条件的右侧定义了余密度单子。

Exercise 20.3.2. 证明分解条件：

$$\alpha = \varepsilon \cdot (\sigma \circ R)$$

对于上面定义的 σ 。提示：绘制相应的弦图并使用伴随关系的三角恒等式。

余密度单子 (Codensity Monad)

我们已经看到，每一个伴随函子 $L \dashv F$ 都会产生一个单子 $F \circ L$ 。事实上，这个单子是 F 沿 F 的右 Kan 扩张。有趣的是，即使 F 没有左伴随，Kan 扩张 $\text{Ran}_F F$ 仍然是一个单子，称为余密度单子，记作 T^F ：

$$T^F = \text{Ran}_F F$$

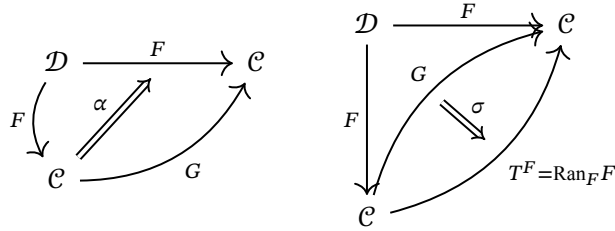
如果我们认真地将 Kan 扩张解释为分数，那么余密度单子将对应于 F/F 。一个使得这个“分数”等于恒等函子的函子被称为余密 (codense)。

为了证明 T^F 是一个单子，我们需要定义单子的单位 (unit) 和乘法 (multiplication)：

$$\eta: \text{Id} \rightarrow T^F$$

$$\mu: T^F \circ T^F \rightarrow T^F$$

这两者都来自于普遍性。对于每一个 (G, α) ，我们都有一个 σ ：



为了得到单位，我们将 G 替换为恒等函子 Id ，并将 α 替换为恒等自然变换。

为了得到乘法，我们将 G 替换为 $T^F \circ T^F$ ，并注意到我们可以使用 Kan 扩张的余单位 (counit)：

$$\varepsilon: T^F \circ F \rightarrow F$$

我们可以选择类型为 α 的：

$$\alpha: T^F \circ T^F \circ F \rightarrow F$$

并将其定义为复合：

$$T^F \circ T^F \circ F \xrightarrow{\text{id} \circ \varepsilon} T^F \circ F \xrightarrow{\varepsilon} F$$

或者，使用 whiskering 记号：

$$\alpha = \varepsilon \cdot (T^F \circ \varepsilon)$$

对应的 σ 给出了单子的乘法。

现在让我们证明，如果我们从一个伴随开始：

$$\mathcal{D}(Lc, d) \cong \mathcal{C}(c, Fd)$$

那么余密度单子由 $F \circ L$ 给出。让我们从任意函子 G 到 $F \circ L$ 的映射开始：

$$[\mathcal{C}, \mathcal{C}](G, F \circ L) \cong \int_c \mathcal{C}(Gc, F(Lc))$$

我们可以使用 Yoneda 引理重写它：

$$\cong \int_c \int_d \mathbf{Set}(\mathcal{D}(Lc, d), \mathcal{C}(Gc, Fd))$$

在这里，对 d 取 end 的效果是将 d 替换为 Lc 。我们现在可以使用伴随：

$$\cong \int_c \int_d \mathbf{Set}(\mathcal{C}(c, Fd), \mathcal{C}(Gc, Fd))$$

并对 c 进行 ninja-Yoneda 积分，得到：

$$\cong \int_d \mathcal{C}(G(Fd), Fd)$$

这反过来定义了一组自然变换：

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

通过 F 的前复合是右 Kan 扩张的左伴随：

$$[\mathcal{D}, \mathcal{C}](G \circ F, F) \cong [\mathcal{C}, \mathcal{C}](G, \text{Ran}_F F)$$

由于 G 是任意的，我们得出结论： $F \circ L$ 确实是余密度单子 $\text{Ran}_F F$ 。

由于每一个单子都可以从某个伴随导出，因此可以得出每一个单子都是某个伴随的余密度单子。

Haskell 中的密度单子

将密度单子 (codensity monad) 翻译到 Haskell 中，我们得到：

```
newtype Codensity f c = C (forall d. (c -> f d) -> f d)
```

以及提取器：

```
runCodensity :: Codensity f c -> forall d. (c -> f d) -> f d
runCodensity (C h) = h
```

这看起来非常类似于延续单子 (continuation monad)。事实上，如果我们选择 f 为恒等函子 (identity functor)，它就会变成延续单子。我们可以将 Codensity 视为接受一个回调函数 $(c \rightarrow f\ d)$ ，并在类型为 c 的结果可用时调用它。

以下是单子实例：

```
instance Monad (Codensity f) where
  return x = C (\k -> k x)
  m >=> kl = C (\k -> runCodensity m (\a -> runCodensity (kl a) k))
```

同样，这几乎与延续单子完全相同：

```
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** 具有延续传递风格 (continuation passing style) 的性能优势。由于它以“由内向外”的方式嵌套延续，因此可以用于优化由 **do** 块生成的绑定长链。

这一特性在处理自由单子 (free monads) 时尤为重要，自由单子会在树状结构中累积绑定。当我们最终解释自由单子时，这些累积的绑定需要遍历不断增长的树。对于每个绑定，遍历都从根节点开始。将此与之前反转列表的示例进行比较，后者通过将函数累积在 FIFO 队列中进行了优化。密度单子提供了相同类型的性能改进。

Exercise 20.3.3. 为 **Codensity** 实现 **Functor** 实例。

Exercise 20.3.4. 为 **Codensity** 实现 **Applicative** 实例。

20.4 左 Kan 扩展

正如右 Kan 扩展被定义为函子预组合的右伴随，左 Kan 扩展被定义为函子预组合的左伴随：

$$[\mathcal{B}, \mathcal{C}](\text{Lan}_P F, G) \cong [\mathcal{E}, \mathcal{C}](F, G \circ P)$$

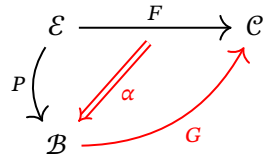
(还有后组合的伴随：它们被称为 Kan 提升。)

或者， $\text{Lan}_P F$ 可以定义为一个配备有称为单位 (unit) 的自然变换的函子：

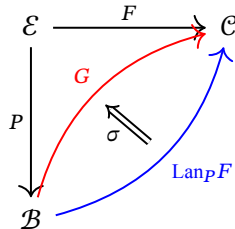
$$\eta : F \rightarrow \text{Lan}_P F \circ P$$

注意，左 Kan 扩展的单位方向与右 Kan 扩展的余单位 (counit) 方向相反。

$(\text{Lan}_P F, \eta)$ 这对是普遍的，意味着对于任何其他对 (G, α) ，其中

$$\alpha: F \rightarrow G \circ P$$


存在唯一的映射 $\sigma: \text{Lan}_P F \rightarrow G$

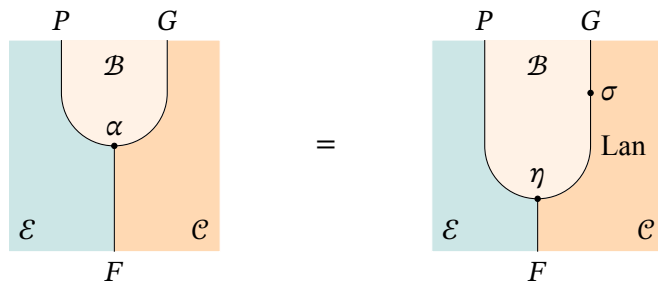


使得 α 被分解为:

$$\alpha = (\sigma \circ P) \cdot \eta$$

同样， σ 的方向与右 Kan 扩展相反。

使用弦图 (string diagrams)，我们可以将普遍条件描绘为:



这建立了两个自然变换集合之间的一一对应关系。对于左边的每个 α ，右边都有一个唯一的 σ :

$$[\mathcal{E}, \mathcal{C}](F, G \circ P) \cong [\mathcal{B}, \mathcal{C}](\text{Lan}_P F, G)$$

左 Kan 扩张作为余端

回顾忍者余 Yoneda 引理。对于每一个余预层 F ，我们有:

$$Fb \cong \int^c \mathcal{B}(c, b) \times Fc$$

左 Kan 扩张将这个公式推广为:

$$(\text{Lan}_P F)b \cong \int^e \mathcal{B}(Pe, b) \times Fe$$

对于一个一般的函子 $F: \mathcal{E} \rightarrow \mathcal{C}$ ，我们用余幂代替乘积：

$$(\text{Lan}_P F) b \cong \int^e \mathcal{B}(Pe, b) \cdot Fe$$

只要所讨论的余端存在，我们就可以通过考虑映射到某个函子 G 来证明这个公式。我们将自然变换的集合表示为关于 b 的端：

$$\int_b \mathcal{C}(\int^e \mathcal{B}(Pe, b) \cdot Fe, Gb)$$

利用余连续性，我们将余端拉出，将其转化为端：

$$\int_b \int_e \mathcal{C}(\mathcal{B}(Pe, b) \cdot Fe, Gb)$$

然后我们代入余幂的定义：

$$\int_b \int_e \mathcal{C}(\mathcal{B}(Pe, b), \mathcal{C}(Fe, Gb))$$

现在我们可以使用 Yoneda 引理对 b 进行积分，将 b 替换为 Pe ：

$$\int_e \mathcal{C}(Fe, G(Pe)) \cong [\mathcal{E}, \mathcal{C}](F, G \circ P)$$

这确实给出了函子预组合的左伴随：

$$[\mathcal{B}, \mathcal{C}](\text{Lan}_P F, G) \cong [\mathcal{E}, \mathcal{C}](F, G \circ P)$$

在 **Set** 中，余幂退化为笛卡尔积，因此我们得到一个更简单的公式：

$$(\text{Lan}_P F) b \cong \int^e \mathcal{B}(Pe, b) \times Fe$$

注意，如果函子 P 有一个右伴随，我们称之为 P^{-1} ：

$$\mathcal{B}(Pe, b) \cong \mathcal{E}(e, P^{-1}b)$$

我们可以使用忍者余 Yoneda 引理得到：

$$(\text{Lan}_P F) b \cong (F \circ P^{-1})b$$

这进一步强化了 Kan 扩张反转 P 并随后应用 F 的直觉。

Haskell 中的左 Kan 扩展

将左 Kan 扩展的公式翻译到 Haskell 时，我们用存在类型（existential type）替换了余端（coend）。符号表示如下：

```
type Lan p f b = exists e. (p e -> b, f e)
```

这是我们使用 **GADT** 编码存在类型的方式：

```
data Lan p f b where
  Lan :: (p e -> b) -> f e -> Lan p f b
```

左 Kan 扩展的单位 (unit) 是一个从函子 **f** 到 (**Lan p f**) 与 **p** 的复合的自然变换：

```
unit :: forall p f e'.
  f e' -> Lan p f (p e')
```

为了实现单位，我们从类型为 (**f e'**) 的值开始。我们需要找到某个类型 **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'**) 上使用恒等函数：

```
unit fe = Lan id fe
```

左 Kan 扩展的计算能力在于其泛性质 (universal property)。给定一个函子 **g** 以及一个从 **f** 到 **g** 与 **p** 的复合的自然变换：

```
type Alpha p f g = forall e. f e -> g (p e)
```

存在一个唯一的自然变换 σ ，从相应的左 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)
```

该变换通过单位 η 将 α 分解：

$$\alpha = (\sigma \circ P) \cdot \eta$$

σ 的 whiskering 意味着在 **b = p e** 处实例化它，因此 α 的分解实现如下：

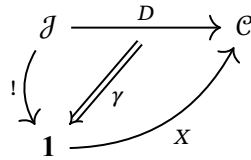
```
factorize :: Functor g => Alpha p f g -> f e -> g (p e)
factorize alpha = sigma alpha . unit
```

Exercise 20.4.1. 为 **Lan** 实现 **Functor** 实例。

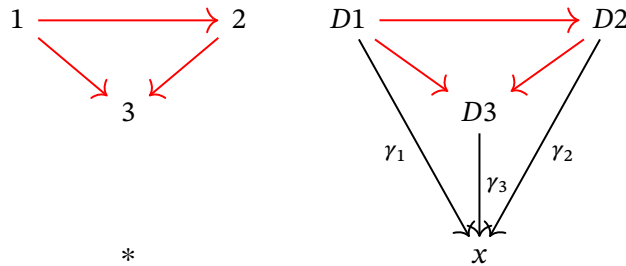
余极限作为 Kan 扩张

正如极限可以定义为右 Kan 扩张一样，余极限也可以定义为左 Kan 扩张。

我们从一个索引范畴 \mathcal{J} 开始，它定义了余极限的形状。函子 D 在目标范畴 \mathcal{C} 中选择这个形状。余锥的顶点由一个从终对象单对象范畴 $\mathbf{1}$ 出发的函子选择。自然变换定义了一个从 D 到 X 的余锥：



这里是一个由三个对象和三个态射（不包括恒等态射）组成的简单形状的示例。对象 x 是函子 X 下单对象 $*$ 的像：



余极限是通用余锥，它由 D 沿函子 $!$ 的左 Kan 扩张给出：

$$\text{Colim } D = \text{Lan}_! D$$

右伴随作为左 Kan 扩张

我们已经看到，当我们有一个伴随 $L \vdash R$ 时，左伴随与右 Kan 扩张相关。对偶地，如果右伴随存在，它可以表示为恒等函子的左 Kan 扩张：

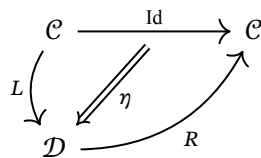
$$R \cong \text{Lan}_L \text{Id}$$

反之，如果恒等函子的左 Kan 扩张存在并且它保持函子 L ：

$$L \circ \text{Lan}_L \text{Id} \cong \text{Lan}_L L$$

那么 $\text{Lan}_L \text{Id}$ 就是 L 的右伴随。 L 沿自身的左 Kan 扩张称为密度余单子。

Kan 扩张的单位与伴随的单位相同：



普遍性的证明与右 Kan 扩张的证明类似。

Exercise 20.4.2. 为密度余单子实现 **Comonad** 实例：

```
data Density f c where
  D :: (f d -> c) -> f d -> Density f c
```

Day 卷积作为 Kan 扩张

我们已经看到 Day 卷积被定义为在么半范畴 \mathcal{C} 上的余预层范畴中的张量积：

$$(F \star G)c = \int^{a,b} \mathcal{C}(a \otimes b, c) \times Fa \times Gb$$

余预层，即 $[\mathcal{C}, \mathbf{Set}]$ 中的函子，也可以使用外部张量积进行张量积。两个对象的外部积不是在同一个范畴中生成一个对象，而是在另一个范畴中选择一个对象。在我们的例子中，两个函子的积最终落在 $\mathcal{C} \times \mathcal{C}$ 上的余预层范畴中：

$$\tilde{\otimes} : [\mathcal{C}, \mathbf{Set}] \times [\mathcal{C}, \mathbf{Set}] \rightarrow [\mathcal{C} \times \mathcal{C}, \mathbf{Set}]$$

两个余预层在 $\mathcal{C} \times \mathcal{C}$ 中的一对对象上的积由以下公式给出：

$$(F \tilde{\otimes} G)\langle a, b \rangle = Fa \times Gb$$

事实证明，两个函子的 Day 卷积可以表示为它们的外部积在 \mathcal{C} 中的张量积下的左 Kan 扩张：

$$F \star G \cong \text{Lan}_{\tilde{\otimes}}(F \tilde{\otimes} G)$$

图示如下：

$$\begin{array}{ccc} \mathcal{C} \times \mathcal{C} & \xrightarrow{F \tilde{\otimes} G} & \mathbf{Set} \\ \tilde{\otimes} \downarrow & \nearrow \text{Lan}_{\tilde{\otimes}}(F \tilde{\otimes} G) & \\ \mathcal{C} & & \end{array}$$

确实，使用左 Kan 扩张的余积公式，我们得到：

$$\begin{aligned} (\text{Lan}_{\tilde{\otimes}}(F \tilde{\otimes} G))c &\cong \int^{\langle a, b \rangle} \mathcal{C}(a \otimes b, c) \cdot (F \tilde{\otimes} G)\langle a, b \rangle \\ &\cong \int^{\langle a, b \rangle} \mathcal{C}(a \otimes b, c) \cdot (Fa \times Gb) \end{aligned}$$

由于这两个函子是 **Set** 值的，余幂退化为笛卡尔积：

$$\cong \int^{\langle a, b \rangle} \mathcal{C}(a \otimes b, c) \times Fa \times Gb$$

并重现了 Day 卷积的公式。

20.5 常用公式

- 余幂:

$$\mathcal{C}(A \cdot b, c) \cong \mathbf{Set}(A, \mathcal{C}(b, c))$$

- 幂:

$$\mathcal{C}(b, A \pitchfork c) \cong \mathbf{Set}(A, \mathcal{C}(b, c))$$

- 右 Kan 扩张:

$$[\mathcal{E}, \mathcal{C}](G \circ P, F) \cong [\mathcal{B}, \mathcal{C}](G, \mathbf{Ran}_P F)$$

$$(\mathbf{Ran}_P F)b \cong \int_e \mathcal{B}(b, Pe) \pitchfork Fe$$

- 左 Kan 扩张:

$$[\mathcal{B}, \mathcal{C}](\mathbf{Lan}_P F, G) \cong [\mathcal{E}, \mathcal{C}](F, G \circ P)$$

$$(\mathbf{Lan}_P F)b \cong \int^e \mathcal{B}(Pe, b) \cdot Fe$$

- **Set** 中的右 Kan 扩张:

$$(\mathbf{Ran}_P F)b \cong \int_e \mathbf{Set}(\mathcal{B}(b, Pe), Fe)$$

- **Set** 中的左 Kan 扩张:

$$(\mathbf{Lan}_P F)b \cong \int^e \mathcal{B}(Pe, b) \times Fe$$

富化

老子曰：“知足者富。”

21.1 富化范畴

这可能会让人感到惊讶，但如果没有一些富化范畴的背景知识，就无法完全解释 Haskell 中 **Functor** 的定义。在本章中，我将尝试展示，至少在概念上，富化并不是从普通范畴论迈出的一大步。

研究富化范畴的额外动机来自于这样一个事实：许多文献，特别是 nLab 网站，以最一般的术语描述概念，这通常意味着以富化范畴的术语来描述。大多数常见的构造只需通过改变词汇来翻译，将 **hom**-集替换为 **hom**-对象，将 **Set** 替换为幺半范畴 \mathcal{V} 。

一些富化概念，如加权极限和余极限，本身非常强大，以至于人们可能会试图用“所有概念都是加权（余）极限”来取代 Mac Lane 的格言“所有概念都是 Kan 扩展”。

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 **hom**-集的概念，定义为两个对象之间的箭头集合，将集合论作为范畴论的前提条件。诚然，箭头仅在局部小范畴中形成一个集合，但考虑到处理太大而不能成为集合的事物需要更多的理论，这只是一个小小的安慰。

如果范畴论能够自举，例如通过用更一般的对象替换 **hom**-集，那将是非常好的。这正是富化范畴背后的思想。然而，这些 **hom**-对象必须来自具有 **hom**-集的某个其他范畴，并且在某些时候我们必须回到集合论的基础。尽管如此，能够用不同的东西替换无结构的 **hom**-集，扩展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，我们有时会谈论广义元素，它们只是指向一个对象的箭头；或者全局元素，它们是从终端对象（有时是从幺半单位 I ）出发的箭头。但最重要的是，集合定义了元素的相等性。

几乎所有我们学到的关于范畴的知识都可以翻译到富化范畴的领域。然而，许多范畴推理涉及交换图，这些图表达了箭头的相等性。在富化设置中，我们没有在对象之间移动的箭头，因此所有这些构造都必须进行修改。

Hom-对象

乍一看，用对象替换 hom-集似乎是一个倒退。毕竟，集合有元素，而对象是无形的斑点。然而，hom-对象的丰富性编码在它们所属范畴的态射中。从概念上讲，集合无结构意味着它们之间有许多态射（函数）。拥有更少的态射通常意味着拥有更多的结构。

定义富化范畴的指导原则是，我们应该能够将普通范畴论作为一个特例恢复。毕竟，hom-集是范畴 **Set** 中的对象。事实上，我们非常努力地用函数而不是元素来表达集合的属性。

话虽如此，用复合和单位定义范畴的定义涉及作为 hom-集元素的态射。因此，让我们首先在不使用元素的情况下重新表述范畴的基本概念。

箭头的复合可以批量定义为 hom-集之间的函数：

$$\circ : \mathcal{C}(b, c) \times \mathcal{C}(a, b) \rightarrow \mathcal{C}(a, c)$$

我们可以使用从单例集出发的函数来讨论单位箭头：

$$j_a : 1 \rightarrow \mathcal{C}(a, a)$$

这表明，如果我们想用某个范畴 \mathcal{V} 中的对象替换 hom-集 $\mathcal{C}(a, b)$ ，我们必须能够乘这些对象来定义复合，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，但事实上，幺半范畴同样适用。正如我们将看到的，幺半范畴的单位和结合律直接转化为复合的单位和结合律。

富化范畴

设 \mathcal{V} 是一个具有张量积 \otimes 、单位对象 I 以及结合子和两个单位子（以及它们的逆）的幺半范畴：

$$\alpha : (a \otimes b) \otimes c \rightarrow a \otimes (b \otimes c)$$

$$\lambda : I \otimes a \rightarrow a$$

$$\rho : a \otimes I \rightarrow a$$

一个在 \mathcal{V} 上富化的范畴 \mathcal{C} 具有对象，并且对于任何一对对象 a 和 b ，都有一个 hom-对象 $\mathcal{C}(a, b)$ 。这个 hom-对象是 \mathcal{V} 中的一个对象。复合使用 \mathcal{V} 中的箭头定义：

$$\circ : \mathcal{C}(b, c) \otimes \mathcal{C}(a, b) \rightarrow \mathcal{C}(a, c)$$

单位由箭头定义：

$$j_a : I \rightarrow \mathcal{C}(a, a)$$

结合性用 \mathcal{V} 中的结合子表示：

$$\begin{array}{ccc} (\mathcal{C}(c, d) \otimes \mathcal{C}(b, c)) \otimes \mathcal{C}(a, b) & \xrightarrow{\alpha} & \mathcal{C}(c, d) \otimes (\mathcal{C}(b, c) \otimes \mathcal{C}(a, b)) \\ \downarrow \circ \otimes id & & \downarrow id \otimes \circ \\ \mathcal{C}(b, d) \otimes \mathcal{C}(a, b) & \xrightarrow{\circ} & \mathcal{C}(c, d) \otimes \mathcal{C}(a, c) \\ & \searrow \circ \quad \swarrow \circ & \\ & \mathcal{C}(a, d) & \end{array}$$

单位律用 \mathcal{V} 中的单位子表示：

$$\begin{array}{ccc} I \otimes \mathcal{C}(a, b) & \xrightarrow{\lambda} & \mathcal{C}(a, b) \\ j_b \otimes id \downarrow & \circ & \uparrow \\ \mathcal{C}(b, b) \otimes \mathcal{C}(a, b) & & \end{array} \quad \begin{array}{ccc} \mathcal{C}(a, b) \otimes I & \xrightarrow{\rho} & \mathcal{C}(a, b) \\ id \otimes j_a \downarrow & \circ & \uparrow \\ \mathcal{C}(a, b) \otimes \mathcal{C}(a, a) & & \end{array}$$

注意，这些都是 \mathcal{V} 中的图，我们在其中确实有形成 **hom**-集的箭头。我们仍然依赖于集合论，但在不同的层次上。

在 \mathcal{V} 上富化的范畴也称为 \mathcal{V} -范畴。在接下来的内容中，我们将假设富化范畴是对称幺半的，因此我们可以形成对偶和积 \mathcal{V} -范畴。

\mathcal{V} -范畴 \mathcal{C} 的对偶范畴 \mathcal{C}^{op} 通过反转 **hom**-对象获得，即：

$$\mathcal{C}^{op}(a, b) = \mathcal{C}(b, a)$$

对偶范畴中的复合涉及反转 **hom**-对象的顺序，因此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 $\mathcal{C} \otimes \mathcal{D}$ 的积的对象是来自每个范畴的对象对。这些对之间的 **hom**-对象定义为张量积：

$$(\mathcal{C} \otimes \mathcal{D})(\langle c, d \rangle, \langle c', d' \rangle) = \mathcal{C}(c, c') \otimes \mathcal{D}(d, d')$$

我们需要张量积的对称性来定义复合。实际上，我们需要交换中间的两个 **hom**-对象，然后才能应用两个可用的复合：

$$\circ : (\mathcal{C}(c', c'') \otimes \mathcal{D}(d', d'')) \otimes (\mathcal{C}(c, c') \otimes \mathcal{D}(d, d')) \rightarrow \mathcal{C}(c, c'') \otimes \mathcal{D}(d, d'')$$

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 21.1.1. 定义 \mathcal{V} -范畴 \mathcal{C}^{op} 中的复合和单位。

Exercise 21.1.2. 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 hom -集由 hom -对象的（幺半全局）元素给出，即 $\mathcal{V}(I, \mathcal{C}(a, b))$ 的元素。

例子

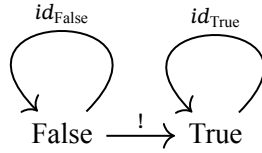
从这个新的角度来看，我们迄今为止研究的普通范畴在幺半范畴 $(\mathbf{Set}, \times, 1)$ 上是平凡富化的，其中笛卡尔积作为张量积，单例集作为单位。

有趣的是，2-范畴可以被视为在 \mathbf{Cat} 上富化。实际上，2-范畴中的 1-细胞本身就是另一个范畴中的对象。2-细胞只是该范畴中的箭头。特别是小范畴的 2-范畴 \mathbf{Cat} 在自身中富化。其 hom -对象是函子范畴，它们是 \mathbf{Cat} 中的对象。

预序

富化并不总是意味着添加更多的东西。有时它看起来更像是贫化，就像在行走箭头范畴上富化的情况。

这个范畴只有两个对象，为了这个构造的目的，我们将其称为 False 和 True 。从 False 到 True 有一个单一的箭头（不包括单位箭头），这使得 False 成为初始对象， True 成为终端对象。



为了使其成为幺半范畴，我们定义张量积，使得：

$$\text{True} \otimes \text{True} = \text{True}$$

所有其他组合都产生 False 。 True 是幺半单位，因为：

$$\text{True} \otimes x = x$$

在幺半行走箭头上富化的范畴称为预序。任何两个对象之间的 hom -对象 $\mathcal{C}(a, b)$ 可以是 False 或 True 。我们将 True 解释为 a 在预序中先于 b ，我们将其写为 $a \leq b$ 。 False 意味着这两个对象无关。

由以下定义的复合的重要属性：

$$\mathcal{C}(b, c) \otimes \mathcal{C}(a, b) \rightarrow \mathcal{C}(a, c)$$

是，如果左边的两个 hom -对象都是 True ，那么右边也必须是 True 。（它不能是 False ，因为没有从 True 到 False 的箭头。）在预序解释中，这意味着 \leq 是传递的：

$$b \leq c \wedge a \leq b \implies a \leq c$$

通过同样的推理，单位箭头的存在：

$$j_a : \text{True} \rightarrow \mathcal{C}(a, a)$$

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， $a \leq a$ 。

注意，预序并不排除循环，特别是，可能有 $a \leq b$ 和 $b \leq a$ 而 a 不等于 b 。

预序也可以在不使用富化的情况下定义为薄范畴——在任何两个对象之间最多有一个箭头的范畴。

自富化

任何笛卡尔闭范畴 \mathcal{V} 都可以被视为自富化的。这是因为每个外部 hom -集 $\mathcal{C}(a, b)$ 都可以被内部 $\text{hom } b^a$ （箭头对象）替换。

事实上，每个幺半闭范畴 \mathcal{V} 都是自富化的。回想一下，在幺半闭范畴中，我们有 hom -函子伴随：

$$\mathcal{V}(a \otimes b, c) \cong \mathcal{V}(a, [b, c])$$

这个伴随的余单位作为评估态射：

$$\varepsilon_{bc} : [b, c] \otimes b \rightarrow c$$

为了在这个自富化范畴中定义复合，我们需要一个箭头：

$$\circ : [b, c] \otimes [a, b] \rightarrow [a, c]$$

诀窍是考虑整个 hom -集并证明我们总能从中选择一个规范元素。我们从集合开始：

$$\mathcal{V}([b, c] \otimes [a, b], [a, c])$$

我们可以使用伴随将其重写为：

$$\mathcal{V}([([b, c] \otimes [a, b]) \otimes a, c)$$

我们现在要做的就是选择这个 hom -集的一个元素。我们通过构造以下复合来实现：

$$([b, c] \otimes [a, b]) \otimes a \xrightarrow{\alpha} [b, c] \otimes ([a, b] \otimes a) \xrightarrow{id \otimes \varepsilon_{ab}} [b, c] \otimes b \xrightarrow{\varepsilon_{bc}} c$$

我们使用了结合子和伴随的余单位。

我们还需要一个定义单位的箭头：

$$j_a : I \rightarrow [a, a]$$

同样，我们可以选择它作为 hom -集 $\mathcal{V}(I, [a, a])$ 的成员。我们使用伴随：

$$\mathcal{V}(I, [a, a]) \cong \mathcal{V}(I \otimes a, a)$$

我们知道这个 hom -集包含左单位子 λ ，所以我们可以用它来定义 j_a 。

21.2 \mathcal{V} -函子

普通函子将对象映射到对象，箭头映射到箭头。类似地，一个充实函子 (enriched functor) F 将对象映射到对象，但它不是作用于单个箭头，而是必须将同态对象 (hom-object) 映射到同态对象。这只有在源范畴 \mathcal{C} 中的同态对象与目标范畴 \mathcal{D} 中的同态对象属于同一范畴时才可能。换句话说，两个范畴必须在相同的 \mathcal{V} 上充实。然后， F 在同态对象上的作用使用 \mathcal{V} 中的箭头定义：

$$F_{ab} : \mathcal{C}(a, b) \rightarrow \mathcal{D}(Fa, Fb)$$

为了清晰起见，我们在 F 的下标中指定对象对。

函子必须保持复合和恒等。这些可以用 \mathcal{V} 中的交换图表示：

$$\begin{array}{ccc} \mathcal{C}(b, c) \otimes \mathcal{C}(a, b) & \xrightarrow{\circ} & \mathcal{C}(a, c) \\ \downarrow F_{bc} \otimes F_{ab} & & \downarrow F_{ac} \\ \mathcal{D}(Fb, Fc) \otimes \mathcal{D}(Fa, Fb) & \xrightarrow{\circ} & \mathcal{D}(Fa, Fc) \end{array} \quad \begin{array}{ccc} & I & \\ j_a \swarrow & & \searrow j_{Fa} \\ \mathcal{C}(a, a) & \xrightarrow{F_{aa}} & \mathcal{D}(Fa, Fa) \end{array}$$

注意，我使用了相同的符号 \circ 表示两种不同的复合，以及相同的 j 表示两种不同的恒等映射。它们的含义可以从上下文中推导出来。

如前所述，所有图表都在范畴 \mathcal{V} 中。

Hom-函子

在一个在幺半闭范畴 (monoidal closed category) \mathcal{V} 上充实的范畴中，hom-函子是一个充实函子：

$$\mathrm{Hom}_{\mathcal{C}} : \mathcal{C}^{op} \otimes \mathcal{C} \rightarrow \mathcal{V}$$

在这里，为了定义一个充实函子，我们必须将 \mathcal{V} 视为自充实的。

这个函子在（对象对）上的作用很清楚：

$$\mathrm{Hom}_{\mathcal{C}}\langle a, b \rangle = \mathcal{C}(a, b)$$

为了定义一个充实函子，我们必须定义 Hom 在同态对象上的作用。这里，源范畴是 $\mathcal{C}^{op} \otimes \mathcal{C}$ ，目标范畴是 \mathcal{V} ，两者都在 \mathcal{V} 上充实。让我们考虑从 $\langle a, a' \rangle$ 到 $\langle b, b' \rangle$ 的同态对象。hom-函子在这个同态对象上的作用是 \mathcal{V} 中的一个箭头：

$$\mathrm{Hom}_{\langle a, a' \rangle \times \langle b, b' \rangle} : (\mathcal{C}^{op} \otimes \mathcal{C})(\langle a, a' \rangle, \langle b, b' \rangle) \rightarrow \mathcal{V}(\mathrm{Hom}\langle a, a' \rangle, \mathrm{Hom}\langle b, b' \rangle)$$

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。因此，我们寻找一个箭头：

$$\mathcal{C}(b, a) \otimes \mathcal{C}(a', b') \rightarrow [\mathcal{C}(a, a'), \mathcal{C}(b, b')]$$

我们可以使用柯里化 (Currying) 的 hom-函子伴随关系来解包内部同态:

$$(\mathcal{C}(b, a) \otimes \mathcal{C}(a', b')) \otimes \mathcal{C}(a, a') \rightarrow \mathcal{C}(b, b')$$

我们可以通过重新排列积并应用两次复合来构造这个箭头。

在充实设置中, 我们最接近定义从 a 到 b 的单个态射的方法是使用单位对象的一个箭头。我们将同态对象的 (么半全局) 元素定义为 \mathcal{V} 中的一个态射:

$$f: I \rightarrow \mathcal{C}(a, b)$$

我们可以定义使用 hom-函子提升这样的箭头意味着什么。例如, 保持第一个参数不变, 我们定义:

$$\mathcal{C}(c, f): \mathcal{C}(c, a) \rightarrow \mathcal{C}(c, b)$$

作为复合:

$$\mathcal{C}(c, a) \xrightarrow{\lambda^{-1}} I \otimes \mathcal{C}(c, a) \xrightarrow{f \otimes id} \mathcal{C}(a, b) \otimes \mathcal{C}(c, a) \xrightarrow{\circ} \mathcal{C}(c, b)$$

类似地, f 的反变提升:

$$\mathcal{C}(f, c): \mathcal{C}(b, c) \rightarrow \mathcal{C}(a, c)$$

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研究的许多熟悉构造都有其充实对应物, 其中积被张量积取代, **Set** 被 \mathcal{V} 取代。

Exercise 21.2.1. 两个预序之间的函子是什么?

充实共预层

共预层 (co-presheaves), 即 **Set**-值函子, 在范畴论中扮演着重要角色, 因此很自然地会问它们在充实设置中的对应物是什么。共预层的推广是一个 \mathcal{V} -函子 $\mathcal{C} \rightarrow \mathcal{V}$ 。这只有在 \mathcal{V} 可以成为一个 \mathcal{V} -范畴时才有可能, 即当它是么半闭的。

一个充实共预层将 \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:

$$F_{ab}: \mathcal{C}(a, b) \rightarrow [Fa, Fb]$$

特别是, Hom-函子是 \mathcal{V} -值 \mathcal{V} -函子的一个例子:

$$\text{Hom}: \mathcal{C}^{op} \otimes \mathcal{C} \rightarrow \mathcal{V}$$

hom-函子是充实 profunctor 的一个特例, 其定义为:

$$\mathcal{C}^{op} \otimes \mathcal{D} \rightarrow \mathcal{V}$$

Exercise 21.2.2. 张量积是 \mathcal{V} 中的一个函子：

$$\otimes : \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V}$$

证明如果 \mathcal{V} 是么半闭的，张量积定义了一个 \mathcal{V} -函子。提示：定义它在内部同态上的作用。

函子强度与充实

当我们讨论单子 (monad) 时，我提到了使它们在编程中工作的重要属性。定义单子的自函子必须是强的，以便我们可以在单子代码中访问外部上下文。

事实证明，我们在 **Haskell** 中定义自函子的方式使它们自动成为强的。原因是强度与充实有关，正如我们所看到的，笛卡尔闭范畴是自充实的。让我们从一些定义开始。

在么半范畴中，自函子 F 的函子强度定义为具有分量的自然变换：

$$\sigma_{ab} : a \otimes F(b) \rightarrow F(a \otimes b)$$

有一些非常明显的相干条件使强度尊重张量积的性质。这是结合性条件：

$$\begin{array}{ccc} (a \otimes b) \otimes F(c) & \xrightarrow{\sigma_{(a \otimes b)c}} & F((a \otimes b) \otimes c) \\ \downarrow \alpha & & \downarrow F(\alpha) \\ a \otimes (b \otimes F(c)) & \xrightarrow{a \otimes \sigma_{bc}} a \otimes F(b \otimes c) \xrightarrow{\sigma_{a(b \otimes c)}} & F(a \otimes (b \otimes c)) \end{array}$$

这是单位条件：

$$\begin{array}{ccc} I \otimes F(a) & \xrightarrow{\sigma_{Ia}} & F(I \otimes a) \\ & \searrow \lambda & \downarrow F(\lambda) \\ & & F(a) \end{array}$$

在一般么半范畴中，这称为左强度，并且有相应的右强度定义。在对称么半范畴中，两者是等价的。

一个充实自函子将同态对象映射到同态对象：

$$F_{ab} : \mathcal{C}(a, b) \rightarrow \mathcal{C}(Fa, Fb)$$

如果我们将么半闭范畴 \mathcal{V} 视为自充实的，同态对象是内部同态，因此一个充实自函子配备了映射：

$$F_{ab} : [a, b] \rightarrow [Fa, Fb]$$

将其与我们在 **Haskell** 中定义的 **Functor** 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, $(a \multimap b)$ 和 $(f \multimap a \multimap f \multimap b)$, 是内部同态。因此, Haskell 的 `Functor` 确实是一个充实函子。

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简单结果:

$$\mathcal{C}(1 \times b, c) \cong \mathcal{C}(1, [b, c])$$

以及终端对象是笛卡尔积的单元的事实。

事实证明, 在自充实范畴 \mathcal{V} 中, 每个强自函子都是自动充实的。确实, 为了证明函子 F 是充实的, 我们需要定义内部同态之间的映射, 即同态集的一个元素:

$$F_{ab} \in \mathcal{V}([a, b], [Fa, Fb])$$

使用同态伴随关系, 这与以下同构:

$$\mathcal{V}([a, b] \otimes Fa, Fb)$$

我们可以通过组合强度和伴随关系的余单位 (求值态射) 来构造这个映射:

$$[a, b] \otimes Fa \xrightarrow{\sigma_{[a, b]a}} F([a, b] \otimes a) \xrightarrow{F\epsilon_{ab}} Fb$$

相反, \mathcal{V} 中的每个充实自函子都是强的。为了证明强度, 我们需要定义映射 σ_{ab} , 或者等价地 (通过同态伴随关系):

$$a \rightarrow [Fb, F(a \otimes b)]$$

回忆同态伴随关系的单位定义, 即余求值态射:

$$\eta_{ab} : a \rightarrow [b, a \otimes b]$$

我们构造以下复合:

$$a \xrightarrow{\eta_{ab}} [b, a \otimes b] \xrightarrow{F_{b, a \otimes b}} [Fb, F(a \otimes b)]$$

这可以直接翻译到 Haskell:

```
strength :: Functor f => (a, f b) -> f (a, b)
strength = uncurry (\a -> fmap (coeval a))
```

其中 `coeval` 的定义如下:

```
coeval :: a -> (b -> (a, b))
coeval a = \b -> (a, b)
```

由于柯里化和求值内置于 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
```

21.3 \mathcal{V} -自然变换

两个函子 F 和 G 从 \mathcal{C} 到 \mathcal{D} 之间的普通自然变换是从同态集 $\mathcal{D}(Fa, Ga)$ 中选择箭头。在富集（enriched）设置中，我们没有箭头，所以我们可以使用单位对象 I 来进行选择。我们定义 \mathcal{V} -自然变换在 a 处的分量为一个箭头：

$$\nu_a : I \rightarrow \mathcal{D}(Fa, Ga)$$

自然性条件有些复杂。标准的自然性方块涉及任意箭头 $f : a \rightarrow b$ 的提升以及以下组合的等式：

$$\nu_b \circ Ff = Gf \circ \nu_a$$

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 $f \in \mathcal{C}(a, b)$ 。方程两边的组合是 $\mathcal{D}(Fa, Gb)$ 的元素。

在左边，我们有箭头 $\nu_b \circ Ff$ 。组合本身是两个同态集乘积的映射：

$$\mathcal{D}(Fb, Gb) \times \mathcal{D}(Fa, Fb) \rightarrow \mathcal{D}(Fa, Gb)$$

类似地，在右边我们有 $Gf \circ \nu_a$ ，这是一个组合：

$$\mathcal{D}(Ga, Gb) \times \mathcal{D}(Fa, Ga) \rightarrow \mathcal{D}(Fa, Gb)$$

在富集设置中，我们必须使用同态对象而不是同态集，并且自然变换的分量选择是使用单位 I 完成的。我们总是可以使用左或右单位的逆来产生单位。

总的来说，自然性条件表示为以下交换图：

$$\begin{array}{ccccc}
 & & I \otimes \mathcal{C}(a, b) & \xrightarrow{\nu_b \otimes F_{ab}} & \mathcal{D}(Fb, Gb) \otimes \mathcal{D}(Fa, Fb) \\
 & \nearrow \lambda^{-1} & & & \searrow \circ \\
 \mathcal{C}(a, b) & & & & \mathcal{D}(Fa, Gb) \\
 & \searrow \rho^{-1} & & & \nearrow \circ \\
 & & \mathcal{C}(a, b) \otimes I & \xrightarrow{G_{ab} \otimes \nu_a} & \mathcal{D}(Ga, Gb) \otimes \mathcal{D}(Fa, Ga)
 \end{array}$$

这也适用于普通范畴，我们可以通过首先从 $\mathcal{C}(a, b)$ 中选择一个 f 来追踪这个图中的两条路径。然后我们可以使用 ν_b 和 ν_a 来选择自然变换的分量。我们还使用 F 或 G 来提升 f 。最后，我们使用组合来重现自然性方程。

如果我们使用我们之前定义的同态函子对同态对象的全局元素的作用，这个图可以进一步简化。自然变换的分量被定义为这样的全局元素：

$$\nu_a : I \rightarrow \mathcal{D}(Fa, Ga)$$

我们有两个这样的提升：

$$\mathcal{D}(d, \nu_b) : \mathcal{D}(d, Fb) \rightarrow \mathcal{D}(d, Gb)$$

和：

$$\mathcal{D}(\nu_a, d) : \mathcal{D}(Ga, d) \rightarrow \mathcal{D}(Fa, d)$$

我们得到的东西看起来更像熟悉的自然性方块：

$$\begin{array}{ccccc} & & \mathcal{D}(Fa, Fb) & & \\ & \nearrow^{F_{ab}} & & \searrow^{\mathcal{D}(Fa, \nu_b)} & \\ \mathcal{C}(a, b) & & & & \mathcal{D}(Fa, Gb) \\ & \searrow_{G_{ab}} & & \nearrow_{\mathcal{D}(\nu_a, Gb)} & \\ & & \mathcal{D}(Ga, Gb) & & \end{array}$$

两个 \mathcal{V} -函子 F 和 G 之间的 \mathcal{V} -自然变换形成一个集合，我们称之为 $\mathcal{V}\text{-nat}(F, G)$ 。

之前我们已经看到，在普通范畴中，自然变换的集合可以写为一个端（end）：

$$[\mathcal{C}, \mathcal{D}](F, G) \cong \int_a \mathcal{D}(Fa, Ga)$$

事实证明，端和余端（coend）可以定义为富集 profunctors，所以这个公式也适用于富集自然变换。不同的是，它不是自然变换的集合 $\mathcal{V}\text{-nat}(F, G)$ ，而是定义了 \mathcal{V} 中自然变换的对象 $[\mathcal{C}, \mathcal{D}](F, G)$ 。

\mathcal{V} -profunctor $P : \mathcal{C} \otimes \mathcal{C}^{op} \rightarrow \mathcal{V}$ 的（余）端的定义与我们看到的普通 profunctors 的定义类似。例如，端是 \mathcal{V} 中的一个对象 e ，配备了一个超自然变换 $\pi : e \rightarrow P$ ，它在这些对象中是普遍的。

21.4 Yoneda 引理

普通的 Yoneda 引理涉及一个 **Set**-值函子 F 和一组自然变换：

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(c, -), F) \cong Fc$$

为了将其推广到充实（enriched）设置中，我们将考虑一个 \mathcal{V} -值函子 F 。如前所述，我们将利用 \mathcal{V} 可以自充实的事实，只要它是闭的，因此我们可以讨论 \mathcal{V} -值的 \mathcal{V} -函子。

Yoneda 引理的弱版本处理的是 \mathcal{V} -自然变换的集合。因此，我们必须将右侧也转换为一个集合。这是通过取 Fc 的（么半群-全局）元素来实现的。我们得到：

$$\mathcal{V}\text{-nat}(\mathcal{C}(c, -), F) \cong \mathcal{V}(I, Fc)$$

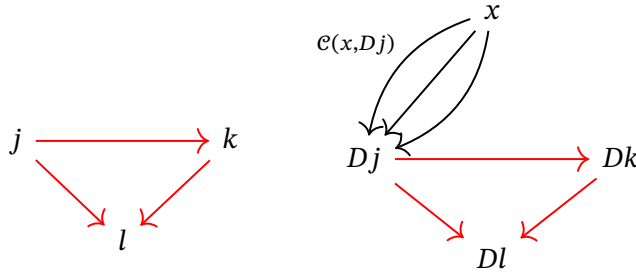
Yoneda 引理的强版本则处理 \mathcal{V} 的对象，并使用 \mathcal{V} 中内部 **hom** 的端 (end) 来表示自然变换的对象：

$$\int_x [\mathcal{C}(c, x), Fx] \cong Fc$$

21.5 加权极限

极限（和余极限）是建立在交换三角形的基础上的，因此它们不能直接转换到充实 (enriched) 环境中。问题在于锥体是由“导线”构建的，即单个态射。你可以将 **hom** 集想象为一束粗导线，每根导线的厚度为零。在构建锥体时，你从给定的 **hom** 集中选择一根导线。我们必须用更粗的东西来替换这些导线。

考虑一个图表，即从索引范畴 \mathcal{J} 到目标范畴 \mathcal{C} 的函子 D 。以 x 为顶点的锥体的导线是从 **hom** 集 $\mathcal{C}(x, Dj)$ 中选择的，其中 j 是 \mathcal{J} 的一个对象。



选择第 j 根导线可以描述为从单元素集 1 出发的函数：

$$\gamma_j : 1 \rightarrow \mathcal{C}(x, Dj)$$

我们可以尝试将这些函数收集成一个自然变换：

$$\gamma : \Delta_1 \rightarrow \mathcal{C}(x, D-)$$

其中 Δ_1 是一个常函子，它将 \mathcal{J} 的所有对象映射到单元素集。自然性条件确保了构成锥体侧面的三角形是交换的。

所有以 x 为顶点的锥体的集合由自然变换的集合给出：

$$[\mathcal{J}, \mathbf{Set}](\Delta_1, \mathcal{C}(x, D-))$$

这种重新表述使我们更接近充实环境，因为它用 **hom** 集而不是单个态射来重新表述问题。我们可以首先考虑 \mathcal{J} 和 \mathcal{C} 都在 \mathcal{V} 上充实，在这种情况下， D 将是一个 \mathcal{V} -函子。

只有一个问题：如何定义一个常 \mathcal{V} -函子 $\Delta_x : \mathcal{C} \rightarrow \mathcal{D}$ ？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 。但它对 **hom** 对象应该做什么？

普通的常函子 Δ_x 将 $\mathcal{C}(a, b)$ 中的所有态射映射到 $id_x \in \mathcal{D}(x, x)$ 中的恒等态射。然而，在充实环境中， $\mathcal{D}(x, x)$ 是一个没有内部结构的对象。即使它恰好是单位 I ，也不能保证对于每个 \mathbf{hom} 对象 $\mathcal{C}(a, b)$ ，我们都能找到一个到 I 的箭头；即使存在一个，它也可能不是唯一的。换句话说，没有理由相信 I 是终端对象。

解决方案是“涂抹奇点”：与其使用常函子选择一根导线，我们应该使用其他“加权”函子 $W: \mathcal{J} \rightarrow \mathbf{Set}$ 来选择 \mathbf{hom} 集内更粗的“圆柱体”。这种以 x 为顶点的加权锥体是自然变换集合中的一个元素：

$$[\mathcal{J}, \mathbf{Set}](W, \mathcal{C}(x, D-))$$

加权极限，也称为索引极限， $\lim^W D$ ，然后被定义为普遍的加权锥体。这意味着对于任何以 x 为顶点的加权锥体，存在一个从 x 到 $\lim^W D$ 的唯一态射，将其分解。分解由定义加权极限的同构的自然性保证：

$$\mathcal{C}(x, \lim^W D) \cong [\mathcal{J}, \mathbf{Set}](W, \mathcal{C}(x, D-))$$

通常的非加权极限通常被称为锥形极限，它对应于使用常函子作为权重。

这个定义几乎可以逐字翻译到充实环境中，只需将 \mathbf{Set} 替换为 \mathcal{V} ：

$$\mathcal{C}(x, \lim^W D) \cong [\mathcal{J}, \mathcal{V}](W, \mathcal{C}(x, D-))$$

当然，这个公式中符号的含义发生了变化。两边现在都是 \mathcal{V} 中的对象。左边是 \mathcal{C} 中的 \mathbf{hom} 对象，右边是两个 \mathcal{V} -函子之间的自然变换对象。

对偶地，加权余极限由自然同构定义：

$$\mathcal{C}(\mathrm{colim}^W D, x) \cong [\mathcal{J}^{op}, \mathcal{V}](W, \mathcal{C}(D-, x))$$

这里，余极限由从对偶范畴 $\mathcal{J}^{op} \rightarrow \mathcal{V}$ 的函子 W 加权。

加权（余）极限，无论是在普通范畴还是在充实范畴中，都起着基础性的作用：它们可以用来重新表述许多熟悉的构造，如（余）端、Kan 扩展等。

21.6 作为加权极限的端

端与积，或者更一般地说，与极限有许多共同之处。如果你仔细观察，投影 $\pi_x: e \rightarrow P(a, a)$ 构成了一个锥的边；只不过我们不再有交换三角形，而是有楔形。事实证明，我们可以将端表示为加权极限。这种表述的优势在于它在富化（enriched）环境中同样适用。

我们已经看到， \mathcal{V} 值的 \mathcal{V} -profunctor 的端可以使用更基本的概念——超自然变换（extranatural transformation）来定义。这反过来又使我们能够定义自然变换的对象，从而能够定义加权极限。现在我们可以继续扩展端的定义，使其适用于更一般的混合变异的 \mathcal{V} -函子，其值在 \mathcal{V} -范畴 \mathcal{D} 中：

$$P: \mathcal{C}^{op} \otimes \mathcal{C} \rightarrow \mathcal{D}$$

我们将使用这个函子作为 \mathcal{D} 中的图。

此时，数学家们开始担心大小问题。毕竟，我们将整个范畴——平方后——作为一个单一的图嵌入到 \mathcal{D} 中。为了避免大小问题，我们假设 \mathcal{C} 是小的；也就是说，它的对象形成一个集合。

我们希望取由 P 定义的图的加权极限。权重必须是一个 \mathcal{V} -函子 $\mathcal{C}^{op} \otimes \mathcal{C} \rightarrow \mathcal{V}$ 。有一个这样的函子总是可用的，即 **hom**-函子 $\text{Hom}_{\mathcal{C}}$ 。我们将使用它来将端定义为加权极限：

$$\int_{\mathcal{C}} P\langle c, c \rangle = \lim^{\text{Hom}} P$$

首先，让我们确信这个公式在普通的 (**Set**-富化) 范畴中有效。由于端是由它们的映射性质定义的，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** 技巧来展示同构。根据定义，我们有：

$$\mathcal{D}(d, \lim^{\text{Hom}} P) \cong [\mathcal{C}^{op} \times \mathcal{C}, \text{Set}](\mathcal{C}(-, =), \mathcal{D}(d, P(-, =)))$$

我们可以将自然变换的集合重写为对象对 $\langle c, c' \rangle$ 上的端：

$$\int_{\langle c, c' \rangle} \text{Set}(\mathcal{C}(c, c'), \mathcal{D}(d, P\langle c, c' \rangle))$$

使用 **Fubini** 定理，这等价于迭代端：

$$\int_{\mathcal{C}} \int_{\mathcal{C}'} \text{Set}(\mathcal{C}(c, c'), \mathcal{D}(d, P\langle c, c' \rangle))$$

我们现在可以应用 **ninja-Yoneda** 引理来对 c' 进行积分。结果是：

$$\int_{\mathcal{C}} \mathcal{D}(d, P\langle c, c \rangle) \cong \mathcal{D}(d, \int_{\mathcal{C}} P\langle c, c \rangle)$$

其中我们使用连续性将端推到 **hom**-函子下。由于 d 是任意的，我们得出结论，对于普通范畴：

$$\lim^{\text{Hom}} P \cong \int_{\mathcal{C}} P\langle c, c \rangle$$

这证明了我们在富化情况下使用加权极限来定义端的合理性。

类似的公式适用于余端，只不过我们使用相反范畴 $\text{Hom}_{\mathcal{C}^{op}}$ 中的 **hom**-函子作为权重的余极限：

$$\int^{\mathcal{C}} P\langle c, c \rangle = \text{colim}^{\text{Hom}_{\mathcal{C}^{op}}} P$$

Exercise 21.6.1. 证明对于普通的 **Set**-富化范畴，余端的加权余极限定义再现了之前的定义。提示：使用余端的映射性质。

Exercise 21.6.2. 证明，只要两边都存在，以下恒等式在普通的 (**Set**-富化) 范畴中成立（它们可以推广到富化环境中）：

$$\begin{aligned} \lim^W D &\cong \int_{j: \mathcal{J}} Wj \multimap Dj \\ \text{colim}^W D &\cong \int^{j: \mathcal{J}} Wj \cdot Dj \end{aligned}$$

提示：使用映射性质与 **Yoneda** 技巧以及幂和余幂的定义。

21.7 Kan 扩展

我们已经看到了如何使用从单点范畴 **1** 出发的函子将极限和余极限表示为 **Kan** 扩展。加权极限让我们摆脱了单点性，而权重的明智选择使我们能够用加权极限来表达 **Kan** 扩展。

首先，让我们推导出普通 **Set**-富范畴的公式。右 **Kan** 扩展定义为：

$$(\text{Ran}_P F)e \cong \int_c \mathcal{B}(e, Pc) \pitchfork Fc$$

我们将考虑从任意对象 d 到它的映射。推导过程遵循几个简单的步骤，主要是通过展开定义。

我们从：

$$\mathcal{D}(d, (\text{Ran}_P F)e)$$

开始，并代入 **Kan** 扩展的定义：

$$\mathcal{D}(d, \int_c \mathcal{B}(e, Pc) \pitchfork Fc)$$

利用 **hom**-函子的连续性，我们可以将 **end** 提取出来：

$$\int_c \mathcal{D}(d, \mathcal{B}(e, Pc) \pitchfork Fc)$$

然后我们使用 **pitchfork** 的定义：

$$\mathcal{D}(d, A \pitchfork d') \cong \mathbf{Set}(A, \mathcal{D}(d, d'))$$

得到：

$$\int_c \mathcal{D}(d, \mathcal{B}(e, Pc) \pitchfork Fc) \cong \int_c \mathbf{Set}(\mathcal{B}(e, Pc), \mathcal{D}(d, Fc))$$

这可以写为一组自然变换：

$$[\mathcal{C}, \mathbf{Set}](\mathcal{B}(e, P-), \mathcal{D}(d, F-))$$

加权极限也是通过自然变换的集合定义的：

$$\mathcal{D}(d, \lim^W F) \cong [\mathcal{C}, \mathbf{Set}](W, \mathcal{D}(d, F-))$$

这引导我们得到最终结果：

$$\mathcal{D}(d, \lim^{\mathcal{B}(e, P-)} F)$$

由于 d 是任意的，我们可以使用 **Yoneda** 技巧得出结论：

$$(\text{Ran}_P F)e = \lim^{\mathcal{B}(e, P-)} F$$

这个公式成为富范畴设置中右 **Kan** 扩展的定义。

类似地，左 **Kan** 扩展可以定义为加权余极限：

$$(\text{Lan}_P F)e = \text{colim}^{\mathcal{B}(P-, e)} F$$

Exercise 21.7.1. 推导普通范畴中左 **Kan** 扩展的公式。

21.8 常用公式

- 米田引理 (Yoneda lemma):

$$\int_x [\mathcal{C}(c, x), Fx] \cong Fc$$

- 加权极限 (Weighted limit):

$$\mathcal{C}(x, \lim^W D) \cong [\mathcal{J}, \mathcal{V}](W, \mathcal{C}(x, D-))$$

- 加权余极限 (Weighted colimit):

$$\mathcal{C}(\operatorname{colim}^W D, x) \cong [\mathcal{J}^{op}, \mathcal{V}](W, \mathcal{C}(D-, x))$$

- 右 Kan 延拓 (Right Kan extension):

$$(\operatorname{Ran}_P F)e = \lim^{\mathcal{B}(e, P-)} F$$

- 左 Kan 延拓 (Left Kan extension):

$$(\operatorname{Lan}_P F)e = \operatorname{colim}^{\mathcal{B}(P-, e)} F$$

依赖类型

我们已经见过依赖于其他类型的类型。它们是通过带有类型参数的类型构造器定义的，例如 `Maybe` 或 `[]`。大多数编程语言都对泛型数据类型（即由其他数据类型参数化的数据类型）提供了一定的支持。

从范畴论的角度来看，这些类型被建模为函子¹。

这一思想的自然推广是让类型由值来参数化。例如，将列表的长度编码在其类型中通常是有优势的。长度为 0 的列表与长度为 1 的列表将具有不同的类型，依此类推。

显然，你不能改变这种列表的长度，因为这会改变它的类型。这在函数式编程中不是问题，因为所有数据类型都是不可变的。当你向列表前添加一个元素时，至少在概念上，你创建了一个新列表。对于长度编码的列表，这个新列表具有不同的类型，仅此而已！

由值参数化的类型称为依赖类型。有些语言如 `Idris` 或 `Agda` 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相当零散。

在编程中使用依赖类型的原因是为了使程序可证明正确。为了实现这一点，编译器必须能够检查程序员所做的假设。

`Haskell` 凭借其强大的类型系统，能够在编译时发现许多错误。例如，除非你为变量的类型提供 `Monoid` 实例，否则它不会让你写 `a <> b`（`mappend` 的中缀表示法）。

然而，在 `Haskell` 的类型系统中，无法表达或强制执行幺半群的单位元和结合律。为此，`Monoid` 类型类的实例必须携带等式证明（不是实际代码）：

```
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。

¹没有 `Functor` 实例的类型构造器可以视为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

本章的内容较为高级，且未在本书的其余部分使用，因此你可以在初次阅读时安全地跳过它。此外，为了避免纤维和函数之间的混淆，我决定在本章的部分内容中使用大写字母表示对象。

22.1 依赖向量

我们将从计数列表或向量的标准示例开始：

```
data Vec n a where
  VNil  :: Vec Z a
  VCons :: a -> Vec n a -> Vec (S n) a
```

如果你包含以下语言编译指示，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`。注意：这是一个值，而不是一个类型。类型检查器能够从数据构造器中对 `n` 的使用中推断出这一点。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
```

如果我们使用编译指示：

```
{-# LANGUAGE KindSignatures #-}
```

并导入库：

```
import Data.Kind
```

我们可以更明确地指定 `n` 是 `Nat`，而 `a` 是 `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 0 的（整数）向量：

```
emptyV :: Vec Z Int
emptyV = VNil
```

它与长度为 1 的向量具有不同的类型：

```
singleV :: Vec (S Z) Int
singleV = VCons 42 VNil
```

依此类推。

我们现在可以定义一个依赖类型的函数，返回向量的第一个元素：

```
headV :: Vec (S n) a -> a
headV (VCons a _) = a
```

该函数保证仅适用于非零长度的向量。这些向量的大小匹配 $(S\ n)$ ，而 $(S\ n)$ 不能是 Z 。如果你尝试用 `emptyV` 调用此函数，编译器将标记错误。

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要求两个向量具有相同大小 n （结果也具有大小 n ）：

```
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特别有用。例如，列表的形状编码在其长度中。一个更高级的例子是将树的形状编码为运行时值。

Exercise 22.1.1. 实现函数 `tailV`，返回非零长度向量的尾部。尝试用 `emptyV` 调用它。