程序员的范畴论

由 Bartosz Milewski 编写

编译和编辑
Igal Tabachnik

程序员的范畴论

Bartosz Milewski

版本 v1.0.0-0-g41e0fc3
2018年10月21日

由 Bartosz Milewski 的一系列博客文章转换而来。
PDF 和书籍由 Igal Tabachnik 编译。

LaTeX 源代码可在 GitHub 上找到：
https://github.com/hmemcpy/milewski-ctfp-pdf

目录

前言

> 有一段时间了，我一直在考虑写一本关于范畴论的书，面向的对象是程序员。 注意，不是计算机科学家，而是程序员-工程师而不是科学家。 我知道这听起来很疯狂，我也很害怕。 我不能否认科学和工程之间存在着巨大的鸿沟，因为我在这两个领域都工作过。 但我一直有一种非常强烈的冲动去解释事物。 我非常钦佩理查德·费曼，他是简单解释的大师。 我知道我不是费曼，但我会尽力而为。 我从发布这个前言开始——它旨在激励读者学习范畴论——希望引发讨论并征求反馈。[1]

在几段文字的空间里，我将尝试说服你，这本书是为你而写的，无论你对在你的“大量空闲时间”学习最抽象的数学分支之一有什么异议，都是毫无根据的。

[1] 你也可以观看我在现场教授这个材料，网址为https://goo.gl/GT2UWU（或在YouTube上搜索“bartosz milewski category theory”）。

我的乐观主义基于几个观察。 首先，范畴论是一个极其有用的编程思想宝库。 Haskell程序员长期以来一直在利用这个资源，这些思想正在慢慢渗透到其他语言中，但这个过程太慢了。我们需要加快速度。

其次，数学有许多不同的种类，它们吸引不同的受众。 你可能对微积分或代数过敏，但这并不意味着你不会喜欢范畴论。 我甚至可以说，范畴论是特别适合程序员思维的数学。 这是因为范畴论不是处理具体事物，而是处理结构。 它处理的是使程序可组合的那种结构。

组合是范畴论的根本之一 - 它是范畴本身的一部分。 我将强烈主张，组合是编程的本质。 在某个伟大的工程师提出子程序的概念之前，我们一直在组合事物。 一段时间以前，结构化编程的原则彻底改变了编程，因为它们使代码块可组合。 然后出现了面向对象编程，它完全是关于组合对象的。 函数式编程不仅仅是关于组合函数和代数数据结构 - 它使并发可组合 - 这在其他编程范式中几乎是不可能的。

第三，我有一个秘密武器，一把屠夫刀，用它来屠杀数学，使其对程序员更易理解。 当你是一名专业的数学家时，你必须非常小心地澄清所有假设，正确地限定每个陈述，并严格构建所有证明。 这使得数学论文和书籍对外行人来说非常难以阅读。 我是一个受过物理学训练的人，而在

物理学家们在使用非正式推理方面取得了惊人的进展。 数学家们嘲笑狄拉克的δ函数，这是伟大的物理学家P.A.M.狄拉克即兴创造的，用于解决一些微分方程。 当他们发现一种完全新的微积分分支——分布理论，正式化了狄拉克的洞察力时，他们停止了嘲笑。

当然，在使用手势推理时，你有可能说出一些明显错误的话，所以我会确保在本书中，非正式论证背后有坚实的数学理论支持。 我在床头柜上放着一本破旧的Saunders Mac Lane的《工作数学家的范畴论》。

由于这是给程序员的范畴论，我将使用计算机代码来说明所有主要概念。 你可能已经意识到函数式语言比更流行的命令式语言更接近数学。 它们还提供了更多的抽象能力。 因此，一个自然的观点是：在范畴论的丰富性对你可用之前，你必须学习Haskell。 但这将意味着范畴论在函数式编程之外没有应用，这显然是不正确的。 所以我将提供很多C++的例子。 诚然，你将不得不克服一些丑陋的语法，模式可能不会从冗长的背景中脱颖而出，你可能会被迫进行一些复制和粘贴，而不是更高的抽象，但这就是C++程序员的命运。

但是对于Haskell来说，你还没有摆脱困境。 你不需要成为一个Haskell程序员，但你需要它作为一种语言来勾勒和记录在C++中实现的想法。这正是我开始使用Haskell的方式。我发现它简洁的语法和强大的类型系统对于理解和实现C++模板、数据结构和算法非常有帮助。 但是由于我不能期望

读者已经了解Haskell，所以我会慢慢介绍它并在进行时解释一切。

如果你是一名有经验的程序员，你可能会问自己：我已经编程很长时间了，从来不用担心范畴论或函数式方法，那么有什么改变吗？ 你肯定会注意到，命令式语言中出现了一系列新的函数式特性。 即使是Java，面向对象编程的堡垒，也让lambda函数进来了。C++最近一直在快速发展——每隔几年就有一个新的标准——试图跟上这个变化的世界。 所有这些活动都是为了准备一个颠覆性的变化，或者我们物理学家所说的相变。 如果你继续加热水，它最终会开始沸腾。 我们现在处于一个青蛙的位置，必须决定是继续在越来越热的水中游泳，还是开始寻找一些替代方案。

推动这一巨大变革的力量之一是多核革命。 盛行的编程范式，面向对象编程，在并发和并行领域中并没有任何优势，反而鼓励危险和有缺陷的设计。数据

隐藏，作为面向对象的基本前提，当与共享和变异相结合时，就成为数据竞争的食谱。 将互斥锁与其保护的数据结合起来的想法很好，但不幸的是，锁不能组合，锁隐藏使死锁更容易发生且更难调试。

即使在没有并发的情况下，软件系统的日益复杂也在测试命令式范式的可扩展性极限。 简单来说，副作用变得失控。 诚然，具有副作用的函数通常很方便且易于编写。

它们的效果原则上可以编码在它们的名称和注释中。 一个名为SetPassword或WriteFile的函数显然会改变某些状态并产生副作用，我们已经习惯处理这种情况。 只有当我们开始将具有副作用的函数与具有副作用的其他函数组合在一起时，以此类推，事情开始变得复杂。 副作用本身并不是坏的，而是它们被隐藏起来使得在更大的规模上无法管理。 副作用无法扩展，而命令式编程则与副作用息息相关。

硬件的变化和软件的日益复杂迫使我们重新思考编程的基础。就像欧洲伟大的哥特式大教堂的建造者一样，我们一直在将我们的技艺推向材料和结构的极限。 法国博韦大教堂是一座未完成的哥特式大教堂，见证了人类与限制的深刻斗争。 它原本打算打破所有先前的高度和轻盈记录，但却遭受了一系列的倒塌。 像铁棒和木支这样的临时措施使其不会解体，但显然有很多事情出了问题。 从现代的角度来看，让这么多哥特式建筑成功完成是一个奇迹。

2 http://zh.wikipedia.org/wiki/博韦大教堂

临时措施防止博韦大教堂倒塌。

没有现代材料科学、计算机建模、有限元分析和一般数学和物理的帮助，完成这些哥特式建筑是一个奇迹。 我希望未来的几代人能像我们在构建复杂的操作系统、Web服务器和互联网基础设施方面展示的编程技能一样钦佩。 而且，坦率地说，他们应该这样做，因为我们所有这些都是基于非常脆弱的理论基础。 如果我们想要前进，我们必须修复这些基础。

第一部分

1

范畴：组合的本质

范畴是一个令人尴尬地简单的概念。 一个范畴由对象和它们一个 之间的箭头组成。 这就是为什么范畴在图形上如此容易表示的原因。 一个对象可以被画成一个圆圈或一个点，而一个箭头就是一个箭头。 （为了多样化，我偶尔会把对象画成小猪，箭头画成烟花。）但是范畴的本质是组合。 或者，如果你愿意，组合的本质就是一个范畴。 箭头可以组合，所以如果你有一个从对象箭头到对象箭头的箭头，还有一个从对象箭头到对象箭头的箭头，那么必须有一个箭头——它们的组合——从对象箭头到对象箭头。

1.1 箭头作为函数

这已经太抽象了吗？不要绝望。让我们谈谈具体的事情。 把箭头，也被称为态射，看作是函数。 你有一个函数你有，它接受类型为你有的参数并返回一个你有。 你还有另一个函数你还，它接受一个你还并返回一个你还。

在一个范畴中，如果存在从 □到 □的箭头和从 □到 □的箭头，那么必须还存在从 □到 □的直接箭头，即它们的组合。 这个图示不是一个完整的范畴，因为它缺少单位态射（稍后会介绍）。

你可以通过将 □的结果传递给 □来组合它们。 你刚刚定义了一个新函数，它接受一个 □并返回一个 □。

在数学中，这种组合用一个小圆圈表示函数之间的关系：□ ∘ □。 注意组合的顺序是从右到左的。 对于一些人来说，这可能会令人困惑。 你可能熟悉Unix中的管道符号，例如：

```
lsof | grep Chrome
```

或者F#中的箭头 >>，它们都是从左到右的。 但在数学和Haskell中，函数是从右到左组合的。 如果你将□ ∘ □读作“f之后g”，会更容易理解。

让我们通过编写一些C代码来更加明确。 我们有一个函数 f，它接受类型为 A的参数并返回类型为 B的值：

```
B f(A a);
```

还有一个函数：

```
C g(B b);
```

它们的组合是：

```
C g_after_f(A a)
{
        return g(f(a));
}
```

在这里，你再次看到了从右到左的组合： g(f(a))；这次是在C中。

我希望我能告诉你C++标准库中有一个模板，它接受两个函数并返回它们的组合，但实际上并没有。所以让我们尝试一些Haskell来改变一下。 这是一个从A到B的函数声明：

```
f :: A -> B
```

类似地：

```
g :: B -> C
```

它们的组合是：

```
g . f
```

一旦你看到Haskell中的简单事物，就会对在C++中无法表达直观的函数概念感到有些尴尬。 事实上，Haskell允许你使用Unicode字符，这样你就可以将组合写成：

```
g ∘ f
```

你甚至可以使用Unicode的双冒号和箭头：

f ∷ A → B

所以这是第一个Haskell课程：双冒号表示“具有类型…”
函数类型是通过在两个类型之间插入箭头来创建的。
通过在它们之间插入一个句点（或Unicode圆圈）来组合两个函数。

1.2 组合的属性

在任何范畴中，组合必须满足两个极其重要的性质。

1. 组合是可结合的。 如果你有三个可以组合的态射，□,□和*h*（即，它们的对象一一对应），你不需要使用括号来组合它们。 在数学符号中，这可以表示为：

$$h \circ (□ \circ □) = (h \circ □) \circ □ = h \circ □ \circ □$$

在（伪）Haskell中：

```
f :: A -> B
g :: B -> C
h :: C -> D
h . (g . f) == (h . g) . f == h . g . f
```

（我说“伪”，因为函数没有定义相等性。）在处理函数时，结合性是相当明显的，但在其他范畴中可能不太明显。

2. 对于每个对象 □，都存在一个箭头，它是一个组合的单位。这个箭头从对象指向自身。 作为组合的单位意味着，当与任何以 □为起点或终点的箭头组合时，它会返回相同的箭头。对象A的单位箭头被称为 id □（在□上的恒等映射）。 在数学符号中，如果 □从 □到 □，则

$$□ \circ \mathrm{id}_{□} = □$$

和

$$\mathrm{id}_{□} \circ □ = □$$

在处理函数时，恒等箭头被实现为只返回其参数的恒等函数。 实现对于每种类型都是相同的，这意味着该函数是普遍多态的。 在C++中，我们可以将其定义为一个模板：

```
template<class T> T id(T x) { return x; }
```

当然，在C++中没有那么简单，因为你不仅要考虑传递的内容，还要考虑传递的方式（即按值传递、按引用传递、按常量引用传递、按移动等等）。

在Haskell中，恒等函数是标准库的一部分（称为Prelude）。这是它的声明和定义：

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

正如你所看到的，Haskell中的多态函数非常简单。 在声明中，你只需用类型变量替换类型。 这是

诀窍：具体类型的名称始终以大写字母开头，类型变量的名称以小写字母开头。

所以这里的 a代表所有
types。 Haskell函数定义由函数名称和形式参数组成-这里只有一个，x。

函数的主体跟在等号后面。 这种简洁性对于新手来说常常令人震惊，但你很快就会发现它完全合理。 函数定义和函数调用是函数式编程的基础，因此它们的语法被简化到最低限度。 不仅参数列表周围没有括号，而且参数之间也没有逗号（当我们定义多个参数的函数时，你会看到后面的情况）。 函数定义和函数调用是函数式编程的基础，因此它们的语法被简化到最低限度。

函数的主体始终是一个表达式 - 函数中没有语句。 函数的结果就是这个表达式 - 在这里，只是简单的x。

这就结束了我们的第二个Haskell课程。
身份条件可以写成（再次，伪Haskell语法）：

```
f . id == f
id . f == f
```

你可能会问自己一个问题：为什么有人要费心去使用一个什么都不做的身份函数？ 然而，为什么我们要费心去使用零这个数字呢？ 零是一个代表无的符号。
古罗马人没有零的数字系统，但他们能够建造出优秀的道路和渡槽，其中一些至今仍然存在。

像零或 id这样的中性值在使用符号变量时非常有用。 这就是为什么罗马人在代数方面不太擅长，而熟悉零的阿拉伯人和波斯人却很擅长。 因此，身份函数非常方便。

作为高阶函数的参数或返回值。 高阶函数是使函数的符号操作成为可能的关键。
它们是函数的代数。

总结一下：一个范畴由对象和箭头（态射）组成。
箭头可以组合，而且组合是可结合的。 每个对象都有一个作为组合单位的身份箭头。

1.3 组合是编程的本质

函数式程序员有一种特殊的解决问题的方式。
他们首先提出非常禅意的问题。 例如，在设计交互式程序时，他们会问：什么是交互？
在实现康威的生命游戏时，他们可能会思考生命的意义。 本着这种精神，我要问：什么是编程？ 在最基本的层面上，编程是告诉计算机要做什么。“取出内存地址x的内容，并将其添加到寄存器EAX的内容中。”但即使在汇编语言中编程，我们给计算机的指令也是更有意义的表达。 我们正在解决一个非平凡的问题（如果它是平凡的，我们就不需要计算机的帮助）。 那么我们如何解决问题呢？ 我们将更大的问题分解为更小的问题。 如果这些较小的问题仍然太大，我们会进一步分解，以此类推。最后，我们编写解决所有小问题的代码。

然后编程的本质就是将那些代码片段组合起来解决更大的问题。
如果我们不能将这些片段重新组合起来，分解就没有意义。

这种层次分解和重新组合的过程并不是计算机强加给我们的。 它反映了人类思维的局限性。 我们的大脑一次只能处理少量的概念。

时间。 心理学中最常引用的论文之一，《神奇的数字七加减二》1，假设我们的大脑只能在短期记忆中保留7±2个“块”信息。 我们对人类短期记忆的理解细节可能在改变，但我们确切知道它是有限的。 底线是我们无法处理对象的混乱或代码的混乱。 我们需要结构，不是因为结构良好的程序看起来舒服，而是因为否则我们的大脑无法高效处理它们。 我们经常描述某段代码为优雅或美丽，但我们真正的意思是它对我们有限的人类思维来说易于处理。 优雅的代码创造出适合我们的心智消化系统吸收的恰到好处的块。

那么程序的组合应该使用哪些正确的块？ 它们的表面积增长速度必须比它们的体积慢。 （我喜欢这个类比，因为几何物体的表面积随着尺寸的平方增长——比体积增长得慢，这样可以直观地理解。）表面积是我们组合块所需的信息。 体积是我们实现它们所需的信息。 这个想法是，一旦一个块被实现，我们就可以忘记它的实现细节，集中精力研究它与其他块的交互。 在面向对象编程中，表面是对象的类声明，或者是它的抽象接口。

在函数式编程中，它是一个函数的声明。 （我稍微简化了一些，但这就是要点。）

范畴论在某种程度上是极端的，因为它积极阻止我们查看对象的内部。 在范畴论中，对象是一个抽象的模糊实体。 你能知道的关于它的一切就是它与其他事物的关系。

1http://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

与其他对象之间的连接方式，使用箭头表示。 这就是互联网搜索引擎通过分析传入和传出链接（除非它们作弊）来对网站进行排名的方式。 在面向对象编程中，理想化的对象只能通过其抽象接口（纯表面，无实体）可见，其中方法扮演箭头的角色。 一旦你必须深入对象的实现才能理解如何与其他对象组合，你就失去了编程范式的优势。

1.4 挑战

1. 尽可能在你最喜欢的编程语言中实现恒等函数（如果你最喜欢的语言是Haskell，则使用第二喜欢的语言）。

2. 在你最喜欢的编程语言中实现组合函数。 它接受两个函数作为参数，并返回它们的组合函数。

3. 编写一个程序来测试你的组合函数是否遵守恒等性。

4. 世界范围内的网络在任何意义上都是一个范畴吗？链接是态射吗？
5. Facebook是一个范畴吗？人作为对象，友谊作为态射？

6. 何时一个有向图是一个范畴？

2

类型和函数

T类型和函数的范畴在编程中起着重要的作用，所以让我们谈谈类型是什么以及为什么我们需要它们。

2.1 谁需要类型?

关于静态与动态类型和强类型与弱类型的优势似乎存在一些争议。 让我用一个思维实验来说明这些选择。 想象一下数百万只猴子在电脑键盘上快乐地随机按键，产生程序，编译并运行它们。

使用机器语言，猴子产生的任何字节组合都会被接受和运行。但是对于高级语言，我们确实欣赏编译器能够检测词法和语法错误的事实。 很多猴子将没有香蕉，但剩下的程序有更好的机会变得有用。类型检查

为非常荒谬的程序提供了另一个屏障。 此外，在动态类型语言中，类型不匹配会在运行时被发现，而在强类型静态检查的语言中，类型不匹配会在编译时被发现，在程序运行之前就消除了许多不正确的程序。所以问题是，我们是想让猴子开心，还是想产生正确的程序？

打字猴思想实验中的通常目标是创作出莎士比亚的完整作品。在这个过程中加入拼写检查和语法检查器会大大增加成功的几率。
类型检查器的类比甚至可以更进一步，确保一旦罗密欧被声明为人类，他就不会长出叶子或者在他强大的引力场中捕获光子。

2.2 类型与可组合性

范畴论是关于组合箭头的。 但并不是任意两个箭头都可以组合。一个箭头的目标对象必须与另一个箭头的相同。

下一个箭头的源对象。 在编程中，我们将一个函数的结果传递给另一个函数。 如果目标函数无法正确解释源函数产生的数据，程序将无法工作。 两个端点必须匹配才能进行组合。 语言的类型系统越强大，描述和机械验证匹配的能力就越好。

我听到的唯一一个严肃的反对强静态类型检查的论点是，它可能会消除一些在语义上正确的程序。
实际上，这种情况极为罕见，并且无论如何，每种语言都提供了一种绕过类型系统的后门，当真正需要时可以使用。 即使是 Haskell 也有unsafeCoerce。 但是这样的设备应该谨慎使用。 弗朗茨·卡夫卡的角色格雷戈尔·萨姆萨在变成一只巨大的虫子时打破了类型系统，我们都知道这是怎么结束的。

我经常听到的另一个论点是，处理类型对程序员来说负担太重。 在我自己用 C++ 编写了几个迭代器声明之后，我可以理解这种情绪，除非有一种叫做类型推断的技术，它可以从上下文中推断出大部分类型。

在 C++ 中，现在可以声明一个变量为 auto，让编译器自己确定其类型。
在Haskell中，除了极少数情况，类型注解是完全可选的。程序员倾向于使用它们，因为它们可以告诉代码的语义，并且可以更容易地理解编译错误。 在Haskell中，以设计类型开始一个项目是一种常见做法。 后来，类型注解驱动实现，并成为编译器强制执行的注释。

强静态类型经常被用作不测试代码的借口。 你可能会听到Haskell程序员说：“如果它编译通过，那就是正确的。”当然，没有保证一个类型正确的程序在产生正确输出方面是正确的。

这种漫不经心的态度的结果是，在几项研究中，Haskell在代码质量方面并没有像人们期望的那样明显领先。
看起来，在商业环境中，修复错误的压力只适用于一定的质量水平，这与软件开发的经济学和最终用户的容忍度有关，与编程语言或方法学关系很小。一个更好的标准是衡量有多少项目进度落后或以大幅度降低的功能交付。 一个更好的标准是衡量有多少项目进度落后或以大幅度降低的功能交付。

至于单元测试可以替代强类型的论点，考虑一下在强类型语言中常见的重构实践：更改特定函数的参数类型。 在强类型语言中，只需修改该函数的声明，然后修复所有构建错误即可。 在弱类型语言中，函数现在期望不同的数据类型，这一事实无法传播到调用点。 单元测试可能会捕捉到一些不匹配，但测试几乎总是一种概率性而非确定性的过程。 测试不能替代证明。

2.3 什么是类型?

类型的最简单直觉是它们是值的集合。 类型Bool（记住，在Haskell中，具体类型以大写字母开头）是一个包含 True和 False的两个元素集合。 类型 Char是包含所有Unicode字符（如 a或 ą）的集合。

集合可以是有限的或无限的。 类型 String（它是 Char列表的同义词）是一个无限集合的例子。当我们声明 x为 Integer时：

x :: 整数

我们说它是整数集的一个元素。在Haskell中，整数是一个无限集，可以用来进行任意精度的算术运算。 还有一个有限集 Int对应于机器类型，就像C++中的int一样。

有一些微妙之处使得类型和集合的这种对应关系变得棘手。 涉及循环定义的多态函数存在问题，而且你不能拥有一个包含所有集合的集合；但是，正如我所承诺的，我不会对数学太过苛刻。 最棒的是，有一个叫做 □ □ 的集合范畴，我们将与之一起工作。 在 □ □ 中，对象是集合，态射（箭头）是函数。

□ □ 是一个非常特殊的范畴，因为我们实际上可以窥视其对象并从中获得很多直觉。 例如，我们知道空集没有元素。 我们知道存在特殊的单元素集。 我们知道函数将一个集合的元素映射到另一个集合的元素。它们可以将两个元素映射到一个元素，但不能将一个元素映射到两个元素。我们知道恒等函数将集合的每个元素映射到其自身，等等。计划是逐渐忘记所有这些信息，而是纯粹地用范畴论术语来表达所有这些概念，即用对象和箭头来表达。

"在理想世界中，我们可以说Haskell类型是集合，Haskell函数是集合之间的数学函数。 "只有一个小问题：数学函数不执行任何代码，它只知道答案。 "Haskell函数必须计算答案。 "如果答案可以在有限步骤内获得，那就不是问题了，无论这个步骤有多大。但是有一些计算涉及到递归，而这些计算可能永远不会终止。 "我们不能只是禁止Haskell中的非终止函数，因为区分终止和非终止函数是不可判定的——这就是著名的停机问题。

"这就是计算机科学家提出的一个绝妙的想法，或者说一个重要的技巧，取决于你的观点，通过在每种类型中添加一个特殊值来扩展，称为底部（bot-tom），用_|_或Unicode ⊥表示。 "这个“值”对应着一个非终止的计算。 "因此，一个声明为： "

```
f :: Bool -> Bool
```

可能返回 True、 False或 _|_；后者意味着它永远不会终止。

有趣的是，一旦你接受底部作为类型系统的一部分，将每个运行时错误都视为底部并允许函数显式返回底部会变得很方便。 后者通常使用表达式 undefined来完成，如下所示：

```
f :: Bool -> Bool
f x = undefined
```

这个定义可以通过类型检查，因为 undefined求值为底部，而底部是任何类型的成员，包括 Bool。你甚至可以这样写：

```
f :: Bool -> Bool
f = undefined
```

（不带 x）因为底部也是该类型的成员
Bool -> Bool。

可能返回底部的函数被称为部分函数，与返回每个可能参数的有效结果的全函数相对。

由于底部的存在，你会看到Haskell类型和函数的范畴被称为 H ask而不是 □ □ ⊿□ 从理论角度来看，这是无休止的复杂性的根源，所以在这一点上

我将使用我的屠夫刀终止这条推理线。 从实用的角度来看，忽略非终止的函数和底部，并将Hask视为真正的从实从实从实。[1]

2.4 为什么我们需要一个数学模型?

作为程序员，你对编程语言的语法和语法非常熟悉。 通常使用正式符号来描述语言的这些方面在语言规范的最开始。 但是语言的含义或语义要更难描述；它需要更多的页面，很少是足够正式的，几乎从不完整。 因此，在语言律师之间永无止境的讨论，以及整个专门研究语言标准细微之处的书籍产业。

有形式工具来描述语言的语义，但是由于它们的复杂性，它们主要用于简化的学术语言，而不是现实生活中的编程巨兽。 一种名为操作语义的工具描述了程序执行的机制。 它定义了一个形式化的理想化解释器。 工业语言（如C++）的语义通常使用非正式的操作推理来描述，通常是以“抽象机器”的形式。问题在于使用操作语义很难证明关于程序的事情。 要展示程序的一个属性，你基本上必须通过理想化的解释器“运行”它。

程序员从不进行形式证明的正确性并不重要。 我们总是“认为”我们编写的程序是正确的。没有人

[1]Nils Anders Danielsson，John Hughes，Patrik Jansson，Jeremy Gibbons，快速和松散的推理在大多数情况下是道德正确的。本文为在大多数情况下忽略底部提供了理由。

坐在键盘前说：“哦，我只是写几行代码看看会发生什么。”我们认为我们编写的代码将执行某些操作，产生期望的结果。 当它没有时，我们通常会感到非常惊讶。 这意味着我们确实对我们编写的程序进行推理，通常是通过在我们的头脑中运行解释器来进行。 只是真的很难跟踪所有的变量。 计算机擅长运行程序 - 人类不擅长！如果我们擅长，我们就不需要计算机。

但是还有一种选择。 它被称为指称语义它是基于数学的。 在指称语义中，每个编程构造都被赋予其数学解释。 有了这个，如果你想证明程序的一个属性，你只需要证明一个数学定理。 你可能认为定理证明很难，但事实上，我们人类已经积累了数千年的数学方法，所以有丰富的知识可以利用。 此外，与专业数学家证明的定理相比，我们在编程中遇到的问题通常相当简单，如果不是微不足道的。

考虑在Haskell中定义一个阶乘函数，这是一种非常适合指示语义的语言：

```
fact n = product [1..n]
```

表达式 `[1..n]`是一个从 `1`到 `n`的整数列表。 函数`product`将列表的所有元素相乘。 这就像是从数学教材中引用的阶乘定义。与C语言进行比较：

```
int fact(int n) {
        int i;
        int result = 1;
        for (i = 2; i <= n; ++i)
                        result *= i;
```

```
    return result;
}
```

还需要我多说吗?

好吧，我承认这是一个小小的挑衅！阶乘函数具有明显的数学含义。 一个敏锐的读者可能会问：从键盘读取字符或发送网络数据包的数学模型是什么？ 很长一段时间以来，这将是一个尴尬的问题，需要进行复杂的解释。 似乎指示语义并不适用于许多重要任务，这些任务对于编写有用的程序至关重要，并且可以通过操作语义轻松解决。 范畴论为此提供了突破。 Eugenio Moggi发现计算效果可以映射到单子上。这证明是一个重要的观察结果，不仅为指示语义带来了新的生机，使纯函数式程序更易用，而且为传统编程提供了新的视角。 稍后我会讨论单子，当我们开发更多范畴工具时。

拥有一个数学模型来进行编程的重要优势之一是可以对软件的正确性进行形式化证明。 当你编写消费者软件时，这可能看起来并不那么重要，但在编程的某些领域，失败的代价可能是巨大的，或者人类生命可能受到威胁。

即使在为卫生系统编写Web应用程序时，您也可能会欣赏到来自Haskell标准库的函数和算法具有正确性证明的思想。

2.5 纯函数和脏函数

我们在C++或任何其他命令式语言中称之为函数的东西，并不是数学家所称之为函数的东西。 数学函数只是一个值到值的映射。

我们可以在编程语言中实现数学函数：这样一个函数，给定一个输入值，将计算出输出值。 一个用于生成一个数的平方的函数可能会将输入值乘以自身。 每次调用时，它都会执行相同的操作，并且保证每次使用相同的输入调用时产生相同的输出。 一个数的平方不会随着月亮的相位而改变。

此外，计算一个数的平方不应该有一个分发美味零食给你的狗的副作用。这样的“函数”不能很容易地建模为数学函数。

在编程语言中，总是给定相同的输入产生相同结果且没有副作用的函数被称为纯函数。 在像Haskell这样的纯函数式语言中，所有函数都是纯函数。
正因为如此，给这些语言赋予指示性语义并使用范畴论对其进行建模更容易。 至于其他语言，总是可以限制自己使用纯子集，或者单独考虑副作用。 稍后我们将看到，单子让我们能够使用纯函数模拟各种效果。 因此，通过限制自己使用数学函数，我们并不会失去任何东西。

2.6 类型的例子

一旦你意识到类型就是集合，你就可以思考一些相当奇特的类型。 例如，对应于空集的类型是什么？不是

虽然这种类型在Haskell中被称为Void，但它并不是C++的void。它是一种没有任何值的类型。 你可以定义一个接受Void的函数，但你永远无法调用它。要调用它，你必须提供一个Void类型的值，但实际上并没有这样的值。至于这个函数可以返回什么类型，没有任何限制。 它可以返回任何类型（尽管它永远不会，因为它无法被调用）。 换句话说，它是一个在返回类型上具有多态性的函数。 Haskeller们给它起了一个名字：

```
absurd :: Void -> a
```

（记住，a是一个可以代表任何类型的类型变量。）这个名字并非巧合。在逻辑学中，有一个更深层次的解释，称为Curry-Howard同构。 类型Void代表虚假，函数absurd的类型对应于“ex falso sequitur quodlibet”这句拉丁谚语中的从虚假中得出任何结论的陈述。

接下来是对应于单例集合的类型。它是一个只有一个可能值的类型。 这个值就是“存在”。你可能不会立即认识到它，但这就是C++的void类型。 想象一下从这个类型到其他类型的函数。 从void到其他类型的函数总是可以调用的。 如果它是一个纯函数，它将始终返回相同的结果。 这是一个这样的函数的例子：

```
int f44() { return 44; }
```

你可能认为这个函数接受“什么都不需要”，但正如我们刚刚看到的，一个接受“什么都不需要”的函数永远不会被调用，因为没有表示“什么都不需要”的值。那么这个函数接受什么呢？ 从概念上讲，它接受一个虚拟值，这个值只有一个实例，所以我们不需要明确地提及它。然而，在Haskell中，有一个表示这个值的符号：一个空的括号对，

因此，由于一个有趣的巧合（或者是巧合吗？），对void类型的函数的调用在C++和Haskell中看起来是相同的。此外，由于Haskell喜欢简洁，同样的符号()被用于类型、构造函数和对应于单例集合的唯一值。所以这是这个函数在Haskell中的表示： 所以，由于一个有趣的巧合（或者是巧合吗？），对void类型的函数的调用在C++和Haskell中看起来是相同的。此外，由于Haskell喜欢简洁，同样的符号()被用于类型、构造函数和对应于单例集合的唯一值。所以这是这个函数在Haskell中的表示

```
f44 :: () -> Integer
f44 () = 44
```

第一行声明了 f44的类型为 ()，发音为“unit”，返回类型为 Integer。 第二行通过模式匹配定义了 f44，它只有一个构造器，即 ()，并返回数字44。 你可以通过提供 ()作为参数来调用这个函数：

```
f44 ()
```

注意，每个unit类型的函数都等价于从目标类型（这里是 Integer）中选择一个元素。 实际上，你可以将 f44看作数字44的另一种表示。 这是一个例子，说明我们可以用函数（箭头）来替代对集合元素的显式提及。 从unit到任意类型 □的函数与该集合 □的元素一一对应。

那么对于返回类型为 void或者在Haskell中为unit的函数呢？ 在C++中，这样的函数用于副作用，但我们知道这些在数学意义上并不是真正的函数。 返回unit的纯函数什么都不做：它会丢弃它的参数。

在数学上，从集合 □到单元素集合的函数将 □的每个元素映射到该单元素集合的单个元素。对于每个□，都存在一个这样的函数。 这是整数的函数：

```
fInt :: 整数 -> ()
fInt x = ()
```

你给它任何整数，它都会给你一个单位。 为了简洁起见，Haskell允许您使用通配符模式，即下划线，表示被丢弃的参数。 这样，您就不必为其发明一个名称。因此，上述代码可以重写为：

```
fInt :: 整数 -> ()
fInt _ = ()
```

请注意，此函数的实现不仅不依赖于传递给它的值，而且甚至不依赖于参数的类型。

可以使用相同的公式为任何类型实现的函数称为参数多态函数。 您可以使用类型参数而不是具体类型，在一个方程中实现整个这样的函数族。 我们应该如何称呼从任何类型到单位类型的多态函数？ 当然，我们将其称为单位：

```
unit :: a -> ()
unit _ = ()
```

在C++中，你会这样写这个函数：

```
template<class T>
void unit(T) {}
```

在类型学中，下一个是一个二元集合。在C++中，它被称为 bool，在Haskell中，可预见地是 Bool。 区别在于，在C++中 bool 是一个内置类型，而在Haskell中可以定义如下：

```
data Bool = True | False
```

（阅读这个定义的方式是 Bool要么是 True要么是 False。）原则上，你也可以在C++中定义一个布尔类型作为一个枚举：

```
enum bool {
    true,
    false
};
```

但是C++ enum实际上是一个整数。 C++11的“enum class”本来可以使用，但是你必须用类名限定它的值，如 bool::true和bool::false，更不用说在每个使用它的文件中都要包含适当的头文件了。从 Bool到纯函数只需从目标类型中选择两个值，一个对应 True，另一个对应 False。

将函数映射到 Bool被称为谓词。 例如，Haskell库 Data.Char充满了像 isAlpha或 isDigit这样的谓词。 在C++中，有一个类似的库，其中定义了 isalpha和 isdigit等函数，但是这些函数返回的是一个整数，而不是布尔值。实际的谓词是在 std::ctype中定义的，其形式为ctype::is(alpha, c)， ctype::is(digit, c)等等。

2.7 挑战

1. 在你喜欢的编程语言中定义一个高阶函数（或函数对象） memoize。 这个函数接受一个纯函数 f作为参数，并返回一个行为几乎与 f完全相同的函数，只是它只对每个参数调用一次原始函数

，将结果存储在内部，并在以相同参数调用时返回此存储的结果。
你可以通过观察其性能来区分记忆化函数和原始函数。 例如，尝试记忆化一个需要很长时间计算的函数。 第一次调用时，你将不得不等待结果，但是在后续调用中，使用相同的参数，你应该立即得到结果。

2. 尝试对你通常用来生成随机数的标准库函数进行记忆化。它有效吗?
3. 大多数随机数生成器可以使用种子进行初始化。
实现一个函数，它接受一个种子，使用该种子调用随机数生成器，并返回结果。对该函数进行记忆化。它有效吗?

4. 以下哪些C++函数是纯函数？ 尝试对它们进行记忆化，并观察在多次调用时会发生什么：记忆化和非记忆化。

(a) 文本中示例中的阶乘函数。
(b) std::getchar()
(c) bool f() {
```
    std::cout << "Hello!" << std::endl;
    return true;
}
```
(d) int f(int x) {
```
    static int y = 0;
    y += x;
    return y;
}
```

5. 从 Bool到 Bool有多少个不同的函数？ 你能实现它们吗?

6. 画出一个范畴的图，该图的唯一对象是类型Void, ()(unit)和 Bool；箭头对应于这些类型之间的所有可能函数。 用函数的名称标记箭头。

3

伟大和小的范畴

是通过研究各种示例，你可以真正欣赏范畴。 范畴以各种形状和大小存在，并经常出现在意想不到的地方。 我们将从非常简单的东西开始。

3.1 没有对象

最简单的范畴是一个没有对象和因此没有态射的范畴。 它本身是一个非常悲伤的范畴，但在其他范畴的上下文中可能是重要的，例如在所有范畴的范畴中（是的，存在这样一个范畴）。 如果你认为空集合有意义，那么为什么不考虑一个空范畴呢？

3.2 简单图

你可以通过连接对象和箭头来构建范畴。 你可以想象从任何有向图开始，并通过简单地添加更多箭头将其变成一个范畴。 首先，在每个节点上添加一个恒等箭头。
然后，对于任意两个箭头，使得一个箭头的末端与另一个箭头的起始点重合（换句话说，任意两个可组合的箭头），添加一个新的箭头作为它们的组合。 每次添加一个新的箭头时，你还必须考虑它与任何其他箭头（除了恒等箭头）以及它自身的组合。 通常会得到无限多的箭头，但这没关系。

从另一个角度来看，你正在创建一个范畴，它对应图中的每个节点，以及所有可能的可组合图边的链作为态射。 （你甚至可以将恒等态射视为长度为零的链的特例。）这样的范畴被称为由给定图生成的自由范畴。

这是一个自由构造的例子，通过扩展给定结构并添加最少数量的项来满足其规则（在这里是范畴的规则）。 我们将在将来看到更多的例子。

3.3 顺序

现在来完全不同的东西！一个范畴，其中态射是对象之间的关系：小于或等于的关系。让我们检查它是否确实是一个范畴。 我们有恒等态射吗？ 每个对象都小于或等于自身：检查！我们有组合吗？ 如果□□□且□□□那么□□□：检查！组合是否为-

结合性？ 检查！具有这样关系的集合被称为预序，因此预序确实是一个范畴。

你还可以有一个更强的关系，它满足一个额外的条件，即如果 □ □ □和□ □ □，那么 □必须等于 □。 这被称为偏序。

最后，你可以强加任意两个对象之间存在关系的条件；这给你一个线性顺序或者全序。

让我们将这些有序集合描述为范畴。 预序是一个范畴，其中从任意对象 □到任意对象 □最多只有一个态射。 这样的范畴还有另一个名称，叫做“薄”。预序是一个薄范畴。

在范畴 □中，从对象 □到对象 □的态射集合被称为同态集，并且写作 □(□, □)（有时也写作 $\mathrm{Hom}_{□}(□, □)$）。 因此，在预序中，每个同态集要么为空集，要么是一个单元素集。 这包括同态集 □(□, □)，即从 □到 □的态射集合，在任何预序中都必须是一个单元素集，只包含恒等态射。 然而，在预序中可能存在循环。 在偏序中是禁止循环的。

识别预序、偏序和全序非常重要，因为涉及到排序。 排序算法，如快速排序、冒泡排序、归并排序等，只能在全序上正确工作。 偏序可以使用拓扑排序进行排序。

3.4 幺半群作为集合

幺半群是一个令人尴尬地简单但令人惊叹的概念。
它是基本算术的概念背后的原理：加法和乘法都构成一个幺半群。 幺半群在编程中无处不在。

它们出现在字符串、列表、可折叠的数据结构、并发编程中的未来、函数响应式编程中的事件等等。传统上，幺半群被定义为具有二元操作的集合。

这个操作所要求的仅仅是它是可结合的，并且存在一个特殊元素，它在这个操作下表现得像一个单位元素。例如，带有零的自然数构成了一个加法幺半群。 可结合性意味着：

$$(a + b) + c = a + (b + c)$$

（换句话说，我们在加法时可以省略括号。）中性元素是零，因为：

$$0 + a = a$$

和

$$a + 0 = a$$

第二个方程式是多余的，因为加法是可交换的 $(a+b = b + a)$，但可交换性不是幺半群的定义的一部分。例如，字符串连接不是可交换的，但它仍然形成一个幺半群。 顺便说一下，字符串连接的中性元素是一个空字符串，它可以附加到字符串的任一侧而不改变它。

在Haskell中，我们可以为幺半群定义一个类型类——一个具有中性元素 mempty和二元操作的类型 mappend:

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
```

一个带有两个参数的函数的类型签名 m -> m -> m，一开始可能看起来很奇怪，但在我们讨论柯里化之后，它将变得非常清晰。 你可以将具有多个箭头的签名解释为两种基本方式：作为具有多个参数的函数，其中最右边的类型是返回类型；或者作为一个参数（最左边的参数）的函数，返回一个函数。 后一种解释可以通过添加括号来强调（尽管括号是多余的，因为箭头是右结合的），如： m -> (m -> m)。 我们稍后会回到这种解释。

请注意，在Haskell中，没有办法表达幺半群的性质（即，mempty是中性元素，mappend是可结合的）。 程序员有责任确保它们被满足。

Haskell类不像C++类那样具有侵入性。 当你定义一个新类型时，你不必事先指定它的类。 你可以自由地拖延并在以后的某个时候声明给定类型是某个类的实例。 举个例子，让我们通过提供mempty和mappend的实现来声明String是一个幺半群（实际上，在标准Prelude中已经为你做了这个）：

```
instance Monoid String where
    mempty = ""
    mappend = (++)
```

在这里，我们重用了列表连接运算符（++），因为String只是一个字符列表。

关于Haskell语法的一点说明：任何中缀运算符都可以通过用括号括起来将其转换为二元函数。 给定两个字符串，你可以通过在它们之间插入++来连接它们：

```
"你好" ++ "世界！"
```

或者通过将它们作为两个参数传递给括号内的 (++)函数：

```
(++) "你好 " "世界！"
```

请注意，函数的参数不用逗号分隔或者用括号括起来。 (这可能是学习Haskell时最难适应的事情。)

值得强调的是，Haskell允许你表达函数的相等性，例如：

```
mappend = (++)
```

从概念上讲，这与表达函数产生的值的相等性是不同的，例如：

```
mappend s1 s2 = (++) s1 s2
```

前者转化为范畴 Hask中态射的相等性（如果我们忽略底部，即无限计算的情况）。 这样的等式不仅更简洁，而且通常可以推广到其他范畴。 后者被称为外延相等性，并且它表明对于任意两个输入字符串，mappend和 (++)的输出是相同的。 由于参数的值有时被称为点（例如：函数 □在点 □处的值），这被称为逐点相等性。 不指定参数的函数相等性被描述为无点。 (顺便说一下，无点等式通常涉及函数的组合，用一个点来表示，所以这可能对初学者有点困惑。)

在C++中，最接近声明一个幺半群的方法是使用（提议的）概念语法。

```
template<class T>
   T mempty = delete;

template<class T>
   T mappend(T, T) = delete;

template<class M>
   concept bool Monoid = requires (M m) {
      { mempty<M> } -> M;
      { mappend(m, m); } -> M;
   };
```

第一个定义使用了值模板（也是提议的）。 多态值是一组值 - 每种类型对应一个不同的值。

关键字 delete表示没有定义默认值：必须根据具体情况指定。同样， mappend没有默认值。

概念 Monoid是一个谓词（因此是 bool类型），用于测试是否存在适当的 mempty和 mappend的定义，适用于给定的类型 M。

通过提供适当的特化和重载，可以实现Monoid概念的实例化：

```
模板<>
std::string mempty<std::string> = {""};

std::string mappend(std::string s1, std::string s2) {
      return s1 + s2;
}
```

3.5 幺半群作为范畴

那就是关于幺半群的“熟悉”定义，它涉及到集合的元素。但是你知道，在范畴论中，我们试图摆脱集合及其元素的概念，而是讨论对象和态射。
所以，让我们稍微改变一下观点，将二元运算符的应用视为将事物在集合中“移动”或“转移”。例如，有一个将每个自
然数加5的操作。它将0映射为5，1映射为6，2映射为7，依此类推。这是在自然数集上定义的一个函数。 这很好：我们有一个函数和一个集合。一般来说，对于任何数字n，都存在一个加n的函数 - “加法器”。

加法器如何组合？ 将添加5的函数与添加7的函数组合起来，得到一个添加12的函数。 因此，加法器的组合可以等同于加法规则。 这也很好：我们可以用函数组合来替代加法。

但是等等，还有更多：还有一个中性元素零的加法器。 加零不会改变事物的位置，所以它是自然数集合中的恒等函数。

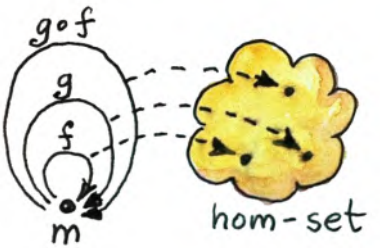
与其给你传统的加法规则，不如给你组合加法器的规则，不会丢失任何信息。 请注意，加法器的组合是结合的，因为函数的组合是结合的；而且我们有零加法器对应于恒等函数。

一个敏锐的读者可能已经注意到，从整数到加法器的映射来自于 mappend的第二种解释，即 m -> (m -> m)的类型签名。 它告诉我们 mappend将一个幺半群集合的元素映射到作用于该集合的函数。

现在我希望你忘记你正在处理的自然数集合，只把它看作一个单一的对象，一个带有一堆态射（加法器）的块。 幺半群是一个单一对象的范畴。 实际上，幺半群这个名字来自希腊语单一的意思。
每个幺半群都可以被描述为一个单一对象的范畴，其具有遵循适当组合规则的一组态射。

字符串连接是一个有趣的情况，因为我们可以选择定义右附加器和左附加器（或者说前附加器）。 这两个模型的组合表是彼此的镜像反转。 你可以很容易地说服自己，在“foo”之后添加“bar”对应于在“bar”之后添加“foo”。

你可能会问这个问题，即每个范畴幺半群——一个单对象范畴——是否定义了一个唯一的带二元运算的集合幺半群。 事实证明，我们总是可以从单对象范畴中提取一个集合。 这个集合就是态射的集合——我们例子中的加法器。 换句话说，我们在范畴 $\mathcal{C}$中有单对象 C的同态集合 $\mathcal{C}(C, C)$of。
我们可以很容易地在这个集合中定义一个二元运算符：两个集合元素的单子积是对应态射的组合的元素。 如果你给我两个 $\mathcal{C}(C, C)$对应于 f和 g的元素，它们的乘积将对应于组合$f \cdot g$。 组合总是存在的，因为这些态射的源和目标是同一个对象。 根据范畴的规则，它是结合的。 恒等态射是这个乘法的中性元素。



单子集合既可以看作态射，也可以看作集合中的点。

这个乘法。 因此，我们总是可以从一个范畴单子集合恢复一个集合单子集合。 对于所有的目的和意图，它们是一样的。

对于数学家来说，只有一个小问题：态射不必形成一个集合。 在范畴的世界中，有比集合更大的东西。 当任意两个对象之间的态射形成一个集合时，这个范畴被称为局部小范畴。 正如我承诺的那样，我将大多数情况下忽略这些细微差别，但我认为我应该为记录提到它们。

范畴论中有很多有趣的现象都源于一个事实，即同态集合的元素既可以看作态射，遵循组合规则，也可以看作集合中的点。在这里，范畴论中的态射组合对应于集合中的单子积 □(□, □)。

3.6 挑战

1. 从以下内容生成一个自由范畴：

    (a) 一个只有一个节点且没有边的图

(b) 一个只有一个节点且有一条（有向）边的图（提示：这条边可以与自身组合）
(c) 一个有两个节点且之间有一条箭头的图
(d) 一个只有一个节点且有26条用字母标记的箭头的图：a, b, c … z.

2. 这是什么类型的排序?

(a) 一个集合的集合，其中包含关系为：□包含于□，如果□的每个元素也是□的元素。
(b) C++类型与以下子类型关系：T1是T2的子类型，如果一个指向T1的指针可以传递给一个期望指向T2的函数而不触发编译错误。

3. 考虑到Bool是一个包含两个值True和False的集合，证明它分别对于运算符 &&(与)和 ||(或)形成两个（集合论上的）幺半群。
4. 将带有AND运算符的Bool幺半群表示为一个范畴：列出态射及其组合规则。
5. 将模3加法表示为一个幺半群范畴。

4

Kleisli范畴

是你已经看到如何将类型和纯函数建模为一个范畴。我还提到过在范畴论中有一种方法来建模副作用或非纯函数。让我们来看一个这样的例子：记录或追踪执行的函数。 在命令式语言中，这通常通过改变一些全局状态来实现，如下所示：

```
字符串记录器;

bool negate(bool b) {
        logger += "不是这样的！ ";
        return !b;
}
```

你知道这不是一个纯函数，因为它的记忆化版本无法生成日志。这个函数有副作用。

在现代编程中，我们尽量避免使用全局可变状态——仅仅是因为并发的复杂性。 而且你永远不会把这样的代码放在库中。

幸运的是，我们可以使这个函数变得纯粹。 你只需要显式地传递日志，输入和输出。让我们通过添加一个字符串参数来实现这一点，并将常规输出与包含更新日志的字符串配对：

```
pair<bool, string> negate(bool b, string logger) {
 return make_pair(!b, logger + "不是这样的！ ");
}
```

这个函数是纯粹的，它没有副作用，每次调用时都返回相同的对，如果需要，它可以被记忆化。 然而，考虑到日志的累积性质，你必须记忆化所有可能导致给定调用的历史记录。 对于每个不同的历史记录，都会有一个单独的记忆条目：

```
negate(true, "这是最好的时光。");
```

和

```
negate(true, "这是最糟糕的时光。");
```

等等。

这也不是一个非常好的库函数接口。 调用者可以忽略返回类型中的字符串，所以这不是一个很大的负担；但是他们被迫传递一个字符串作为输入，这可能不方便。

有没有一种更少侵入性的方法来做同样的事情？ 有没有一种方法来分离关注点？ 在这个简单的例子中，主要目的是

函数 negate是将一个布尔值转换为另一个布尔值。 日志是次要的。 当然，被记录的消息是特定于函数的，但将这些消息聚合成一个连续的日志的任务是一个独立的关注点。 我们仍然希望函数产生一个字符串，但我们希望它不再负责生成日志。所以这是一个折衷的解决方案：

```
pair<bool, string> negate(bool b) {
    return make_pair(!b, "不是这样的！");
}
```

这个想法是日志在函数调用之间进行聚合。

为了看到这是如何完成的，让我们切换到一个稍微现实一些的例子。 我们有一个从字符串到字符串的函数，将小写字符转换为大写：

```
string toUpper(string s) {
    string result;
    int (*toupperp)(int) = &toupper; // toupper is overloaded
    transform(begin(s), end(s), back_inserter(result), toupperp);
    return result;
}
```

还有一个将字符串分割成字符串向量的函数，在空白边界上进行分割：

```
vector<string> toWords(string s) {
    return words(s);
}
```

实际的工作是在辅助函数 words中完成的：

```
    vector<string> words(string s) {
        vector<string> result{""};
    for (auto i = begin(s); i != end(s); ++i)
    {
        if (isspace(*i))
            result.push_back("");
        else
            result.back() += *i;
    }
    return result;
}
```

我们希望修改函数toUpper和toWords，使它们在返回值的基础上附加一个消息字符串。

我们将“装饰”这些函数的返回值。让我们以一种通用的方式来定义一个模板Writer，它封装了一个由任意类型A的值和一个字符串组成的对：

```
template<class A>
using Writer = pair<A, string>;
```

下面是装饰后的函数：

```
Writer<string> toUpper(string s) {
    string result;
    int (*toupperp)(int) = &toupper;
```

```
    transform(begin(s), end(s), back_inserter(result), toupperp);
    return make_pair(result, "toUpper ");
}

Writer<vector<string>> toWords(string s) {
    return make_pair(words(s), "toWords ");
}
```

我们希望将这两个函数组合成另一个装饰后的函数，它将字符串转换为大写并拆分为单词，同时生成这些操作的日志。下面是我们可能这样做的方式：

```
Writer<vector<string>> process(string s) {
    auto p1 = toUpper(s);
    auto p2 = toWords(p1.first);
    return make_pair(p2.first, p1.second + p2.second);
}
```

我们已经实现了我们的目标：日志的聚合不再是个体函数的关注点。 它们产生自己的消息，然后在外部连接成一个更大的日志。现在想象一下一个完全按照这种风格编写的程序。 这是一个重复、容易出错的噩梦。但我们是程序员。 我们知道如何处理重复的代码：我们将其抽象化！然而，这不是你平常的抽象化——我们必须抽象化函数组合本身。 但组合是范畴论的本质，所以在编写更多代码之前，让我们从范畴论的角度分析问题。

4.1 Writer 范畴

为了搭载一些额外的功能，装饰一堆函数的返回类型的想法证明是非常有成果的。 我们将看到更多的例子。起点是我们常规的类型和函数范畴。 我们将保留类型作为对象，但重新定义我们的态射为装饰函数。

例如，假设我们想要装饰函数 isEven它从 int到 bool。 我们将其转化为一个由装饰函数表示的态射。 重要的是，尽管装饰函数返回一个对，但这个态射仍然被认为是对象 int和 bool之间的箭头：

```
pair<bool, string> isEven(int n) {
    return make_pair(n % 2 == 0, "isEven ");
}
```

根据范畴的法则，我们应该能够将这个态射与从对象 bool到其他对象的另一个态射组合起来。 特别地，我们应该能够将其与我们之前的 negate组合起来：

```
pair<bool, string> negate(bool b) {
    return make_pair(!b, "不是这样的！ ");
}
```

显然，我们不能像组合常规函数那样组合这两个态射，因为存在输入/输出不匹配的问题。 它们的组合应该看起来更像这样：

```
pair<bool, string> isOdd(int n) {
    pair<bool, string> p1 = isEven(n);
    pair<bool, string> p2 = negate(p1.first);
```

```
    return make_pair(p2.first, p1.second + p2.second);
}
```

所以这是我们正在构建的新范畴中两个态射的组合的步骤：

1. 执行与第一个态射对应的修饰函数

2. 提取结果对的第一个分量并将其传递给与第二个态射对应的修饰函数3. 连接第一个结果的第二个分量（字符串）和第二个结果的第二个分量（字符串）4. 返回一个新的对，将最终结果的第一个分量与连接的字符串组合在一起。

如果我们想将这个组合抽象为一个在C++中的高阶函数，我们必须使用一个由三个类型参数化的模板，这三个类型对应于我们范畴中的三个对象。 它应该接受两个符合我们规则的可组合的修饰函数，并返回一个第三个修饰函数：

```
template<class A, class B, class C>
function<Writer<C>(A)> compose(function<Writer<B>(A)> m1,
                               function<Writer<C>(B)> m2)
{
    return [m1, m2](A x) {
        auto p1 = m1(x);
        auto p2 = m2(p1.first);
        return make_pair(p2.first, p1.second + p2.second);
    };
}
```

现在我们可以回到之前的例子，并使用这个新模板实现 toUpper和 toWords的组合：

```
Writer<vector<string>> process(string s) {
    return compose<string, string, vector<string>>(toUpper,
     toWords)(s);
}
```

在传递类型给 compose模板时仍然存在很多噪音。 只要你有一个支持带有返回类型推导的通用lambda函数的C++14兼容编译器（此代码由Eric Niebler提供），就可以避免这个问题：

```
auto const compose = [](auto m1, auto m2) {
    return [m1, m2](auto x) {
        auto p1 = m1(x);
        auto p2 = m2(p1.first);
        return make_pair(p2.first, p1.second + p2.second);
    };
};
```

在这个新定义中， process的实现简化为：

```
    Writer<vector<string>> process(string s) {
        return compose(toUpper, toWords)(s);
}
```

但我们还没有完成。我们已经在我们的新范畴中定义了组合，但是什么是恒等态射？ 这些不是我们常规的恒等函数！它们必须是从类型A返回到类型A的态射，这意味着它们是以下形式的装饰函数：

```
Writer<A> identity(A);
```

它们必须在组合方面表现得像单位。 如果你看一下我们的组合定义，你会发现恒等态射应该不改变其参数，并且只对日志贡献一个空字符串：

```
template<class A> Writer<A> identity(A x) {
    return make_pair(x, "");
}
```

你可以很容易地确信我们刚刚定义的范畴确实是一个合法的范畴。 特别是，我们的组合是平凡的结合的。 如果你跟踪每对的第一个分量发生的情况，它只是一个常规的函数组合，而函数组合是结合的。
第二个组件正在被连接，并且连接也是
可结合的。

敏锐的读者可能会注意到，将这个构造推广到任何幺半群都很容易，而不仅仅是字符串幺半群。 我们会在compose中使用mappend，在identity中使用mempty（代替 +和""）。 实际上，没有理由限制我们只记录字符串。一个好的库编写者应该能够确定使库工作所需的最低限度的约束 - 在这里，日志库的唯一要求是日志具有幺半群属性。

4.2 Haskell 中的 Writer

在Haskell中，同样的事情更加简洁，我们还从编译器那里得到了更多的帮助。 让我们首先定义 Writer类型：

```
type Writer a = (a, String)
```

在这里，我只是定义了一个类型别名，类似于C++中的 typedef（或 using）。类型 Writer通过类型变量 a进行参数化，并且等价于一个 a 和 String的对。 对于对，语法很简洁：只是括号中的两个项，用逗号分隔。

我们的态射是从任意类型到某个 Writer type的函数：

```
a -> Writer b
```

我们将组合声明为一个有趣的中缀运算符，有时被称为“鱼”：

```
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
```

这是一个具有两个参数的函数，每个参数都是一个函数，并返回一个函数。 第一个参数的类型是 (a -> Writer b)，第二个参数的类型是 (b -> Writer c)，结果的类型是 (a -> Writer c)。

这是这个中缀运算符的定义 - 鱼符号两侧的两个参数 m1 和 m2：

```
m1 >=> m2 = \x ->
    let (y, s1) = m1 x
                (z, s2) = m2 y
    in (z, s1 ++ s2)
```

结果是一个带有一个参数 x的lambda函数。 lambda函数用反斜杠表示 - 将其视为带有截肢腿的希腊字母λ。 let表达式允许您声明辅助变量。 这里，对 m1的调用结果与一对变量 (y, s1)进行模式匹配；对 m2的调用结果，使用来自第一个模式的参数 y进行匹配，与 (z, s2)进行匹配。

在Haskell中，模式匹配对是常见的，而不是像在C++中使用访问器那样。除此之外，这两种实现之间有一个相当直接的对应关系。

整个let表达式的值在其in子句中指定：这里是一个由第一个组件为z和第二个组件为两个字符串的连接的对。

我还将为我们的范畴定义恒等态射，但出于稍后将变得清晰的原因，我将其称为return。

```
return :: a -> Writer a
return x = (x, "")
```

为了完整起见，让我们来看看embellished函数upCase和toWords的Haskell版本：

```
upCase :: String -> Writer String
upCase s = (map toUpper s, "upCase ")

toWords :: String -> Writer [String]
toWords s = (words s, "toWords ")
```

函数map对应于C++的transform。 它将字符函数toUpper应用于字符串s。 辅助函数words在标准Prelude库中定义。

最后，通过鱼运算符的帮助，两个函数的组合完成:

```
process :: String -> Writer [String]
process = upCase >=> toWords
```

4.3 Kleisli 范畴

你可能已经猜到我并没有即兴发明这个范畴。 这是所谓的Kleisli范畴的一个例子 - 一个基于单子的范畴。 我们还没有准备好讨论单子，但我想给你一个他们能做什么的味道。对于我们有限的目的，Kleisli范畴的对象是基础编程语言的类型。 从类型 𝐴到类型 𝐵的态射是从 𝐴到使用特定装饰的 𝐵派生类型的函数。 每个Kleisli范畴都定义了自己的组合这样的态射的方式，以及关于该组合的恒等态射。 （稍后我们将看到，不精确的术语“装饰”对应于范畴中的自函子的概念。）

在这篇文章中，我使用的特定单子被称为写入单子，它用于记录或跟踪函数的执行。 它也是将效果嵌入纯计算的更一般机制的一个例子。 你之前已经看到，我们可以在集合的范畴中对编程语言的类型和函数进行建模（通常忽略底部）。 在这里，我们将这个模型扩展到一个稍微不同的范畴，一个范畴，其中态射由装饰函数表示，它们的组合不仅仅是将一个函数的输出传递给另一个函数的输入。

我们还有一个可以玩弄的自由度：组合本身。 事实证明，这正是使得能够对使用副作用传统上在命令式语言中实现的程序进行简单的指称语义的自由度。

4.4 挑战

对于其参数的所有可能值都没有定义的函数被称为部分函数。 从数学意义上讲，它实际上不是一个函数，因此它不符合标准的范畴模型。 然而，它可以由一个返回装饰类型optional的函数来表示：

```
template<class A> class optional {
        bool _isValid;
        A _value;
public:
        optional()                      : _isValid(false) {}
        optional(A v) : _isValid(true), _value(v) {}
        bool isValid() const { return _isValid; }
        A value() const { return _value; }
};
```

举个例子，这是装饰函数的实现
安全根：

```
optional<double> safe_root(double x) {
        if (x >= 0) return optional<double>{sqrt(x)};
        else return optional<double>{};
}
```

这是挑战：

1. 为部分函数构建Kleisli范畴（定义组合和恒等）。

2. 实现装饰函数safe_reciprocal，如果参数不为零，则返回有效的倒数。

3. 组合 safe_root 和 safe_reciprocal 来实现 safe_root_reciprocal，在可能的情况下计算sqrt(1/x)。
。

5

积和余积

古希腊剧作家欧里庇得斯曾经说过：“一个人就像他经常交往的人一样。”我们是通过我们的关系来定义自己的。 在范畴论中，这一点比任何地方都更为真实。 如果我们想要在一个范畴中单独指出一个特定的对象，我们只能通过描述它与其他对象（包括自身）的关系模式来做到这一点。
这些关系是由态射来定义的。

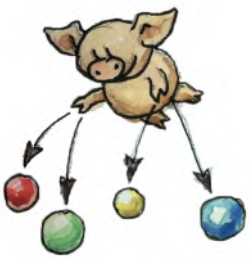
范畴论中有一个常见的构造叫做“通用构造”，用于根据它们的关系来定义对象。
其中一种方法是选择一个模式，一个由对象和态射构成的特定形状，并在范畴中寻找它的所有出现。 如果这是一个足够常见的模式，并且范畴很大，那么很有可能会有很多很多的匹配。 关键是在这些匹配中建立某种排序，并选择可以被认为是最佳匹配的对象。

这个过程让人想起我们进行网络搜索的方式。 查询就像是一个模式。 一个非常通用的查询会给你很大的召回率：很多结果。 其中一些可能是相关的，其他的则不是。为了排除不相关的结果，你需要细化你的查询。 这会增加查询的精确度。 最后，搜索引擎会对结果进行排序，希望你感兴趣的结果会排在列表的顶部。

5.1 初始对象

最简单的形状是一个单独的对象。 显然，在给定范畴中，这种形状的实例与对象的数量一样多。 这可供选择的太多了。 我们需要建立某种排名，并尝试找到排名最高的对象。 我们唯一可以使用的手段是态射。 如果你将态射看作箭头，那么可能存在从范畴的一端到另一端的整体流动。 这在有序范畴中是正确的，例如在偏序中。 我们可以通过将对象 □比对象 □“更初级”，如果存在从 □到 □的箭头（态射）来推广对象的优先级概念。 然后我们将初始对象定义为具有指向所有其他对象的箭头的对象。 显然，并没有保证这样的对象存在，这没关系。 更大的问题是可能存在太多这样的对象：召回率很高，但是精确度不足。 解决方案是借鉴有序范畴的思路- 它们在任意两个对象之间最多允许一个箭头：只有一种方式可以小于或等于另一个对象。 这导致我们得到了初始对象的定义：

> 初始对象是指在范畴中，对于任意对象都存在且唯一的一个态射。



然而，即使如此也不能保证初始对象的唯一性（如果存在的话）。 但它确保了下一个最好的事情：唯一性直到同构。 同构在范畴论中非常重要，所以我会简要谈谈它们。 暂时先同意唯一性直到同构在定义初始对象时的使用。

以下是一些例子：在偏序集合（通常称为 poset）中，初始对象是最小元素。 有些偏序集合没有初始对象，比如所有整数（包括正数和负数）构成的集合，其态射关系为小于等于。

在集合和函数的范畴中，初始对象是空集。记住，空集对应于Haskell类型 Void（C++中没有对应类型），而从 Void到任何其他类型的唯一多态函数被称为 absurd：

```
absurd :: Void -> a
```

正是这个态射家族使得 Void成为范畴中的初始对象。

5.2 终结对象

让我们继续使用单对象模式，但是改变一下我们对对象进行排序的方式。 我们将说对象 □比对象 □“更终结”，如果存在一个从 □到 □的态射（注意方向的反转）。 我们将寻找一个比范畴中的任何其他对象都更终结的对象。 同样，我们将坚持唯一性：

> 终结对象是范畴中从任何对象到它的唯一一个态射的对象。

而且，终结对象是唯一的，同构的，我很快会证明。 但首先让我们看一些例子。 在偏序集中，如果存在的话，终结对象是最大的对象。 在集合范畴中，终结对象是一个单例集合。 我们已经讨论过单例集合——它们对应于C++中的 void类型和Haskell中的单位类型 ()。它是一个只有一个值的类型——在C++中是隐式的，在Haskell中是显式的，用 ()表示。 我们还确定了从任何类型到单位类型的纯函数只有一个且唯一的存在：

```
unit :: a -> ()
unit _ = ()
```

因此，终结对象的所有条件都得到满足。

请注意，在这个例子中，唯一性条件是至关重要的，因为还有其他集合（实际上，除了空集之外的所有集合）都有来自每个集合的入射态射。例如，对于每种类型，都有一个布尔值函数（谓词）定义为：

```
yes :: a -> Bool
yes _ = True
```

但是，布尔值 Bool不是一个终结对象。 对于每种类型，至少还有一个布尔值 Bool-值函数：

```
no :: a -> Bool
no _ = False
```

坚持唯一性使我们得到了恰到好处的精确性，将终结对象的定义缩小到了一个类型。

5.3 对偶性

你不禁会注意到我们如何定义初始对象和终结对象之间的对称性。 两者之间唯一的区别就是态射的方向。 事实证明，对于任何范畴 C，我们可以通过反转所有箭头来定义对偶范畴 $\mathcal{C}^{op}$。 只要我们同时重新定义组合，对偶范畴就会自动满足范畴的所有要求。 如果原始态射$f : a \to b$和$g : b \to c$组合成$h : a \to c$，那么反转的态射 $f^{op} : b \to a$和 $g^{op} : c \to b$将组合成 $h^{op} : c \to a$，其中 $h^{op} = f^{op} \circ g^{op}$。

并且反转恒等箭头是一个（双关语警告！）无操作。

对于范畴论中的每个数学家来说，对偶性是一个非常重要的性质，因为它使每个数学家的工作效率翻倍。
对于你提出的每个构造，都有它的对立面；对于你证明的每个定理，你都会得到一个免费的定理。 对立范畴中的构造通常以“co”为前缀，所以你有乘积和余积，单子和余子，锥和余锥，极限和余极限，等等。不过，没有余单子，因为两次反转箭头会使我们回到原始状态。

由此可见，终结对象是对立范畴中的初始对象。

5.4 同构

作为程序员，我们很清楚定义相等性是一个非平凡的任务。 两个对象相等意味着什么？ 它们必须占据内存中的相同位置（指针相等）吗？ 或者仅仅它们所有组成部分的值相等就足够了？ 如果一个复数以实部和虚部表示，另一个以模和角度表示，它们是否相等？ 你可能会认为数学家已经弄清了相等性的含义，但他们并没有。 他们面临着相等性的多个竞争定义的问题。 有命题相等、内涵相等、外延相等和同伦类型论中的路径相等。 然后还有更弱的同构概念，甚至更弱的等价关系。

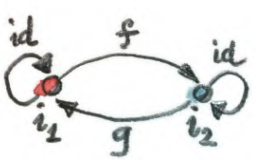
直觉上，同构对象看起来是相同的-它们具有相同的形状。 这意味着一个对象的每个部分都对应于另一个对象的某个部分，形成一对一的映射。 就我们的仪器所能告诉的，这两个对象是完全相同的副本。
从数学上讲，这意味着从对象 □到对象-

ject □，并且存在从对象 □到对象 □的映射，它们互为反函数。在范畴论中，我们用态射代替映射。同构是可逆的态射；或者是一对互为反函数的态射。

我们通过组合和恒等性来理解反函数：态射 □是态射 □的反函数，如果它们的组合是恒等态射。 这实际上是两个方程，因为有两种组合两个态射的方式：

f . g = id
g . f = id

当我说初始（终端）对象是同构唯一的时候，我指的是任意两个初始（终端）对象是同构的。
这实际上很容易理解。假设我们有两个初始对象 $□_1$和 $□_2$。 由于 $□_1$是初始的，存在唯一的态射 □从 $□_1$到 $□_2$。 同样地，由于 $□_2$是初始的，存在唯一的态射□从 $□_2$到 $□_1$。 这两个态射的复合是什么？



这个图中的所有态射都是唯一的。

复合□ ◦ □必须是从 $□_1$到 $□_1$的态射。 但是 $□_1$是初始的，所以只能有一个从 $□_1$到 $□_1$的态射。 由于我们在一个范畴中，我们知道存在从 $□_1$到 $□_1$的恒等态射，而且由于只有一个位置，那就是它。因此□ ◦ □是

等于恒等。 同样地，□ ◦ □必须等于恒等，因为从 $□_2$回到 $□_2$只能有一个态射。 这证明了 □和 □必须互为逆。 因此，任意两个初始对象是同构的。

注意，在这个证明中我们使用了从初始对象到自身的态射的唯一性。 如果没有这一点，我们无法证明“同构”的部分。 但是为什么我们需要 □和□的唯一性呢？ 因为初始对象不仅在同构意义下是唯一的，而且在唯一同构意义下也是唯一的。 原则上，两个对象之间可能存在多个同构，但在这里不是这样的情况。

这种“唯一同构性”是所有通用构造的重要属性。

5.5 乘积

下一个通用构造是乘积。 我们知道两个集合的笛卡尔积是什么：它是一组有序对。 但是连接乘积集合与其组成集合的模式是什么？ 如果我们能够弄清楚这一点，我们就能够将其推广到其他范畴。

我们只能说有两个函数，即投影函数，从乘积到每个组成部分。 在Haskell中，这两个函数分别被称为fst和snd，并且它们分别选择一对的第一个和第二个分量：

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

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

在这里，函数通过模式匹配它们的参数来定义：匹配任何一对的模式是(x, y)，它将其分量提取到变量x和y中。

使用通配符可以进一步简化这些定义：

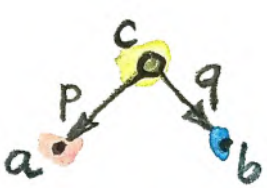
```
fst (x, _) = x
snd (_, y) = y
```

在C++中，我们可以使用模板函数，例如：

```
template<class A, class B> A
fst(pair<A, B> const & p) {
        return p.first;
}
```

凭借这些看似非常有限的知识，让我们尝试在集合范畴中定义一种对象和态射的模式，从而构建两个集合的积， □和 □。 这种模式包括一个对象 □和两个连接它与 □和 □的态射 □和 □：

```
p :: c -> a
q :: c -> b
```



所有符合这种模式的 □都将被视为积的候选对象。
可能会有很多候选对象。

例如，让我们选择两个Haskell类型， Int和
Bool，并获取它们积的样本。

这是一个例子：整数. 整数和布尔的乘积是否可以被视为一个候选项？ 是的，可以 - 这里是它的投影：

```
p :: 整数 -> 整数
p x = x

q :: 整数 -> 布尔
q _ = 真
```

这很糟糕，但它符合标准。

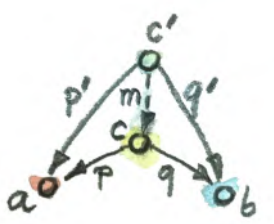
这是另一个例子： (整数, 整数, 布尔). 它是一个由三个元素组成的元组，或者称为三元组。 这里有两个使它成为一个合法候选项的态射（我们在三元组上使用模式匹配）：

```
p :: (整数, 整数, 布尔) -> 整数
p (x, _, _) = x

q :: (整数, 整数, 布尔) -> 布尔
q (_, _, b) = b
```

你可能已经注意到，我们的第一个候选项太小了 - 它只覆盖了整数维度；第二个候选项太大了 - 它不必要地重复了整数维度。

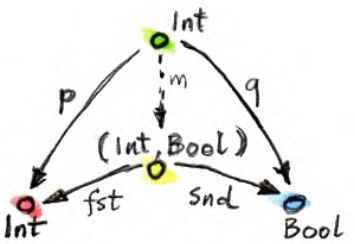
但我们还没有探索通用构造的另一部分：排名。 我们希望能够比较我们模式的两个实例。 我们希望将一个候选对象 □及其两个投影 p和 q与另一个候选对象 □′及其两个投影 p’ 和 q’ 进行比较。 我们希望能够说 □比 □′更好，如果存在一个从 □′到 □的态射 □ - 但这太弱了。 我们还希望它的投影比□′的投影“更好”或“更普遍”。 这意味着投影 p’ 和 q’ 可以通过使用 □从 p和 q重建：

p' = p . m
q' = q . m



从另一个角度来看，这些方程可以看作是 □因子分解了p’ 和 q’ 。 假装这些方程是在自然数中，点表示乘法: □是 p’ 和 q’ 共有的因子。

为了建立一些直觉，让我向你展示一下具有两个规范投影的对 (Int, Bool)确实比我之前提到的两个候选对更好。



第一个候选对的映射 m为：

```
m :: Int -> (Int, Bool)
m x = (x, True)
```

事实上，两个投影 p和 q可以重构为：

```
p x = fst (m x) = x
q x = snd (m x) = True
```

第二个示例的 m同样是唯一确定的：

```
m (x, _, b) = (x, b)
```

我们能够证明 (Int, Bool)比这两个候选者都要好。 让我们看看为什么相反的情况不成立。 我们能找到一些m'能够帮助我们从 p和 q重构 fst和 snd吗？

```
fst = p . m'
snd = q . m'
```

在我们的第一个例子中， q总是返回 True，而我们知道存在第二个分量为 False的对。 我们无法从中重构 snd q。

第二个例子不同：在运行 p或 q之后，我们保留了足够的信息，但是有多种方式可以因式分解 fst和 snd。 因为 p和 q都忽略了三元组的第二个分量，我们的 m'可以在其中放任何东西。我们可以有：

```
m' (x, b) = (x, x, b)
```

或者

```
m' (x, b) = (x, 42, b)
```

等等。

综上所述，对于任何具有两个投影 p和 q的类型 c，存在一个唯一的 m从 c到笛卡尔积 (a, b)，使其因式分解它们。 实际上，它只是将 p和 q组合成一对。

```
m :: c -> (a, b)
m x = (p x, q x)
```

这使得笛卡尔积 (a, b)成为我们最佳的匹配，这意味着这个普遍构造在集合范畴中起作用。 它选择了任意两个集合的积。

现在让我们忘记集合，使用相同的普遍构造在任何范畴中定义两个对象的积。 这样的积并不总是存在，但当存在时，它是唯一的，唯一的同构。

> 两个对象 □和 □的积是对象 □配备了两个投影，使得对于任何其他配备了两个投影的对象 □′，存在一个唯一的态射 □从 □′到 □，使得这些投影分解。

产生分解函数 m的（高阶）函数有时被称为分解器。 在我们的情况下，它将是函数：

```
factorizer :: (c -> a) -> (c -> b) -> (c -> (a, b))
factorizer p q = \x -> (p x, q x)
```

5.6 余积

像范畴论中的每个构造一样，积也有一个对偶，被称为余积。 当我们颠倒积模式中的箭头时，我们得到一个对象 □，配备有两个注入, i和 j：从 □和 □到 □的态射。

```
i :: a -> c
j :: b -> c
```

排名也被颠倒了：对象 □比对象 □′“更好”，如果存在一个从 □到 □′的态射 □，使得它分解了注入：

```
i' = m . i
j' = m . j
```

“最佳”的这样一个对象，它与任何其他模式连接的唯一态射被称为余积，如果存在的话，它是唯一的，直到唯一同构。

> 两个对象 □和 □的余积是带有两个注入的对象 □，使得对于任何其他带有两个注入的对象 □′，存在一个唯一的态射 □从 □到 □′，使得这些注入因子化。

在集合范畴中，余积是两个集合的不相交并集。
不相交并集 □和 □的元素要么是 □的元素，要么是 □的元素。 如果两个集合有重叠部分，不相交并集将包含两个公共部分的副本。 你可以将不相交并集的元素视为带有指定来源标识符的标记。

对于程序员来说，更容易通过类型来理解余积：它是两种类型的标记联合。 C++支持联合，但它们没有标记。 这意味着在你的程序中，你必须以某种方式跟踪联合的哪个成员是有效的。 要创建一个带标记的联合，你必须定义一个标记 - 一个枚举 - 并将其与联合结合起来。 例如，一个带有 int和char const *的标记联合可以这样实现：

```
struct Contact {
    enum { isPhone, isEmail } tag;
```

```
    union { int phoneNum; char const * emailAddr; };
};
```

这两个注入可以作为构造函数或函数来实现。 例如，这是第一个注入作为函数 PhoneNum：

```
Contact PhoneNum(int n) {
    Contact c;
    c.tag = isPhone;
    c.phoneNum = n;
    return c;
}
```

它将一个整数注入到 Contact中。

标记联合也被称为变体，在boost库中有一个非常通用的实现，boost::variant。

在Haskell中，你可以通过用竖线分隔数据构造函数来将任何数据类型组合成标记联合。这个 Contact示例可以转换为以下声明：

```
data Contact = PhoneNum Int | EmailAddr String
```

在这里， PhoneNum和 EmailAddr既作为构造函数（注入），也作为模式匹配的标签（稍后会详细介绍）。 例如，这是使用电话号码构造联系人的方法：

```
helpdesk :: Contact
helpdesk = PhoneNum 2222222
```

与Haskell中作为原始对的基本对的规范实现不同，余乘积的规范实现是一个名为 Either的数据类型，在标准Prelude中定义如下：

```
data Either a b = Left a | Right b
```

它由两个类型参数化， a和 b，并且有两个构造函数： Left接受类型为 a的值， Right接受类型为 b的值。就像我们为乘积定义了

因子化器一样，我们也可以为余乘积定义一个因子化器。 给定一个候选类型 c和两个候选注入函数i和 j，余乘积的因子化器产生因子化函数：

```
factorizer :: (a -> c) -> (b -> c) -> Either a b -> c
factorizer i j (Left a) = i a
factorizer i j (Right b) = j b
```

5.7 不对称性

我们已经看到了两组对偶定义：终态对象的定义可以通过反转箭头的方向从初态对象的定义中得到；类似地，余积的定义可以从积的定义中得到。 然而，在集合范畴中，初态对象与终态对象非常不同，余积与积也非常不同。 我们稍后会看到，积的行为类似于乘法，终态对象扮演着一的角色；而余积的行为更像是求和，初态对象扮演着零的角色。 特别地，对于有限集合，积的大小是各个集合大小的乘积，而余积的大小是它们的和。

这表明集合范畴在箭头的反转方面不对称。

请注意，虽然空集合对任何集合都有唯一的态射（荒谬函数），但它没有任何回来的态射。 单例集合对任何集合都有唯一的态射，但它也有出去的态射。

对每个集合（除了空集）都有出去的态射。 正如我们之前所看到的，来自终对象的出去的态射在选择其他集合的元素（空集没有元素，所以没有东西可选）方面起着非常重要的作用。

单例集合与积的关系使其与余积不同。 考虑使用单例集合，用单位类型 ()表示，作为另一个（远远不如的）产品模式的候选。 为其配备两个投影 p和 q：从单例到每个组成集合的函数。 每个选择一个具体元素，来自我们的候选者，即单例集合，到积的也有（唯一的）态射 m。

这个态射从乘积集合中选择一个元素 - 它选择一个具体的对。 它还将这两个投影分解为：

```
p = fst . m
q = snd . m
```

当作用于单值 ()时，即单例集合的唯一元素，这两个方程变为：

```
p () = fst (m ())
q () = snd (m ())
```

由于 m ()是由 m选择的乘积中的元素，这些方程告诉我们从第一个集合中由 p选择的元素 p ()是由 m选择的对中的第一个分量。 类似地， q ()等于第二个分量。 这完全符合我们对乘积的理解，即乘积的元素是来自构成集合的元素的对。

对于余乘积，没有这样简单的解释。 我们可以尝试将单例集合作为余乘积的候选，试图从中提取元素，但这里会有两个注入。

而不是两个投影从中出来。它们不会告诉我们任何关于它们的来源（事实上，我们已经看到它们忽略了输入参数）。 从余积到我们的单元素的唯一态射也不会。 当从初始对象的方向看时，集合范畴看起来与从终止端看起来非常不同。

这不是集合的固有属性，而是函数的属性，我们将其用作 □ □ □ 中的态射。 函数在一般情况下是不对称的。让我解释一下。

函数必须对其定义域集合中的每个元素进行定义（在编程中，我们称之为 total函数），但它不必覆盖整个值域。 我们已经看到了一些极端情况：来自单元素集合的函数 - 仅选择值域中的一个元素。 （实际上，来自空集的函数才是真正的极端情况。）当定义域的大小远小于值域的大小时，我们通常认为这样的函数将定义域嵌入到值域中。 例如，我们可以将来自单元素集合的函数视为将其单个元素嵌入到值域中。 我称它们为嵌入函数，但数学家更喜欢给相反的名称：紧密填充其值域的函数被称为满射或到。

另一个不对称的来源是函数允许将域集的许多元素映射到目标域的一个元素。
它们可以将它们合并。 极端情况是将整个集合映射到一个单例集。 你已经见过多态的单元函数，它就是这样做的。 合并只能通过组合来实现。 两个合并函数的组合比单个函数更加合并。 数学家给非合并函数起了一个名字：它们被称为单射或一对一。

当然，有一些既不是嵌入也不是合并的函数。 它们被称为双射，它们是真正对称的，因为它们是可逆的。 在集合范畴中，同构等同于双射。

5.8 挑战

1. 证明终结对象在唯一同构的情况下是唯一的。

2. 在偏序集中，两个对象的乘积是什么？ 提示：使用通用构造。
3. 在偏序集中，两个对象的余积是什么？
4. 在你喜欢的编程语言中实现Haskell的Either的等效泛型类型。
5. 证明Either是比int更好的余积，配备了两个注入函数：

```
int i(int n) { return n; }
int j(bool b) { return b ? 0: 1; }
```

提示：定义一个函数

```
int m(Either const & e);
```

来分解i和j。

6. 继续上一个问题：你如何论证int配备了两个注入函数i和j时不能比Either更好？
7. 继续进行：这些注入函数怎么样？

```
int i(int n) {
    if (n < 0) return n;
    return n + 2;
```

```
}

int j(bool b) { return b ? 0: 1; }
```

8. 提出一个比Either更差的int和bool的余积候选，因为它允许从它到Either的多个可接受的态射。

5.9 参考文献

1. 猫猫们，乘积和余积[1]视频。

1https://www.youtube.com/watch?v=upCSDIO9pjc

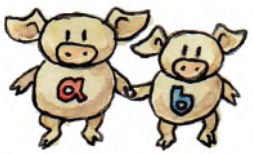
6

简单的代数数据类型

我们已经看到了两种基本的类型组合方式：使用乘积和余积。 事实证明，在日常编程中，很多数据结构只需要使用这两种机制就可以构建。

这个事实具有重要的实际意义。 许多数据结构的属性是可组合的。 例如，如果你知道如何比较基本类型的值是否相等，并且知道如何将这些比较推广到乘积和余积类型，你可以自动推导出复合类型的相等运算符。 在Haskell中，你可以自动推导出大部分复合类型的相等性、比较性、转换为字符串等操作。

让我们更仔细地看一下在编程中出现的乘积类型和和类型。



6.1乘积类型

在编程语言中，两种类型的乘积的典型实现是一个二元组。 在Haskell中，二元组是一个原始的类型构造器；在C++中，它是一个相对复杂的在标准库中定义的模板。

对偶不严格可交换：一个对偶 (Int, Bool)不能替代另一个对偶 (Bool, Int)，即使它们携带相同的信息。 然而，它们在同构意义上是可交换的——同构由 swap函数给出（它是自身的逆函数）：

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

你可以将这两个对偶简单地视为使用不同的格式来存储相同的数据。 这就像大端序和小端序一样。

通过将对偶嵌套在对偶中，您可以将任意数量的类型组合成一个乘积，但有一种更简单的方法：嵌套对偶等同于元组。 这是不同方式嵌套对偶同构的结果。 如果您想以这个顺序将三种类型 a、 b和 c组合成一个乘积，有两种方法可以实现：

```
((a, b), c)
```

或者

```
(a, (b, c))
```

这些类型是不同的 - 你不能将一个类型传递给期望另一个类型的函数 - 但它们的元素是一一对应的。 有一个函数将一个映射到另一个：

```
alpha :: ((a, b), c) -> (a, (b, c))
alpha ((x, y), z) = (x, (y, z))
```

而且这个函数是可逆的：

```
alpha_inv :: (a, (b, c)) -> ((a, b), c)
alpha_inv (x, (y, z)) = ((x, y), z)
```

所以它是一个同构。 这只是重新打包相同数据的不同方式。

你可以将创建一个乘积类型解释为类型的二元操作。 从这个角度来看，上述同构看起来非常像我们在幺半群中看到的结合律：

$$(\square\ \square\ \square)\square\ \square = \square\ \square(\square\ \square\ \square)$$

除了，在幺半群的情况下，组合乘积的两种方式是相等的，而在这里它们只是“同构上相等”。如果我们可以接受同构，并且

不坚持严格的相等，我们甚至可以进一步证明单位类型 `()`是乘积的单位，就像1是乘法的单位一样。 实际上，将某个类型的值 `a` 与单位配对并不添加任何信息。类型：

```
(a, ())
```

与 `a`同构。这是同构映射：

```
rho :: (a, ()) -> a
rho (x, ()) = x

rho_inv :: a -> (a, ())
rho_inv x = (x, ())
```

这些观察可以通过说 □ □ □(集合的范畴）是一个幺半范畴来形式化。 它是一个既是范畴又是幺半的概念，意味着你可以乘法对象（在这里，取它们的笛卡尔积）。
我将在将来更详细地讨论幺半范畴，并给出完整的定义。

在Haskell中，有一种更一般的定义产品类型的方式 - 特别是，当它们与和类型结合时，我们很快就会看到。 它使用具有多个参数的命名构造函数。 例如，可以将一对定义为：

```
data Pair a b = P a b
```

在这里， Pair a b是由两个其他类型参数化的类型的名称， a和 b； P是数据构造函数的名称。 通过将两个类型传递给 Pair类型构造函数来定义一对类型。 通过将适当类型的两个值传递给构造函数 P来构造一对值。 例如，让我们将一个值 stmt定义为一个 String的一对。
和 布尔：

```
stmt :: 对 String Bool 的配对
stmt = P "这个语句是" False
```

第一行是类型声明。 它使用类型构造器 Pair，用 String和 Bool替换了泛型定义中的 a和 b。 第二行通过将具体的字符串和具体的布尔值传递给数据构造器 P来定义实际值。

类型构造器用于构造类型；数据构造器用于构造值。 Type constructors 用于构造类型；data constructors 用于构造值。

由于Haskell中类型和数据构造函数的命名空间是分开的，因此你经常会看到相同的名称同时用于两者，如下所示：

```
数据对 a b = 对 a b
```

如果你足够仔细，甚至可以将内置的对类型视为这种声明的一种变体，其中名称Pair被二元运算符(,)替换。 实际上，你可以像使用其他命名构造函数一样使用(,)，并使用前缀表示法创建对：

```
stmt = (,) "这个语句是" False
```

类似地，你可以使用 (,,) 来创建三元组，以此类推。

而不是使用通用的对或元组，你也可以定义特定的命名产品类型，如下所示：

```
数据类型 Stmt = Stmt String Bool
```

它只是一个 String和 Bool的乘积，但它有自己的名称和构造函数。这种声明风格的优势在于，你可以定义许多具有相同内容但具有不同含义和功能的类型，而这些类型不能互相替代。

使用元组和多参数构造函数进行编程可能会变得混乱和容易出错 - 需要跟踪每个组件代表的是什么。 通常最好给组件命名。 具有命名字段的乘积类型在Haskell中称为记录，而在C中称为struct。

6.2记录

让我们看一个简单的例子。 我们想通过将两个字符串，名称和符号；以及一个整数，原子序数；组合成一个数据结构来描述化学元素。 我们可以使用一个元组 (String, String, Int)并记住每个组件代表的是什么。 我们可以通过模式匹配提取组件，就像这个函数检查元素的符号是否是其名称的前缀（如 He是 Helium的前缀）一样：

```
startsWithSymbol :: (String, String, Int) -> Bool
startsWithSymbol (name, symbol, _) = isPrefixOf symbol name
```

这段代码容易出错，而且很难阅读和维护。 最好定义一个记录：

```
data Element = Element { name :: String
                       , symbol :: String
                       , atomicNumber :: Int }
```

这两种表示是同构的，可以通过这两个相互逆转的转换函数来证明：

```
tupleToElem :: (String, String, Int) -> Element
tupleToElem (n, s, a) = Element { name = n
                                , symbol = s
                                , atomicNumber = a }

elemToTuple :: Element -> (String, String, Int)
 elemToTuple e = (name e, symbol e, atomicNumber e)
```

注意，记录字段的名称也可以作为访问这些字段的函数。 例如，atomicNumber e可以从e中检索atomicNumber字段。 我们将atomicNumber用作类型的函数：

```
atomicNumber :: 元素 -> 整数
```

使用记录语法为 Element，我们的函数startsWithSymbol变得更易读：

```
startsWithSymbol :: 元素 -> 布尔值
startsWithSymbol e = isPrefixOf (symbol e) (name e)
```

我们甚至可以使用Haskell的技巧，通过用反引号括起来将函数 isPr efixOf转换为中缀运算符，并使其读起来几乎像一个句子：

```
startsWithSymbol e = symbol e `isPrefixOf` name e
```

在这种情况下，括号可以省略，因为中缀运算符
比函数调用的优先级低。

6.3和类型

就像集合范畴中的乘积导致了乘积类型一样，余积导致了和类型。 Haskell中和类型的规范实现是：

```
data Either a b = Left a | Right b
```

就像对偶一样， Either是可交换的（同构），可以嵌套，并且嵌套顺序是无关紧要的（同构）。 因此，我们可以定义一个等价于三元组的和类型：

```
data OneOfThree a b c = Sinistral a | Medial b | Dextral c
```

等等。

原来 □ □ 也是一个（对称的）幺半范畴，关于余积它是保持的。 二元运算的角色由不相交的和扮演，单位元的角色由初始对象扮演。

在类型方面，我们有 Either作为幺半操作符， Void作为其中性元素。 你可以将 Either看作是加法， Void看作是零。 事实上，将 Void添加到一个和类型中不会改变其内容。例如：

```
Either a Void
```

与 a同构。 这是因为无法构造一个 Right版本的该类型 - 不存在类型为 Void的值。 构造 Either a Void的唯一成员是使用 Left构造函数，它们只是封装了一个类型为 a的值。 因此，符号上有□ + 0 = □。

在Haskell中，和它们的C++等价物，联合体或变体，在很大程度上不常见。 这有几个原因。

首先，最简单的和类型只是枚举，并且在C++中使用枚举实现。Haskell的和类型的等价物是：

```
data Color = 红色 | 绿色 | 蓝色
```

在C++中是：

```
enum { 红色, 绿色, 蓝色 };
```

一个更简单的和类型是：

```
data Bool = True | False
```

在C++中是原始的 bool 类型。

在C++中，用于编码值的存在或不存在的简单和类型使用特殊技巧和“不可能”的值，如空字符串、负数、空指针等进行实现。如果是有意的，Haskell中使用 Maybe类型来表示这种可选性：

data Maybe a = 无 | 有 a

Maybe类型是两种类型的和。 如果将这两个构造器分开成单独的类型，你就可以看到这一点。 第一个构造器看起来像这样：

数据 NothingType = Nothing

它是一个枚举类型，只有一个值叫做 Nothing。 换句话说，它是一个单例，等同于单位类型 ()。第二部分：

数据 JustType a = Just a

它只是对类型 a的封装。 我们可以将 Maybe编码为：

数据 Maybe a = Either () a

更复杂的和类型在C++中通常使用指针来模拟。 指针可以是空指针，也可以指向特定类型的值。 例如，一个Haskell列表类型可以定义为一个（递归）和类型：

数据 List a = Nil | Cons a (List a)

可以使用空指针技巧在C++中实现空列表：

```
template<class A>
class List {
    Node<A> * _head;
public:
    List() : _head(nullptr) {} // Nil
    List(A a, List<A> l)                    // Cons
      : _head(new Node<A>(a, l))
    {}
};
```

注意，两个Haskell构造函数 Nil和 Cons被翻译成了两个具有类似参数的重载 List构造函数（对于Nil来说是没有参数，对于Cons来说是一个值和一个列表）。 List类不需要标签来区分和区分和类型的两个组成部分。 相反，它使用特殊的空指针值来编码 _head为Nil。然而，Haskell和C++类型之间的主要区别在于Haskell数据

结构是不可变的。 如果你使用特定的构造函数创建一个对象，那么这个对象将永远记住使用了哪个构造函数以及传递给它的参数。因此，作为 Just "energy"创建的 Maybe对象永远不会变成 Nothing。同样，一个空列表将永远是空的，而一个包含三个元素的列表将始终具有相同的三个元素。

正是这种不可变性使得构造可逆。 给定一个对象，你总是可以将其拆解成构造时使用的部分。 这种拆解是通过模式匹配来完成的，并且它将构造函数重用作为模式。 构造函数的参数（如果有的话）将被替换为变量（或其他模式）

List数据类型有两个构造函数，因此对于任意 List的解构都使用与这些构造函数对应的两个模式。
一个匹配空 Nil列表，另一个匹配 Cons构造的列表。
例如，这是一个关于 List的简单函数的定义：

```
maybeTail :: List a -> Maybe (List a)
maybeTail Nil = Nothing
maybeTail (Cons _ t) = Just t
```

maybeTail定义的第一部分使用 Nil构造函数作为模式，并返回 Nothing。 第二部分使用 Cons构造函数作为模式。 它用通配符替换了构造函数的第一个参数，因为我们对它不感兴趣。 Cons的第二个参数绑定到变量 t（严格来说，这些东西被称为变量，尽管它们从不变化：一旦绑定到表达式，变量就不会改变）。 返回值是 Just t。 现在，根据 List的创建方式，它将匹配其中一个子句。 如果是使用 Cons创建的，将检索到传递给它的两个参数（并且第一个参数将被丢弃）。

在C++中，使用多态类层次结构实现了更加复杂的和类型。 具有共同祖先的类族可以被理解为一种变体类型，其中vtable作为隐藏标签。在Haskell中，通过对构造函数进行模式匹配和调用专门的代码来完成的操作，在C++中通过根据vtable指针分派调用到虚函数来实现。

在C++中，你很少见到将union用作和类型，因为union有严格的限制。 甚至不能将std::string放入union中，因为它有一个拷贝构造函数。

6.4类型的代数

单独使用，积类型和和类型可以用来定义各种有用的数据结构，但真正的力量来自于将两者结合起来。再次，我们在调用组合的力量。

让我们总结一下我们迄今为止所发现的内容。我们已经看到了两个交换的幺半群结构作为类型系统的基础：和类型以 Void作为中性元素，积类型以单位类型 ()作为中性元素。 我们希望将它们类比为加法和乘法。 在这个类比中， Void对应于零，而单位 ()对应于一。

让我们看看我们能够将这个类比推到多远。 例如，零乘以零等于零吗？ 换句话说，具有一个组件为 Void的乘积类型是否与 Void同构？ 例如，是否有可能创建一个由 Int和 Void组成的对？

要创建一对，你需要两个值。 虽然你可以轻松地得到一个整数，但是没有 Void类型的值。因此，对于任何类型 a，类型 (a, Void)是不可居住的——没有值——因此等价于 Void。 换句话说，$a \times 0 = 0$。

将加法和乘法联系起来的另一件事是分配律：

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

对于乘积类型和和类型，它是否也成立？ 是的，它成立——通常是同构的。左边对应的类型是：

```
(a, Either b c)
```

右边对应的是类型：

```
要么 (a, b) (a, c)
```

这是将它们转换为另一种方式的函数：

```
prodToSum :: (a, Either b c) -> Either (a, b) (a, c)
prodToSum (x, e) =
    case e of
```

```
        Left y -> Left (x, y)
        Right z -> Right (x, z)
```

这是另一种方式的函数：

```
sumToProd :: Either (a, b) (a, c) -> (a, Either b c)
sumToProd e =
    case e of
        Left (x, y) -> (x, Left y)
        Right (x, z) -> (x, Right z)
```

在函数内部用于模式匹配的 case of语句。
每个模式后面都跟着一个箭头和要评估的表达式当模式匹配时。
例如，如果你用值调用 prodToSum：

```
prod1 :: (Int, Either String Float)
prod1 = (2, Left "Hi!")
```

在情况 e 的情况下，将等于 Left "嗨！"。 它将匹配模式Left y，将 "嗨！"替换为 y。 由于 x已经匹配到 2，因此 case of子句和整个函数的结果将为 Left (2, "嗨！")，正如预期的那样。

我不打算证明这两个函数是彼此的反函数，但是如果你仔细思考，它们必须是！它们只是简单地重新打包了两个数据结构的内容。 它是相同的数据，只是格式不同。

数学家给这两个交织在一起的幺半群起了一个名字：它被称为半环。 它不是一个完整的环，因为我们无法定义类型的减法。 这就是为什么半环有时被称为环，这是对“没有 n”（负数）的一个双关语。 但是除此之外，我们可以得到一个

从翻译关于自然数的语句（构成一个环）到关于类型的语句中，我们可以得到很多有用的信息。 下面是一张有些有趣条目的翻译表：

数字类型	
0	空
1	()
□ + □	Either a b = Left a | Right b
□ × □	(a, b) orPair a b = Pair a b
2 = 1 + 1	data Bool = True | False
1 + □	data Maybe = Nothing | Just a

列表类型非常有趣，因为它被定义为一个方程的解。 我们要定义的类型出现在方程的两边：

```
数据 List a = Nil | Cons a (List a)
```

如果我们进行常规的替换，并将List a替换为x，我们得到以下方程：

```
x = 1 + a * x
```

我们无法使用传统的代数方法来解决它，因为我们无法对类型进行减法或除法。 但我们可以尝试一系列的替换，将右边的x逐步替换为(1 + a*x)，并使用分配律。 这导致了以下一系列的替换：

```
x = 1 + a*x
x = 1 + a*(1 + a*x) = 1 + a + a*a*x
x = 1 + a + a*a*(1 + a*x) = 1 + a + a*a + a*a*a*x
```

...
x = 1 + a + a*a + a*a*a + a*a*a*a...

我们最终得到了一个无限的乘积（元组）的和，可以解释为：列表要么为空，1；要么是一个单元素，a；要么是一对元素，a*a；要么是一个三元组，a*a*a；等等... 好吧，这正是列表的定义 -

一串a的字符串！列表还有更多的内容，我们在学习了函子和不动点之后会回到它们以及其他递归数据结构。

解方程的符号变量 - 这就是代数！这就是给这些类型命名的原因：代数数据类型。

最后，我应该提到类型代数的一个非常重要的解释。 注意，两种类型 a和 b的乘积必须包含类型 a和类型 b的值，这意味着两种类型都必须有实例。 另一方面，两种类型的和包含类型 a或类型 b的值，所以只要其中一种类型有实例就足够了。 逻辑和和或也构成一个半环，它也可以映射到类型理论中：

逻辑	类型
假	空
真	()
□ || □	Either a b = Left a | Right b
□ && □	(a, b)

这个类比更深入，并且是逻辑和类型理论之间的柯里-霍华德同构的基础。 当我们讨论函数类型时，我们会回到这个问题。

6.5挑战

1. 展示 Maybe a和 Either () a之间的同构。
2. 这是在Haskell中定义的一个和类型：

   ```
   data Shape = Circle Float
              | Rect Float Float
   ```

   当我们想要定义一个像 area这样作用于 Shape的函数时，我们通过对两个构造函数进行模式匹配来实现：

   ```
   area :: Shape -> Float
   area (Circle r) = pi * r * r
   area (Rect d h) = d * h
   ```

   在C++或Java中实现 Shape作为一个接口，并创建两个类： Circle和 Rect。 将 area实现为一个虚函数。
3. 继续上一个例子：我们可以很容易地添加一个新的函数 circ来计算一个 Shape的周长。 我们可以在不修改 Shape的定义的情况下完成这个操作：

   ```
   circ :: Shape -> Float
   circ (Circle r) = 2.0 * pi * r
   circ (Rect d h) = 2.0 * (d + h)
   ```

   将 circ添加到你的C++或Java实现中。 你需要修改原始代码的哪些部分?
4. 进一步继续：向 Shape添加一个新的形状 Square，并进行所有必要的更新。 在Haskell与C++或Java中，你需要修改哪些代码? （即使你不是Haskell程序员，修改应该是相当明显的。）5. 证明对于类型（在同构的情况下），□ + □ = 2 × □成立。 请记住，根据我们的翻译表， 2对应于 Bool。

7

函子

尽管听起来像是老生常谈，但我还是要说一下关于函子的事情一个：函子是一个非常简单但强大的概念。范畴论中充满了这些简单而强大的思想。函子是范畴之间的映射。给定两个范畴 □和 □，一个函子□将 □中的对象映射到 □中的对象——它是一个对象的函数。如果 □是 □中的一个对象，我们将其在 □中的映像写作□ □（不使用括号）。但是一个范畴不仅仅是对象——它还包括连接它们的态射。函子也映射态射——它是一个态射的函数。

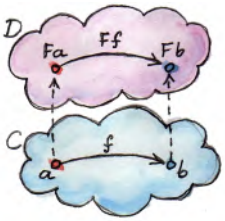
但它不会随意映射态射——它保留连接关系。
因此，如果一个态射 □在 □中连接对象 □和对象 □，

□ : : □ → □

"□在 □中的图像，□ □，将把 □的图像连接到 □的图像上：

□ □ : : □ □ → □ □

"（这是数学和Haskell符号的混合，希望现在有意义了。当应用函子到对象或态射时，我不会使用括号。）正如你所看到的，函子保留了范畴的结构：在一个范畴中连接的东西也会在另一个范畴中连接。

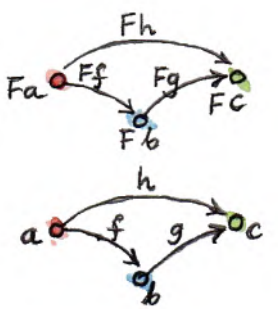


" "但范畴的结构还有更多：还有态射的组合。 "如果 *h*是 □和 □的组合：

$$h = □.□$$

"我们希望它在 □下的图像是 □和 □的图像的组合：

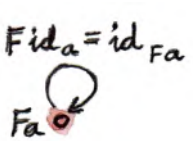
$$□\ h = □\ □\ .\ □\ □$$



"最后，我们希望在 □中的所有恒等态射被映射为 □中的恒等态射：

$$□\ \mathrm{id}_{□} = \mathrm{id}_{\mathrm{id}\ □}$$

在这里，$_{\square}$是对象 $\square$的恒等态，而 $\mathrm{id}_{\square\ \square}$是$\square\ \square$的恒等态。 请注意，这些条件使得函子比普通函数更加受限。 函子必须保持范畴的结构。 如果你将范畴想象成由一系列对象通过态射连接在一起的集合，那么函子不允许在这个结构中引入任何破坏。 它可以将对象合并在一起，可以将多个态射粘合成一个，但是它绝不能破坏事物。



这种不破坏的约束类似于你在微积分中所了解的连续性条件。 从这个意义上说，函子是“连续的”（尽管对于函子存在着更严格的连续性概念）。 就像函数一样，函子既可以进行折叠又可以进行嵌入。 当源范畴远小于目标范畴时，嵌入方面更加突出。 在极端情况下，源范畴可以是平凡的单对象范畴——一个只有一个对象和一个态射（恒等态射）的范畴。 从单对象范畴到任何其他范畴的函子只需在该范畴中选择一个对象。 这完全类似于从单元素集合到目标集合的态射选择元素的性质。 最大折叠函子被称为常函子 $\Delta_{\square}$。 它将源范畴中的每个对象映射到目标范畴中的一个选定对象 $\square$。 它还将源范畴中的每个态射映射到恒等态射 $\mathrm{id}_{\square}$。 它就像一个黑洞，将一切都压缩成一个奇点。 当我们讨论极限和余极限时，我们将会更多地了解这个函子。

7.1编程中的函子

让我们回到现实并谈论编程。 我们有我们的类型和函数的范畴。 我们可以谈论将这个范畴映射到自身的函子 - 这样的函子被称为自函子。 那么在类型的范畴中，什么是自函子？ 首先，它将类型映射到类型。 我们已经看到了这种映射的例子，也许没有意识到它们只是这样的映射。 我正在谈论通过其他类型参数化的类型定义。 让我们看几个例子。

7.1.1 Maybe函子

Maybe的定义是从类型a到类型Maybe a的映射：

```
data Maybe a = 无 | 有 a
```

这里有一个重要的细微之处：Maybe本身不是一个类型，它是一个类型构造器。 你必须给它一个类型参数，比如Int或Bool，才能将其转换为类型。没有任何参数的Maybe表示类型上的一个函数。 但是我们能把Maybe变成一个函子吗？ （从现在开始，当我在编程的上下文中谈论函子时，我几乎总是指自函子。）函子不仅是对象（这里是类型）的映射，还是态射（这里是函数）的映射。 对于从a到b的任何函数：

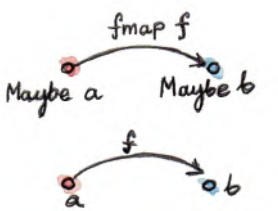
```
f :: a -> b
```

我们希望从 Maybe a到 Maybe b生成一个函数。 为了定义这样一个函数，我们需要考虑两种情况，对应于 Maybe的两个构造器。 Nothing的情况很简单：我们只需返回 Nothing。 如果参数是 Just，我们将应用函数 f到其内容上。 因此， Maybe下的 f的图像是这个函数：

```
f' :: Maybe a -> Maybe b
f' Nothing = Nothing
f' (Just x) = Just (f x)
```

（顺便说一下，在Haskell中你可以在变量名中使用撇号，这在这种情况下非常方便。）在Haskell中，我们将函子的态射映射部分实现为一个名为fmap的高阶函数。 对于 Maybe，它具有以下签名：

```
fmap :: (a -> b) -> (Maybe a -> Maybe b)
```



我们经常说， fmap提升了一个函数。 提升后的函数作用于 Maybe值。 通常情况下，由于柯里化，这个签名可以有两种解释：作为一个参数的函数（本身是一个函数（a -> b）），返回一个函数（Maybe a -> Maybe b）；或者作为两个参数的函数返回 Maybe b：

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

根据我们之前的讨论，这是我们实现 fmap的方式 Maybe:

```
fmap _ Nothing = Nothing
 fmap f (Just x) = Just (f x)
```

为了证明类型构造器 Maybe与函数 fmap构成一个函子，我们必须证明 fmap保持恒等性和组合性。 这些被称为“函子定律”，但它们只是确保范畴结构的保持。

7.1.2等式推理

为了证明函子定律，我将使用等式推理，这是Haskell中常见的证明技巧。它利用了Haskell函数被定义为相等性的事实：左边等于右边。 你总是可以将一个替换为另一个，可能需要重新命名变量以避免名称冲突。 将其视为内联函数，或者反过来，将表达式重构为函数。 让我们以恒等函数为例：

```
id x = x
```

如果你在某个表达式中看到，例如， id y，你可以用 y（内联）替换它。 此外，如果你看到 id应用于一个表达式，比如id (y + 2)，你可以用表达式本身 (y + 2)替换它。 而且，这种替换方式是双向的：你可以用 id e（重构）替换任何表达式 e。 如果一个函数通过模式匹配来定义，你可以独立使用每个子定义。 例如，给定 fmap的上述定义，你可以用 Nothing替换 fmap f Nothing，或者反过来。 让我们看看这在实践中是如何工作的。 让我们从保持身份的角度开始：

```
fmap id = id
```

有两种情况需要考虑： Nothing和 Just。 这是第一种情况（我使用Haskell伪代码将左边转换为右边）：

```
    fmap id Nothing
= { fmap的定义 }
    Nothing
= { id的定义 }
    id Nothing
```

注意，在最后一步中，我反向使用了id的定义。 我用id Nothing替换了表达式Nothing。 在实践中，你可以通过“两头烧蜡烛”的方式进行证明，直到你在中间得到相同的表达式——在这里是Nothing。 第二种情况也很简单：

```
    fmap id (Just x)
= { fmap的定义 }
    Just (id x)
= { id的定义 }
    Just x
= { id的定义 }
    id (Just x)
```

现在，让我们证明fmap保持组合：

```
fmap (g . f) = fmap g . fmap f
```

首先是Nothing的情况：

```
    fmap (g . f) Nothing
= { fmap的定义 }
    Nothing
= { fmap的定义 }
    fmap g Nothing
= { fmap的定义 }
     fmap g (fmap f Nothing)
```

然后是只是情况：

```
    fmap (g . f) (Just x)
= { fmap的定义 }
    Just ((g . f) x)
= { 组合的定义 }
    Just (g (f x))
= { fmap的定义 }
    fmap g (Just (f x))
= { fmap的定义 }
     fmap g (fmap f (Just x))
= { 组合的定义 }
    (fmap g . fmap f) (Just x)
```

值得强调的是，等式推理对于具有副作用的C++风格的“函数”是不起作用的。考虑以下代码：

```
     int square(int x) {
          return x * x;
}

int counter() {
       static int c = 0;
       return c++;
}

double y = square(counter());
```

使用等式推理，你可以将 square内联，得到：

```
double y = counter() * counter();
```

这绝对不是一个有效的转换，它不会产生相同的结果。 尽管如此，如果你将 square实现为宏，C++编译器将尝试使用等式推理，结果将是灾难性的。

7.1.3可选的

函子在Haskell中很容易表达，但它们可以在任何支持通用编程和高阶函数的语言中定义。 让我们考虑C++中的Maybe的类似物，模板类型optional。这是一个实现的草图（实际实现要复杂得多，涉及参数可能传递的各种方式，复制语义以及C++特有的资源管理问题）：

```
template<class T>
class optional {
    bool _isValid; // 标签
    T _v;
public:
    optional()          : _isValid(false) {}                    // 无
    optional(T x) : _isValid(true) , _v(x) {} // 有
    bool isValid() const { return _isValid; }
    T val() const { return _v; } };
```

这个模板提供了函子定义的一部分：类型的映射。 它将任何类型T映射到一个新类型optional<T>。 让我们定义它对函数的作用：

```
template<class A, class B>
std::function<optional<B>(optional<A>)>
fmap(std::function<B(A)> f) {
    return [f](optional<A> opt) {
        if (!opt.isValid())
            return optional<B>{};
        else
            return optional<B>{ f(opt.val()) };
    };
}
```

这是一个高阶函数，它接受一个函数作为参数并返回一个函数。这是它的非柯里化版本：

```
template<class A, class B>
optional<B> fmap(std::function<B(A)> f, optional<A> opt) {
    if (!opt.isValid())
        return optional<B>{};
    else
        return optional<B>{ f(opt.val()) };
}
```

还有一种选择，可以将fmap作为optional的模板方法。在C++中，这种选择过多使得抽象化函子模式成为一个问题。 函子应该是一个继承的接口吗（不幸的是，你不能有模板虚函数）？ 它应该是一个柯里化的或非柯里化的自由模板函数吗？ C++编译器能够正确推断缺失的类型吗，还是应该明确指定？

考虑一个情况，输入函数 f将一个 int映射到一个 bool。编译器将如何确定 g的类型：

```
auto g = fmap(f);
```

特别是如果将来有多个重载 fmap的函子，编译器将如何处理？（我们很快会看到更多函子。）

7.1.4类型类

那么Haskell如何处理抽象的函子呢？ 它使用类型类机制。 类型类定义了一组支持共同接口的类型。 例如，支持相等性的对象的类定义如下：

```
class Eq a where
    (==) :: a -> a -> Bool
```

这个定义说明了类型 a是类 Eq的实例，如果它支持操作符 (==)，该操作符接受两个类型为 a的参数，并返回一个 Bool。 如果你想告诉Haskell一个特定的类型是 Eq的，你必须将其声明为该类的一个实例，并提供 (==)的实现。
例如，给定一个二维点的定义(两个乘积类型)
浮点数):

```
数据类型 Point = Pt 浮点数 浮点数
```

你可以定义点的相等性:

```
实例 Eq Point where
    (Pt x y) == (Pt x' y') = x == x' && y == y'
```

在这里，我使用了运算符 (==)(我正在定义的那个)作为中缀位于两个模式 (Pt x y)和 (Pt x' y')之间。 函数的主体部分在单个等号之后。 一旦声明了 Point为 Eq的实例，你可以直接比较点的相等性。 请注意，与C++或Java不同，你不必在定义 Point时指定 Eq类（或接口）- 你可以在客户端代码中稍后进行。 类型类也是Haskell中唯一用于函数（和运算符）重载的机制。 我们将需要这个来重载不同函子的 fmap。然而，有一个复杂之处：函子不是作为一种类型定义的，而是作为类型的映射，即类型构造器。 我们需要一个类型类，它不是一族类型，就像 Eq那样，而是一族类型构造器。 幸运的是，Haskell的类型类既适用于类型构造器，也适用于类型。 所以这是 Functor类的定义:

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

它规定，如果存在一个具有指定类型签名的函数 fmap，那么 f就是一个范畴论。 小写字母 f是一个类型变量，类似于类型变量 a和 b。 然而，编译器能够通过观察其用法来推断出它代表的是一个类型构造器而不是一个类型：作用于其他类型，如 f a和 f b。 因此，在声明 Functor的实例时，你必须给它一个类型构造器，就像 Maybe一样：

```
instance Functor Maybe where
        fmap _ Nothing = Nothing
        fmap f (Just x) = Just (f x)
```

顺便说一下， Functor类以及它对于很多简单数据类型的实例定义，包括 Maybe，在标准Prelude库中。

7.1.5 C++中的函子

我们能在C++中尝试相同的方法吗？ 类型构造器对应于模板类，例如optional，因此通过类比，我们将使用模板模板参数F对fmap进行参数化。 这是它的语法：

```
template<template<class> F, class A, class B>
F<B> fmap(std::function<B(A)>, F<A>);
```

我们希望能够为不同的函子专门化这个模板。 不幸的是，在C++中，对于模板函数的部分专门化是被禁止的。你不能写：

```
template<class A, class B>
optional<B> fmap<optional>(std::function<B(A)> f, optional<A> opt)
```

相反，我们必须退回到函数重载，这使我们回到了未柯里化的 fmap 的原始定义：

```
template<class A, class B>
optional<B> fmap(std::function<B(A)> f, optional<A> opt) {
    if (!opt.isValid())
        return optional<B>{};
    else
        return optional<B>{ f(opt.val()) };
}
```

这个定义是有效的，但只是因为 fmap 的第二个参数选择了重载。 它完全忽略了更通用的定义 fmap。

7.1.6列表函子

为了对范畴论在编程中的作用有一些直觉，我们需要看更多的例子。 任何由另一个类型参数化的类型都是一个函子的候选。 通用容器是由存储的元素类型参数化的，所以让我们看一个非常简单的容器，即列表：

```
数据 List a = Nil | Cons a (List a)
```

我们有类型构造器 List，它是从任何类型到类型List a的映射 a到类型 List a的映射。 为了证明 List是一个函子，我们需要定义函数的提升：给定一个函数 a -> b定义一个函数 List a -> List b:

```
fmap :: (a -> b) -> (List a -> List b)
```

作用于 List a的函数必须考虑两种情况，对应于两个列表构造器。 对于 Nil情况来说很简单——只需返回 Nil ——对于空列表，你无法做太多事情。 对于 Cons情况来说有点棘手，因为它涉及到递归。 所以让我们暂停一下，考虑一下我们要做什么。我们有一个由 a组成的列表，一个将 a转换为 b的函数 f，我们想生成一个由 b组成的列表。 显而易见的做法是使用 f将列表的每个元素从 a转换为 b。 在实践中，我们如何做到这一点，考虑到一个（非空）列表被定义为一个头部和一个尾部的 Cons？ 我们将 f应用于头部，并将提升的（使用fmap定义的） f应用于尾部。 这是一个递归定义，因为我们正在定义提升的
f以提升 f的形式：

```
 fmap f (Cons x t) = Cons (f x) (fmap f t)
```

请注意，在右边， fmap f被应用于一个比我们正在定义的列表更短的列表 - 它被应用于它的尾部。
我们递归地向更短的列表递归，所以我们最终会达到空列表，或 Nil。 但是正如我们之前决定的那样， fmap f作用于 Nil返回 Nil，从而终止递归。 为了得到最终结果，我们使用 Cons构造函数将新头部 (f x)与新尾部 (fmap f t)组合起来。 将所有这些放在一起，这是列表函子的实例声明：

```
instance Functor List where
      fmap _ Nil = Nil
       fmap f (Cons x t) = Cons (f x) (fmap f t)
```

如果你更熟悉C++，考虑一下std::vector的情况，它可以被认为是最通用的C++容器。 对于std::vector的 fmap的实现只是对其进行了薄封装
std::transform:

```
template<class A, class B>
std::vector<B> fmap(std::function<B(A)> f, std::vector<A> v) {
        std::vector<B> w;
        std::transform( std::begin(v)
                                , std::end(v)
                                , std::back_inserter(w)
                                , f);
        return w;
}
```

例如，我们可以用它来对一系列数字的元素进行平方：

```
std::vector<int> v{ 1, 2, 3, 4 };
auto w = fmap([](int i) { return i*i; }, v);
std::copy( std::begin(w)
                , std::end(w)
                , std::ostream_iterator(std::cout, ", "));
```

大多数C++容器都是通过实现可以传递给std::transform的迭代器而成为函子的（std::transform是fmap的更原始的表亲）。 不幸的是，在迭代器和临时变量的常规混乱下，函子的简洁性丧失了（请参见上面fmap的实现）。 我很高兴地说，新提出的C++范围库使范畴的本质更加明显。

7.1.7读者函子

现在你可能已经形成了一些直觉 - 例如，函子是某种容器 - 让我给你展示一个一开始看起来非常不同的例子。 考虑一种类型映射 `a` 到返回 `a`的函数类型。 我们还没有深入讨论函数类型 - 完整的范畴论处理即将到来 - 但作为程序员，我们对此有一些了解。

)构建，它接受两个类型：参数类型和结果类型。 你已经在中缀形式 a -> b中看到过它，但它也可以在前缀形式中使用，当加上括号时： (->) a b

就像普通函数一样，多个参数的类型函数可以部分应用。

因此，当我们只提供一个类型参数给箭头时，它仍然期望另一个类型。这就是为什么：) a

就像普通函数一样，多个参数的类型函数可以部分应用。

是一个类型构造器。 它需要另一个类型 b来生成一个完整的类型 a -> b。 就目前而言，它定义了一个由 a参数化的类型构造器家族。 让我们看看这是否也是一个函子家族。 处理两个类型参数可能会有点混乱，所以让我们进行一些重命名。 让我们将参数类型称为 r，结果类型称为 a，与我们之前的函子定义保持一致。 因此，我们的类型构造器接受任何类型 a并将其映射到类型 r -> a。 为了证明它是一个函子，我们希望将一个函数 a -> b提升为一个接受 r -> a并返回r -> b的函数。 这些是使用类型构造器(->) r分别作用于 a和 b形成的类型。 这是 fmap应用于这种情况的类型签名：

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

我们需要解决以下难题：给定一个函数 f :: a -> b和一个函数 g :: r -> a，创建一个函数 r -> b。 我们只有一种方式可以组合这两个函数，而且结果正好是我们需要的。 所以这是我们的 fmap的实现：

```
instance Functor ((->) r) where
        fmap f g = f . g
```

它就是这样工作的！如果你喜欢简洁的符号，这个定义可以进一步简化，因为组合可以用前缀形式重写：

```
 fmap f g = (.) f g
```

并且参数可以省略，从而得到两个函数的直接相等：

```
fmap = (.)
```

将类型构造器 (->) r与上述 fmap的实现结合起来，被称为读取器函子。

7.2函子作为容器

我们在编程语言中看到了一些函子的例子，它们定义了通用的容器，或者至少包含了某种类型的值。 读取器函子似乎是一个例外，因为我们不认为函数是数据。 但是我们已经看到纯函数可以被记忆化，并且函数执行可以转换为表查找。表是数据。 相反，由于Haskell的惰性求值，传统的容器，比如列表，实际上可以被实现为一个函数。 例如，考虑一个无限自然数列表，可以被紧凑地定义为：

```
nats :: [整数]
nats = [1..]
```

在第一行，一对方括号是Haskell的内置类型构造函数，用于列表。 在第二行，方括号用于创建一个列表字面量。显然，像这样的无限列表无法存储在内存中。编译器将其实现为一个按需生成整数的函数。 Haskell有效地模糊了数据和代码之间的区别。 列表可以被视为一个函数，而函数可以被视为将参数映射到结果的表。 如果函数的定义域是有限的且不太大，后者甚至可能是实用的。 然而，将strlen实现为表查找是不实际的，因为有无限多个不同的字符串。 作为程序员，我们不喜欢无限，但在范畴论中，你会学会吃掉无限。 无论是所有字符串的集合还是过去、现在和未来的所有可能状态的集合，我们都可以处理它！因此，我喜欢将函子对象（由自函子生成的类型的对象）视为包含其参数化类型的值或值，即使这些值在物理上不存在。 函子的一个例子是C++的std::future，它可能在某个时刻包含一个值，但不能保证；如果你想访问它，你可能会阻塞等待另一个线程执行完毕。 另一个例子是Haskell的IO对象，它可能包含用户输入，或者显示在监视器上的未来版本的“Hello World！”。 根据这种解释，函子对象是可能包含其参数化类型的值或值的东西。 或者它可能包含生成这些值的方法。 我们对能够访问这些值一点也不关心——这完全是可选的，超出了函子的范围。

我们只关心能够使用函数来操作这些值。如果可以访问这些值，那么我们应该能够看到这种操作的结果。如果不能访问，那么我们只关心

这是为了展示我们对于能够访问函子对象内部值的能力不关心，只要演示操作正确地组合以及使用恒等函数进行操作不改变任何内容。这里有一个完全忽略其参数的类型构造器：

```
数据常量 c a = 常量 c
```

Const类型构造函数接受两个类型，c和 a。 就像我们之前对箭头构造函数所做的那样，我们将部分应用它来创建一个函子。 数据构造函数（也称为 Const）只接受一个类型为 c的值。 它不依赖于 a。 对于这个类型构造函数， fmap的类型是：

```
fmap :: (a -> b) -> Const c a -> Const c b
```

由于函子忽略了其类型参数，因此fmap的实现可以自由地忽略其函数参数 - 函数没有任何操作对象：

```
instance Functor (Const c) where
    fmap _ (Const v) = Const v
```

这在C++中可能会更清晰一些（我从来没有想过我会说这些话！），因为类型参数和值之间有更强的区别 - 类型参数是编译时的，而值是运行时的：

```
template<class C, class A>
struct Const {
    Const(C v) : _v(v) {}
    C _v;
};
```

fmap的C++实现也忽略了函数参数，并且基本上重新转换了 Const 参数而不改变其值：

```
template<class C, class A, class B>
Const<C, B> fmap(std::function<B(A)> f, Const<C, A> c) {
                return Const<C, B>{c._v};
}
```

尽管它很奇怪，但 Const函子在许多构造中起着重要作用。 在范畴论中，它是我之前提到的 Δ□函子的特殊情况 - 一个黑洞的内函子情况。 我们将在未来看到更多它的应用。

7.3函子组合

说服自己相信范畴之间的函子组合，就像集合之间的函数组合一样，并不难。 两个函子的组合，在作用于对象时，就是它们各自对象映射的组合；在作用于态射时也是如此。 经过两个函子的转换，恒等态射变为恒等态射，态射的组合变为态射的组合。 实际上，并没有太多复杂的东西。特别是，组合自函子非常容易。 还记得函数 maybeTail吗？ 我将使用Haskell内置的列表实现来重写它：

```
maybeTail :: [a] -> Maybe [a]
maybeTail [] = Nothing
maybeTail (x:xs) = Just xs
```

(我们曾经称之为 Nil的空列表构造器被替换为空的方括号 []。 Cons 构造器被替换为

中缀运算符 :(冒号)。 maybeTail的结果是一个由两个函子 Maybe和 []组成的类型的组合，作用于一个。每个函子都配备了自己的 fmap版本，但是如果我们想要将一些函数 f应用于组合的内容：一个 Maybe列表，怎么办呢?
我们必须突破两层函子。 我们可以使用 fmap来突破外层 Maybe。但是我们不能直接将 f放入 Maybe中，因为 f不能在列表上工作。我们必须发送 (fmap f)来操作内部列表。 例如，让我们看看如何对 Maybe整数列表的元素进行平方：

```
square x = x * x

mis :: Maybe [Int]
mis = Just [1, 2, 3]

mis2 = fmap (fmap square) mis
```

编译器在分析类型后，将确定对于外部的fmap，应使用 Maybe实例的实现，对于内部的，则使用列表函子的实现。 上面的代码可能不会立即显而易见地被重写为：

```
mis2 = (fmap . fmap) square mis
```

但请记住， fmap可以被视为只有一个参数的函数：

```
 fmap :: (a -> b) -> (f a -> f b)
```

在我们的情况下， (fmap . fmap)的第二个 fmap以以下参数作为其参数：

```
square :: Int -> Int
```

并返回以下类型的函数：

[Int] -> [Int]

然后，第一个 fmap接受该函数并返回一个函数：

Maybe [Int] -> Maybe [Int]

最后，将该函数应用于 mis。 因此，两个函子的组合是一个函子，其 fmap是相应fmap的组合。回到范畴论：显然，函子的组合是可结合的（对象的映射是可结合的，态射的映射也是可结合的）。而且，在每个范畴中还有一个平凡的恒等函子：它将每个对象映射到其自身，将每个态射映射到其自身。 因此，函子具有与某个范畴中的态射相同的属性。 但是，这个范畴是什么？ 它必须是一个对象是范畴，态射是函子的范畴。

这是一个范畴的范畴。 但是一个包含所有范畴的范畴将会包括它自己，我们将陷入与使所有集合不可能的相同类型的悖论。 然而，有一个包含所有小范畴的范畴叫做然而然而然而（它很大，所以不能成为它自己的成员）。 小范畴是指对象形成一个集合，而不是比集合更大的东西。请注意，在范畴论中，即使是无限可数的集合也被认为是“小”的。我想提到这些事情，因为我发现在许多抽象层次上我们可以识别出相同的结构重复出现是非常令人惊奇的。 我们稍后会看到函子也形成范畴。

7.4挑战

1. 我们可以通过定义将Maybe类型构造器转变为函子吗:

```
fmap _ _ = Nothing
```

它忽略了它的两个参数吗？（提示：检查函子定律。）

2. 为读者函子证明函子定律。 提示：这非常简单。
3. 用你第二喜欢的语言实现读者函子
   （第一喜欢的当然是Haskell）。
4. 为列表函子证明函子定律。 假设定律对你应用它的列表的尾部部分成立（换句话说，使用归纳法）。

8

函子性

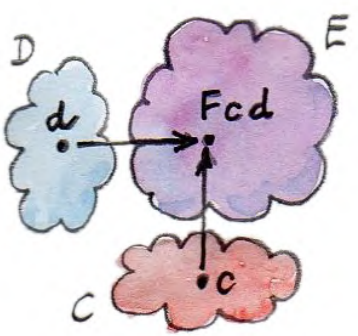
N现在你知道了什么是函子，并且看到了一些例子，让我们看看如何从小的函子构建更大的函子。
特别有趣的是看看哪些类型构造器（对应于范畴中对象之间的映射）可以扩展为函子（包括映射对象之间的态射）。

8.1双函子

由于函子是 □ □ □(范畴的范畴）中的态射，关于态射的很多直觉 — 尤其是函数 — 也适用于函子。 例如，就像你可以有两个参数的函数一样，你可以有两个参数的函子，或者双函子。 对于对象，双函子将来自范畴 □和范畴 □的每对对象映射到范畴 □中的一个对象。注意这是

只是说它是从一个范畴的笛卡尔积o另一个范畴 𝒟的映射。

这很直观。 但是，函子性意味着双函子也必须映射态射。



不过，这次它必须映射一对态射，一个来自 𝒞，一个来自 𝒟，到一个 ℰ中的态射。

同样，一对态射只是一个在乘积范畴𝒞 × 𝒟到 ℰ中的单个态射。 我们将笛卡尔积范畴中的态射定义为从一个对象对到另一个对象对的一对态射。 这些态射对可以以明显的方式进行组合： 组合是可结合的，并且有一个恒等态射 - 一对恒等态射(id,id)。

$$(g, g') \circ (f, f') = (g \circ f, g' \circ f')$$

因此，范畴的笛卡尔积确实是一个范畴。 所以，范畴的笛卡尔积确实是一个范畴。

但是一个更简单的思考双函子的方法是它们在两个参数上都是函子。 所以不需要将函子的函子律 — 结合律和恒等保持律 — 翻译成双函子，只需要分别对每个参数进行检查即可。 如果你有一个从一对范畴到第三个范畴的映射，并且你证明它在每个参数上都是函子（即保持另一个参数不变），那么这个映射自动成为一个双函子。 通过函子性我指的是它像一个诚实的函子一样作用于态射。

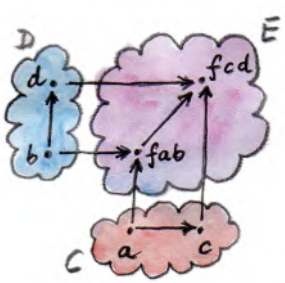
让我们在Haskell中定义一个双函子。在这种情况下，所有三个范畴都是相同的：Haskell类型的范畴。双函子是一种类型构造器

它是一个接受两个类型参数的结构体。下面是直接从库Control.Bifunctor中取出的Bifunctor类型类的定义：

```
双函子 f 的类
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
  bimap g h = first g . second h
   first :: (a -> c) -> f a b -> f c b
  first g = bimap g id
  second :: (b -> d) -> f a b -> f a d
  second = bimap id
```

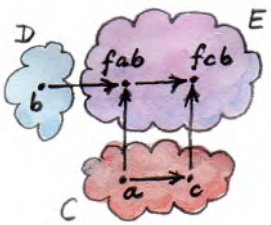
类型变量 f表示双函子。 你可以看到，在所有类型签名中，它总是应用于两个类型参数。 第一个类型签名定义 bimap：同时映射两个函数。 结果是一个提升的函数，(f a b -> f c d)，操作由双函子的类型构造器生成的类型。 有一个默认的bimap实现，用 first和 second来定义，这表明只需要分别在每个参数上具有函子性质就能定义一个双函子。 （在范畴论中，这一般不成立，因为这两个映射可能不可交换： first g . second h可能与

第二个 h . 第一个 g.)

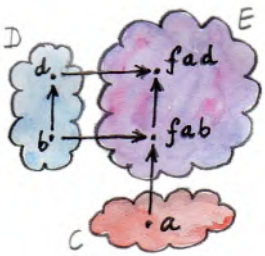


双映射

另外两个类型签名，第一个和第二个，分别是第一个和第二个参数中的 fmap，分别证明了 f的函子性。



第一个



第二个

类型类定义提供了 bimap的默认实现。

在声明 Bifunctor的实例时，你可以选择实现 bimap并接受 first和 second的默认值，或者同时实现 first和 second并接受bimap的默认值（当然，你也可以实现这三个方法，但是你需要确保它们之间的关系）。

8.2乘积和余积双函子

一个重要的双函子的例子是范畴积 — 由一个通用构造定义的两个对象的积。 如果对于任意一对对象都存在积，那么从这些对象到积的映射是双函子的。这在一般情况下都成立，在Haskell中尤其如此。下面是对于一个pair构造器的 Bifunctor实例 — 最简单的积类型：

```
实例Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)
```

选择并不多： bimap只是将第一个函数应用于对对的第一个组件，并将第二个函数应用于第二个组件。 鉴于类型，代码几乎可以自己编写：

```
bimap :: (a -> c) -> (b -> d) -> (a, b) -> (c, d)
```

这里的双函子的作用是创建类型对，例如：

```
(,) a b = (a, b)
```

通过对偶性，如果在一个范畴中为每对对象定义了余积，则它也是一个双函子。 在Haskell中，这可以通过 Either类型构造函数作为 Bifunctor的实例来说明：

```
实例Bifunctor Either where
    bimap f _ (Left x) = Left (f x)
    bimap _ g (Right y) = Right (g y)
```

这段代码也可以自动生成。

现在，还记得我们讨论过的幺半范畴吗？ 一个幺半范畴定义了作用于对象上的二元运算符，以及一个单位对象。 我提到过，我提我提我提是一个关于笛卡尔积的幺半范畴，其中单例集合作为单位元。 它也是一个关于不交并的幺半范畴，其中空集作为单位元。 我没有提到的是，幺半范畴的一个要求是二元运算符是一个双函子。 这是一个非常重要的要求 - 我们希望幺半积与范畴的结构相容，而范畴的结构是由态射定义的。 我们现在离完整定义幺半范畴更近了一步（在此之前我们还需要了解自然性）。

8.3函子代数数据类型

我们已经看到了几个参数化数据类型的例子，它们被证明是函子 - 我们能够为它们定义 fmap。 复杂的数据类型是由简单的数据类型构建而来的。 特别地，代数数据类型（adts）是使用求和和乘积创建的。 我们刚刚看到求和和乘积是函子。我们也知道函子可以组合。 因此，如果我们能够证明adts的基本构建块是函子，我们就会知道参数化的代数数据类型也是函子。那么参数化代数数据类型的基本构建块是什么？ 首先，有些项与函子的类型参数

无关，比如 Maybe中的 Nothing或 List中的 Nil。 它们等同于 Const函子。 记住， Const函子忽略了它的类型参数（实际上是第二个类型参数，我们感兴趣的那个，第一个类型参数保持不变）。

然后还有一些元素，它们只是简单地封装了类型参数本身，比如 Maybe中的 Just。 它们等同于恒等函子。
我之前提到过恒等函子，作为范畴 Cat中的恒等态射，但没有给出在 Haskell 中的定义。这里是它：

```
data Identity a = Identity a

instance Functor Identity where
    fmap f (Identity x) = Identity (f x)
```

你可以将 Identity看作是一种最简单的容器，它总是存储一个（不可变的）类型为 a的值。

代数数据结构中的其他所有内容都是使用积和和构建的这两个基本元素。

有了这个新的知识，让我们重新审视一下 Maybe类型构造器：

```
data Maybe a = 无 | 有 a
```

它是两种类型的和，并且我们现在知道和是函子的。第一部分的 Nothing可以表示为作用在 a上的 Const () （ Const的第一个类型参数设置为 unit —— 后面我们会看到更有趣的 Const的用法）。 第二部分只是恒等函子的另一个名称。 我们可以将 Maybe定义为以下形式（同构）：

```
type Maybe a = Either (Const () a) (Identity a)
```

所以 Maybe是由双函子 Either和两个函子，
Const ()和 Identity的组合构成的。 （Const实际上是一个双函子，但在这里我们总是部分应用它。）

我们已经看到，函子的组合是一个函子 -
我们很容易相信双函子也是如此。
我们只需要弄清楚双函子与两个函子的组合在态射上的作用方式。 给定两个态射，我们只需用一个函子提升一个，用另一个函子提升另一个。 然后我们用双函子提升所得到的提升态射的对。

我们可以用Haskell来表示这个组合。让我们定义一个数据类型，它的参数是一个双函子 bf（它是一个类型变量，是一个类型构造器，接受两个类型作为参数），两个函子 fu和 gu（分别是接受一个类型变量的类型构造器），以及两个普通的类型 a和 b。 我们将 fu应用于 a， gu应用于 b，然后将 bf应用于得到的两个类型：

```
newtype BiComp bf fu gu a b = BiComp (bf (fu a) (gu b))
```

这是对象或类型的组合。 请注意，在Haskell中我们将类型构造器应用于类型，就像我们将函数应用于参数一样。语法是相同的。

如果你有点迷失，请按照以下顺序将BiComp应用于Either，Const ()，Identity，a和b。 你将恢复我们的基本版本Maybe b（忽略a）。

新的数据类型BiComp在a和b上是一个双函子，但前提是bf本身是一个双函子，fu和gu是函子。编译器必须知道bf将有一个bimap的定义，以及fu和gu将有fmap的定义。 在Haskell中，这被表达为实例声明中的前提条件：一组类约束后跟一个双箭头：

```
实例 (Bifunctor bf, Functor fu, Functor gu) =>
         Bifunctor (BiComp bf fu gu) where
  bimap f1 f2 (BiComp x) = BiComp ((bimap (fmap f1) (fmap f2))
                                   □ x)
```

对于 BiComp的 bimap的实现是基于 bf的 bimap和 fu和 gu的两个 fmap。 当使用 BiComp时，编译器会自动推断所有类型并选择正确的重载函数。

在 bimap的定义中， x的类型为：

```
bf (fu a) (gu b)
```

这是一个相当冗长的表达方式。 外层的 bimap穿透了外层的 bf，而两个 fmap分别深入了 fu和 gu。 如果 f1和 f2的类型分别为：

```
f1 :: a -> a'
f2 :: b -> b'
```

那么最终结果的类型是 bf (fu a') (gu b')：

```
bimap :: (fu a -> fu a') -> (gu b -> gu b')
                         -> bf (fu a) (gu b) -> bf (fu a') (gu b')
```

如果你喜欢拼图游戏，这些类型操作可以提供
数小时的娱乐。

所以事实证明，我们不需要证明 Maybe是一个函子- 这个事实是从它作为两个函子原语的和构造出来的方式中得出的。

一个有洞察力的读者可能会问：如果代数数据类型的函子实例的推导是如此机械化的，难道不能由编译器自动化执行吗？ 确实可以，并且已经实现了。您需要在源文件的顶部包含以下行以启用特定的Haskell扩展：

```
{-# LANGUAGE DeriveFunctor #-}
```

然后在您的数据结构中添加 deriving Functor：

```
数据 Maybe a = Nothing | Just a 派生自函子
```

对应的 fmap将为您实现。代数数据结构的规律性使得

不仅可以派生 Functor的实例，还可以派生其他几个类型类的实例，包括我之前提到的 Eq类型类。 还有一种选择是教编译器派生您自己的类型类的实例，但这有点高级。 然而，思想是一样的：您提供基本构建块、和以及积的行为，让编译器解决其余的问题。

8.4 C++中的函子

如果您是C++程序员，显然在实现函子方面您是自己的。 然而，您应该能够在C++中识别某些类型的代数数据结构。如果将这样的数据结构转换为通用模板，您应该能够快速实现它的fmap。

让我们来看一个树形数据结构，在Haskell中我们将其定义为递归的和类型：

```
数据 Tree a = 叶子 a | 节点 (Tree a) (Tree a)
    派生 Functor
```

正如我之前提到的，C++中实现和类型的一种方式是通过类层次结构。 在面向对象的语言中，将 fmap实现为基类 Functor的虚函数，然后在所有子类中进行重写是很自然的。 不幸的是，这是不可能的，因为 fmap是一个模板，不仅参数化了它所作用的对象的类型（ this指针），还参数化了应用于它的函数的返回类型。在C++中，虚函数不能被模板化。我们将 fmap实现为一个通用的自由函数，并用dynamic_cast替换模式匹配。

为了支持动态转换，基类必须至少定义一个虚函数，所以我们将析构函数定义为虚函数（这在任何情况下都是一个好主意）：

```
template<class T>
struct Tree {
    virtual ~Tree() {};
};
```

叶子只是一个伪装成恒等函子的身份：

```
template<class T>
struct Leaf : public Tree<T> {
        T _label;
        Leaf(T l) : _label(l) {}
};
```

节点是一个乘积类型：

```
template<class T>
struct Node : public Tree<T> {
        Tree<T> * _left;
        Tree<T> * _right;
        Node(Tree<T> * l, Tree<T> * r) : _left(l), _right(r) {}
};
```

在实现 fmap时，我们利用了对 Tree类型的动态分派。 叶子情况应用了 fmap的恒等版本，而节点情况则被视为由两个 Tree函子组成的双函子。 作为一个C++程序员，你可能不习惯用这些术语来分析代码，但这是一种良好的范畴论思维的练习。

```
template<class A, class B>
Tree<B> * fmap(std::function<B(A)> f, Tree<A> * t) {
        Leaf<A> * pl = dynamic_cast <Leaf<A>*>(t);
        if (pl)
                return new Leaf<B>(f (pl->_label));
        Node<A> * pn = dynamic_cast<Node<A>*>(t);
        if (pn)
                return new Node<B>( fmap<A>(f, pn->_left)
                                                , fmap<A>(f, pn->_right));
        return nullptr;
}
```

为了简单起见，我决定忽略内存和资源管理问题，但在实际代码中，你可能会使用智能指针（根据你的策略选择unique或shared）。

将其与Haskell中的 fmap实现进行比较：

```
instance Functor Tree where
    fmap f (Leaf a) = Leaf (f a)
    fmap f (Node t t') = Node (fmap f t) (fmap f t')
```

这个实现也可以由编译器自动推导出来。

8.5写入器函子

我承诺会回到我之前描述的Kleisli范畴。 该范畴中的态射被表示为返回 Writer数据结构的“装饰”函数。

```
type Writer a = (a, String)
```

我说过装饰与自函子有些关联。
而且， Writer类型构造器在 a上是函子的。 我们甚至不需要为它实现 fmap，因为它只是一个简单的乘积类型。

但是克莱斯利范畴和函子之间的关系是什么 —— 一般来说？作为一个范畴，克莱斯利范畴定义了组合和恒等。 让我提醒你，组合是由鱼运算符给出的：

```
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
m1 >=> m2 = \x ->
    let (y, s1) = m1 x
                (z, s2) = m2 y
    in (z, s1 ++ s2)
```

而恒等态射由一个名为 return的函数给出：

```
return :: a -> Writer a
return x = (x, "")
```

事实证明，如果你长时间地观察这两个函数的类型（我是说，长时间地观察），你可以找到一种方法将它们组合起来，以产生一个具有正确类型签名的函数，作为 fmap。像这样：

```
fmap f = id >=> (\x -> return (f x))
```

在这里，鱼操作符结合了两个函数：其中一个是熟悉的id，另一个是一个lambda函数，它将return应用于f对lambda参数的结果。 最难理解的部分可能是对id的使用。 鱼操作符的参数难道不应该是一个接受“正常”类型并返回一个装饰类型的函数吗？嗯，并不完全是这样。没有人说a必须是一个“正常”类型。

它是一个类型变量，所以它可以是任何类型，特别是它可以是一个装饰类型，比如Writer b。所以id将接受Writer a并将其转换为Writer a。 鱼操作符将提取出a的值，并将其作为x传递给lambda函数。 在那里，f将将其转换为b，并且return将对其进行装饰，使其成为Writer b。 将所有这些组合在一起，我们得到一个接受Writer a并返回Writer b的函数，正是fmap应该产生的结果。 将所有这些组合在一起，我们最终得到一个接受Writer a并返

请注意，这个论证非常普遍：你可以用任何类型构造器替换 Writer 只要它支持一个fish操作符和return，你也可以定义 fmap。 所以Kleisli范畴中的装饰总是一个函子。 （尽管不是每个函子都会产生一个Kleisli范畴。）

你可能会想知道我们刚刚定义的 fmap是否与编译器通过 deriving Functor为我们派生的 fmap相同。 有趣的是，确实如此。这是因为Haskell实现了多态函数。 它被称为参数多态性，并且它是所谓的免费定理的来源。 其中一个定理说，如果存在一个给定类型构造器的 fmap的实现，它保留了恒等性，那么它必须是唯一的。

8.6协变和逆变函子

现在我们已经回顾了writer函子，让我们回到reader函子。 它基于部分应用的函数箭头类型构造器：

```
(->) r
```

我们可以将其重写为类型同义词:

```
type Reader r a = r -> a
```

正如我们之前所见，对于该 Functor实例，其读取方式如下:

```
instance Functor (Reader r) where
    fmap f g = f . g
```

但是就像对偶类型构造函数或 Either类型构造函数一样，函数类型构造函数也需要两个类型参数。 对偶类型和Either在两个参数上都是函子 - 它们是双函子。 函数构造函数也是双函子吗?

让我们尝试使其在第一个参数上成为函子。 我们将从一个类型同义词开始 - 它与 Reader类似，但参数顺序相反:

```
type Op r a = a -> r
```

这次我们固定返回类型为 r，而变化的是参数类型 a。 让我们看看我们是否可以以某种方式匹配类型，以实现 fmap，其类型签名如下:

```
fmap :: (a -> b) -> (a -> r) -> (b -> r)
```

只有两个函数，分别接受一个和返回另一个和结果，根本无法构建一个接受另一个和返回结果的函数！如果我们能够以某种方式反转第一个函数，使其接受另一个并返回一个，那就不同了。 我们无法反转任意函数，但我们可以转到相反的范畴。

简短回顾：对于每个范畴 $\mathcal{C}$，都有一个对偶范畴 $\mathcal{C}^{op}$。 它是一个具有与 $\mathcal{C}$相同对象的范畴，但所有箭头都被颠倒了。

考虑一个从 $\mathcal{C}^{op}$到另一个范畴 $\mathcal{D}$的函子：

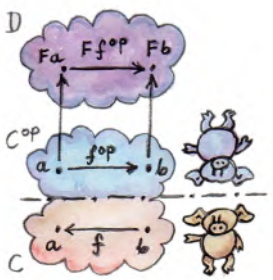
$$F :: \mathcal{C}^{op} \to \mathcal{D}$$

这样一个函子将范畴 $\mathcal{C}^{op}$中的态射 $f^{op} :: b \to a$映射到范畴 $\mathcal{D}$中的态射 $F\,f^{op} :: F\,b \to F\,a$。 但是态射 f^{op}秘密地对应于原始范畴 $\mathcal{C}$中的某个态射 $f :: a \to b$。注意反转。

现在， F是一个常规函子，但我们可以基于 F定义另一个映射，它不是一个函子 - 让我们称之为 F。 这是一个从 $\mathcal{C}$到 $\mathcal{D}$的映射。它以与 F相同的方式映射对象，但是当映射态射时，它将它们反转。 它接受一个态射 $f :: a \to b$在 $\mathcal{C}$中，首先将其映射到相反的态射 $f^{op} :: b \to a$，然后在其上使用函子 F，得到 $F\,f^{op} :: F\,b \to F\,a$。

考虑到 $F\,a$与 $F\,b$相同，
$F\,a$与 $F\,b$相同，整个过程可以描述为： $F :: (a \to b) \to (F\,b \to F\,a)$

这是一个“有点不同的函子”。将范畴的映射反转为态射的方向的映射被称为逆变函子。
注意，逆变函子只是从对偶范畴到普通范畴的普通函子。 顺便说一下，我们迄今为止学习的那种普通函子被称为协变函子。



这是在Haskell中定义逆变函子（实际上是逆变自函子）的类型类：

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

我们的类型构造器 `Op` 是它的一个实例：

```
instance Contravariant (Op r) where
    -- (b -> a) -> Op r a -> Op r b
    contramap f g = g . f
```

注意，函数 `f`在 `Op`的内容之前（也就是右边）插入了自身，函数 `g`。

对于Op，contramap的定义可以更加简洁，如果你注意到它只是参数翻转的函数组合运算符。 有一个特殊的函数用于翻转参数，称为
flip:

```
flip :: (a -> b -> c) -> (b -> a -> c)
 flip f y x = f x y
```

使用它，我们得到：

```
contramap = flip (.)
```

8.7 Profunctors

我们已经看到函数箭头运算符在第一个参数上是逆变的，在第二个参数上是协变的。 这样的东西有一个名字吗？
事实证明，如果目标范畴是 □ □ □ 这样的东西被称为profunctor。 因为逆变函子等价于对偶范畴中的协变函子，所以profunctor的定义如下：

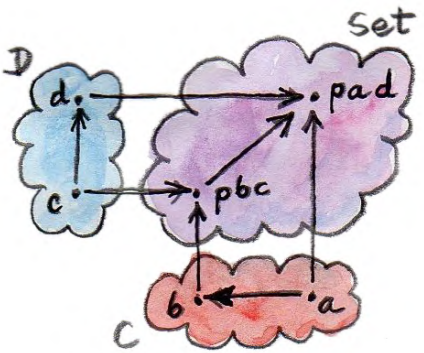
□ □ □ × □ → □ □ □

由于Haskell类型在第一近似中是集合，我们将Profunctor这个名字应用于一个具有两个参数的类型构造器 p，它在第一个参数上是逆变函子，在第二个参数上是函子。 这是从Data.Profunctor库中提取的适当的类型类：

```
自函子 p 的类
      dimap :: (a -> b) -> (c -> d) -> p b c -> p a d
    dimap f g = lmap f . rmap g
    lmap :: (a -> b) -> p b c -> p a c
    lmap f = dimap f id
    rmap :: (b -> c) -> p a b -> p a c
    rmap = dimap id
```

所有三个函数都带有默认实现。 就像使用Bifunctor一样，当声明一个Profunctor的实例时，你有一个选择

要么实现dimap并接受lmap和
rmap的默认值，要么同时实现lmap和rmap并接受dimap的默认值。
dimap。



dimap

现在我们可以断言函数箭头运算符是一个的实例
Profunctor:

```
instance Profunctor (->) where
    dimap ab cd bc = cd . bc . ab
    lmap = flip (.)
    rmap = (.)
```

Profunctors在Haskell镜头库中有应用。 当我们讨论端和余端时
，我们将再次见到它们。

8.8Hom-函子

上述示例是更一般的陈述的反映，即将一对对象 □和 □映射到它的

它们之间的态射集合，即同态集 $\hom(a, b)$，是一个函子。 它是从乘积范畴 $\mathcal{C}^{op} \times \mathcal{C}$到集合范畴 $\mathcal{S}$ 的函子。

让我们定义它对态射的作用。 在 $\mathcal{C}^{op} \times \mathcal{C}$中，一个态射是来自 (a, b)的一对态射：

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

这对的提升必须是从集合$\hom(a, b)$到集合 $\hom(a', b')$的态射（函数）。 只需选择任何元素 *h* of $\hom(a, b)$（它是从 a到 b的态射）并将其分配给：

$$g \circ h \circ f$$

它是 $\hom(a', b')$的一个元素。

正如你所看到的，同态函子是一个协函子的特例。

8.9 挑战

1. 证明数据类型：

   ```
   数据对 a b = 对 a b
   ```

   是一个双函子。 为了额外的学分，实现 Bifunctor的所有三个方法，并使用等式推理来证明这些定义在可以应用时与默认实现兼容。

2. 展示标准定义的 Maybe和这个展开的定义之间的同构：

   ```
   类型 Maybe' a = Either (Const () a) (Identity a)
   ```

提示：定义两个映射来连接这两个实现。
为了额外的学分，使用等式推理来证明它们互为逆元。

3. 让我们尝试另一种数据结构。 我称之为 PreList，因为它是 List 的前身。 它用类型参数 b替代了递归。

```
数据 PreList a b = Nil | Cons a b
```

通过递归将 PreList 应用于自身，可以恢复我们之前对 List 的定义（我们将在讨论固定点时看到如何实现）。

证明 PreList 是 Bifunctor 的一个实例。

4. 证明以下数据类型在 a 和 b 中定义了 bifunctors：

```
数据 K2 c a b = K2 c

数据 Fst a b = Fst a

数据 Snd a b = Snd b
```

为了额外的学分，可以将解决方案与 Conor McBride 的论文《左边是小丑，右边是小丑》进行对比。5. 在除了 Haskell 之外的语言中定义一个 bifunctor。在该语言中为通用对实现 bimap。

6. 在两个模板参数 Key 和 T 中，std::map 应该被视为 bifunctor 还是 profunctor？ 你会如何重新设计这个数据类型以实现这一点？

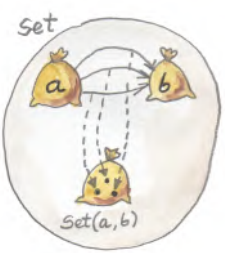
1http://strictlypositive.org/CJ.pdf

9

函数类型

到目前为止，我一直在概述函数类型的含义。 函数类型与其他类型不同。

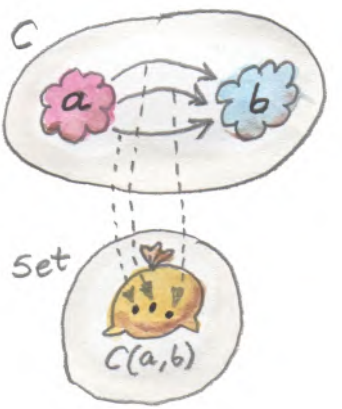
以整数为例：它只是一个整数的集合。布尔是一个两个元素的集合。但是一个函数类型以整 → □不仅仅是这样：它是对象□和□之间的态射的集合。 在任何范畴中，两个对象之间的态射集被称为同态集。 恰好在集合范畴恰好恰好恰好中，每个同态集本身也是该范畴中的一个对象，因为它毕竟是一个集合。



在集合中的同态集只是一个集合

其他范畴不具备相同的特性在其他范畴中，同态集是范畴之外的。 它们甚至被称为外部同态集。

范畴的自指性质使得函数类型特殊。但在某些范畴中，有一种方法可以保留源代码和数学符号。



范畴C中的同态集是一个外部集合

构造表示同态集的对象。 这些对象被称为内部同态集。

9.1 通用构造

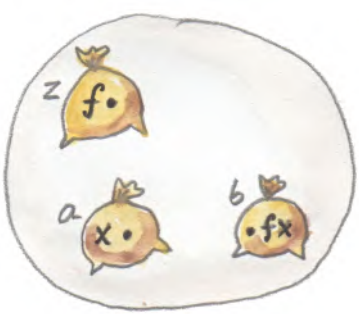
让我们暂时忘记函数类型是集合，并尝试从头开始构造一个函数类型，或者更一般地说，一个内部同态集。 像往常一样，我们将从范畴中获取线索，但小心避免使用集合的任何属性，以便构造将自动适用于其他范畴。

函数类型可以被视为复合类型，因为它与参数类型和结果类型之间有关系。 我们已经看到了复合类型的构造 - 那些涉及对象之间关系的构造。 我们使用普遍构造来定义乘积类型和余积类型。 我们可以使用相同的技巧来定义-

找到一个函数类型。 我们需要一个涉及三个对象的模式：我们正在构建的函数类型，参数类型和结果类型。

连接这三种类型的明显模式被称为函数应用或求值。 给定一个函数类型的候选项，让我们称之为给定（注意，如果我们不在给定给定给定类别中，这只是一个像其他对象一样的对象），和参数类型给定（一个对象），应用将这对映射到结果类型给定（一个对象）。 我们有三个对象，其中两个是固定的（表示参数类型和结果类型的对象）。

我们还有一个映射的应用。 我们如何将这个映射纳入我们的模式中？ 如果我们被允许查看对象的内部，我们可以将函数如果（如果的元素）与参数如果（如果的元素）配对，并将其映射到如果 □（□对□的应用，它是□的元素）。

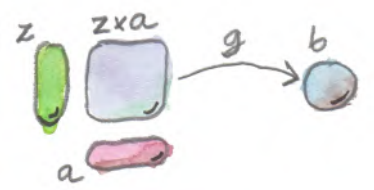


在集合中，我们可以从一组函数 □中选择一个函数 □，并且我们可以从集合（类型） □中选择一个参数 □。 我们得到了集合（类型） □中的一个元素□ □。

但是，我们不仅可以处理单个对 (□ , □)，我们也可以讨论函数类型 □和参数的整个乘积。

类型 □。 乘积□ ×□是一个对象，我们可以选择作为我们的应用映射的箭头 □从该对象到 □。 在集合中， □将每个对 (□ , □)映射到□ □。

所以这就是模式：两个对象 □和 □的乘积通过态射 □与另一个对象 □相连。



一种由对象和态射组成的模式，是通用构造的起点

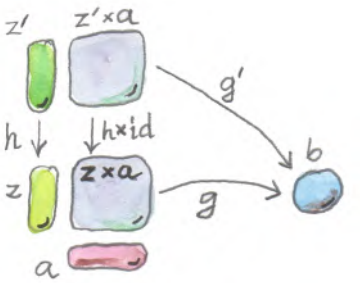
这个模式是否足够具体，可以通过通用构造来确定函数类型？ 并不是在每个范畴中都是如此。 但在我们感兴趣的范畴中是可以的。还有一个问题：是否可以在不先定义乘积的情况下定义函数对象？ 有些范畴中没有乘积，或者并非所有对象对都有乘积。

答案是否定的：如果没有乘积类型，就没有函数类型。
我们稍后会再讨论指数。

让我们回顾一下通用构造。 我们从一组对象和态射开始。 这是我们不太明确的查询，通常会得到很多结果。 特别地，在 □ □ □ 中，几乎所有东西都相互连接。 我们可以取任意对象 □，与，与形成乘积，然后就会有一个从它到 □的函数（除非 □是空集）。

这就是我们应用秘密武器的时候：排名。 通常情况下，这是通过要求候选对象之间存在唯一的映射来完成的 - 这种映射以某种方式分解我们的构造。在我们的

情况下，我们将规定 □连同从□ × □到 □的态射 □，比某个具有自己应用 □′的其他 □′更好，当且仅当存在唯一的映射 *h*从 □′到 □，使得□′的应用通过 □的应用分解。 （提示：在看图片的同时阅读这个句子。）



为函数对象的候选者建立排名

现在是棘手的部分，也是我将这个特定的普遍构造推迟到现在的主要原因。给定态射*h* : : □′→ □，我们希望闭合同时具有 □′和 □与 □交叉的图表。 实际上，我们需要的是，给定从 □′到 □的映射 *h*，从 □′× □到□ × □的映射。 现在，在讨论了积的函子性之后，我们知道如何做到这一点。因为积本身是一个函子（更准确地说是一个自函子双函子），可以提升成对的态射。 换句话说，我们不仅可以定义对象的积，还可以定义态射的积。

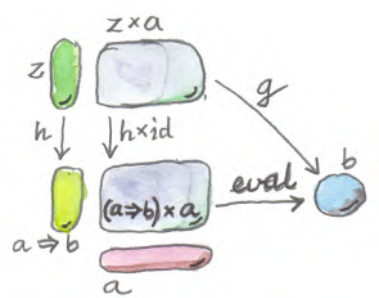
由于我们不会触及乘积的第二个分量 □′× □，我们将提升态射对 (h, id)，其中 id是一个恒等态射在 □上。

所以，这是我们如何从另一个应用程序 □中分解出一个应用程序 □′的方法：

$$□' = □ \circ (h \times \mathrm{id})$$

关键在于乘积对态射的作用。

通用构造的第三部分是选择普遍最佳的对象。 让我们称这个对象为□ ⇒ □（将其视为一个对象的符号名称，不要与Haskell类型类约束混淆 - 我将在后面讨论不同的命名方式）。 这个对象带有它自己的应用程序 - 从 (□ ⇒ □) × □到 □的态射，我们将其称为 □ □ □ □。 如果任何其他函数对象的候选者可以通过唯一地映射到它，并且其应用态射 □通过 □ □ □因子分解，则对象□ ⇒ □是最佳的。 根据我们的排名，这个对象比任何其他对象都要好。



通用函数对象的定义。 这是与上面相同的图表，但现在对象□ ⇒ □是普遍的。

正式地说：

从 □到 □的函数对象是一个对象□ ⇒ □以及这个态射

□ □ □ :□ ((□ ⇒ □) × □) → □

使得对于任何其他对象 □和一个态射

□ : : □ × □ → □

存在唯一的态射

h : : □ → (□ ⇒ □)

使得 □ 通过 □ □ □ □

□ = □ □ □ ∘□ (h × id)

当然，并不能保证在给定的范畴中对于任何一对对象 □和 □都存在这样的对象□ ⇒ □。 但在但在但在但在中总是存在。 此外，在此外此外此外中，这个对象同构于同态集合□ □ (□□, □)。

这就是为什么在Haskell中，我们将函数类型 a -> b解释为范畴函数对象□ ⇒ □。

9.2 柯里化

让我们再次审视所有函数对象的候选者。
然而，这一次，让我们将态射 □看作是两个变量 □和 □的函数。

□ : : □ × □ → □

作为一个从积对象到态射的态射，它与一个两个变量的函数非常接近。 特别地，在 □ □ 中， □是从值对（一个来自集合 □，一个来自集合 □）到函数的函数。

另一方面，通用性质告诉我们，对于每个这样的□，都存在一个将 □映射到函数对象□ ⇒ □的唯一态射 *h*。

$$h :: □ \to (□ \Rightarrow □)$$

在 □ □ 中，这意味着 *h*是一个接受类型为 □的变量，并返回从 □到 □的函数。 这使得 *h*成为一个高阶函数。 因此，通用构造建立了两个变量函数和返回函数的一对一对应关系。 这种对应关系被称为柯里化， *h*被称为 □的柯里化版本。

这种对应关系是一对一的，因为对于任何给定的 □，都存在一个唯一的 *h*，并且对于任何给定的 *h*，您始终可以使用以下公式重新创建两个参数函数 □：

$$□ = □ \ □ \ □ \circ □ (h \times \mathrm{id})$$

函数 □可以称为非柯里化版本的 *h*。

柯里化基本上内置在Haskell的语法中。一个返回函数的函数：

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

通常被认为是两个变量的函数。 这就是我们读取未加括号的签名的方式：

```
a -> b -> c
```

这种解释在我们定义多参数函数的方式中是显而易见的。例如：

```
catstr :: String -> String -> String
catstr s s' = s ++ s'
```

同样的函数可以被写成返回一个函数的一参数函数 - 一个lambda表达式：

```
catstr' s = \s' -> s ++ s'
```

这两个定义是等价的，任何一个都可以部分应用于一个参数，产生一个一参数函数，如下所示：

```
greet :: String -> String
greet = catstr "Hello "
```

严格来说，一个两个变量的函数是一个接受一对（一个乘积类型）的函数：

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

在两种表示之间进行转换是微不足道的，完成这个转换的两个（高阶）函数被称为curry和uncurry。
uncurry:

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

和

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

注意 curry是函数对象的通用构造的分解器。 如果以这种形式重写，这一点尤为明显：

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

(作为提醒：分解器从候选者中产生分解函数。)

在非函数式语言（如C++）中，柯里化是可能的，但并不简单。你可以将C++中的多参数函数视为接受元组的Haskell函数（尽管为了更加混淆，你可以在C++中定义接受显式std::tuple的函数，以及可变参数函数和接受初始化列表的函数）。你可以使用模板std::bind部分应用C++函数。

例如，给定两个字符串的函数：

```
std::string catstr(std::string s1, std::string s2) {
    return s1 + s2;
}
```

你可以定义一个字符串的函数：

```
using namespace std::placeholders;

auto greet = std::bind(catstr, "Hello ", _1);
std::cout << greet("Haskell Curry");
```

Scala比C++或Java更加函数式，介于两者之间。 如果你预计要定义的函数将会被部分应用，你可以使用多个参数列表来定义它：

```
def catstr(s1: String)(s2: String) = s1 + s2
```

当然，这需要库编写者有一定的先见之明。

9.3 指数

在数学文献中，函数对象或两个对象 □和 □之间的内部同态对象通常被称为指数，并用 □□表示。 注意参数类型在指数中。 这种表示法一开始可能看起来很奇怪，但如果你考虑函数和积之间的关系，它就变得很合理。 我们已经看到在内部同态对象的通用构造中必须使用积，但这种联系更加深入。

当你考虑在有限类型之间的函数时，这一点最容易理解- 类型具有有限数量的值，例如 Bool、 Char，甚至 Int或 Double。 这样的函数，至少在原则上，可以完全进行记忆化或转化为数据结构进行查找。这就是函数之间的等价性的本质，函数是态射，函数类型是对象。

例如，从 Bool到（纯）函数完全由一对值指定：一个对应于 False，一个对应于True。 从 Bool到，比如说， Int的所有可能函数的集合是所有 Int对的集合。这与积 Int × Int相同，或者稍微创造性地使用符号， Int^2。

再举一个例子，让我们看看C++类型 char，它包含256个值（Haskell Char更大，因为Haskell使用Unicode）。在C++标准库的一部分中，通常使用查找来实现几个函数。 像 isupper或 isspace这样的函数使用表来实现，这相当于256个布尔值的元组。 元组是一个积类型，所以我们处理的是256个布尔值的积： bool × bool × bool × ... × bool。 我们从算术中知道，迭代的积定义了一个幂。如果你“乘法”

如果将bool单独重复256次（或者 char），你得到bool的 char次方, 或 boolchar.

在定义为256元组的 bool类型中有多少个值?
确切地说是 2^{256}个。 这也是从 char到bool的不同函数的数量，每个函数对应一个唯一的256元组。 你可以类似地计算出从 bool到 char的函数数量是 256^2，以此类推。函数类型的指数表示在这些情况下非常合理。

我们可能不想完全记忆化从 int或 double的函数。 但是函数和数据类型之间的等价性，虽然不总是实用，但确实存在。 还有无限类型，例如列表、字符串或树。 对于这些类型的函数的急切记忆化将需要无限的存储空间。 但是Haskell是一种惰性语言，所以惰性评估（无限）数据结构和函数之间的边界是模糊的。这种函数与数据的二元性解释了Haskell的函数类型与范畴论的指数对象的等同性，这更符合我们对数据的理解。

9.4 笛卡尔闭范畴

虽然我将继续使用集合范畴作为类型和函数的模型，但值得一提的是，还有一类更大的范畴可以用于此目的。 这些范畴被称为笛卡尔闭范畴，而这些这些这些只是这类范畴的一个例子。

笛卡尔闭范畴必须包含：

1. 终对象，
2. 任意一对对象的积，以及
3. 任意一对对象的指数。

如果你将指数视为迭代的积（可能是无限次），那么你可以将笛卡尔闭范畴视为支持任意元数积的范畴。 特别地，终对象可以被视为零个对象的积，或者是一个对象的零次幂。

从计算机科学的角度来看，笛卡尔闭范畴的有趣之处在于它们为简单类型λ演算提供了模型，而简单类型λ演算是所有有类型编程语言的基础。

终端对象和乘积有它们的对偶：初始对象和余积。 一个笛卡尔闭范畴也支持这两个，并且在其中乘积可以分配到余积上。

$$A \times (B + C) = A \times B + A \times C$$

$$(A + B) \times C = A \times C + A \times C$$

被称为双笛卡尔闭范畴。 我们将在下一节中看到，双笛卡尔闭范畴，其中我们我们我们是一个主要的例子，具有一些有趣的性质。

9.5 指数和代数数据类型

将函数类型解释为指数非常适合代数数据类型的方案。 事实证明，高中代数中关于数字零和一、求和、乘积和指数的所有基本恒等式在任何双笛卡尔闭范畴理论中几乎保持不变，分别适用于初始和终结对象、余积、乘积和指数。 我们还没有工具来证明它们（如伴随或Yoneda引理），但我仍然会在这里列出它们作为有价值的直觉来源。

9.5.1 零次幂

$$□^0 = 1$$

在范畴论的解释中，我们用初始对象替换0，用终结对象替换1，并用同构代替等式。 指数是内部同态对象。 这个特定的指数表示从初始对象到任意对象 □的态射集合。 根据初始对象的定义，存在唯一的这样的态射，因此 □(0, □)是一个单元素集合。单元素集合是 □ □ 中的终结对象，因此这个恒等式在 □ □ 中显然成立。 我们所说的是它在任何双笛卡尔闭范畴中都成立。

在Haskell中，我们用 Void替换0；用单位类型 ()替换1；用函数类型替换指数。 这个论断是，从 Void到任意类型 a的函数集合等价于单位类型——也就是一个单元素集合。 换句话说，只有一个函数 Void -> a。 我们之前见过这个函数：它被称为 absurd。

这有点棘手，有两个原因。 其中一个原因是在Haskell中，我们没有真正的无人居住的类型 - 每个类型都包含“永不结束的计算”的“结果”，或者底部。 第二个原因是所有的 absurd的实现都是等价的，因为无论它们做什么，没有人能够执行它们。 没有任何值可以传递给 absurd。 （如果你设法给它一个永不结束的计算，它将永远不会返回！）

9.5.2 一次幂

$$1^□ = 1$$

这个等式在 □ □ 中的解释重新阐述了终结对象的定义：从任何对象到终结对象都存在唯一的态射

终结对象。 一般来说，从 □到终结对象的内部同态对象与终结对象本身是同构的。

在Haskell中，从任何类型□到单元的函数只有一个。 我们之前见过这个函数 - 它被称为 unit。 你也可以将它看作是 const函数部分应用于 ()。

9.5.3 第一次幂

$$\square^1 = \square$$

这是一个重新陈述的观察，从终结对象到对象 a的态射可以用来选择对象的“元素”。 这样的态射集合同构于对象本身。 在 □ □ 中，以及在Haskell中，同构是集合元素和选择这些元素的函数之间的关系，即 () -> a。

9.5.4 和的指数

$$\square^{\square + \square} = \square^{\square} \times \square^{\square}$$

范畴论上，这意味着两个对象的余积的指数同构于两个指数的积。 在Haskell中，这个代数恒等式有一个非常实际的解释。 它告诉我们，从两种类型的和到函数等价于从各个类型到函数的一对函数。 这只是我们在对和上定义函数时使用的情况分析。 我们通常将一个带有 case语句的函数定义拆分成两个（或更多）函数，分别处理每个类型构造器。 例如，从和类型 (Either Int Double)中取一个函数：

```
f :: Either Int Double -> String
```

它可以被定义为从Int和Double分别到String的一对函数
Double:

f (Left n) = 如果 n < 0 则返回 "负整数" 否则返回 "正整数"
f (Right x) = 如果 x < 0.0 则返回 "负浮点数" 否则返回 "正
 浮点数"

在这里，n是一个整数，x是一个浮点数。

9.5.5 指数的指数

$$(C^B)^A = C^{B \times A}$$

这只是一种纯粹基于指数对象的柯里化表达方式。 返回一个函数的函数等同于一个来自乘积（一个二元函数）的函数。

9.5.6 乘积上的指数

$$(A \times B)^C = A^C \times B^C$$

在Haskell中：返回一对的函数等同于一对函数，每个函数产生一对的一个元素。

令人惊讶的是，那些简单的高中代数恒等式如何被提升到范畴论，并在函数式编程中有实际应用。

9.6 Curry-Howard同构

我已经提到了逻辑和代数数据类型之间的对应关系。 Void类型和单位类型()对应于false和true。 积类型和和类型对应于逻辑连词

∧(与) 和 ∨(或)。 在这个方案中，我们刚刚定义的函数类型对应于逻辑蕴含 ⇒。 换句话说，类型 a -> b可以理解为“如果a，则b”。

根据柯里-霍华德同构，每种类型都可以被解释为命题 - 一个可能为真或假的陈述或判断。 如果类型有实例，则该命题被认为为真；如果没有实例，则被认为为假。 特别地，如果与之对应的函数类型有实例，则逻辑蕴含为真，这意味着存在一个具有该类型的函数。 函数的实现因此是一个定理的证明。 编写程序等同于证明定理。 让我们看几个例子。

让我们来看一下我们在函数对象的定义中引入的函数 eval。它的签名是：

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

它接受一个由函数和其参数组成的对，并产生一个适当类型的结果。 这是Haskell实现的态射：

$$□ □ □ :: □ \ (□ \Rightarrow □) \times □ \to □$$

它定义了函数类型□ ⇒ □（或指数对象 $□^{□}$）。
让我们使用Curry-Howard同构将这个签名翻译成逻辑谓词：

$$((□ \Rightarrow □) \wedge □) \Rightarrow □$$

你可以这样理解这个语句：如果 □是从□推导出来的，并且 □是真的，那么 □必须是真的。 这在直觉上是完全合理的，并且自古以来被称为modus ponens。 我们可以通过实现这个函数来证明这个定理：

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

如果你给我一个由函数 f接受一个 a并返回一个 b的对，以及一个具体值 xof类型 a，我可以通过简单地将函数 f应用于 x来产生一个类型为b的具体值。 通过实现这个函数，我刚刚证明了类型 ((a -> b), a) -> b是有实例的。
因此假言推理在我们的逻辑中是正确的。

那么一个明显错误的谓词怎么样？ 例如：如果 □或 □是真的，那么 □必须是真的。

$$□ \vee □ \Rightarrow □$$

这显然是错误的，因为你可以选择一个假的 □和一个真的□，这就是一个反例。

将这个谓词映射到一个使用Curry-Howard同构的函数签名中，我们得到：

```
Either a b -> a
```

尽管你努力，你无法实现这个函数——如果你用 Right值调用它，你无法产生一个类型为 a的值。 （记住，我们谈论的是纯函数。）

最后，我们来看一下 absurd函数的含义：

```
absurd :: Void -> a
```

考虑到 Void翻译成假，我们得到：

$$□ □ □ □ \Rightarrow □$$

任何事物都可以从虚假中推导出来（ex falso quodlibet）。 这是Haskell中这个语句（函数）的一个可能证明（实现）：

```
荒谬 (Void a) = 荒谬 a
```

其中 `Void` 被定义为：

```
新类型 Void = Void Void
```

一如既往，类型 `Void`很棘手。 这个定义使得构造一个值变得不可能，因为为了构造一个值，你需要提供一个值。因此，函数荒谬永远不会被调用。

这些都是有趣的例子，但是Curry-Howard同构有实际的应用吗？ 在日常编程中可能不会有。
但是有一些编程语言，比如Agda或Coq，利用Curry-Howard同构来证明定理。

计算机不仅帮助数学家完成他们的工作，而且正在彻底改变数学的基础。 该领域最新的热门研究课题被称为同伦类型论，它是类型论的一个发展。 它充满了布尔值、整数、积类型和余积类型、函数类型等等。而且，为了消除任何疑虑，这个理论正在Coq和Agda中进行形式化。计算机正在以多种方式改变世界。

9.7 参考文献

1. Ralph Hinze, Daniel W. H. James, 以同构的方式推理！[1]. 这篇论文包含了我在本章中提到的所有高中代数恒等式的范畴论证明。

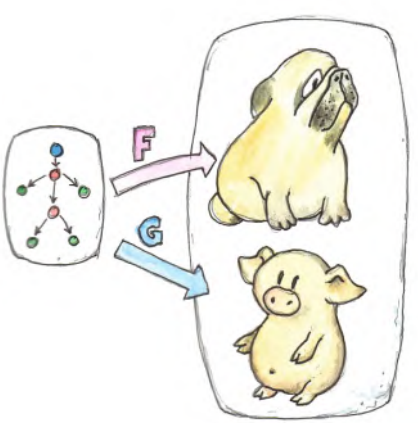
1http://www.cs.ox.ac.uk/ralf.hinze/publications/WGP10.pdf

10

自然变换

我们将函子视为在保持其结构的范畴之间的映射。

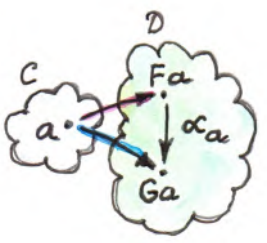
一个函子将一个范畴“嵌入”到另一个范畴中。 它可以将多个事物合并为一个，但它永远不会断开连接。 一种思考方式是，通过函子，我们在另一个范畴中建模了一个范畴。 源范畴作为目标范畴的一部分的结构的模型，蓝图。



可能有很多种方式将一个范畴嵌入到另一个范畴中。 有时它们是等价的，

有时非常不同。 一个可能将整个源范畴折叠成一个对象，另一个可能将每个对象映射到不同的对象，每个态射映射到不同的态射。 相同的蓝图可以以许多不同的方式实现。 自然变换帮助我们比较这些实现。 它们是函子的映射 - 保持其函子性质的特殊映射。

考虑两个函子 □和 □在范畴 □和 □之间。 如果你只关注范畴 □中的一个对象 □，它会被映射到两个对象：□ □和 □ □。
因此，函子的映射应该将□ □映射到 □ □。



注意到□ □和 □ □是同一个范畴 □中的对象。 在同一个范畴中，对象之间的映射不应该违背范畴的规则。 我们不希望在对象之间建立人为的联系。 因此，使用已有的连接，即态射，是很自然的选择。
自然变换是一组态射的选择：对于每个对象□，它选择从□ □到 □ □的一个态射。 如果我们将自然变换称为 □，那么这个态射被称为 □在对象 □上的分量，或者 □□。

□ □: : □ □ → □ □

请记住，在 □是 □中的一个对象，而 □□是 □中的一个态射。

如果对于某个 □，在 □中□ □和 □ □之间没有态射，则 □和 □之间也没有自然变换。

当然，这只是故事的一半，因为函子不仅映射对象，还映射态射。 那么自然变换对这些映射做了什么呢？ 事实证明，态射的映射是固定的 - 在 □和□之间的任何自然变换中，□ □必须被转换为□ □。 而且，这两个函子对态射的映射极大地限制了我们在定义与之兼容的自然变换时的选择。考虑在 □中两个对象 □和 □之间的态射 □。 它被映射为两个态射，在 □中为□ □和 □ □：

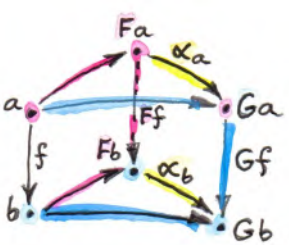
□ □ : : □ □ → □ □

□ □: : □ □→ □ □

自然变换 α提供了两个额外的态射，
完成了 D中的图表：

□ □: : □ □ → □ □

□ □: : □ □ → □ □

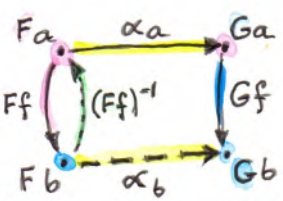


现在我们有两种从□ □到 □ □的方法。 为了确保它们相等，我们必须施加自然性条件，对于任何 □都成立。

□ □∘ □ □= □ □∘ □ □

自然性条件是一个相当严格的要求。 例如，
如果态射□ □是可逆的，自然性就确定了 □□与
□□之间的关系。 它将 □□沿着 □传输：

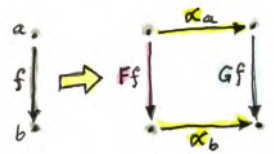
□ □= (□ □) ∘ □ □ ∘ (□ □)$^{-1}$



如果两个对象之间存在多个可逆态射，所有这些传输都必须一致。 然而，一般来说，态射并不是可逆的；但是你可以看到，两个函子之间存在自然变换的存在远非保证。 因此，与自然变换相关的函子的稀缺性或丰富性可能会告诉你很多关于它们操作的范畴结构。 当我们讨论极限和Yoneda引理时，我们将看到一些例子。

从组件的角度来看，一个自然变换可以说是将对象映射到态射。 由于自然性条件的存在，

也可以说它将态射映射到可交换的方块 - 对于范畴对于中的每个态射，在范畴对于中都有一个可交换的自然性方块。



自然变换的这个性质在许多范畴构造中非常有用，这些构造通常包括可交换图。
通过巧妙选择函子，许多这些可交换条件可以转化为自然性条件。 当我们讨论极限、余极限和伴随时，我们将看到一些例子。

最后，自然变换可以用来定义函子的同构。 说两个函子是自然同构几乎就是说它们是相同的。自然同构被定义为其组件都是同构（可逆态射）的自然变换。

10.1 多态函数

我们讨论了函子（或者更具体地说，自函-子）在编程中的作用。 它们对应于将类型映射到类型的类型构造器。 它们还将函数映射到函数，并且这种映射是由一个高阶函数 fmap（或者在C++中是 transform、 then、and等等）实现的。

为了构造一个自然变换，我们从一个对象开始，这里是一个类型□。 一个函子 F将其映射到类型□ □。另一个函子 G

将其映射到 □ □。 自然变换 alpha在□处的分量是从□ □到 □ □的函数。在伪Haskell中：

```
alphaa :: F a -> G a
```

自然变换是一个多态函数，适用于所有类型□：

```
alpha :: forall a . F a -> G a
```

在Haskell中， forall a是可选的（实际上需要打开语言扩展ExplicitForAll）。 通常，你会这样写：

```
alpha :: F a -> G a
```

请记住，这实际上是一个由参数化的函数族 a。这是Haskell语法简洁性的另一个例子。 在C++中，类似的构造会稍微冗长一些：

```
template<class A> G<A> alpha(F<A>);
```

Haskell的多态函数和C++的泛型函数之间有一个更深刻的区别，这体现在这些函数的实现和类型检查方式上。 在Haskell中，多态函数必须统一地定义在所有类型上。 一个公式必须适用于所有类型。 这被称为参数多态。

另一方面，C++默认支持特定多态，这意味着模板不必对所有类型都定义良好。 模板是否适用于给定类型是在实例化时决定的，具体类型被替换为类型参-数。 类型检查被延迟，这往往导致难以理解的错误消息。

在C++中，还有一种函数重载和模板特化的机制，允许为不同类型的相同函数定义不同的版本。 在Haskell中，这种功能由类型类和类型族提供。

Haskell的参数多态性有一个意想不到的结果：
任何类型为多态函数的函数：

```
alpha :: F a -> G a
```

其中 F和 G是函子，自动满足自然性条件。 这是范畴论符号表示法中的表达式（□是一个函数□ : : □ → □）：

□ □◦ □ □= □ □◦ □ □

在Haskell中，函子 G对于态射 f的作用是使用 fmap实现的。 我首先用伪Haskell写出来，带有明确的类型注解：

```
fmapG f . alphaa = alphab . fmapF f
```

由于类型推断，这些注解是不必要的，以下等式成立：

```
fmap f . alpha = alpha . fmap f
```

这仍然不是真正的Haskell - 函数相等性在代码中无法表达- 但它是程序员在等式推理中可以使用的一个恒等式；或者由编译器用来实现优化。

Haskell中自然性条件自动成立的原因与“免费定理”有关。 在Haskell中，用于定义自然变换的参数多态性对实现施加了非常强的限制 - 一个公式适用于所有类型。 这些限制转化为关于这些函数的等式定理。在

变换函子的函数的情况下，免费定理是自然性条件。[1]

在Haskell中，关于函子的一种思考方式是将它们视为广义容器。 我们可以延续这个类比，将自然变换视为将一个容器的内容重新打包到另一个容器中的配方。 我们不会触及物品本身：我们不修改它们，也不创建新的物品。 我们只是将它们（其中的一些）复制到一个新的容器中，有时多次复制。

自然性条件成为一个陈述，即我们首先通过应用 fmap修改项目，然后重新打包；或者首先重新打包，然后在新容器中修改项目，使用其自己的 fmap实现。 这两个操作，重新打包和 fmapping，是正交的。"一个移动鸡蛋，另一个煮鸡蛋。"

让我们在Haskell中看几个自然变换的例子。第一个是列表函子和 Maybe函子之间的变换。 它返回列表的头部，但仅当列表非空时：

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

它是一个多态函数，适用于任何类型 a，没有限制，因此它是参数多态的一个例子。 它适用于任何类型 a，没有限制，因此它是参数多态的一个例子。 因此，它是两个函子之间的自然变换。 但为了说服自己，让我们验证自然性条件。

```
fmap f . safeHead = safeHead . fmap f
```

1你可以在我的博客Parametricity: Money for Nothing and Theorems for Free中了解更多关于自由定理的内容。

我们有两种情况要考虑；一个空列表：

```
fmap f (safeHead []) = fmap f Nothing = Nothing

safeHead (fmap f []) = safeHead [] = Nothing
```

和一个非空列表：

```
fmap f (safeHead (x:xs)) = fmap f (Just x) = Just (f x)

safeHead (fmap f (x:xs)) = safeHead (f x : fmap f xs) = Just (f
 x)
```

我在列表的实现中使用了 `fmap` ：

```
fmap f [] = []
fmap f (x:xs) = f x : fmap f xs
```

以及对于 `Maybe`的实现：

```
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

有趣的是，当其中一个函子是平凡的 `Const`函子时。从 `Const`函子到另一个函子的自然变换看起来就像是一个在返回类型或参数类型中具有多态性的函数。

例如，长度可以被看作是从列表函子到常数整数函子的自然变换：

```
length :: [a] -> Const Int a
length [] = Const 0
length (x:xs) = Const (1 + unConst (length xs))
```

在这里，unConst 用于去除 Const 构造器：

```
unConst :: Const c a -> c
unConst (Const x) = x
```

当然，在实践中，length 的定义如下：

```
length :: [a] -> Int
```

这实际上隐藏了它是一个自然变换的事实。

从 Const 函子中找到一个参数化多态函数有点困难，因为它需要从无中创建一个值。我们能做的最好的是：

```
scam :: Const Int a -> Maybe a
scam (Const x) = Nothing
```

另一个我们已经见过的常见函子，并且在 Yoneda 引理中将发挥重要作用的是 Reader 函子。 我将重新将其定义为一个新类型：

```
newtype Reader e a = Reader (e -> a)
```

它由两种类型参数化，但只在第二个类型上（协变地）具有函子性质：

```
instance Functor (Reader e) where
    fmap f (Reader g) = Reader (\x -> f (g x))
```

对于每个类型 `e`，你可以定义一个从 `Reader e`到任何其他函子 `f`的自然变换家族。 我们稍后会看到，这个家族的成员总是与 `f e`的元素一一对应（Yoneda引理）。

例如，考虑有一个元素 ()的有点平凡的单位类型 ()。 函子 Reader ()接受任何类型 a并将其映射为一个函数类型 () -> a。 这些只是从集合 a中选择单个元素的所有函数。 这些函数的数量与 a中的元素数量相同。 现在让我们考虑从这个函子到Maybe函子的自然变换：

```
alpha :: Reader () a -> Maybe a
```

这些只有两个，愚蠢的和显而易见的：

```
愚蠢的 (Reader _) = Nothing
```

和

```
显而易见的 (Reader g) = Just (g ())
```

（你可以对 g做的唯一事情就是将其应用于单位值 ()。）而且，

正如Yoneda引理所预测的那样，它们对应于 Maybe ()类型的两个元素，即 Nothing和 Just ()。
我们稍后会回到Yoneda引理 - 这只是一个小小的引子。

10.2 超越自然性

在两个函子之间的参数多态函数（包括 Const函子的边缘情况）始终是一个自然变换。 由于所有标准代数数据类型都是函子，因此在这些类型之间的多态函数是自然变换。

我们还可以使用函数类型，并且它们对于返回类型是函子的情况也是函子的。 我们可以使用它们来构建函子（例如 Reader

函子)并定义高阶函数的自然变换。

然而，函数类型在参数类型上不是协变的。
它们是逆变的。 当然，逆变函子等价于从相反范畴到协变函子。
在范畴论意义上，两个逆变函子之间的多态函数仍然是自然变换
，只是它们作用于从相反范畴到Haskell类型的函子。

你可能还记得我们之前看过的一个逆变函子的例子：

```
newtype Op r a = Op (a -> r)
```

这个函子在 a上是逆变的：

```
instance Contravariant (Op r) where
    contramap f (Op g) = Op (g . f)
```

我们可以编写一个多态函数，从 Op Bool到 Op String：

```
predToStr (Op f) = Op (\x -> if f x then "T" else "F")
```

但由于这两个函子不是协变的，所以这不是一个自然变换在 Hask 中。 然而，因为它们都是逆变的，它们满足“相反”的自然性条件：

```
contramap f . predToStr = predToStr . contramap f
```

注意函数 f必须与 fmap相反的方向，因为 contramap的签名如下：

```
contramap :: (b -> a) -> (Op Bool a -> Op Bool b)
```

是否存在一些类型构造器既不是协变的也不是逆变的函子？这里有一个例子：

```
a -> a
```

这不是一个函子，因为相同的类型 a在负（逆变）和正（协变）位置上都被使用。 你无法为这种类型实现fmap或 contramap。 因此，具有以下签名的函数：

```
(a -> a) -> f a
```

其中 f是任意的函子，不能是自然变换。 有趣的是，还有一种自然变换的推广，称为双自然变换，用于处理这种情况。 当我们讨论端时，我们会涉及到它们。

10.3 函子范畴

既然我们有了函子之间的映射——自然变换——那么自然而然地会问，函子是否构成一个范畴。 事实上，它们确实构成一个范畴！ 对于每一对范畴 □和 □，都有一个函子的范畴。 这个范畴中的对象是从 □到 □的函子，而态射是这些函子之间的自然变换。

我们需要定义两个自然变换的复合，但这非常容易。 自然变换的分量是态射，而我们知道如何组合态射。

实际上，让我们从函子 □到 □取一个自然变换 □。 它在对象 □的组成部分是一些态射:

□ □: : □ □ → □ □

我们想要将 □与 □组合起来，它是从函子 □到 □的自然变换。 在 □处的 □的分量是一个态射:

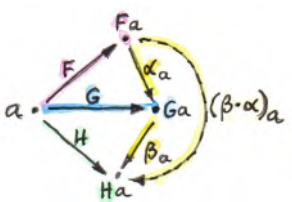
□ □: : □ □→ □ □

这些态射是可组合的，它们的组合是另一个态射:

□ □◦ □ □: : □ □ → □ □

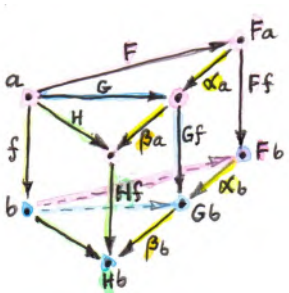
我们将使用这个态射作为自然变换□ · □的分量——两个自然变换 □和□的组合:

(□ · □)□ = □ □◦ □ □



通过观察图表，我们可以确信这个组合的结果确实是从 F 到 H 的自然变换:

□ □ ◦ (□ · □)□ = (□ · □)□ ◦ □ □



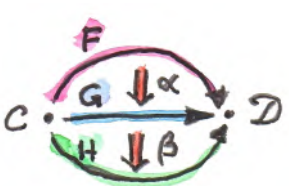
自然变换的组合是结合的，因为它们的分量，也就是常规态射，对于它们的组合是结合的。

最后，对于每个函子 F，都存在一个恒等自然变换 $1_{\square}$，它的分量是恒等态射:

$$\mathrm{id}_{\mathrm{id}\,\square\,::\,\square\,\square\,\to\,\square\,\square}$$

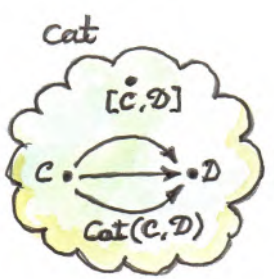
所以，确实，函子形成一个范畴。

关于符号的一点说明。 按照Saunders Mac Lane的说法，我使用点表示我刚刚描述的自然变换的组合。
问题在于有两种组合自然变换的方式。 这个被称为垂直组合，因为在描述它的图表中，函子通常垂直堆叠。垂直组合在定义函子范畴时很重要。 我将很快解释水平组合。



范畴 𝒞和 𝒟之间的函子范畴写作 𝒟 ℰ (𝒞, 𝒟)，或 [𝒞, 𝒟]，有时也写作 𝒟 𝒞。 最后一种符号表明函子范畴本身可以被认为是另一个范畴中的函数对象（指数）。这确实是这种情况吗？

让我们来看看我们迄今为止构建的抽象层次结构。我们从一个范畴开始，它是一组对象和态射。 范畴本身（或者严格来说是小范畴，其对象形成集合）是一个更高层次范畴 𝒟 ℰ 𝒞中的对象。 该范畴中的态射是函子。 在 𝒟 ℰ 𝒞中的一个Hom集合是一组函子。 例如 𝒟 ℰ (𝒞, 𝒟)是两个范畴 𝒞和 𝒟之间的一组函子。



一个函子范畴 [𝒞, 𝒟]也是两个范畴之间的函子集合（以自然变换为态射）。 它的对象与 𝒟 ℰ (𝒞, 𝒟)的成员相同。 此外，函子范畴作为一个范畴，必须本身是 𝒟 ℰ 𝒞的一个对象（恰好函子范畴在两个小范畴之间是小范畴）。 我们在一个范畴中的Hom集合和同一范畴中的对象之间有一个关系。 情况与我们在上一节中看到的指数对象完全相同。 让我们看看我们如何在 𝒟 ℰ 𝒞中构建后者。

正如你可能记得的那样，为了构建一个指数，我们首先需要定义一个乘积。 在 □ □ 中，这相对容易，因为小范畴是对象的集合，我们知道如何定义集合的笛卡尔积。 因此，乘积范畴 □ × □ 中的对象只是一对对象 (□, □)，一个来自 □，一个来自 □。 类似地，两个这样的对 (□, □) 和 (□′, □′) 之间的态射是一对态射 (□ , □)，其中 □ : : □ → □′ 和 □ : : □ → □′。 这些对态射按分量组合，总是存在一个恒等对，它只是一对恒等态射。 长话短说， □ □ 是一个完全的笛卡尔闭范畴，其中对于任意一对范畴都存在一个指数对象 □ □。 而在 □ □ 中，“对象”指的是一个范畴，所以 □□ 是一个范畴，我们可以将其视为 □ 和 □ 之间的函子范畴。

10.4 2-范畴

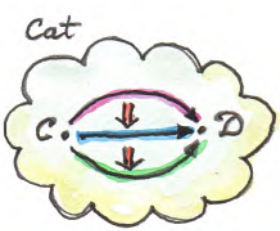
说完这些，让我们仔细看一下 □ □ ⁁ 根据定义，在 □ □ 中的任何 Hom 集合都是一组函子。 但是，正如我们所见，两个对象之间的函子具有比仅仅是一个集合更丰富的结构。它们形成一个范畴，其中自然变换充当态射。 由于函子在 □ □ 中被视为态射，自然变换就是态射之间的态射。

这种更丰富的结构是 □-范畴的一个例子，它是范畴的一种推广，除了对象和态射（在这种情况下可能被称为 1-态射）之外，还有 2-态射，它们是态射之间的态射。

在 □ □ 被视为 □-范畴的情况下，我们有：

- 对象：（小）范畴

- 1-态射：范畴之间的函子
- 2-态射：函子之间的自然变换



与两个范畴 □和 □之间的Hom-集合不同，我们有一个Hom-范畴 - 函子范畴 □□。 我们有常规的函子组合：从 □□到 □□的函子 □与从 □□到□□的函子 □组合得到从□□到□□的函子□ ◦ □。 但我们还有每个Hom-范畴内部的组合 - 函子之间的自然变换或2-态射的垂直组合。

在 □-范畴中有两种组合方式，问题是：它们如何相互作用?

让我们选择两个函子或1-态射，在 □ □ 中：

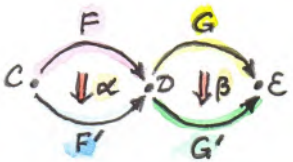
$$□ : : □ \to □$$

$$□ : : □ \to □$$

以及它们的组合：

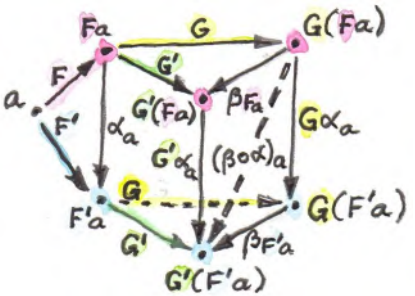
$$□ \circ □ : : □ \to □$$

假设我们有两个自然变换， □和 □，它们分别作用于函子 □和 □：
$□ : : □ \to □' □ : : □ \to □'$



请注意，我们不能对这对进行垂直组合，因为 □的目标与 □的源不同。 实际上，它们是两个不同的函子范畴的成员： □□和 □□。 然而，我们可以对函子 □′和 □′进行组合，因为 □′的目标是 □′的源——即范畴 □。 函子 □′ ∘ □′和□ ∘ □之间的关系是什么？

在我们的掌握下，有 □和 □，我们能定义一个从□ ∘ □到 □′ ∘ □′的自然变换吗？ 让我简要介绍一下构造过程。



像往常一样，我们从一个对象 □在 □中开始。 它的图像分裂成两个对象在 □中： □ □和 □′□。 还有一个态射，即 □的一个分量，连接这两个对象：

□ □: : □ □ → □′ □

当从 □到 □时，这两个对象进一步分裂成四个对象： □(□ □)， □′(□ □)， □(□′□)， □′(□′□)。 我们还有四个态射。

形成一个正方形。 其中两个态射是自然变换 □的分量：

$$□\ □□_{::}\ □(□\ □) \to □'(□\ □)$$
$$□\ □□ :: □(□,□) \to □'(□,□)$$

另外两个是 □□在两个函子（函子映射态射）下的像：

$$□\ □\ □_{□}: □(□\ □) \to □(□'\ □)$$
$$□'□\ □:: □'(□\ □) \to □'(□'□)$$

这是很多态射。 我们的目标是找到一条从□(□ □)到 □′(□′□)的态射，这是连接两个函子□。□和 □′。□′的自然变换的分量的候选： 实际上，从 □(□ □)到 □′(□′□)有两条路径可选：

$$□'□\ □。□\ □□$$
$$□\ □□。□\ □\ □$$

幸运的是，它们是相等的，因为我们形成的正方形是幸运的自然性正方形。

我们刚刚定义了从我们。□到□′。□′的自然变换的一个组成部分。 对于这个变换的自然性证明非常直观，只要你有足够的耐心。

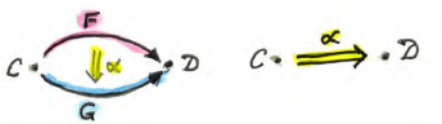
我们将这个自然变换称为水平组合的我们和我们：

$$□。□ :: □。□ \to □'。□'$$

同样，按照 Mac Lane 的说法，我使用小圆圈表示水平组合，尽管你也可能遇到星号。

这是一个范畴论的经验法则：每当你有组合时，你应该寻找一个范畴。 我们有自然变换的垂直组合，它是函子范畴的一部分。 但是水平组合呢？ 它属于哪个范畴？

找出答案的方法是侧着看找出找出找出。 将自然变换视为函子之间的箭头，而不是函子之间的箭头。 自然变换位于两个通过它进行转换的范畴之间。 我们可以将其视为连接这两个范畴的桥梁。



让我们专注于 □ □ 的两个对象 - 范畴 □和 □。 存在一组自然变换，它们连接从 □到 □的函子之间。
这些自然变换是从 □到 □的新箭头。 同样地，存在自然变换，它们连接从 □到 □的函子之间，我们可以将其视为从 □到□的新箭头。 水平合成是这些箭头的合成。

我们还有一个从 □到 □的恒等箭头。 它是将 □上的恒等函子映射到自身的恒等自然变换。
请注意，水平合成的恒等元素也是垂直合成的恒等元素，但反之则不成立。
最后，这两种合成满足交换律：

$$(□' \cdot □') \circ (□ \cdot □) = (□' \circ □) \cdot (□' \circ □)$$

我将引用Saunders Mac Lane的话：读者可以写下明显的图表来证明这个事实。

还有一个符号可能在将来会派上用场。 在这个新的侧面解释中 □ □ 有两种从对象到对象的方法：使用函子或使用自然变换。 然而，我们可以将函子箭头重新解释为一种特殊的自然变换：作用于该函子的恒等自然变换。 所以你经常会看到这个符号：

□ ◦ □

其中 □是从 □到 □的函子， □是从 □到 □的两个函子之间的自然变换。 由于你不能将函子与自然变换组合，因此这被解释为恒等自然变换 1□在 □之后的水平组合。

类似地：

□ ◦ □

是□在 1□之后的水平组合。

10.5 结论

这就结束了本书的第一部分。 我们已经学习了范畴论的基本词汇。 你可以将对象和范畴看作名词；将态射、函子和自然变换看作动词。
态射连接对象，函子连接范畴，自然变换连接函子。

但我们也看到，在一个抽象层次上看起来像是一个动作的东西，在下一个层次上变成了一个对象。 一组态射变成了一个函数对象。 作为一个对象，它可以是另一个态射的源或目标。 这就是高阶函数的思想。

函子将对象映射到对象，因此我们可以将其用作类型构造器或参数化类型。 函子还映射态射，因此它是

一个高阶函数 — fmap。 有一些简单的函子，比如Const、积和余积，可以用来生成各种代数数据类型。 函数类型也是函子，既协变又逆变，并且可以用来扩展代数数据类型。

函子可以被看作是函子范畴中的对象。
作为这样的对象，它们成为态射的源和目标：自然变换。 自然变换是一种特殊类型的多态函数。

10.6 挑战

1. 从Maybe函子到列表函子定义一个自然变换。 证明它的自然性条件。
2. 在Reader ()和列表函子之间定义至少两个不同的自然变换。 有多少个不同的()列表？
3. 使用Reader Bool和Maybe继续上一个练习。
4. 证明自然变换的水平组合满足自然性条件（提示：使用组件）。 这是一个很好的追踪图的练习。

5. 写一篇关于如何享受写下证明交换律所需的明显图表的短文。 为不同的Op函子之间的变换的相反自然性条件创建一些测试用例。这是一个选择：

```
op :: Op Bool Int
op = Op (\x -> x > 0)
```

和

```
f :: String -> Int
```

```
f x = read x
```

第二部分

11

声明式编程

在本书的第一部分中，我认为范畴论和编程都是关于可组合性的。 在编程中，你会不断地将问题分解，直到达到你可以处理的细节级别，然后逐个解决每个子问题，并从底向上重新组合解决方案。 大致上有两种方法可以做到这一点：告诉计算机要做什么，或者告诉它如何做。一种被称为声明式，另一种是命令式。

即使在最基本的层面上，你也可以看到这一点。 组合本身可以以声明式方式定义；例如， h是 g after f的组合：

```
h = g . f
```

或者以命令式方式定义；例如，先调用 f，记住该调用的结果，然后使用该结果调用 g：

```
h x = let y = f x
          in g y
```

程序的命令式版本通常被描述为按时间顺序排列的一系列动作。 特别是，在 f执行完成之前，调用 g是不可能发生的。 至少，在惰性语言中，通过call-by-need参数传递，实际执行可能会有所不同。

实际上，根据编译器的聪明程度，声明式代码和命令式代码的执行方式可能几乎没有区别。 但是，这两种方法在问题解决方式、代码可维护性和可测试性方面有时会有很大的差异。

主要问题是：面对一个问题，我们是否总是可以选择声明式和命令式方法来解决它？ 而且，如果存在声明式解决方案，它是否总是可以转化为计算机代码？ 这个问题的答案远非明显，如果我们能找到答案，我们可能会彻底改变对宇宙的理解。

让我详细说明一下。 在物理学中也有类似的二元性，这可能指向一些深层原则，或者告诉我们一些关于我们思维方式的东西。 理查德费曼在他自己关于量子电动力学的工作中提到了这种二元性作为灵感。

有两种表达方式-
大多数物理定律的表达方式。 一种使用局部或无穷小的考虑- 我们观察系统在一个小邻域内的状态，并预测它在下一个瞬时将如何演变。这通常是

通常使用微分方程来表示，这些方程需要在一段时间内进行积分或求和。

注意这种方法类似于命令式思维：我们通过按照一系列小步骤的结果来达到最终解决方案。实际上，物理系统的计算机模拟通常通过将微分方程转化为差分方程并进行迭代来实现。 这就是在小行星游戏中制作太空船动画的方式。 在每个时间步长中，太空船的位置会通过添加一个小增量来改变，该增量是通过将速度乘以时间间隔来计算的。 而速度则通过与加速度成比例的小增量来改变，加速度由力除以质量给出。

这些是与牛顿运动定律对应的微分方程的直接编码：

$$\square = \square \frac{\square\,\square}{\square\,\square}$$

$$\square = \square - \square$$

类似的方法可以应用于更复杂的问题，比如使用麦克斯韦方程传播电磁场，甚至使用格点QCD（量子色动力学）研究质子内部的夸克和胶子的行为。

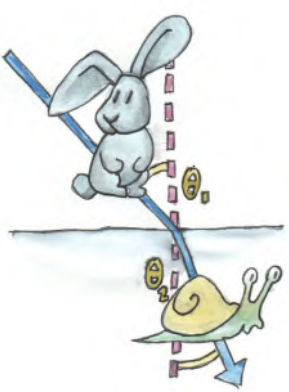
这种局部思考与空间和时间的离散化相结合，这种离散化是数字计算机使用所鼓励的，它在斯蒂芬·沃尔夫勒姆试图将整个宇宙的复杂性简化为细胞自动机系统的英勇尝试中找到了极端的表达方式。

另一种方法是全局的。我们观察系统的初始和最终状态，并通过最小化某个函数来计算连接它们的轨迹。最简单的例子是费马原理，即最短时间原理。

它表明光线沿着最小化飞行时间的路径传播。 特别是在没有反射或折射物体的情况下，从点 □到点 □的光线将走最短路径，即直线。 但是光在密集（透明）材料中传播速度较慢，如水或玻璃。 因此，如果你选择在空气中选择起点，在水下选择终点，光线更有利于在空气中旅行更长的距离，然后通过水中的捷径。 最短时间路径使光线在空气和水的边界处发生折射，从而得到斯涅尔定律： 光线在界面上发生折射的角度满足斯涅尔定律：

$$\frac{\square\ \square(\square_1)}{\square\ \square(\square_2)} = \frac{\square_1}{\square_2}$$

其中 $\square_1$是空气中的光速，而 $\square_2$是水中的光速。

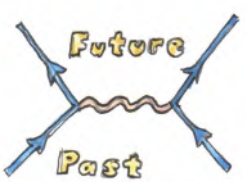


所有的经典力学都可以从最小作用量原理推导出来。 通过积分可以计算任何轨迹的作用量，作用量是动能和势能之间的差异。

能量（注意：是差异，而不是总和——总和将是总能量）。 当你发射迫击炮以击中给定目标时，抛射物首先会上升，重力势能较高，并在那里花费一些时间来积累对作用量的负贡献。

它还会在抛物线的顶部减速，以最小化动能。 然后它会加速通过低势能区域。

费曼最伟大的贡献是意识到最小作用量原理可以推广到量子力学。 在那里，问题再次以初始状态和最终状态的形式进行描述。 费曼路径积分用于计算这些状态之间的转换概率。



关键是我们可以以一种奇特而无法解释的方式描述物理定律。 我们可以使用局部视图，其中事物按顺序和小增量发生。或者我们可以使用

全局视图，在其中我们声明初始和最终条件，中间的一切都会自动跟随。

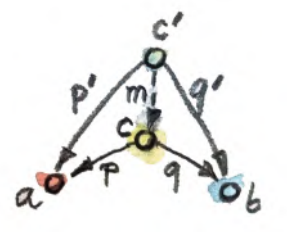
全局方法也可以用于编程，例如在实现光线追踪时。 我们声明眼睛的位置和光源的位置，并找出连接它们的光线可能采取的路径。 我们不明确地将每条光线的飞行时间最小化，但我们确实使用了斯涅尔定律和反射的几何学效应。

局部方法和全局方法之间最大的区别在于它们对空间和更重要的是时间的处理。 局部方法拥抱此时此地的即时满足感，而全局方法则采取长期静态的视角，仿佛未来已经被预定，我们只是分析某个永恒宇宙的属性。

在功能反应式编程（frp）方法中，没有比这更好的例证了。 与其为每个可能的用户操作编写单独的处理程序，所有这些处理程序都可以访问某些共享的可变状态，frp将外部事件视为无限列表，并对其应用一系列转换。从概念上讲，我们所有未来操作的列表都在那里，作为程序的输入数据可用。 从程序的角度来看，从程的数字列表、伪随机数列表或通过计算机硬件传输的鼠标位置列表之间没有区别。 在每种情况下，如果你想获取第在每个项目，你必须先遍历前在每−1个项目。 当应用于时间事件时，我们称之为因果性。

那么它与范畴论有什么关系呢？ 我将论证范畴论鼓励一种全局方法，因此支持声明式编程。 首先，与微积分不同，它没有内置的距离、邻域或时间概念。 我们所拥有的只是抽象的对象。

对象和它们之间的抽象连接。 如果你可以通过一系列步骤从 □到 □ ，那么你也可以一跃而至。 此外，范畴论的主要工具是普遍构造，它是一种全局方法的典范。 我们已经在实践中看到过它，例如在范畴积的定义中。 通过指定其属性来完成，这是一种非常声明性的方法。 它是一个带有两个投影的对象，并且它是最优的这样的对象-它优化了某个属性：将其他这样的对象的投影因子分解的属性。



将其与费马的最小时间原理或最小作用量原理进行比较。

相反，将其与传统的笛卡尔积定义进行对比，后者更加命令式。 你描述了如何从一个集合中选择一个元素和从另一个集合中选择另一个元素来创建一个产品的元素。这是一个创建一对的方法。 还有另一种方法可以拆分一对。

在几乎所有的编程语言中，包括函数式语言如Haskell，在构造上都内置了乘积类型、余积类型和函数类型，而不是通过通用构造来定义；尽管有人试图创建范畴编程语言（参见，例如，Tatsuya Hagino的论文[1]）。

尽管有人试图创建范畴编程语言（参见，例如，Tatsuya Hagino的论文[1]），但几乎每种编程语言，包括函数式语言如Haskell，在构造上都内置了乘积类型、余积类型和函数类型。

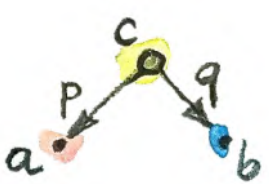
无论是直接使用还是间接使用，范畴论的定义都能够证明现有的编程结构，并且产生新的编程结构。 最重要的是，范畴论为以声明性方式推理计算机程序提供了一种元语言。 它还鼓励在将问题转化为代码之前对问题规范进行推理。

1http://web.sfc.keio.ac.jp/~hagino/thesis.pdf

12

极限和余极限

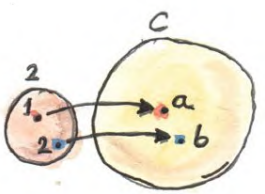
在范畴论中，似乎一切都与一切相关，并且一切都可以从许多角度来看待。 以乘积的通用构造为例。 现在我们对函子和自然变换有了更多了解，我们能否简化并可能推广它呢？让我们试试。



构造乘积始于选择两个对象 □和 □，我们想要构造它们的乘积。但是选择对象意味着什么呢？ 我们能否用更范畴论的术语来重新表述这个动作？

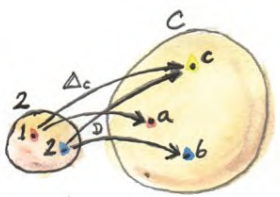
两个对象形成一个模式 - 一个非常简单的模式。我们可以将这个模式抽象成一个范畴 - 一个非常简单的范畴，但仍然是一个范畴。

这个模式可以被抽象成一个范畴 - 一个非常简单的范畴，但是它仍然是一个范畴。 这是一个我们将称之为 □的范畴。 它只包含两个对象，1和 2，除了两个必需的恒等态之外没有其他态射。 现在我们可以将在 □中选择两个对象重新表述为定义一个从范畴 □到 □的函子 □的行为。 一个函子将对象映射到对象，因此它的图像只有两个对象（如果函子将对象折叠起来，也可以只有一个对象，这也是可以的）。 它还将态射映射到态射 - 在这种情况下，它只是将恒等态射映射到恒等态射。



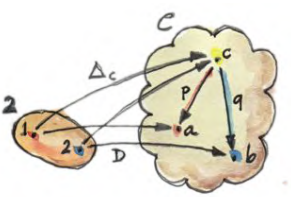
这种方法的优点在于它建立在范畴论的概念之上，避免了不准确的描述，比如“选择对象”，这些描述直接来自我们祖先的狩猎采集词汇。 而且，顺便说一下，它也很容易推广，因为我们可以使用比 □更复杂的范畴来定义我们的模式。

但是让我们继续。 定义乘积的下一步是选择候选对象 □。 在这里，我们可以用一个从单元范畴到 □的函子来重新表达选择。 实际上，如果我们使用Kan扩展，这将是正确的做法。但是由于我们还没有准备好使用Kan扩展，我们可以使用另一个技巧：一个从相同范畴 □到 □的常量函子 Δ。 在□中选择 □可以使用 $\Delta_{□}$。 记住，$\Delta_{□}$将所有对象映射到 □，将所有态射映射到 $\mathrm{id}_{□}$。



现在我们有两个函子， Δ□和 □在 □和 □之间，所以自然而然地问起自然变换。 由于 □中只有两个对象，自然变换将有两个组成部分。 在 □中，对象 1通过 Δ □映射到□，通过 □映射到 □。 因此，在 Δ□和 □之间的自然变换的组成部分在 1处是从 □到 □的态射。我们可以称之为 □。 类似地，第二个组成部分是从 □到 □的态射——在 □中对象 2的映像下通过 □。

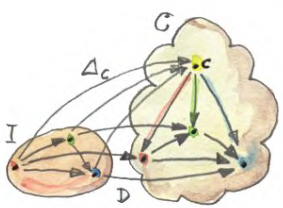
但这些恰好就像我们在原始定义中使用的两个投影一样。 因此，我们可以只谈论选择函子和自然变换，而不是选择对象和投影。 在这种简单情况下，我们的变换的自然性条件是平凡满足的，因为 □中除了恒等态射之外没有其他态射。



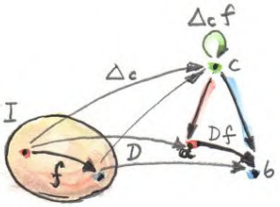
这种构造的一般化适用于除 □之外的范畴，例如包含非平凡态射的范畴，将在 Δ□和 □之间的变换上施加自然性条件。我们称之为

变换为一个锥，因为 Δ的图像是一个由自然变换的分量形成的锥形/金字塔的顶点。 □的图像形成锥形的底部。

一般来说，要构建一个锥形，我们从定义模式的范畴 □开始。 它是一个小的、通常是有限的范畴。 我们选择一个从□到 □的函子 □，并称其为（或其图像为）一个图表。 我们在 □中选择一些 □ 作为我们锥形的顶点。 我们用它来定义常量函子 $\Delta_{□}$从 □到□。 从 $\Delta_{□}$到 □的自然变换就是我们的锥形。 对于有限的□，它只是连接 □ 和图表的一堆态射： □下的 □的图像。



自然性要求这个图中的所有三角形（金字塔的墙壁）都是可交换的。 事实上，取任意态射 □ 在 □中。 函子 □将其映射为一个态射 □ □在 □中，这个态射构成了某个三角形的底部。 常量函子 $\Delta_{□}$将 □ 映射为□上的恒等态射。Δ将态射的两端压缩成一个对象，自然性正方形变成了一个可交换的三角形。 这个三角形的两条边是自然变换的分量。

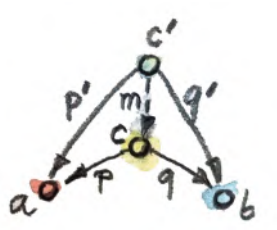


这就是一个圆锥。 我们感兴趣的是通用圆锥 —
就像我们为定义乘积选择了一个通用对象一样。

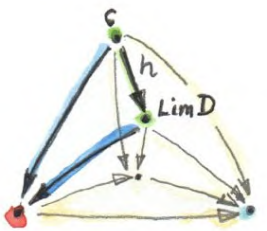
有很多方法可以做到这一点。例如，我们可以基于给定的函子 □定义一个圆锥范畴。 该范畴中的对象是圆锥。 然而，并不是 □ 中的每个对象 □都可以成为一个圆锥的顶点，因为可能不存在 Δ□ 和 □之间的自然变换。

为了使其成为一个范畴，我们还必须定义锥之间的态射。 这些态射将完全由它们的顶点之间的态射确定。 但并不是任何态射都可以。请记住，在我们构造乘积时，我们强加了候选对象（顶点）之间的态射必须是投影的公共因子的条件。例如：

```
p' = p . m
q' = q . m
```



在一般情况下，这个条件转化为以因子化态射为一边的三角形都是可交换的条件。



连接两个锥形图的可交换三角形，其中因子化态射为 *h*（这里，下方的锥形图是通用的，以 Lim□为其顶点）

我们将把这些因子化态射作为我们锥形图范畴中的态射。 很容易验证这些态射确实可以组合，并且恒等态射也是一个因子化态射。 因此，锥形图形成一个范畴。

现在我们可以将通用锥定义为范畴中的终对象。 终对象的定义表明，从任何其他对象到该对象都存在唯一的态射。 在我们的情况下，这意味着从任何其他锥的顶点到通用锥的顶点存在唯一的分解态射。 我们将这个通用锥称为图 □的极限 Lim□（在文献中，你经常会看到一个指向 □下方的左箭头在 Lim符号下）。 通常，作为一种简写，我们将这个锥的顶点称为极限（或极限对象）。

直觉是极限将整个图表的属性体现在一个单一的对象中。 例如，我们两个对象的极限是两个对象的乘积。 乘积（连同

两个投影）包含了关于两个对象的信息。而且，成为普遍的意味着它没有多余的垃圾。

12.1 极限作为自然同构

这个极限的定义还是有些不令人满意。 我的意思是，它是可行的，但是我们仍然有这个交换性条件，用于连接任意两个锥体的三角形。 如果我们能用一些自然性条件来替代它，那将更加优雅。但是怎么做呢？

我们不再处理一个锥体，而是处理一个整个集合（实际上是一个范畴）的锥体。 如果极限存在（让我们明确一点——没有任何保证），其中一个锥体就是通用锥体。 对于每个其他锥体，我们都有一个唯一的分解态射，将其顶点映射到通用锥体的顶点，我们将其命名为 $\square$，通用锥体的顶点我们称为 $\mathrm{Lim}\square$。 （实际上，我可以省略“其他”这个词，因为恒等态射将通用锥体映射到自身，并且它通过自身平凡地分解。）让我重复一下重要的部分：给定任何锥体，都存在一种唯一的特殊态射。 我们将锥体映射到特殊态射，这是一种一对一的映射。

这个特殊的态射是同态集合 $\square(\square, \mathrm{Lim}\square)$的一个成员。 这个同态集合的其他成员不那么幸运，因为它们不能将锥的映射分解。 我们想要的是能够为每个 $\square$选择一个来自集合 $\square(\square, \mathrm{Lim}\square)$的态射——一个满足特定交换性条件的态射。 听起来像是在定义一个自然变换吗？ 当然是！但是与这个变换相关的函子是哪些呢？

一个函子是将 $\square$映射到集合 $\square(\square, \mathrm{Lim}\square)$的映射。 它是从 $\square$到 $\square$ $\square$的一个函子——它将对象映射到集合。 实际上它是一个逆变函子。 下面是我们如何定义它对态射的作用：让我们取一个

从 C' 到 C 的态射 F：

$$F :: C' \to C$$

我们的函子将 C' 映射到集合 $C(C', \mathrm{Lim}D)$。 为了定义这个函子对 F 的作用（换句话说，提升 F），我们需要定义 $C(C, \mathrm{Lim}D)$ 和 $C(C', \mathrm{Lim}D)$ 之间的对应映射。 所以让我们选择 $C(C, \mathrm{Lim}D)$ 中的一个元素 u，并看看我们是否可以将其映射到 $C(C', \mathrm{Lim}D)$ 中的某个元素 。 一个同态集合的元素是一个态射，因此我们有：

$$u :: C \to \mathrm{Lim}D$$

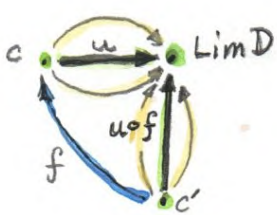
我们可以将 u 与 F 预合成，得到：

$$u \cdot F :: C' \to \mathrm{Lim}D$$

这是 $C(C', \mathrm{Lim}D)$ 的一个元素，因此，我们找到了一个映射的态射：

```
contramap :: (c' -> c) -> (c -> LimD) -> (c' -> LimD)
contramap f u = u . f
```

注意 C 和 C' 的顺序颠倒，这是一个反变函子的特点。



为了定义一个自然变换，我们需要另一个同时也是从 □到 □ □的函子。 但这次我们考虑一组锥。 锥就是自然变换，所以我们在看的是自然变换□ □ □(Δ□, □)的集合。 从 c到这个特定自然变换集合的映射是一个（反变）函子。我们如何证明这一点呢？ 同样，让我们定义它在态射上的作用：

$$□ : : □' \to □$$

□的提升应该是从 □到 □的两个函子之间的自然变换的映射:

$$□ □ □(Δ□, □) \to □ □ □(Δ□', □)$$

我们如何映射自然变换？ 每个自然变换都是一组态射的选择 - 它的组成部分 - 每个元素对应一个态射。 □的元素。 某个 □的一个组成部分（Δ□, □）在 □（ □中的一个对象）上是一个态射：

$$□ □: : Δ□ □\to □ □$$

或者，使用常量函子 Δ的定义，

$$□ □: : □ \to □ □$$

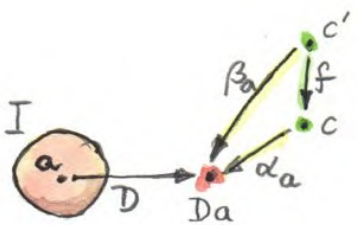
给定 □和 □，我们必须构造一个 □，它是Δ□', □中的一个成员。 它在 □处的组成部分应该是一个态射：

$$□ □: : □' \to □ □$$

我们可以通过与 □的前合成来轻松地从前者 (□□) 得到后者 (□□)：

$$□ □= □ □□$$

相对容易地证明这些组件确实相加得到一个自然变换。



给定我们的态射 □，我们因此建立了两个自然变换之间的映射，逐个组件。 这个映射为函子定义了 contramap：

$$□ \to □ □ □(Δ□, □)$$

我刚刚做的是向你展示，我们有两个（逆变）从 □到 □ □ □的函子。 我没有做任何假设 - 这些函子总是存在的。

顺便说一下，这两个函子中的第一个在范畴论中扮演着重要的角色，在我们讨论Yoneda引理时会再次见到它。 从任意范畴□到 □ □ □的逆变函子有一个名字：它们被称为“预层”。 这个被称为representablepresheaf。 第二个函子也是一个预层。

既然我们有了两个函子，我们可以谈论它们之间的自然变换。 所以，不多说了，这就是结论：从范畴 □到范畴 □的函子 □有极限 $\mathrm{Lim}□$，当且仅当这两个我刚刚定义的函子之间存在一个自然同构：

$$□(□, \mathrm{Lim}Li) \simeq □ □ □(Δ□, □)$$

让我提醒你什么是自然同构。它是一个自然变换，其每个分量都是一个同构，也就是说是一个可逆的态射。

我不打算证明这个陈述。 这个过程相当直接，尽管有些乏味。 在处理自然变换时，你通常关注分量，它们是态射。 在这种情况下，由于两个函子的目标都是 □ □，□ 自然同构的分量将是函数。 这些是高阶函数，因为它们从同态集合到自然变换集合。 同样，你可以通过考虑函数对其参数的作用来分析一个函数：这里的参数将是一个态射——一个 □(□, Lim□)的成员——而结果将是一个自然变换——一个□ □ □(Δ□, □)的成员，或者我们所称的锥。 这个自然变换又有它自己的分量，它们是态射。

所以这是一种完全的态射，如果你能跟踪它们，你就能证明这个陈述。

最重要的结果是，这个同构的自然性条件恰好是锥的映射的可交换性条件。

作为即将到来的吸引力的预览，让我提到集合□ □ □(Δ□, □)可以被看作是函子范畴中的一个同态集合；因此，我们的自然同构关联了两个同态集合，这指向了一个更一般的关系，称为伴随关系。

12.2 极限的例子

我们已经看到，范畴积是由一个简单的范畴 □生成的图的极限。

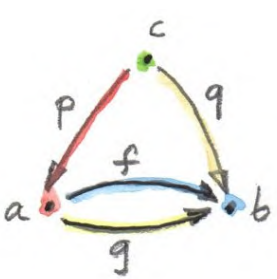
甚至还有一个更简单的极限的例子：终对象。 第一个冲动是认为一个单例范畴会导致一个终对象，但事实上比这更严峻：终对象是由一个空范畴生成的极限。 从一个空范畴到一个函子不选择任何对象，所以一个锥缩小到只有顶点。 通用锥是唯一的顶点，它从任何其他顶点都有一个唯一的态射到它。 你会认出这是终对象的定义。

下一个有趣的极限被称为等值器. 它是由一个具有两个平行态射的两元范畴生成的极限（以及恒等态射，如常）。 这个范畴选择了一个由两个对象 □和 □以及两个态射组成的 □中的图表：

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

为了构建这个图表上的锥体，我们需要添加顶点 □和两个投影：

```
p :: c -> a
q :: c -> b
```



我们有两个必须满足的三角形：

```
q = f . p
q = g . p
```

这告诉我们 □由这些方程之一唯一确定，比如 q = f . p，我们可以从图中省略它。 所以我们只剩下一个条件：

```
f . p = g . p
```

思考的方式是，如果我们将注意力限制在 □ □ 止，函数 □的图像选择了 □的一个子集。 当限制在这个子集上时，函数 □和 □是相等的。

例如，将 □视为由坐标 □和 □参数化的二维平面。 将 □视为实数线，并取：

```
f (x, y) = 2 * y + x
g (x, y) = y - x
```

这两个函数的等值器是实数集合（顶点， □）和函数：

```
p t = (t, (-2) * t)
```

注意， (□ □)在二维平面上定义了一条直线。
沿着这条直线，两个函数是相等的。

当然，还有其他集合 □′和函数 □′可能导致相等性：

```
f . p' = g . p'
```

但它们都通过 唯一地分解。 例如，我们可以将单例集合 ()视为 ′，并取函数：

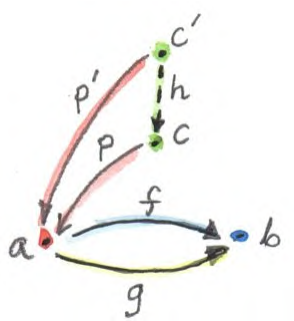
p'() = (0, 0)

这是一个好的锥体，因为 (0, 0)= (0, 0)。 但它并不是普遍的，因为通过 *h*的唯一分解。

p' = p . h

使用

h () = 0



等值器可以用来解决 = 类型的方程。
但它更加通用，因为它是通过对象和
态射来定义的，而不是代数上的。

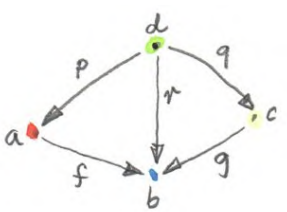
解方程的一个更加通用的思想体现在另一个极限中——拉回。在这里，我们仍然有两个我们想要等同的态射，但这次它们的定义域不同。 我们从一个形状为1 → 2 ← 3的三对象范畴开始。该图

对应于这个范畴的包括三个对象， □, □, 和 □, 以及两个态射:

```
f :: a -> b
g :: c -> b
```

这个图表通常被称为一个　余锥　.
　　在这个图表上构建的锥包括顶点, □, 和三个态射:

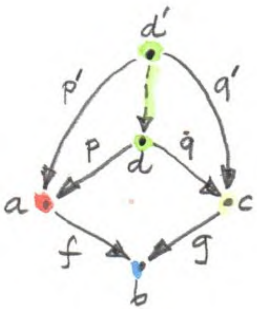
```
p :: d -> a
q :: d -> c
r :: d -> b
```



可交换性条件告诉我们 □完全由其他态射确定，并且可以从图中省略。 所以我们只剩下以下条件:

```
g . q = f . p
```

一个拉回是这种形状的普遍锥.



再次，如果你将焦点缩小到集合，你可以将对象□看作是由 □和 □中的元素对组成的，其中 □作用于第一个分量等于 □作用于第二个分量. 如果这还是太泛化了，考虑一下特殊情况，其中 □是一个常数函数，比如 □ _= 1.23(假设 □是一组实数). 那么你实际上是在解方程:

f x = 1.23

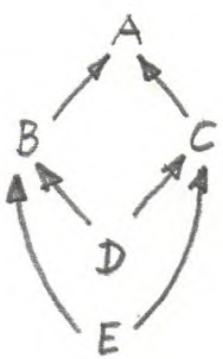
在这种情况下，选择在这是无关紧要的（只要它不是一个空集），所以我们可以将其视为一个单例集。集合在这可以是三维向量的集合，在这可以是向量长度。 然后，回拉是一对（然后，())的集合，其中）的是长度为1.23的向量（满足方程√‾(□2+□2+□2)= 1.23），()是单例集的虚元素。

但是回拉还有更广泛的应用，也适用于编程。
例如，将C++类视为一个范畴，其中态射是连接子类和超类的箭头。 我们将继承视为传递性质，所以如果C从B继承，B从A继承，那么我们将说C从A继承（毕竟，你可以传递一个指向C的指针）

在这里，我们假设C继承自C，因此我们对每个类都有一个恒等箭头。这样，子类化与子类型化是一致的。C++还支持多重继承，因此您可以构建一个菱形继承图，其中两个类B和C继承自A，并且第四个类D从B和C中多重继承。通常，D会获得两个A的副本，这很少是可取的；但是您可以使用虚拟继承，在D中只有一个A的副本。在这个图中，D作为一个回溯意味着什么？这意味着任何从B和C中多重继承的类E也是D的子类。这在C++中不能直接表

达，因为子类型化是名义的（C++编译器不会推断这种类关系-它需要“鸭子类型”）。但是我们可以超出子类型化关系，而是询问从E到D的转换是否安全。如果D是B和C的基本组合，没有额外的数据和方法覆盖，那么这个转换将是安全的。

当然，如果B和C之间存在名称冲突，就不会有回拉。



在类型推断中，还有一种更高级的回拉使用。 经常需要统一两个表达式的类型。 例如，假设编译器想要推断函数的类型：

```
twice f x = f (f x)
```

它将为所有变量和子表达式分配初步类型。 特别是，它将分配：

```
f           :: t0
x           :: t1
f x         :: t2
f (f x) :: t3
```

从中推断出：

```
twice :: t0 -> t1 -> t3
```

它还会根据函数应用的规则得出一组约束：

```
t0 = t1 -> t2 -- 因为f应用于x
t0 = t2 -> t3 -- 因为f应用于(f x)
```

这些约束必须通过找到一组类型（或类型变量）来统一，当这些未知类型在两个表达式中替换时，产生相同的类型。 其中一种替换是：

```
t1 = t2 = t3 = Int
twice :: (Int -> Int) -> Int -> Int
```

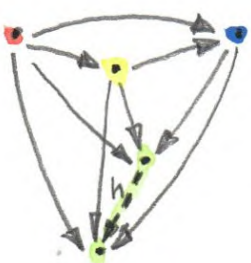
但是，显然，这不是最一般的。最一般的替换是使用一个拉回得到的。 我不会详细介绍，因为它们超出了本书的范围，但你可以自己验证结果应该是：

```
twice :: (t -> t) -> t -> t
```

带有 t一个自由类型变量。

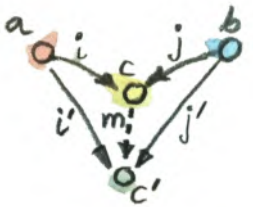
12.3 余极限

就像范畴论中的所有构造一样，极限在相反的范畴中有它们的对偶映射。 当你反转锥形图中的所有箭头的方向时，你得到一个共锥，其中的通用共锥被称为余极限。 注意，反转也会影响因子态射，它现在从通用共锥流向任何其他共锥。



带有一个因子态射 *h*连接两个顶点的共锥。

余极限的一个典型例子是余积，它对应于由 □生成的图表，在定义乘积时我们使用了该范畴。



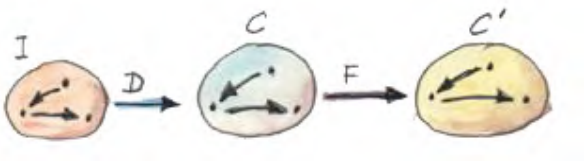
产品和余积都以不同的方式体现了一对对象的本质。

就像终结对象是一个极限一样，初始对象是一个对应于基于空范畴的图表的余极限。

拉回的对偶被称为推出。 它基于一个称为跨度的图表，由范畴 1 ← 2 → 3生成。

12.4 连续性

我之前说过，函子接近于范畴之间的连续映射的概念，因为它们不会破坏现有的连接（态射）。 从范畴 $\mathcal{C}$到范畴 $\mathcal{C}'$的连续函子 F 的实际定义包括保持极限的要求。 在范畴 $\mathcal{C}$中，每个图表 D都可以通过简单地组合两个函子映射到范畴 $\mathcal{C}'$中的图表$F \circ D$。 对于函子 F的连续性条件规定，如果图表 D有一个极限 $\mathrm{Lim}\, D$，则图表$F \circ D$也有一个极限，并且它等于 $F(\mathrm{Lim}\, D)$。



请注意，因为函子将态射映射到态射，并将组合映射到组合，所以锥的图像始终是一个锥。 一个可交换的三角形始终被映射为一个可交换的三角形（函子保持组合）。 对于分解态射也是如此：分解态射的图像也是一个分解态射。 因此，每个函子几乎都是连续的。 可能出错的是唯一性条件。 在□′中，分解态射可能不是唯一的。

在□中可能还有其他“更好的锥”，在中可′中不可用。

同态函子是连续函子的一个例子。 回想一下，同态函子回想(□, □)在第一个变量中是逆变的，在第二个变量中是协变的。 换句话说，它是一个函子：

□ □ □× □ → □ □ □

当第二个参数固定时，同态集函子（变成可表示的预层）将□中的余极限映射到 □ □ 中的极限；当第一个参数固定时，它将极限映射到极限。

在Haskell中，同态函子是任意两种类型映射到一个函数类型的映射，因此它只是一个参数化的函数类型。 当我们固定第二个参数，比如 String，我们得到了逆变函子：

```
newtype ToString a = ToString (a -> String)
instance Contravariant ToString where
    contramap f (ToString g) = ToString (g . f)
```

连续性意味着当 ToString应用于余极限时，例如余积 Either b c，它将产生一个极限；在这种情况下，是两个函数类型的积：

```
ToString (Either b c) ~ (b -> String, c -> String)
```

实际上，任何一个要么 b的函数都可以通过一个情况语句来实现，其中两种情况由一对函数处理。

类似地，当我们固定范畴中的第一个参数时，我们得到了熟悉的读取器函子。 它的连续性意味着，例如，任何返回乘积的函数都等价于一组函数的乘积；特别地：

```
r -> (a, b) ~ (r -> a, r -> b)
```

我知道你在想什么：你不需要范畴论来弄清楚这些事情。你是对的！不过，我发现这样的结果可以从第一原则推导出来，而不需要借助位和字节、处理器架构、编译器技术，甚至是λ演算。

如果你好奇“极限”和“连续性”这些名词的来源，它们是从微积分中对应概念的一般化。 在微积分中，极限和连续性是通过开放邻域来定义的。 定义拓扑的开集构成了一个范畴（一个偏序集）。

12.5 挑战

1. 你如何描述C++类别中的推出(pushout)?
2. 证明恒等函子 $\mathrm{Id}::\square \to \square$的极限是初始对象。

3. 给定集合的子集构成一个范畴。 在该范畴中，如果第一个集合是第二个集合的子集，则连接两个集合的箭头被定义为该范畴中的态射。 在这样的范畴中，两个集合的拉回(pullback)是什么？推出(pushout)又是什么？ 初始对象和终结对象是什么？
4. 你能猜出什么是等值器(coequalizer)吗?
5. 证明在具有终结对象的范畴中，向终结对象的拉回(pullback)是一个积(product)。
6. 类似地，证明如果存在初始对象，则从初始对象的推出(pushout)是一个余积(coproduct)。

13

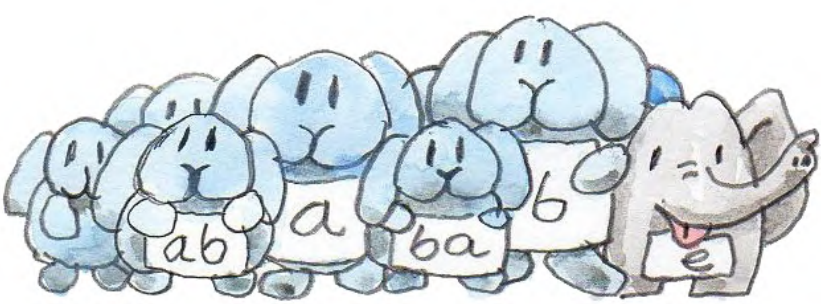
Free Monoids

Mon幺半群是范畴论和编程中的重要概念。范畴对应于强类型语言，幺半群对应于无类型语言。这是因为在幺半群中，你可以组合任意两个箭头，就像在无类型语言中你可以组合任意两个函数（当然，在执行程序时可能会出现运行时错误）。

我们已经看到，一个幺半群可以被描述为一个只有一个对象的范畴，其中所有的逻辑都被编码在态射组合的规则中。这个范畴模型与更传统的集合论定义的幺半群完全等价，其中我们通过将两个集合的元素“相乘”得到第三个元素。这个“相乘”的过程可以进一步分解为首先形成一对元素，然后将这对元素与现有元素进行标识——它们的“乘积”。

当我们放弃乘法的第二部分——将一对元素与现有元素进行标识时，会发生什么？例如，我们可以从任意集合开始，形成所有可能的元素对，并将它们称为新元素。 然后我们将这些新元素与所有可能的元素配对，依此类推。这是一个连锁反应——我们将不断添加新元素。 结果是一个无限集合，几乎是一个幺半群但是一个幺半群还需要一个单位元素和结合律。 没问题，我们可以添加一个特殊的单位元素并标识一些对 — 只需要支持单位元素和结合律。

让我们通过一个简单的例子来看看这是如何工作的。 让我们从一个包含两个元素的集合开始，{□, □}。 我们将称它们为自由幺半群的生成元。首先，我们将添加一个特殊元素 □作为单位元。 接下来，我们将添加所有元素对并称之为“乘积”。 元素 □和 □的乘积将是对 (□, □)。 元素 □和 □的乘积将是对 (□, □)，元素 □和 □的乘积将是 (□, □)，元素 □和 □的乘积将是 (□, □)。
我们还可以用 □来形成对，比如 (□, □), (□, □)等等，但我们会用 □, □等来代表它们。所以在这一轮中，我们只会添加 (□, □), (□, □)和 (□, □)以及 (□, □)，最终得到集合 {□, □, □, (□, □), (□, □), (□, □), (□, □)}。



在下一轮中，我们将继续添加元素，如：$(□,(□,□)),((□,□),□)$，等等。在这一点上，我们必须确保结合律成立，因此我们将$(□,(□,□))$与$((□,□),□)$等同，等等。换句话说，我们不需要内部括号。

你可以猜测这个过程的最终结果是什么：我们将创建所有可能的□和□的列表。实际上，如果我们将□表示为空列表，我们可以看到我们的“乘法”只是列表的连接。

这种构造方式，你可以持续生成所有可能的元素组合，并执行最小数量的等同操作-只需足够维持法则-被称为自由构造。 我们刚刚从生成器集合$\{□,□\}$构造了一个自由幺半群。

13.1 Haskell中的自由幺半群

在Haskell中，一个包含两个元素的集合等同于类型 Bool，而由该集合生成的自由幺半群等同于类型 [Bool]（布尔列表）。 （我故意忽略了无限列表的问题。）在Haskell中，幺半群由类型类
定义：

```
class Monoid m where
      mempty :: m
      mappend :: m -> m -> m
```

这只是说每个 Monoid必须有一个中性元素，称为 mempty，和一个二元函数（乘法）称为 mappend。 单位元和结合律无法在Haskell中表示，必须由程序员在每次实例化幺半群时进行验证。

任何类型的列表形成幺半群的事实由以下实例定义描述：

```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

它说明空列表 `[]`是单位元素，列表连接
`(++)`是二元操作。

正如我们所见，类型为 `a`的列表对应于以集合 `a`作为生成器的自由幺半群。 具有乘法的自然数集合不是自由幺半群，因为我们识别了许多乘积。
比如说比较：

```
2 * 3 = 6
[2] ++ [3] = [2, 3] // 不同于 [6]
```

这很简单，但问题是，在范畴论中，我们不允许查看对象内部，我们能否执行这种自由构造呢?
我们将使用我们的得力工具：通用构造。

第二个有趣的问题是，是否可以通过识别比法则所需的最小数量更多的元素，从某个自由幺半群中获得任何幺半群？ 我将向你展示，这直接来自通用构造。

13.2 自由幺半群的通用构造

如果你还记得我们之前对通用构造的经验，你可能会注意到，这与构造某物并不那么相关，而是选择一个最适合给定模式的对象。 所以，如果我们想要使用通用构造来“构造”一个自由幺半群，我们需要考虑一大堆幺半群，从中选择一个。我们需要一个完整的幺半群范畴来选择。 但是，幺半群是否构成一个范畴?

让我们首先看一下作为集合的幺半群，它们通过单位元和乘法定义了附加结构。 我们将选择那些保持幺半群结构的函数作为态射。 这样的结构保持函数被称为同态。 幺半群同态必须将两个元素的乘积映射为两个元素映射的乘积：

h (a * b) = h a * h b

并且它必须将单位映射到单位。

例如，考虑从整数列表到整数的同态。 如果我们将 [2]映射到2，将 [3]映射到3，那么我们必须将 [2, 3]映射到6，因为连接

[2] ++ [3] = [2, 3]

变成乘法

2 * 3 = 6

现在让我们忘记单个幺半群的内部结构，只将它们视为具有相应态射的对象。 你得到一个幺半群的范畴 □ □ ⊾□

好的，也许在我们忘记内部结构之前，让我们注意到一个重要的性质。 □ □ 的每个对象都可以被简单地映射到一个集合。它只是它的元素的集合。 这个集合被称为底层集合。实际上，我们不仅可以将这个这个这个的对象映射到集合，还可以将的对的对的对的态射（同态）映射到函数。 再次，这似乎有点平凡，但它很快就会变得有用。 从□ □ 到□ □ 的对象和态射的映射实际上是一个函子。 由于这个函子“忘记”了单子结构 - 一旦我们进入一个普通的集合，我们不再区分单位元素或关心乘法 - 它被称为遗忘函子。

遗忘函子经常在范畴论中出现。 遗忘函子经常在范畴论中出现。

我们现在对 □ □ 有两种不同的视角。 我们可以像处理其他范畴的对象和态射一样处理它。 在这个视角中，我们看不到幺半群的内部结构。 我们只能说 □ □ 中的一个特定对象通过态射与自身和其他对象连接。 态射的“乘法”表——合成规则——是从另一个视角派生出来的：作为集合的幺半群。 通过转向范畴论，我们并没有完全失去这个视角——我们仍然可以通过我们的遗忘函子来访问它。

为了应用通用构造，我们需要定义一个特殊的属性，让我们能够在幺半群的范畴中搜索并选择最佳的自由幺半群候选者。 但是自由幺半群是由其生成元定义的。 不同的生成元选择会产生不同的自由幺半群（一个 Bool列表与一个 Int列表不同）。 我们的构造必须从一组生成元开始。 所以我们又回到了集合！这就是遗忘函子发挥作用的地方。 我们可以用它来透视我们的幺半群。

我们可以在这些斑点的透视图中识别出生成元。以下是它的工作原理：

我们从一组生成器开始， □。那是一个 □ □ 中的集合。

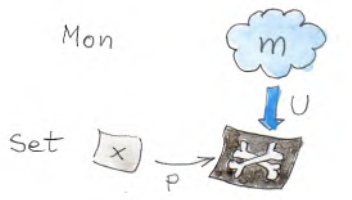
我们要匹配的模式由一个幺半群 □— 一个 □ □ 对象 — 和一个函数 □在 □ □ 中组成：

```
p :: x -> U m
```

其中 □是我们从 □ □ 到 □ □ 的遗忘函子。 这是一个奇怪的异构模式 — 一半在 □ □ 中，一半在 □ □ 中。

这个想法是函数 □将识别出□的生成器集合
在X射线图像中。 函数可能是无关紧要的

在识别集合内部的点方面很差（它们可能会将其折叠）。 这将由通用构造来解决，它将选择最佳的模式代表。



我们还必须定义候选人之间的排名。 假设我们还有另一个候选人：一个幺半群 □和一个能够识别其X射线图像中生成元的函数：

```
q :: x -> U n
```

我们将说 □比 □更好，如果存在幺半群的同态（即保持结构的同态）：

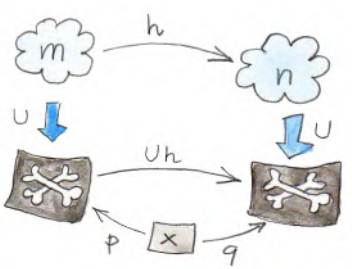
```
h :: m -> n
```

在 □的作用下（记住， □是一个函子，所以它将态射映射到函数），它在 □上分解：

```
q = U h . p
```

如果你将 □看作是选择 □中的生成元；将 □看作是选择 □中的“相同”生成元；那么你可以将 *h*看作是在这两个幺半群之间映射这些生成元。 请记住，根据定义， *h*

保留了单调结构。 这意味着一个单调中两个生成器的乘积将映射到第二个单调中相应两个生成器的乘积，依此类推。



这个排名可以用来找到最佳候选人 - 自由单调。
下面是定义：

> 我们将说 □（连同函数 □）是具有生成器 □的自由单调，当且仅当存在一个唯一的态射 *h*从 □到任何其他单调 □（连同函数 □），满足上述分解性质。

顺便说一下，这回答了我们的第二个问题。 函数□ *h*是将□ □的多个元素合并为□ □的单个元素的函数。 这种合并对应于识别自由单调的一些元素。 因此，具有生成器 □的任何单调都可以通过识别一些元素从基于 □的自由单调中获得。 自由单调是只进行了最少限度的识别的单调。

当我们谈论伴随时，我们将回到自由幂等元。

13.3 挑战

1. 你可能会认为（就像我一开始那样）幺半群同态保持单位元的要求是多余的。 毕竟，我们知道对于所有

   ```
   h a * h e = h (a * e) = h a
   ```

   所以 *h*的作用类似于右单位元（类似地，也是左单位元的类比）。 问题在于 *h*，对于所有的 可能只覆盖目标幺半群的一个子幺半群。 在 *h*的图像之外可能存在一个“真正的”单位元。 证明保持乘法的幺半群同构必然保持单位元。

2. 考虑从整数列表与连接的幺半群到整数与乘法的幺半群的同态。 空列表 []的图像是什么？ 假设所有单元素列表都映射到它们包含的整数，即 [3]映射到3，等等。那么 [1, 2, 3, 4]的图像是什么？ 有多少不同的列表映射到整数12？ 两个幺半群之间还有其他的同态吗？

3. 由一个单元素集合生成的自由幺半群是什么？ 你能看到它同构于什么吗？

14

可表示函子

是时候谈谈集合了。 数学家对集合论有着爱恨交加的关系。 它是数学的汇编语言——至少过去是这样。范畴论试图在某种程度上摆脱集合论。 例如，众所周知，所有集合的集合是不存在的，但是所有集合的范畴例如例如例如是存在的。所以这是好的。 另一方面，我们假设范畴中任意两个对象之间的态射形成一个集合。我们甚至称之为同态集。 公平地说，范畴论的一个分支中，态射不形成集合。 相反，它们是另一个范畴中的对象。 那些使用同态对象而不是同态集的范畴被称为富范畴。 然而，在接下来的内容中，我们将坚持使用具有老式同态集的范畴。

集合是除了范畴对象之外最接近无特征的东西。 集合有元素，但你对这些元素不能说太多。 如果你有一个有限集合，你可以数一下元素。

你可以使用基数数来大致计算无限集合的元素。
例如，自然数集比实数集小，尽管两者都是无限的。 但是，也许令人惊讶的是，有理数集和自然数集的大小是相同的。

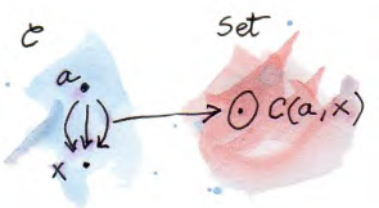
除此之外，关于集合的所有信息都可以通过它们之间的函数编码 - 特别是可逆的称为同构的函数。 就所有目的而言，同构集合是相同的。在我招致基础数学家的愤怒之前，让我解释一下等号和同构之间的区别是非常重要的。 事实上，这是最新的数学分支 - 同伦类型论（HoTT）的主要关注点之一。 我提到HoTT是因为它是一个纯数学理论，它从计算中汲取灵感，而其中的主要支持者弗拉基米尔·沃伊多夫斯基在研究Coq定理证明器时有了重大的顿悟。 数学和编程之间的互动是双向的。

关于集合的重要教训是，比较不同元素的集合是可以的。 例如，我们可以说给定的自然变换集合同构于某个态射集合，因为集合就是集合。在这种情况下，同构意味着对于每个从一个集合到另一个集合的自然变换，都存在一个唯一的从另一个集合到该集合的态射。 它们可以相互配对。 如果它们是来自不同范畴的对象，你不能比较苹果和橙子，但你可以比较苹果集合和橙子集合。通常将范畴问题转化为集合论问题可以给我们提供必要的洞察力，甚至让我们证明有价值的定理。

14.1 Hom函子

每个范畴都配备了一个到每个每个每个的规范映射族。 这些映射实际上是函子，因此它们保留了范畴的结构。 让我们构建一个这样的映射。

让我们修复一个对象 □在 □中，并选择另一个对象 □也在 □中。 hom-集合 □(□, □)是一个集合，是 □ □ 中的一个对象。 当我们改变 □，保持 □不变时， □(□, □)也会在 □ □ 中改变。 因此，我们有一个从 □到□ □ 的映射。



如果我们想强调我们将hom-集合作为第二个参数的映射，我们使用符号 □(□, −)，其中破折号作为参数的占位符。

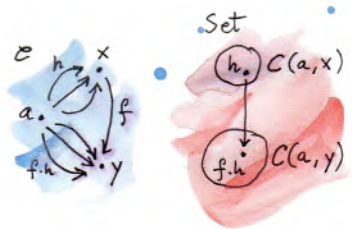
这种对象的映射很容易扩展到态射的映射。 让我们取一个态射 □在 □中的两个任意对象□和 □之间。 对象 □被映射到集合 □(□, □)，对象 □被映射到 □(□, □)，在我们刚刚定义的映射下。 如果这个映射是一个函子， □必须被映射到两个集合之间的函数： □(□, □)→ □(□, □)

让我们逐点定义这个函数，也就是分别为每个参数定义。 对于参数，我们应该选择 □(□, □)中的任意元素——让我们称之为 *h*。 如果态射的目标与 *h*的源匹配，它们就可以组合起来。

这样就得到了组合：

$$□ \circ h :: □ \to □$$

是从 □到 □的态射。 因此它是 □(□, □)的成员。



我们刚刚找到了从 □(□, □)到 □(□, □)的函数，它可以作为 □的图像。 如果没有混淆的危险，我们将把这个提升的函数写作： □(□, □)，它在态射 *h*上的作用为：

$$□(□, □)h = □ \circ h$$

由于这个构造在任何范畴中都适用，它也必须适用于Haskell类型的范畴。 在Haskell中，同态函子更为人所知为读取器函子：

```
类型 读取器 a x = a -> x

实例 Functor (读取器 a) where
                fmap f h = f . h
```

现在让我们考虑一下，如果我们固定同态集的源而不是目标，会发生什么。 换句话说，我们正在问的问题是映射 □(−, □)是否也是一个函子。 它是的，但与其说是协变的，不如说是逆变的。 这是因为相同类型的匹配

从态射的起点到终点的结果通过 □的后合成来实现，而不是像 □（□，−）那样的前合成。

我们已经在Haskell中看到了这个逆变函子。我们称之为 Op：

```
type Op a x = x -> a

instance Contravariant (Op a) where
    contramap f h = h . f
```

最后，如果我们让两个对象都变化，我们得到一个协变于第二个参数的逆变于第一个参数的 profunctor □(−, =)（为了强调这两个参数可以独立变化，我们使用双破折号作为第二个占位符）。 我们之前已经见过这个 profunctor，当我们讨论函子性时：

```
instance Profunctor (->) where
    dimap ab cd bc = cd . bc . ab
    lmap = flip (.)
    rmap = (.)
```

重要的教训是这个观察在任何范畴中都成立：对象到同态集的映射是函子性的。由于逆变性等价于从对偶范畴的映射，我们可以简洁地陈述这个事实：

□(−, =) ∷ □ □ □K □ → □ □ □

14.2 可表示函子

我们已经看到，对于范畴 □中的每个对象 □的选择，我们得到一个从 □到 □ □的函子。 这种结构保持映射到 □ □通常是

称为表示。 我们将我们中的对象和态射表示为集合和函数在我们我们我们中。

函子函子(□, −)本身有时被称为可表示的。 更一般地，任何与某个选择的更一自然同构于同态函子的函子更一被称为可表示的。 这样的函子必须是这样这样这样值的，因为这样(□, −)是。

我之前说过，我们经常将同构的集合视为相同的。更一般地，我们将范畴中同构的对象视为相同的。这是因为对象除了通过态射与其他对象（和自身）的关系之外没有其他结构。

例如，我们之前讨论过以集合为模型的幺半范畴例如例如例如。 但我们小心地选择了那些保持幺半结构的函数作为态射。 因此，如果因此因此因此中的两个对象是同构的，意味着它们之间存在一个可逆的态射，它们具有完全相同的结构。 如果我们窥视它们所基于的集合和函数，我们会发现一个幺半群的单位元被映射到另一个幺半群的单位元，并且两个元素的乘积被映射到它们映射的乘积。

相同的推理可以应用于函子。 在两个范畴之间的函子形成一个范畴，其中自然变换起到态射的作用。 因此，如果存在一个可逆的自然变换，那么两个函子是同构的，可以被视为相同的。

让我们从这个角度分析可表示函子的定义。 为了使为了成为可表示函子，我们要求：在为了中存在一个对象为了；存在一个从为了（为了，−）到为了的自然变换α；存在一个相反方向的自然变换β；它们的组合是恒等自然变换。

让我们看一下α在某个对象让我上的分量。 它是一个在它是它是它是中的函数：

$$\square\ \square :: \square(\square, \square) \to \square\ \square$$

这个变换的自然性条件告诉我们，对于任何从这个到这个的态射这个，以下图表是可交换的：

$$\square\ \square \circ \square\ \square = \square\ \square \circ \square(\square, \square)$$

在Haskell中，我们将用多态函数替换自然变换：

```
alpha :: forall x. (a -> x) -> F x
```

带有可选的 forall量词。自然性条件

```
fmap f . alpha = alpha . fmap f
```

由于参数性质的原因，自然性条件会自动满足（这是我之前提到的那些定理之一），理解为左边的fmap是由函子 □定义的，而右边的fmap是由reader函子定义的。 由于reader的fmap只是函数的预合成，我们可以更加明确。 作用于 *h*，一个属于□(□, □)的元素，自然性条件简化为：

```
fmap f (alpha h) = alpha (f . h)
```

另一个变换， beta，则相反：

```
beta :: forall x. F x -> (a -> x)
```

它必须遵守自然性条件，并且必须是 alpha的逆变换：

alpha . beta = id = beta . alpha

我们将在后面看到，从 □(□, −)到任何 □ □ -值函子的自然变换总是存在的（Yoneda引理），但不一定可逆。

让我用Haskell和列表函子以及 Int 作为示例。 这是一个完成任务的自然变换：

```
alpha :: forall x. (Int -> x) -> [x]
alpha h = map h [12]
```

我随意选择了数字12，并用它创建了一个单元素列表。然后，我可以对这个列表进行 fmap函数 h的映射，并得到一个与 h返回类型相同的列表。 （实际上，有与整数列表一样多的这样的变换。）

自然性条件等价于 map （ fmap的列表版本）的可组合性：

```
map f (map h [12]) = map (f . h) [12]
```

但是如果我们试图找到逆变换，我们将不得不从任意类型的列表 x 转换为返回 x的函数：

```
beta :: forall x. [x] -> (Int -> x)
```

你可能会想从列表中获取一个 x，例如使用 head，但是对于空列表来说这是行不通的。 请注意，在这里没有选择type a（而不是 Int ）可以起作用。 因此，列表函子是不可表示的。

还记得我们谈论过的Haskell（endo-）函子有点像容器吗？ 同样地，我们可以将可表示的

函子看作是用于存储函数调用的记忆化结果的容器（Haskell中的hom-sets成员只是函数）。 表示对象，即类型 □在 □（□，−）中，被认为是关键类型，我们可以使用它来访问函数的表格化值。我们称之为 alpha的变换被称为 tabulate，它的逆变换 beta被称为 index。

这是一个（稍微简化的）可表示的类定义：

```
class Representable f where
    type Rep f :: *
    tabulate :: (Rep f -> x) -> f x
    index                    :: f x -> Rep f -> x
```

请注意，表示类型，我们的 □，在这里被称为 Rep f，是Representable的定义的一部分。 星号只是表示 Rep f是一个类型（与类型构造函数或其他更奇特的种类相对）。

无限列表或流，不能是空的，是可表示的。

```
data Stream x = Cons x (Stream x)
```

你可以将它们视为以一个整数为参数的函数的记忆化值。 （严格来说，我应该使用非负自然数，但我不想让代码变得复杂。）为了 tabulate这样一个函数，你需要创建一个无限的值流。

当然，这只有在Haskell是惰性的情况下才可能。 这些值是按需求计算的。 你可以使用 index来访问记忆化的值：

```
实例可表示的流，其中
    类型 Rep Stream = 整数
    tabulate f = Cons (f 0) (tabulate (f . (+1)))
    index (Cons b bs) n = if n == 0 then b else index bs (n - 1)
```

有趣的是，你可以实现一个单一的记忆化方案来
覆盖具有任意返回类型的整个函数族。

对于逆变函子，可表示性的定义类似，只是我们保留 □(−, □)的第二个参数固定。 或者，等价地，我们可以考虑从 □□ □到 □ □ □的函子，因为 □□ □(□, −)与 □(−, □)相同。

表示性有一个有趣的变化。 记住，在笛卡尔闭范畴中，同态集可以内部地被视为指数对象。 同态集 □(□, □)等价于 □□，对于可表示的函子 □，我们可以写成： −□= □。

让我们取两边的对数，只是为了好玩起见：□ = log□当然
，这只是一种纯粹的形式转换，但如果你了解对数的一些性质，它会非常有帮助。 特别是，事实证明，基于乘积类型的函子可以用和类型表示，而和类型函子一般不能表示（例如：列表函子）。

最后，注意可表示函子给我们提供了两种不同的实现方式——一种是函数，一种是数据结构。
它们具有完全相同的内容——使用相同的键检索到相同的值。 这就是我所说的“相同性”的意义。
就内容而言，两个自然同构的函子是相同的。 另一方面，这两种表示通常以不同的方式实现，并且可能具有不同的性能特征。 记忆化被用作性能增强，可以大大减少运行时间。 能够生成相同基础计算的不同表示非常有价值。 令人惊讶的是，尽管范畴论并不关注性能，但它提供了大量探索具有实际价值的替代实现的机会。

14.3 挑战

1. 证明同态函子将 C中的恒等态射映射到 □ □ 中相应的恒等函数。
2. 证明 Maybe 不是可表示的。
3. Reader函子可表示吗?
4. 使用流表示，对一个函数进行记忆化，该函数将其参数平方。
5. 证明 tabulate和 index对于Stream来说确实是互逆的。（提示：使用归纳法。）6. 函子：

   Pair a = Pair a a

   是可表示的。 你能猜出代表它的类型吗?

   实现 制表和 索引。

14.4 参考文献

1. 关于可表示函子的Catsters视频[1]。

1https://www.youtube.com/watch?v=4QgjKUzyrhM

15

Yoneda引理

M范畴论中的大多数构造都是从数学的其他更具体领域的结果推广而来的。 像乘积、余积、幺半群、指数等等，在范畴论之前就已经被人们熟知了。 它们在数学的不同分支中可能以不同的名称被人们所知。 在集合论中是笛卡尔积，在序理论中是meet，在逻辑中是合取——它们都是范畴论中抽象概念的具体例子。

尤内达引理在这方面非常突出，它是关于范畴的一项广泛陈述，几乎没有其他数学分支中的先例。 有人说它最接近的类比是凯莱定理在群论中的应用（每个群同构于某个集合的置换群）。

尤内达引理的背景是任意范畴 $\mathcal{C}$ 以及从 $\mathcal{C}$ 到 $\mathbf{Set}$ 的函子 F。 我们在前一节中看到，一些 $\mathbf{Set}$ 值函子是可表示的，即与一个同态函子同构。

函子。 尤内达引理告诉我们，所有的 □ □ 值函子都可以通过自然变换从同态函子中获得，并且它明确列举了所有这样的变换。

当我谈到自然变换时，我提到自然性条件可能非常严格。 当你在一个对象上定义自然变换的一个分量时，自然性可能足够“传递”这个分量到通过一个态射与它相连的另一个对象。 源范畴和目标范畴之间的箭头越多，传递自然变换的分量就有越多的约束。源范源范源范恰好是一个非常富有箭头的范畴。

Yoneda引理告诉我们，一个自然变换在一个同态函子和任何其他函子之间的映射 □上是完全由其在一个点上的单个分量的值来确定的！自然变换的其余部分只是根据自然性条件而来。

让我们回顾一下Yoneda引理中涉及的两个函子之间的自然性条件。 第一个函子是同态函子。 它将范畴 □中的任何对象 □映射到态射集 □(□, □)— 对于范畴 □中的一个固定对象 □。 我们还看到它将从□ → □的任何态射 □映射到 □(□, □)。

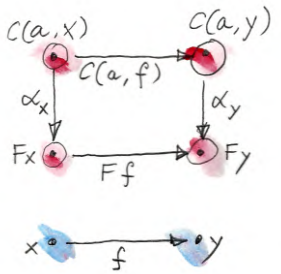
第二个函子是任意的 □ □ -值函子 □。

让我们称这两个函子之间的自然变换为 □.

因为我们在 □ □ 中操作，自然变换的分量，如 □□或 □□，只是集合之间的普通函数:

$$\square\ \square :: \square(\square, \square) \to \square\ \square$$

$$\square\ \square :: \square(\square, \square) \to \square\ \square$$



由于这些只是函数，我们可以查看它们在特定点的值。 但是在集合 $\mathcal{D}(\mathcal{C}, \mathcal{C})$中，什么是一个点？ 这里的关键观察是：集合 $\mathcal{D}(\mathcal{C}, \mathcal{C})$ 中的每个点也是从 $\mathcal{C}$到 $\mathcal{C}$的态射 h.

因此，对于 $\mathcal{C}$的自然性方形：

$$\mathcal{C}\ \mathcal{C} \circ \mathcal{D}(\mathcal{C}, \mathcal{C}\ ) = \mathcal{C}\ \mathcal{C} \circ \mathcal{C}\ \mathcal{C}$$

在作用于 h时，逐点变为：

$$\mathcal{C}\ \mathcal{C}(\mathcal{D}(\mathcal{C}, \mathcal{C})h\ ) = (\mathcal{C}\ \mathcal{C})(\mathcal{C}\ \mathcal{C}\ h)$$

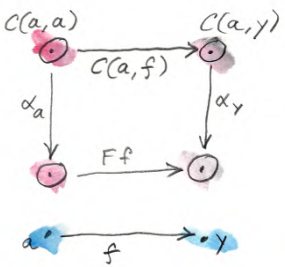
你可能还记得前一节中，对于态射 $\mathcal{C}$，对于函子 $\mathcal{D}(\mathcal{C}, -)$的作用被定义为预合成:

$$\mathcal{D}(\mathcal{C}, \mathcal{C})h = \mathcal{C} \circ h$$

这导致：

$$\mathcal{C}\ \mathcal{C}(\mathcal{C} \circ h) = (\mathcal{C}\ \mathcal{C})(\mathcal{C}\ \mathcal{C}\ h)$$

只需将其特化为$\mathcal{C} = \mathcal{C}$，就可以看出这个条件有多么强大。



在这种情况下， *h*成为从 □到 □的态射。 我们知道至少存在一个这样的态射，*h* = $\mathrm{id}_{□}$。让我们将其代入：

□ □ ≡ (□ □)(□ □d□)

请注意刚刚发生的事情：左边是 $□_{□}$对 □中任意元素 □（$□_{□}$，$_{□}$）的作用。 它完全由 $□_{□}$在 $\mathrm{id}_{□}$处的单个值确定。 我们可以选择任何这样的值，它将生成一个自然变换。 由于 $□_{□}$的值在集合□ □中，□ □中的任何点都将定义一些 □。

相反地，对于任何自然变换 □从 □(□, −)到 □，你可以在 $\mathrm{id}_{□}$处评估它，得到一个在□ □中的点。

我们刚刚证明了Yoneda引理：

> 自然变换从 □(□, −)到 □与□ □中的元素之间存在一一对应关系。

换句话说，

□ □(□□(□, −), □) ≅ □ □

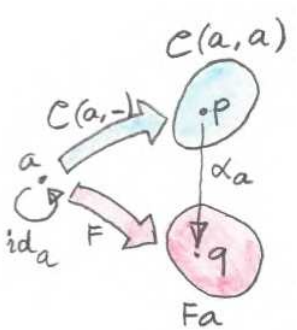
或者，如果我们使用记号 [□, □ □]表示从 □到□ □的函子范畴，那么自然变换的集合就是该范畴中的一个同态集合，我们可以写成：

[□, □ □](□□(□, −), □) ≅ □ □

我稍后会解释这个对应实际上是一个自然同构。

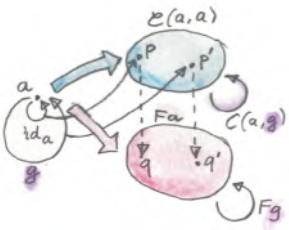
现在让我们试着对这个结果有一些直觉。 最令人惊奇的是，整个自然变换仅从一个核聚变点结晶出来：我们在 $\mathrm{id}_{\square}$处分配给它的值。 它遵循自然性条件从那个点开始扩散。 它淹没了 □在 □ □ 中的图像。
所以让我们首先考虑一下 □在 □(□, −)下的图像。

让我们从 □本身的图像开始。 在同态函子 □(□, −)下，□被映射到集合 □(□, □)。 另一方面，在函子 □下，它被映射到集合□ □。 自然变换的分量 □□是从集合 □(□, □)到集合□ □的某个函数。 让我们只关注集合 □(□, □)中的一个点，即对应于态射 $\mathrm{id}_{\square}$的点。 为了强调它只是集合中的一个点，让我们称之为 □。 分量 □□应该将 □映射到□ □中的某个点 □。 我将向你展示，任何 □的选择都会导致唯一的自然变换。



第一个论断是选择一个点 □唯一确定了函数的其余部分 □□。 事实上，让我们选择任何其他点 □′在 □(□, □)中，对应于某个从 □到 □的态射 □。 这就是Yoneda引理的魔力所在： □可以被视为一个点 □′在

集合 □(□, □)中。 同时，它选择了两个集合之间的函数。
事实上，在同态函子下，态射 □被映射为一个函数□(□, □)；在 □下，它被映射为□ □。



现在让我们考虑 □(□, □)对我们的原始 □的作用，正如你记得的那样，它对应于 $\mathrm{id}_{□}$。 它被定义为前合成，□ ∘ $\mathrm{id}_{□}$，它等于 □，对应于我们的点 □′。 因此，态射□被映射到一个函数，当作用于 □时产生 □′，即□。 我们已经完全回到了起点！

现在考虑□ □对 □的作用。 它是一些 □′，是□ □中的一个点。 为了完成自然性正方形， □′必须在 □□下映射到 □′。 我们选择了一个任意的 □′（一个任意的 □）并推导出它在□□下的映射。 函数 □□因此完全确定。

第二个论断是，对于与 □相连的任何对象□在 □中， □ □是唯一确定的。 推理是类似的，只是现在我们有两个更多的集合， □(□, □)和□ □，并且在同态函子下，态射 □从 □到□被映射为：

$$□(□, □) ∷ □(□, □) → □(□, □)$$

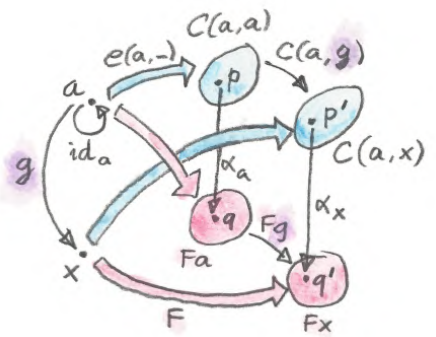
并且在 □下：

$$□ □ ∷ □ □ → □ □$$

同样地，我们对于我们的 □的 □(□, □)的作用是通过前合成给出的：□ ∘ id□，这对应于 □(□, □)中的一个点 □′。 自然性确定了 □□对 □′的值为：

$$□′ = (□ □)□$$

由于 □′是任意的，整个函数 □□因此被确定。



如果 □中的对象与 □没有连接，会怎么样？ 它们都被映射到一个单独的集合下 □(□, −) — 空集。回想一下，空集是集合范畴中的初始对象。 这意味着从该集合到任何其他集合都存在唯一的函数。我们称之为荒谬函数。 所以在这里，我们对于自然变换的组成部分没有选择：它只能是荒谬。

理解Yoneda引理的一种方式是意识到 □ □值函子之间的自然变换只是一组函数，而函数一般是有损的。 函数可能会丢失信息，而且可能只覆盖其值域的一部分。 唯一不具有损失的函数是可逆的函数 — 同构。 由此可见，最佳的结构保持 □ □值

函子是可表示的。 它们要么是同态函子，要么是与同态函子自然同构的函子。 任何其他函子 □都是通过一种有损转换从同态函子获得的。 这种转换不仅可能丢失信息，还可能只覆盖函子 □在 □ □ □中的一小部分。

15.1 Haskell中的Yoneda

我们在Haskell中已经遇到了同态函子，以读取器函子的方式出现：

```
类型 读取器 a x = a -> x
```

读取器通过预合成来映射态射（这里是函数）：

```
实例 Functor (读取器 a) where
    fmap f h = f . h
```

Yoneda引理告诉我们，读取器函子可以自然地映射到任何其他函子。

自然变换是一种多态函数。 因此，给定一个函子 F，我们可以从读取器函子向其进行映射：

```
alpha :: forall x . (a -> x) -> F x
```

像往常一样， forall是可选的，但我喜欢明确地写出来以强调自然变换的参数多态性。

Yoneda引理告诉我们，这些自然变换与 F a的元素一一对应：

```
对于所有的 x，(a -> x) -> F x  ≅  F a
```

这个等式的右边是我们通常认为的一个数据结构。 还记得将函子解释为广义容器吗？F a是一个包含 a的容器。 但是左边是一个多态函数，它以一个函数作为参数。 Yoneda引理告诉我们这两种表示是等价的-它们包含相同的信息。

另一种说法是：给我一个类型为的多态函数：

```
alpha :: forall x . (a -> x) -> F x
```

我将生成一个包含 a的容器。 这个技巧是我们在证明Yoneda引理时使用的：我们用 id调用这个函数来得到一个元素
F a:

```
 alpha id :: F a
```

反过来也成立：给定一个类型为 F a的值：

```
fa :: F a
```

可以定义一个多态函数：

```
alpha h = fmap h fa
```

具有正确类型的。 您可以轻松地在这两种表示之间来回切换。

拥有多种表示的优点是，一种表示可能比另一种更容易组合，或者在某些应用中更高效。

这个原则最简单的例证是编译器构建中经常使用的代码转换：延续传递样式或cps。它是Yoneda引理应用于恒等函子的最简单应用。 将 F替换为identity产生：

forall r . (a -> r) -> r ≅ a

这个公式的解释是，任何类型 a都可以被一个接受 a的“处理程序”的函数替换。 处理程序是一个接受a并执行剩余计算（续传）的函数。 （类型 r通常封装了某种状态码。）这种编程风格在UI、异步系统和并发编程中非常常见。 cps的缺点是它涉及控制反转。 代码被分为生产者和消费者（处理程序），并且不容易组合。 任何做过一定量的非平凡网络编程的人都熟悉与有状态处理程序交互的代码混乱的噩梦。 正如我们稍后将看到的，巧妙地使用函子和单子可以恢复cps的一些组合属性。

15.2 Co-Yoneda

通常情况下，通过反转箭头的方向，我们可以得到一个额外的构造。 Yoneda引理可以应用于对偶范畴 $\mathbf{C}^{op}$，给我们提供了一个逆变函子之间的映射。

或者，我们可以通过固定我们的同态函子的目标对象而不是源对象来推导出共Yoneda引理。 我们可以从范畴 $\mathbf{C}$到 $\mathbf{Set}$ 得到逆变同态函子$\mathbf{C}(-, a)$。 Yoneda引理的逆变版本建立了从该函子到任何其他逆变函子 F的自然变换与集合$F\,a$的元素之间的一一对应关系。

$$[\mathbf{C}^{op}, \mathbf{Set}](\mathbf{C}(-, a), F) \cong F\,a$$

这是共Yoneda引理的Haskell版本：

forall x . (x -> a) -> F x ≅ F a

注意，在一些文献中，被称为逆变版本的是Yoneda引理。

15.3 挑战

1. 证明在Haskell中形成Yoneda同构的两个函数 phi和 psi是彼此的逆函数。

   ```
   phi :: (forall x . (a -> x) -> F x) -> F a
   phi alpha = alpha id

   psi :: F a -> (forall x . (a -> x) -> F x)
   psi fa h = fmap h fa
   ```

2. 离散范畴是指除了恒等态射之外，只有对象而没有态射的范畴。 对于从这样一个范畴到函子的Yoneda引理如何工作?

3. 一个单位列表 [()]除了长度之外没有其他信息。
   因此，作为一种数据类型，它可以被视为整数的编码。
   使用列表函子的Yoneda引理构造这个数据类型的另一种表示。

15.4 参考文献

1. Catsters[1]视频。

1https://www.youtube.com/watch?v=TLMxHB19khE

16

Yoneda嵌入

我们之前已经看到，当我们固定一个对象 □在范畴 □中时，映射 □(□, −)是从 □到 □ □ 的（协变）函子。

$$□ \to □(□, □)$$

（目标范畴是 □ □ ，因为同态集合 □(□, □)是一个集合。）我们称这个映射为同态函子 —— 我们之前也定义了它在态射上的作用。

现在让我们在这个映射中改变 □。 我们得到一个将同态函子 □(□, −)分配给任何 □的新映射。

$$□ \to □(□, −)$$

这是从范畴 □到函子的对象的映射，函子是函子范畴中的对象（请参阅关于函子范畴的部分自然变换）。让我们使用符号 [□, □ □]□ 表示函子

从范畴 $\mathcal{C}$到 $\mathcal{C}^{\mathcal{C}^{op}}$的范畴。 你可能还记得同态函子是典型的可表示函子。

每当我们有两个范畴之间的对象映射时，自然而然地会问这样的映射是否也是一个函子。 换句话说，我们是否可以将一个态射提升到另一个范畴中的态射。 在范畴 $\mathcal{C}$中，态射只是 $\mathcal{C}$中的一个元素（a, b），但在函子范畴 $[\mathcal{C}, \mathcal{C}^{op}]$中，态射是一个自然变换。 因此，我们正在寻找从态射到自然变换的映射。

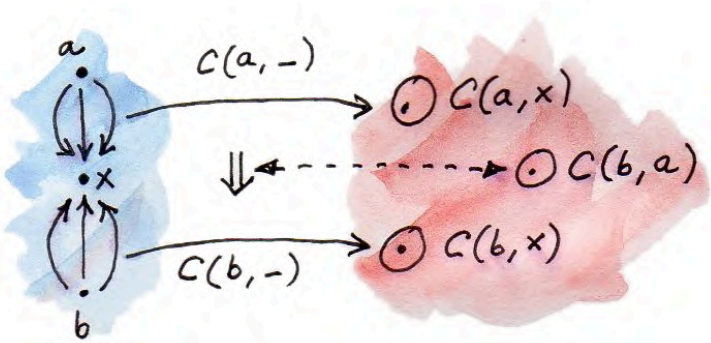
让我们看看是否可以找到与一个态射 $f :: a \to b$相对应的自然变换。 首先，让我们看看 a和 b被映射到哪里。它们被映射到两个函子： $\mathcal{C}(a, -)$和 $\mathcal{C}(b, -)$。 我们需要一个在这两个函子之间的自然变换。

这里的诀窍是：我们使用Yoneda引理：

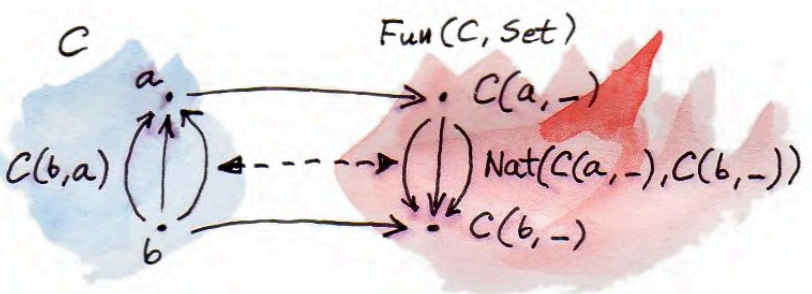
$$[\mathcal{C}, \mathcal{C}^{op}](\mathcal{C}(a, -), F) \cong F\, a$$

并将通用的 F替换为同态函子 $\mathcal{C}(b, -)$。我们得到：

$$[\mathcal{C}, \mathcal{C}^{op}](\mathcal{C}(a, -), \mathcal{C}(b, -)) \cong \mathcal{C}(b, a)$$



这正是我们正在寻找的两个同态函子之间的自然变换，但有一个小小的变化：我们有一个自然变换和一个态射（属于 $\mathcal{D}(F, G)$）之间的映射，它以“错误”的方向进行。但没关系；这只意味着我们正在查看的函子是反变的。



实际上，我们得到的远远超出了我们的预期。从 $\mathcal{D}$ 到 $[\mathcal{C}, \mathcal{D}]$ 的映射不仅是一个逆变函子 - 它是一个完全忠实的函子。 完全性和忠实性是描述函子如何映射同态集的属性。

一个忠实的函子在同态集上是单射的，这意味着它将不同的态射映射到不同的态射。 换句话说，它不会将它们合并在一起。

一个全函子在同态集上是满射的，这意味着它将一个同态集映射到另一个同态集，完全覆盖后者。

一个完全忠实的函子 $\mathcal{D}$ 是同态集上的双射 - 两个集合的一对一匹配。 对于源范畴 $\mathcal{C}$ 中的每一对对象 $\mathcal{C}$ 和 $\mathcal{C}$，在 $\mathcal{D}(F, G)$ 和目标范畴 $\mathcal{D}(\mathcal{D}\ F, \mathcal{D}\ G)$ 之间存在一个双射，其中 F 是 $\mathcal{D}$ 的目标范畴（在我们的情况下，是函子范畴 $[\mathcal{C}, \mathcal{D}\ ]$）。 请注意，这并不意味着 $\mathcal{D}$ 是一个双射。

关于对象的。 在 $\mathcal{C}$中可能存在一些不在 $\mathcal{C}$的图像中的对象，对于这些对象我们无法对同态集合做出任何断言。

16.1 嵌入

我们刚刚描述的（反变）函子，将 $\mathcal{C}$中的对象映射到 $[\mathcal{C}, \mathbf{Set}]$中的函子：

$$\mathcal{C} \to \mathbf{Set}(\mathcal{C}, -)$$

定义了Yoneda嵌入。 它将一个范畴 $\mathcal{C}$（严格来说，是 $\mathcal{C}$的逆范畴，因为是反变的）嵌入到函子范畴$[\mathcal{C}, \mathbf{Set}]$中。 它不仅将 $\mathcal{C}$中的对象映射到函子，还忠实地保留它们之间的所有连接。

这是一个非常有用的结果，因为数学家对函子范畴，特别是以 $\mathbf{Set}$为目标的函子，有很多了解。 通过将一个任意范畴 $\mathcal{C}$嵌入到函子范畴中，我们可以获得很多关于它的洞察力。

当然，Yoneda嵌入有一个对偶版本，有时被称为共Yoneda嵌入。 观察我们可以通过固定每个同态集的目标对象（而不是源对象）来开始， $\mathbf{Set}(-, \mathcal{C})$。 这将给我们一个逆变同态函子。 从 $\mathcal{C}$到 $\mathbf{Set}$的逆变函子是我们熟悉的预层（参见，例如，极限和余极限）。共Yoneda嵌入定义了一个范畴 $\mathcal{C}$在预层范畴中的嵌入。 它在态射上的作用如下：

$$[\mathcal{C}, \mathbf{Set}](\mathbf{Set}(-, \mathcal{C}), \mathbf{Set}(-, \mathcal{D})) \cong \mathcal{C}(\mathcal{D}, \mathcal{C})$$

再次，数学家对预层范畴非常了解，所以能够将任意范畴嵌入其中是一个巨大的优势。

16.2 在Haskell中的应用

在Haskell中，Yoneda嵌入可以表示为一方面是读取器函子之间的自然变换的同构，另一方面是函数（在相反方向上）之间的同构：

```
forall x. (a -> x) -> (b -> x) ≅ b -> a
```

(记住，读者函子等价于 ((->) a)。)这个等式的左边是一个多态函数，给定一个从 a到 x的函数和一个类型为 b的值，可以产生一个类型为 x的值(我在取消函数 b -> x周围的括号 —— uncurrying)。 只有当我们的函数知道如何将 b转换为 a时，才能对所有的 x进行这样的转换。 它必须秘密地访问一个函数 b -> a。

给定这样一个转换器 btoa，可以定义左边，称之为fromY，如下：

```
fromY :: (a -> x) -> b -> x
 fromY f b = f (btoa b)
```

相反地，给定一个函数 fromY，我们可以通过调用恒等函数来恢复转换器：

```
fromY id :: b -> a
```

这建立了函数类型 fromY和之间的双射
btoa.

另一种看待这个同构的方式是它是一个来自b到a的函数的cps编码。 参数 a -> x是一个连续的
续函数（处理程序）。 结果是一个从 b到 x的函数，当

用类型 b的值调用时，将执行预组合的续函数
与被编码的函数。

Yoneda嵌入还解释了Haskell中一些替代的数据结构表示方式。特别是，它提供了一个非常有用的表示[1]of来自Control.Lens库的镜头。

16.3 预序例子

这个例子是由Robert Harper提出的。它是将Yoneda嵌入应用于由预序定义的范畴。 预序是一个具有元素之间的排序关系的集合，传统上写作 □（小于或等于）。 预序中的“pre”表示我们只要求关系是传递的和自反的，但不一定是反对称的（因此可能存在循环）。

具有预序关系的集合产生一个范畴。 对象是这个集合的元素。从对象 □到 □的态射存在的条件是，如果无法比较对象或者□ □ □不成立，则不存在；如果□ □ □成立，并且它从 □指向 □，则存在。从一个对象到另一个对象的态射永远不会超过一个。 因此，在这样的范畴中，任何同态集合要么是空集，要么是一个元素的集合。这样的范畴被称为 thin。

很容易让自己相信这个构造确实是一个范畴：箭头是可组合的，因为如果□ □ □且□ □ □，则□ □ □；而且组合是结合的。 我们还有恒等箭头，因为每个元素都与自身（基础关系的自反性）相等或小于。

1https://bartoszmilewski.com/2015/07/13/from-lenses-to-yoneda-embedding/

我们现在可以将共Yoneda嵌入应用于预序范畴。
特别是，我们对其在态射上的作用感兴趣：

[□, □ □](□(−, □), □(−, □)) ≅ □(□, □)

当且仅当□ □ □时，右手边的同态集是非空的，此时它是一个单元素集。因此，如果□ □ □，则左边存在一个单一的自然变换。 否则就没有自然变换。

那么，在预序中，同态函子之间的自然变换是什么？ 它应该是一族函数，将集合 □(−, □)和 □(−, □)之间进行映射。
在预序中，这些集合中的每一个都可以是空集或单元素集。 让我们看看我们可以使用的函数类型。有一个从空集到自身的函
数（在空集上的恒等函数），一个从空集到单元素集的函数（它不做任何操作，因为它只需要对空集的元素进行定义，而空集中没有元素），以及一个从单元素集到自身的函数（在一个单元素集上的恒等函数）。 唯一不允许的组合是从单元素集到空集的映射（当作用于单个元素时，这样的函数的值将是什么？）。因此，我们的自然变换永远不会将一个单元素同态集连接到一个空的同态集。 换句话说，如果□ □ □（单元素同态集□(□, □)），那
么 □(□, □)不能是空的。 一个非空的 □(□, □)意味着□小于或等于 □。 因此，问题中自然变换的存在要求对于每个 □，如果□ □ □，则□ □ □。

对于所有□, □ □ □ ⇒ □ □ □

另一方面，共Yoneda告诉我们，这个自然变换的存在等价于 □(□, □)非空，或者□ □ □.

综上所述，我们得到：

$$\square\ \square\ \square \text{ 当且仅当对于所有 } \square, \square\ \square\ \square \Rightarrow \square\ \square\ \square$$

我们也可以直接得到这个结果。 直觉告诉我们，如果$\square\ \square\ \square$那么所有低于 $\square$的元素也必须低于 $\square$. 反过来说，当你在右边将 $\square$替换为 $\square$时，可以得出$\square\ \square\square$. 但你必须承认，通过Yoneda嵌入得到这个结果更加令人兴奋。

16.4 自然性

Yoneda引理建立了自然变换集合与一个对象之间的同构关系. 自然变换是函子范畴中的态射 $[\square, \square\ \square]$$\square$任意两个函子之间的自然变换集合是该范畴中的同态集合. Yoneda引理是以下同构关系:

$$[\square, \square\ \square]\left(\square\square(\square, -), \square\right) \cong \square\ \square$$

这个同构关系在 $\square$和 $\square$两者都是自然的. 换句话说，它在 $(\square, \square)$中是自然的，其中$(\square, \square)$是从乘积范畴 $[\square, \square\ \square]\square\times\ \square$中取出的一对. 请注意，我们现在将 $\square$视为函子范畴中的一个对象.

让我们思考一下这意味着什么. 自然同构是两个函子之间的可逆自然变换. 事实上，我们同构的右侧是一个函子. 它是从$[\square, \square\ \square]\square\ \times\ \square$到 $\square\ \square$的一个函子. 它对于一对 $(\square, \square)$的作用是一个集合 —— 对函子 $\square$在对象 $\square$处的求值结果. 这被称为求值函子.

左手边也是一个函子，它将 $(\square, \square)$映射到一组自然变换 $[\square, \square\ \square]\left(\square(\square, -), \square\right)$。

为了证明这些确实是函子，我们还应该定义它们对态射的作用。 但是对于一对 (□ , □)和(□, □)，什么是态射呢？ 它是一对态射，(Φ, □)；第一个是函子之间的态射——自然变换——第二个是 □ 中的常规态射。

评估函子将这对 (Φ, □)映射到两个集合之间的函数，□ □和 □ □。 我们可以轻松地从 Φ在 □处的分量（将□ □映射到 □ □）和由 □提升的态射 □构造这样的函数：

$$(□ □) ◦ Φ□$$

注意，由于 Φ的自然性，这与以下相同：

$$Φ□ ◦ (□ □)$$

我不打算证明整个同构的自然性 - 在你确定了函子是什么之后，证明就相当机械化了。这是因为我们的同构是由函子和自然变换构建的。 它根本不可能出错。

16.5 挑战

1. 用Haskell表达共Yoneda嵌入。
2. 证明我们在fromY和btoa之间建立的双射是一个同构（两个映射是彼此的逆映射）。
3. 计算一个幺半群的Yoneda嵌入。 哪个函子对应于幺半群的单个对象？ 哪些自然变换对应于幺半群的态射？

4. 协变Yoneda嵌入在预序中的应用是什么？ （由Gershom Bazerman提出的问题。）

5. Yoneda嵌入可以用于嵌入任意函子cat-egory [□, □]到函子范畴 [[□, □], □ □]。 弄清楚它在态射上的工作原理（在这种情况下是自然变换）。

第三部分

17

一切都关于态射

如果我还没有说服你范畴论完全是关于态射的话，那我就没有做好我的工作。 由于下一个主题是伴随，伴随是通过同态集合的同构来定义的，所以回顾一下同态集合的基本概念是有意义的。 此外，你会发现伴随提供了一个更通用的语言来描述我们之前研究过的许多构造，所以回顾一下它们可能会有所帮助。

17.1 函子

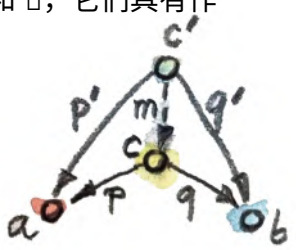
首先，你应该真正把函子看作是态射的映射——这是Haskell定义的Functor类型类所强调的观点，它围绕着 fmap展开。 当然，函子也映射对象——态射的端点——否则我们就无法讨论保持组合性。 对象告诉我们哪些对

态射是可组合的。 一个态射的目标必须等于另一个态射的源 - 如果它们要被组合起来的话。 所以，如果我们希望态射的组合被映射到提升态射的组合，它们的端点的映射基本上已经确定了。

17.2 交换图

很多态射的性质都是用交换图表来表达的。 如果一个特定的态射可以用多种方式描述为其他态射的组合，那么我们就有了一个交换图表。

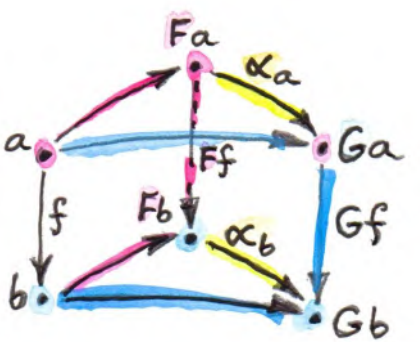
特别是，交换图表构成了几乎所有的泛性构造的基础（除了初始对象和终结对象）。 我们在产品、余积、各种其他（共）极限、指数对象、自由幺半群等的定义中都见过这一点。产品是一个简单的泛性构造的例子。 我们选择两个对象 □和 □，并查看是
否存在一个对象 □，以及一对态射 □和 □，它们具有作为它们的乘积的泛性属性。



一个乘积是极限的特殊情况。
极限是通过锥体来定义的。
一般的锥体是由交换图构建的。 这些图的可交换性可以用函子映射的适当自然性条件来替代。 这样，可交换性就被降低到自然变换的高级语言中的汇编语言的角色。

17.3 自然变换

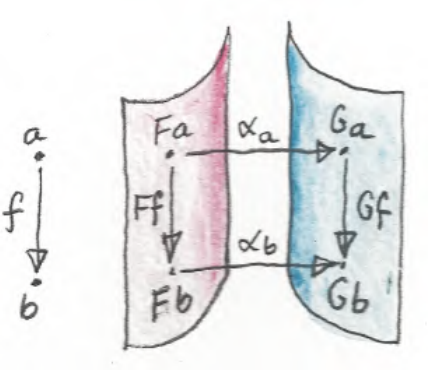
一般来说，自然变换在需要从态射到交换方块的映射时非常方便。 自然性方块的两个相对的边是某个态射 □在两个函子 □和 □下的映射。 另外两个边是自然变换的分量（也是态射）。



自然性意味着当你移动到“相邻”的分量时（相邻是指通过一个态射相连），你不会违背范畴或函子的结构。 无论你是先使用自然变换的一个分量来连接对象之间的差距，然后再使用函子跳到它的相邻分量；还是反过来，都没有关系。 这两个方向是正交的。自然变换使你左右移动，而函子使你上下或前后移动，可以这么说。 你可以将函子的 image想象成目标范畴中的一张纸。

一个自然变换将对应于F的一个这样的表格映射到对应于G的另一个表格。

我们在Haskell中看到了这种正交性的例子。在那里，函子的作用修改了容器的内容而不改变其形状，而自然变换则将未修改的内容重新打包到不同的容器中。 这些操作的顺序无关紧要。



我们在极限的定义中看到了锥被自然变换取代的情况。 自然性确保了每个锥的边都是可交换的。 然而，极限是根据锥之间的映射来定义的。 这些映射还必须满足可交换性条件。 （例如，乘积的定义中的三角形必须可交换。）这些条件也可以用自然性来取代。 你可能还记得，通用锥或极限被定义为（反变）同态函子之间的自然变换：

$$\square :: \square \to \square(\square, \mathrm{Lim}\mathrm{Li})$$

以及将 C中的对象映射到锥形图的（也是逆变的）函子，它们本身是自然变换：

$$\square :: \square \to \square \ \square (\Delta\square, \square)$$

在这里， $\Delta_{\square}$是常量函子， $\square$是定义了 $\square$中图表的函子。 函子 $\square$和 $\square$在 $\square$中的态射有明确定义的操作。 恰好这个特定的自然变换在 $\square$和 $\square$之间是一个同构。

17.4 自然同构

自然同构——即每个分量都可逆的自然变换——是范畴论中“两个事物相同”的方式。这样的变换的一个分量必须是对象之间的同构——具有逆的态射。

如果你将函子的图像视为图层，那么自然同构就是这些图层之间的一对一可逆映射。

17.5 Hom-集合

但是什么是态射？ 它们比对象具有更多的结构：与对象不同，态射有两个端点。 但是如果你固定源对象和目标对象，两者之间的态射形成一个无聊的集合（至少对于局部小范畴来说）。 我们可以给这个集合的元素起名字，比如 □或 □，以区分彼此 - 但是它们之间的真正区别是什么呢？

给定同态集合中的态射之间的本质区别在于它们与其他态射的组合方式（来自相邻的同态集合）。 如果存在一个态射 *h*，它与 □的组合（无论是前合成还是后合成）与 □的组合不同，例如：

$$h \circ \square \neq h \circ \square$$

那么我们可以直接“观察”出 □和 □之间的差异。 但是即使差异不是直接可观察的，我们可以使用函子来聚焦于同态集合。 一个函子 □可以将这两个态射映射为不同的态射：

$$\square\ \square \neq \square\ \square$$

在一个更丰富的范畴中，相邻的同态集提供了更多的分辨率，例如，

$$h' \circ \square\,\square \neq h' \circ \square\,\square$$

其中 h'不在 □的像中。

17.6 Hom-集合同构

许多范畴构造依赖于同态集之间的同构。 但是由于同态集只是集合，它们之间的简单同构并不能告诉你太多。 对于有限集合，同构只表示它们具有相同数量的元素。 如果集合是无限的，它们的基数必须相同。 但是同态集的任何有意义的同构都必须考虑到复合。 而复合涉及到不止一个同态集。 我们需要定义跨越整个同态集合的同构，并且需要施加一些与复合相互作用的兼容性条件。 而一个自然同构正好符合要求。

但是同态集的自然同构是什么？ 自然性是函子之间映射的一个属性，而不是集合。 所以我们实际上在谈论的是同态集值函子之间的自然同构。 这些函子不仅仅是集合值函子。 它们对于态射的作用是由适当的同态函子引起的。 通过同态函子，态射被规范地映射，可以使用预合成或后合成（取决于函子的协变性）。

Yoneda嵌入是这种同构的一个例子。 它将它将中的同态集映射到函子范畴中的同态集；而且它是自然的。Yoneda嵌入中的一个函子是Yo中的同态函子，另一个将对象映射到同态集之间的自然变换集合。

极限的定义也是同态集之间的自然同构（再次在函子范畴中）：

$$\square(\square, \mathrm{Lim}\mathrm{Li}) \simeq \square\ \square(\Delta\square, \square)$$

事实证明，我们对指数对象或自由幺半群的构造也可以被重写为同态集之间的自然同构。

这不是巧合 - 我们将在下面看到，这些只是自然同构的不同例子，它们被定义为同态集之间的自然同构。

17.7 Hom-集合的非对称性

还有一个观察结果将帮助我们理解伴随
一般来说，同态集不是对称的 同态集 $\square(\square, \square)$通常与同态集 $\square(\square, \square)$非常不同 这种不对称性的最终证明是将偏序视为范畴 在偏序中，从 □到 □的同态存在当且仅当 □小于或等于 □

如果 □和 □不同，则不能存在从 □到 □的同态 因此，如果同态集 $\square(\square, \square)$非空，这在这种情况下意味着它是一个单元素集合，那么同态集 $\square(\square, \square)$必须为空，除非$\square = \square$
这个范畴中的箭头有一个明确的方向流动。

一个基于不一定是反对称的关系的预序，也是“大多数”是有方向的，除了偶尔的循环。
方便起见，我们可以将任意范畴看作是预序的一般化。

预序是一个薄范畴 - 所有的同态集要么是单元素集，要么是空集。 我们可以将一般范畴可视化为一个“厚”预序。

17.8 挑战

1. 考虑一些自然性条件的退化情况，并绘制相应的图表。 例如，如果函子 □或 □将对象 □和 □（即□ ∶ □ → □的端点）映射到同一个对象，例如□ □ = □ □或□ □ = □ □，会发生什么？ （注意，这样你会得到一个锥体或余锥体。）然后考虑□ □ = □ □或□ □ = □ □的情况。 最后，如果你从一个自环的态射开始 - □ ∶ □ → □，会发生什么？

18

伴随

在数学中，我们有各种方式来表达一件事物类似于另一件事物。 最严格的是相等。 如果没有办法区分一个事物和另一个事物，那么它们是相等的。 一个可以替代另一个在任何可想象的情境中。 例如，你有没有注意到我们在谈论交换图时使用了态射的相等性？ 这是因为态射形成了一个集合（同态集合），而集合元素可以进行相等比较。

但是相等通常太强了。 有很多例子表明，事物在所有意图和目的上都是相同的，但实际上并不相等。例如，对偶类型 (Bool, Char) 并不严格等于(Char, Bool)，但我们理解它们包含相同的信息。 这个概念最好通过两种类型之间的同构来捕捉——一种可逆的态射。 由于它是一个态射，它保留了结构；而“iso”意味着它是一个往返旅程的一部分，最终回到原点。

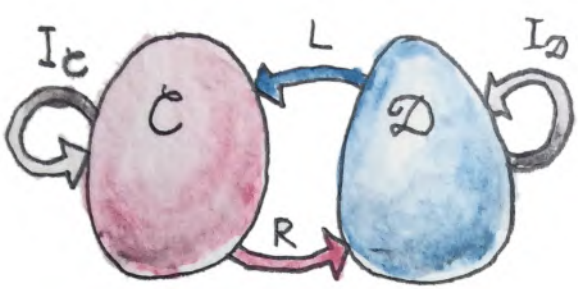
无论你从哪一边开始，你都会停在同一个位置。 对于成对的情况，这个同构被称为交换：

```
交换 :: (a,b) -> (b,a)
交换 (a,b) = (b,a)
```

交换恰好是它自己的逆。

18.1 伴随和单位/余单位对

当我们谈论范畴同构时，我们用范畴之间的映射，也就是函子来表达。 如果存在一个从范畴如果到范畴如果的函子如果（“右”），它是可逆的，我们希望能够说两个范畴如果和如果是同构的。 换句话说，存在另一个从范畴换句返回到范畴换句的函子换句（“左”），当与函子换句组合时，等于恒等函子换句。 有两种可能的组合，有两 ◦□和□◦□；还有两个可能的恒等函子：一个在范畴□中，另一个在范畴□中。



但是这里有个棘手的部分：两个函子相等是什么意思？ 我们对这个相等性是什么意思：

□ ◦ □ = □ □

或者这个：

□ ◦ □ = □ □

以对象的相等性来定义函子的相等性是合理的。 当作用于相等的对象时，两个函子应该产生相等的对象。 但是一般来说，在任意范畴中我们没有对象相等性的概念。 这不是定义的一部分。 （深入探讨“相等性到底是什么”的话题，我们将进入同伦类型论的领域。）

你可能会争辩说函子是范畴范畴中的态射，所以它们应该是可比较的。 事实上，只要我们讨论的是小范畴，其中对象构成一个集合，我们确实可以使用集合元素的相等性来比较对象的相等性。

但是，请记住，但是但是但是实际上是一个 □-范畴。 在 □-范畴中，同态集具有额外的结构——存在作用于 1-同态之间的 2-同态。 在 □ □ 中， 1-同态是函子， 2-同态是自然变换。 因此，当谈论函子时，将自然同构视为相等性的替代品更为自然（无法避免这个双关语！）。

所以，不是范畴的同构，而是更一般的等价概念更有意义。 如果我们能够找到两个来回之间的两个函子，它们的组合（无论哪种方式）都与恒等函子自然同构。 换句话说，存在一种双向自然变换-

在□ ◦ □和恒等函子 □□之间，以及
在□ ◦ □和恒等函子 □□之间，存在另一种自然变换。

伴随性甚至比等价性更弱，因为它不要求两个函子的组合与恒等函子同构。 相反，它规定存在一种从恒等函子 □□到□ ◦ □的单向自然变换，以及从□ ◦ □到 □□的另一种自然变换。 这里是这两个自然变换的签名：

$$\square :: \square \ \square \to \square \circ \square$$
$$\square :: \square \circ \square \to \square \ \square$$

□被称为单位，而 □被称为伴随的对偶。

注意这两个定义之间的不对称性。 一般来说，我们没有剩下的两个映射：

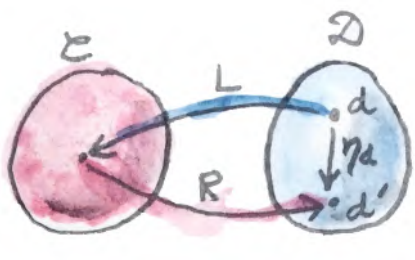
$$\square \circ \square \to \square \ \square \qquad \text{不一定}$$
$$\square \ \square \to \square \circ \square \qquad \text{不一定}$$

由于这种不对称性，函子 □被称为左伴随到函子 □，而函子 □是函子 □的右伴随。 （当然，左和右只有在你以一种特定的方式绘制你的图表时才有意义。）

伴随的紧凑表示法是：

$$\square \dashv \square$$

为了更好地理解伴随，让我们更详细地分析单位和对偶。

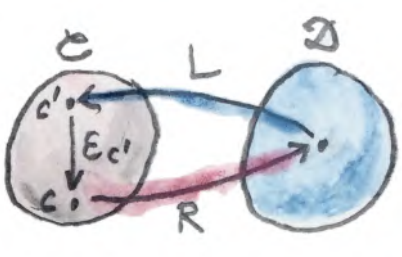


让我们从单位开始。 它是一个自然变换，所以它是一族态射。 给定一个对象 𝑐在 𝒞中， 𝜂的分量是一个态射，它在𝑐 𝒞和 (𝐹 ∘ 𝐺) 𝑐之间；在图中被称为𝜂′：

$$\eta_c :: c \to (F \circ G)\, c$$

注意到组合𝐹 ∘ 𝐺是 𝒞中的一个自函子。

这个方程告诉我们，我们可以选择 𝑐作为我们的起始点，并使用往返函子𝐹 ∘ 𝐺选择我们的目标对象 𝑐′。 然后我们发射一个箭头 - 形态 $f_{c'}$ - 到我们的目标。



同样，余单位ε的分量可以描述为：

$$\varepsilon_c :: (F \circ G)\, c \to c$$

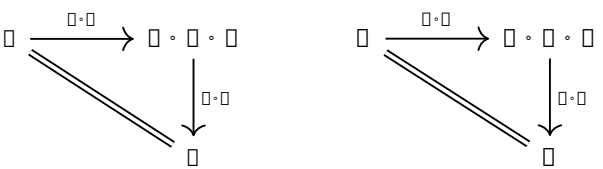
其中 □′是 (□ ∘ □)□。 它告诉我们，我们可以选择 □作为我们的目标，并使用往返函子□ ∘ □选择源 □′。 然后，我们发射箭头 - 形态 □□′ - 从源到目标。

另一种看待单位和余单位的方式是，单位让我们能够在任何可以插入一个恒等函子的地方引入另一 ∘ □的组合；而余单位则让我们能够用恒等函子替换□ ∘ □的组合。 这导致了一些“显而易见”的一致性条件，确保引入后跟消除不会改变任何东西：

$$□ = □ \circ □ \; □ \to □ \circ □ \circ □ \to □ \; □ \circ □ = □$$

$$□ = □ \; □ \circ □ \to □ \circ □ \circ □ \to □ \circ □ \; □ = □$$

这些被称为三角恒等式，因为它们使以下图表交换：



这些是函子范畴中的图表：箭头是自然变换，它们的组合是自然变换的水平组合。 在组件中，这些恒等式变为：

$$□ \; □ \; □ \; □ \; □ = \mathrm{id}\mathrm{id}\mathrm{id}$$

$$□ \; □ \; □ \; □ \; □ = \mathrm{id}\mathrm{id}\mathrm{id}$$

我们经常在Haskell中以不同的名称看到单位和余单位。 单位被称为return（或者在Applicative的定义中称为pure）：

return :: d -> m d

并计数为 提取 :

提取 :: w c -> c

在这里， m是对应于□。□的（自）函子，而 w是对应于□。□的（自）函子。 正如我们稍后将看到的，它们是单子和余单子的定义的一部分。

如果你将一个自函子看作一个容器，那么单元（或返回）是一个多态函数，它在任意类型的值周围创建一个默认的盒子。 余单元（或提取）则相反：它从容器中检索或生成一个单个值。

我们稍后将看到，每对伴随函子都定义了一个单子和一个余单子。 相反，每个单子或余单子都可以分解为一对伴随函子 - 尽管这种分解不是唯一的。

在Haskell中，我们经常使用单子，但很少将它们分解为对偶函子对，主要是因为这些函子通常会使我们离开 Hask。

然而，我们可以在Haskell中定义函子的对偶。这是从Data.Functor.Adjunction中摘取的定义的一部分：

```
class (Functor f, Representable u) =>
      Adjunction f u | f -> u, u -> f where
  unit :: a -> u (f a)
 counit :: f (u a) -> a
```

这个定义需要一些解释。 首先，它描述了一个多参数类型类 - 这两个参数是 f和 u。 它建立了一个称为 Adjunction的关系，连接了这两个类型构造器。

垂直线后面的附加条件指定了函数依赖关系。 例如， f -> u表示 u由 f确定（这里是在类型构造器上的函数）。相反地，u -> f表示如果我们知道u，那么f是唯一确定的

f和 u之间的关系是一个函数（这里是在类型构造器上的函数）。反过来， u -> f表示如果我们知道 u，那么 f是唯一确定的。

我将在稍后解释为什么在Haskell中，我们可以强加条件，即右伴随 u是一个可表示的函子。

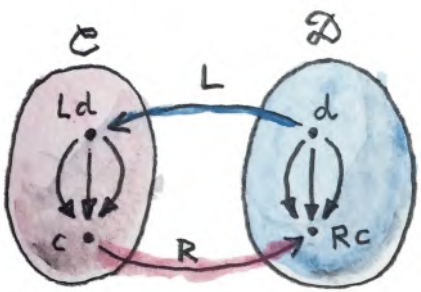
18.2 伴随和Hom-集合

有一个等价的定义，即通过同态集合的自然同构来定义伴随。 这个定义与我们迄今为止所学习的普遍构造很好地联系在一起。每当你听到有关某个唯一态射的陈述时，它将某个构造因子化为一些构造时，你应该将其视为将某个集合映射到一个同态集合。

这就是“选择一个唯一态射”的意义。此外，因子化通常可

以用自然变换来描述。 因子化涉及到交换图表 - 某个态射等于两个态射（因子）的复合。 自然变换将态射映射到交换图表。 因此，在普遍构造中，我们从一个态射到一个交换图表，然后到一个唯一态射。 我们最终得到了从态射到态射的映射，或者从一个同态集合到另一个（通常在不同的范畴中）。 如果这个映射是可逆的，并且如果它可以在所有同态集合上自然地扩展，那么我们就有了一个伴随。

通用构造和伴随-函子之间的主要区别在于后者是全局定义的——对于所有的同态集合。 例如，使用通用构造，你可以定义两个特定对象的积，即使在该范畴中不存在其他对象对的积。 正如我们很快将看到的，如果一个范畴中存在任意一对对象的积，它也可以通过伴随来定义。



这是使用同态集合的伴随的另一种定义。 与之前一样，我们有两个函子□ : : □ → □和□ : : □ → □。 我们选择两个任意的对象：源对象 □在 □中，目标对象 □在 □中。 我们可以使用 □将源对象 □映射到 □中。 现在我们在 □中有两个对象： □ □和 □。 它们定义了一个同态集合：□(□ □, □)

同样地，我们可以使用 □将目标对象映射为 □。 现在我们在 □中有两个对象， □和 □ □。 它们也定义了一个同态集合：□(□, □

□)

我们说 □是 □的左伴随，如果存在同态集合的同构：

□(□ □, □) ≅ □(□, □ □)

这在 □和 □两者都是自然的。 自然性意味着源 □可以在 □上平滑变化；目标 □可以在 □上平滑变化。 更准确地说，我们有一个自然变换 □在从 □到 □ □的以下两个（协变）函子之间。 这些函子在对象上的作用如下：□ → □(□ □, □)□ → □(□, □ □)

另一个自然变换， □，在以下（反变）函子之间起作用：

$$\square \to \square(\square\ \square, \square)$$
$$\square \to \square(\square, \square\ \square)$$

两个自然变换都必须可逆。

很容易证明两个伴随定义是等价的。 例如，让我们从同态集的同构派生单位变换：

$$\square(\square\ \square, \square) \cong \square(\square, \square\ \square)$$

由于这个同构对于任何对象 □都有效，它也对于□ = □ □有效：

$$\square(\square\ \square, \square\ \square) \cong \square(\square, (\square \circ \square)\square)$$

我们知道左边必须至少包含一个态射，即恒等态射。 自然变换将把这个态射映射到 □（□， （□ ◦□） □）的元素，或者插入恒等函子□，一个态射在：

$$\square(\square\ \square, (\square \circ \square)\square)$$

我们得到一个以 □为参数的态射族。 它们形成了从函子 □到函子□ ◦□的自然变换（自然性条件很容易验证）。 这正是我们的单位，□。相反，从单位和余单位的存在，我们可以定义同态集之间的变换。 例如，让我们在同态集 □（□ □, □）中选择一个任意的态射 □。 我们想要定义一个作用于 □的 □，产生一个在 □（□, □ □）中的态射。

实际上没有太多选择。 我们可以尝试的一件事是使用我们来提升 □。 这将产生从□(□ □)到 □ □的态射 □ □— 一个是 □((□ ◦ □)□, □ □)的元素的态射。

对于 □的一个组件，我们需要从 □到 □ □的态射。
这不是一个问题，因为我们可以使用 □□的一个组件从 □到 (□ ◦ □)□ 。我们得到：

□ □= □ □◦ □ □

另一个方向是类似的，推导 □也是如此。

回到Haskell定义的 Adjunction，自然变换 □和 □被多态的（对于 a和 b）函数leftAdjunct和rightAdjunct替代。 函子 □和 □被称为 f和 u：

```
class (Functor f, Representable u) =>
  Adjunction f u | f -> u, u -> f where
    leftAdjunct :: (f a -> b) -> (a -> u b)
    rightAdjunct :: (a -> u b) -> (f a -> b)
```

单元/余单元形式和
leftAdjunct/rightAdjunct形式之间的等价性由以下映射证明：

```
单元                = leftAdjunct id
余单元              = rightAdjunct id
leftAdjunct f = fmap f . 单元
rightAdjunct f = 余单元 . fmap f
```

从范畴论描述到Haskell代码的翻译非常有教育意义。 我强烈鼓励这个作为一个练习。

我们现在准备解释为什么在Haskell中，右伴随自动成为一个可表示的函子。 这是因为，初步的近似，我们可以将Haskell类型的范畴视为集合的范畴。

当右范畴 $\mathcal{D}$是 $\mathcal{C}^{op}$时，右伴随 R是从 $\mathcal{C}$到 $\mathcal{C}^{op}$的函子。 如果我们能找到一个对象$A \in \mathcal{C}$在 $\mathcal{C}$中，使得同态函子 $\mathcal{C}(-, -)$与R自然同构，那么这样的函子就是可表示的。 事实证明，如果 R是从 $\mathcal{C}^{op}$到 $\mathcal{C}$的某个函子 F的右伴随，这样的对象总是存在的——它是单元素集合()在 F下的像：

$$A = F()$$

事实上，伴随性告诉我们以下两个同态集是自然同构的：

$$\mathcal{C}(F(), -) \cong \mathcal{C}^{op}(1, R -) = \mathcal{C}(R -)$$

对于给定的 X，右边是从单元集 ()到 RX的函数集。 我们之前已经看到，每个这样的函数从集合RX中选择一个元素。 这样的函数集同构于集合 RX。 所以我们有：

$$\mathcal{C}(F(), -) \cong R$$

这表明 F确实是可表示的。

18.3 通过伴随构造积

我们之前使用普遍构造引入了几个概念。 当全局定义时，其中许多概念更容易使用伴随性来表达。 最简单的非平凡例子是乘积。 乘积的普遍构造的要点是能够通过普遍乘积将任何类似乘积的候选对象分解。

更准确地说，两个对象 A和 B的乘积是带有两个态射 $A \leftarrow P \rightarrow B$和$A$ B的对象 ($A \times B$)（在Haskell符号中为(a, b)），使得对于任何其他带有两个

对于态射□ : : □ → □和□ : : □ → □，存在唯一的态射
□ : : □ → (□, □)，通过□ □ □和 □ □ □来分解 □和 □。

正如我们之前所见，在Haskell中，我们可以实现一个分解器来生成这个从两个投影中得到的态射：

```
factorizer :: (c -> a) -> (c -> b) -> (c -> (a, b))
factorizer p q = \x -> (p x, q x)
```

很容易验证分解条件成立：

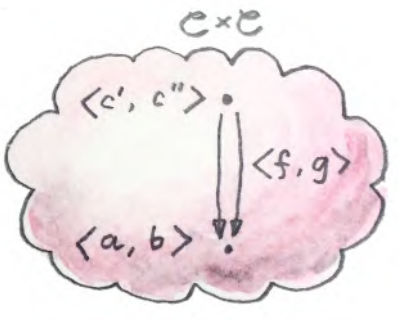
```
fst . 分解器 p q = p
snd . 分解器 p q = q
```

我们有一个映射，它接受一对态射 p和 q并产生
另一个态射 m = 分解器 p q。

我们如何将其转化为两个同态集之间的映射以便定义一个伴随？ 关键是走出 Hask并将一对态射视为乘积范畴中的单个态射。

让我提醒你什么是一个乘积范畴。取两个任意的范畴 □和 □。乘积范畴□ × □中的对象是一对对象，一个来自 □，一个来自 □。态射是一对态射，一个来自 □，一个来自 □。

要在某个范畴 □中定义一个乘积，我们应该从乘积范畴 □×□开始。 来自 □的态射对是乘积范畴□ × □中的单个态射。



起初可能有点困惑，我们使用一个乘积范畴来定义一个乘积。 然而，这些是非常不同的乘积。
我们不需要一个通用构造来定义一个乘积范畴。
我们所需要的只是一对对象和一对态射的概念。

然而，然而中的一对对象不是中的中的一个对象。 它是一个不同范畴中的对象，它是 × 。 我们可以形式化地将这对对象写作 , ，其中，其和是的对象。 另一方面，通用构造是为了定义对象另一 × （或在Haskell中的 (a, b)），它是同一范畴），中的一个对象。 这个对象被认为以通用构造指定的方式表示对 , 的。 它并不总是存在的，即使对于某些对象存在，也可能对于它并中的其他对象对不存在。

现在让我们将因子化器看作是同态集的映射。 第一个同态集在乘积范畴第一 × 中，第二个同态集在中，中。 × 中的一般态射将是一对态射 , ：

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

具有 □″可能与 □′不同。 但是为了定义一个乘积，我们对□ × □中的一个特殊态射感兴趣，即共享相同源对象 □的一对 □和 □。 没问题：在伴随定义中，左侧同态集的源不是任意对象——它是右侧范畴中某个对象上左函子 □的结果。 符合要求的函子很容易猜到——它是从 □到□ × □的对角函子 Δ，其对对象的作用是：

$$\Delta\,\square = \square, \square$$

我们的伴随中的左侧同态集应该是：

$$(\square \times \square)(\Delta\,\square, \square, \square)$$

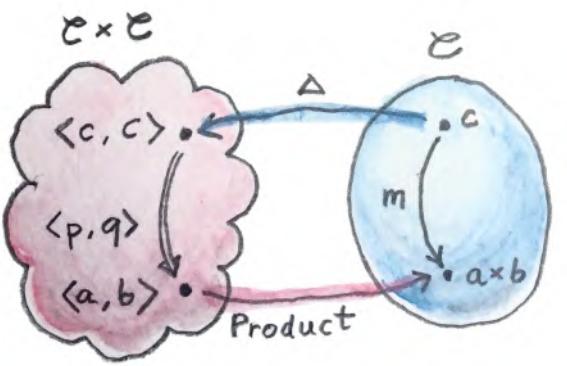
这是一个乘积范畴中的同态集。 它的元素是我们认识到是我们的因子化器的参数的态射对：

$$(\square \to \square) \to (\square \to \square) \cdots$$

右手边的同态集合存在于 □中，并且它在中， × □中作用于目标对象上的某个函子 □的源对象 □和结果之间。 这就是将对□ □, □ □的映射到我们的乘积对象□ × □的函子。 我们将同态集合中的这个元素识别为结果的

分解器：

$$\cdots \to (\square \to (\square, \square))$$



我们仍然没有完整的伴随关系。 为此，我们首先需要我们的分解器是可逆的 - 我们正在构建同态集合之间的一个同构。
分解器的逆应该从一个态射 □开始 - 从某个对象 □到乘积对象□ × □的态射。 换句话说， □应该是以下元素的一个：

□(□, □ × □)

逆因子化器应该将 □映射到一个态射□ □, □ □在□ × □中，从□ □, □ □到□ □, □ □，换句话说，一个作为元素的态射：

(□ × □)(Δ □, □ □, □ □)

如果存在这种映射，我们可以得出结论，对角函子存在右伴随函子。 该函子定义了一个乘积。

在Haskell中，我们可以通过将 m分别与 fst和 snd组合来构造逆因子化器。

```
p = fst . m
q = snd . m
```

为了完成两种定义乘积的方式等价的证明，我们还需要证明在 $\mathcal{C}$、 $\mathcal{D}$和 $\mathcal{E}$中，同态集之间的映射是自然的。 我将把这留给专注的读者作为练习。

总结我们所做的：范畴乘积可以全局地定义为对角函子的右伴随：

$$(\mathcal{C} \times \mathcal{C})(\Delta_{\mathcal{C}}(c), (a, b)) \cong \mathcal{C}(c, a \times b)$$

在这里， $\mathcal{C} \times \mathcal{C}$是我们右伴随函子 $\mathcal{C} \to \mathcal{C} \times \mathcal{C}$对于对 (a, b)的作用的结果。 注意，从 $\mathcal{C} \times \mathcal{C}$到任何函子都是双函子，所以 $\mathcal{C} \to \mathcal{C} \times \mathcal{C}$是一个双函子。 在Haskell中，双函子简单地写作 (,)。 你可以将它应用于两种类型并得到它们的乘积类型，例如：

```
(,) Int Bool ~ (Int, Bool)
```

18.4 通过伴随构造指数

指数 a^b，或者函数对象 $b \Rightarrow a$，可以使用通用构造来定义。 如果这个构造对于所有对象的对都存在，它可以被看作是一个伴随。 再次，关键是集中注意力在这个陈述上：

> 对于任何其他对象 c 具有一个态射 $f :: c \times b \to a$
> 存在一个唯一的态射 $h :: c \to (b \Rightarrow a)$

这个陈述建立了同态集之间的映射关系。

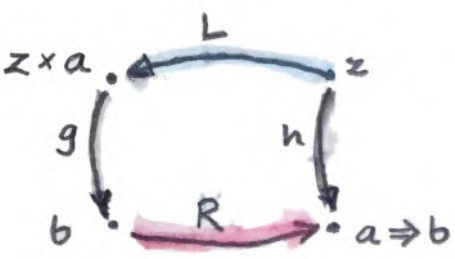
在这种情况下，我们处理的是同一范畴中的对象，因此这两个伴随函子是自函子。 左（自）函子 $(-) \times b$，当

作用于对象 □时，产生□ × □。 它是一个对应于与某个固定的 □取积的函子。

右（自）函子 □，当作用于 □时，产生函数对象□ ⇒ □（或 □□ ）。再次强调， □是固定的。 这两个函子之间的伴随通常写作：

$$- \times \square \dashv (-)^{\square}$$

在通用构造中，底层的同态集映射可以通过重新绘制我们在通用构造中使用的图表来看得最清楚。



注意， □ □ □态射[1]实际上就是这个伴随的余单位：

$$(\square \Rightarrow \square) \times \square \to \square$$

其中：

$$(\square \Rightarrow \square) \times \square = (\square \circ \square)\square$$

我之前提到过，通用构造定义了一个唯一的对象，与同构无关。这就是为什么我们有“the”乘积和“the”指数。这个性质同样适用于伴随：如果一个函子有一个伴随函子，那么这个伴随函子是唯一的，与同构无关。

1参见第9章关于通用构造。

18.5 挑战

1. 推导 □的自然性正方形，即两个（逆变）函子之间的变换：

$$□ → □(□ □, □)$$
$$□ → □(□, □ □)$$

2. 从伴随的第二个定义中的同态集同构推导出余单位 □。

3. 完成两个伴随定义等价性的证明。

4. 证明余积可以通过一个伴随定义。 从余积的因子化定义开始
   。
5. 证明余积是对角函子的左伴随。
6. 在Haskell中定义乘积和函数对象之间的伴随关系。

19

自由/遗忘伴随

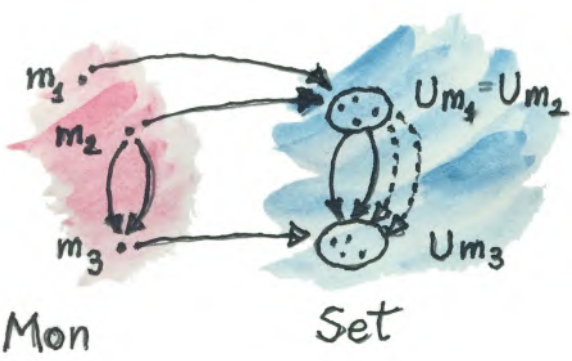
自由构造是伴随的一个强大应用。 自由函子被定义为遗忘函子的左伴随。 遗忘函子通常是一个简单的函子，它遗忘了一些结构。 例如，许多有趣的范畴是建立在集合之上的。 但是范畴对象，即抽象了这些集合的对象，没有内部结构 - 它们没有元素。 然而，这些对象通常携带着集合的记忆，也就是说，从给定范畴 □ 到 □ □ 的映射 - 函子。 与 □中某个对象对应的集合被称为它的底层集合。

幺半群是具有底层集合的对象 - 元素的集合。 有一个遗忘函子 □从幺半群 □ □ 的范畴到集合的范畴，它将幺半群映射到它们的底层集合。 它还将幺半群态射（同态）映射到集合之间的函数。

我喜欢把幺半群 □ □ 看作是具有分裂个性的。 一方面，它是一堆带有乘法和单位元素的集合。 另一方面，它是一个只有结构在它们之间的态射中编码的无特征对象的范畴。 每个保持乘法和单位的集合函数都会产生一个 □ □ 中的态射。

需要记住的事情：

- 可能有许多映射到相同集合的幺半群，以及
- 幺半群态射（同态）的数量少于（或最多与）它们底层集合之间的函数的数量。



幺半群 M_1和 M_2有相同的底层集合。在 M_2和 M_3的底层集合之间，有更多的函数而不是态射。

函子 □是遗忘函子 □的左伴随函子，它从生成器集合构建自由幺半群。 这个伴随关系是从我们之前讨论过的自由幺半群的普遍构造中得出的。[1]

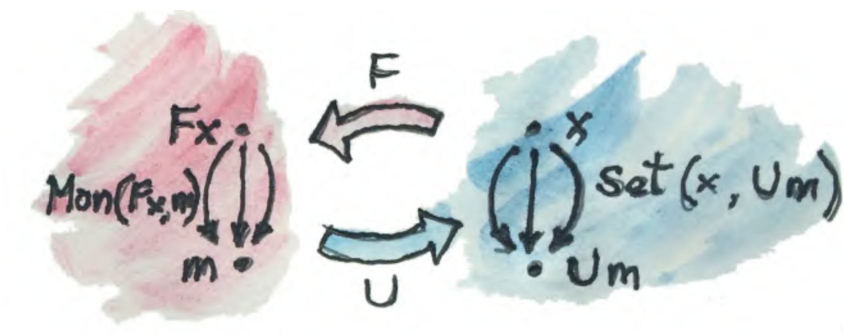
1参见第13章关于自由幺半群。

从同态集的角度来看，我们可以将这个伴随关系写成：

□ □(□□ □, □) ≅ □ □(□□, □ □)

这个（对 □ 和 □）自然同构告诉我们：

- 对于由 □生成的自由幺半群□ □和任意幺半群 □之间的每个幺半群同态，存在一个唯一的函数将生成器集合 □嵌入到底层集合□中。 这是一个 □ □(□□, □ □)中的函数。
- 对于将 □嵌入到某个□的底层集合中的每个函数，存在一个唯一的幺半群态射，它连接了由 □生成的自由幺半群和幺半群 □。 （这就是我们在通用构造中称之为*h*的态射。）



直观上，□ □是可以基于 □构建的“最大”幺半群。 如果我们能够查看幺半群的内部，我们会发现属于 □ □ □(□ □, □）的任何态射都会将这个自由幺半群嵌入到某个其他幺半群 □中。 它可能通过识别一些元素来实现这一点。 特别地，它将□ □的生成元（即 □的元素）嵌入到 □中。

伴随性质表明，嵌入 □的函数从 □ □ (Ⅲ, □ □)到右边的嵌入唯一地确定了左边的幺半群的嵌入，反之亦然。

在Haskell中，列表数据结构是一个自由幺半群（有一些注意事项：参见Dan Doel的博客文章[2]）。 列表类型 [a]是一个具有类型的自由幺半群表示生成器的集合a。例如，类型 [Char]包含单位元素——空列表 []——和单个元素如['a'], ['b']——自由幺半群的生成器。 其余部分通过应用“乘积”生成。在这里，两个列表的乘积就是将一个列表附加到另一个列表的末尾。 附加是可结合的和有单位元（即，存在一个中性元素——这里是空列表）。 由Char生成的自由幺半群就是由Char的所有字符串组成的集合。 在Haskell中，它被称为 String：

```
类型 String = [Char]
```

(type定义了一个类型同义词——对现有类型的另一个名称).

另一个有趣的例子是仅由一个生成器构建的自由幺半群。 它是单位列表的类型，[()]. 它的元素是 [], [()],[(), ()]等等。每个这样的列表可以用一个自然数来描述——它的长度。 在单位列表中没有更多的信息编码。
将两个这样的列表连接起来会产生一个新的列表，其长度是其组成部分的长度之和。 很容易看出，类型 [()]与自然数的加法幺半群（带零）是同构的。 这里是两个互为逆的函数，证明了这个同构：

```
toNat :: [()] -> Int
toNat = length
```

2http://comonad.com/reader/2015/free-monoids-in-haskell/

```
toLst :: Int -> [()]
toLst n = replicate n ()
```

为了简单起见，我使用了类型 Int而不是 Natural，但思想是一样的。 函数 replicate创建一个长度为 n的列表，预先填充了一个给定的值 - 在这里是单位元。

19.1 一些直觉

接下来是一些手势的论证。 这种类型的论证远非严谨，但它们有助于形成直觉。

为了对自由/遗忘伴随性有一些直觉，需要记住函子和函数的本质是有损失的。 函子可能会合并多个对象和态射，函数可能会将集合中的多个元素合并在一起。此外，它们的图像可能只覆盖其余域的一部分。

在□□中，“平均”的同态集将包含一整个函数谱，从最不损失的函数（例如，单射或可能的同构）开始，到将整个定义域折叠为单个元素的常数函数（如果存在的话）。

我倾向于将任意范畴中的态射视为有损的。这只是一种心理模型，但它是一个有用的模型，特别是在思考伴随性时——尤其是其中一个范畴是□□的情况。

从形式上讲，我们只能谈论可逆的（同构）或不可逆的态射。正是后一种可能被视为有损的。 还有一种称为单态射和满态射的概念，它们推广了单射（非折叠）和满射（覆盖整个余域）函数的思想。

但是可能存在既是单态射又是满态射的态射，而且仍然是不可逆的。

在自由 □ 遗忘伴随性中，我们在左边有更受限制的范畴 □，在右边有更不受限制的范畴 □。
在 □中的态射“更少”，因为它们必须保留一些附加结构。 在 □ □ □的情况下，它们必须保留乘法和单位。 在 □中的态射不必保留太多结构，所以它们“更多”。

当我们将遗忘函子 □应用于 □中的对象 □时，我们认为它揭示了 □的“内部结构”。 事实上，如果 □是 □ □ □，我们将 □视为 □的内部结构的定义 - 其底层集合。（在任意范畴中，我们不能单独讨论对象的内部情况，只能通过其与其他对象的连接来了解，但在这里我们只是在概括地描述。）

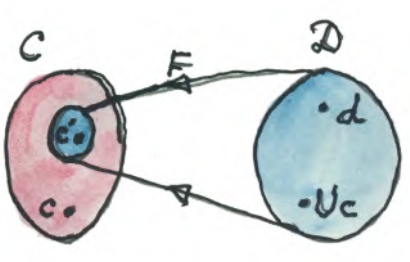
如果我们使用 □将两个对象 □′和 □映射起来，我们期望在一般情况下，同态集合的映射 □(□′, □)只会覆盖只会(□ □′, □ □)的一个子集。 这是因为 □(□′, □)中的态射必须保持额外的结构，而□(□ □′, □ □)中的态射则不需要。

但是由于伴随是特定同态集合的同构，我们在选择 □′时必须非常挑剔。 在伴随中，□′不是从 □的任意位置选择的，而是从（可能是

自由函子的较小图像 □：

$$□(□\ □, □) \cong □(□, □\ □)$$

因此， □的图像必须由许多从任意 □到 □的态射组成。 实际上，从 □ □到 □的保结构态射的数量必须与从□到□ □的非保结构态射的数量相同。 这意味着 □的图像必须由本质上无结构的对象组成（因此，没有结构需要通过态射来保持）。 这种“无结构”的对象被称为自由对象。



在幺半群的例子中，自由幺半群除了由单位和结合律生成的结构外没有其他结构。 除此之外，所有的乘法都会产生全新的元素。

在自由幺半群中，2 □ 3不是 6—它是一个新元素 [2, 3]。 由于 [2, 3]和 6没有等同，所以从这个自由幺半群到任何其他幺半群 □的态射可以将它们分别映射。 但它也可以将 [2, 3]和 6（它们的乘积）映射到 □的同一个元素。 或者在加法幺半群中将 [2, 3]和 5（它们的和）等同起来，等等。不同的等同关系会得到不同的幺半群。

这导致了另一个有趣的直觉：自由幺半群不是执行幺半操作，而是累积传递给它的参数。它们不是将2和3相乘，而是将2和3记住在一个列表中。

这种方案的优势在于我们不必指定将要使用的幺半操作。我们可以继续累积参数，并且只在最后将运算符应用于结果。 然后我们可以选择要应用的运算符。 我们可以将数字相加，或者将它们相乘，或者执行模2加法等等。自由幺半群将表达式的创建与其评估分开。 当我们讨论代数时，我们将再次看到这个想法。 我们将在谈论代数时再次看到这个想法。

这种直觉可以推广到其他更复杂的自由构造-法。 例如，我们可以在计算之前累积整个表达式树。 这种方法的优点是我们可以转换这样的树，使评估更快或者占用更少的内存。例如，在实现矩阵微积分时就会这样做，因为急切的评估会导致大量的临时数组分配来存储中间结果。

19.2 挑战

1. 考虑一个由单例集合构建的自由幺半群作为其生成元。证明从这个自由幺半群到任何幺半群 □的态射和从单例集合到 □的底层集合的函数之间存在一一对应关系。

20

单子：程序员的定义

程序员们围绕单子发展了一个完整的神话体系。它被认为是编程中最抽象和最困难的概念之一。 有人“懂了”，也有人不懂。 对于许多人来说，理解单子的概念就像是一种神秘的体验。 单子抽象了许多不同的构造的本质，在日常生活中我们没有一个好的类比。 我们只能在黑暗中摸索，就像那些摸到大象不同部分的盲人一样，兴高采烈地喊道：“这是一根绳子”，“这是一根树干”，或者“这是一个卷饼！”让我澄清一下：单子周围的整个神秘感是一个误解的结果。 单子是一个非常简单的概念。 正是单子的应用多样性导致了混淆。

作为这篇文章研究的一部分，我查了一下胶带（又称鸭嘴胶带）及其应用。 这是一些你可以用它做的小样例：

- 封闭管道
- 修理阿波罗13号上的CO_2吸收器
- 治疗疣
- 修复苹果iPhone 4的掉话问题
- 制作舞会礼服
- 建造悬索桥

现在想象一下，如果你不知道什么是胶带，你会根据这个列表来弄清楚。祝你好运！

所以我想在“单子就像…”的陈词滥调中再添加一项：单子就像胶带。 它的应用非常广泛，但其原理非常简单：它将事物粘在一起。 更准确地说，它组合事物。

这在一定程度上解释了为什么很多程序员，特别是那些来自命令式背景的程序员，对单子的理解有困难。 问题在于我们不习惯用函数组合的方式来思考编程。 这是可以理解的。 我们经常给中间值取名字，而不是直接从一个函数传递到另一个函数。 我们还会将短的粘合代码内联，而不是将它们抽象成辅助函数。 这是一个用C语言实现的命令式风格的向量长度函数：

```
double vlen(double * v) {
        double d = 0.0;
    int n;
    for (n = 0; n < 3; ++n)
                d += v[n] * v[n];
    return sqrt(d);
}
```

将此与（程式化的）Haskell版本进行比较，使函数组合明确：

```
vlen = sqrt . sum . fmap (flip (^) 2)
```

（在这里，为了使事情更加神秘，我部分应用了指数运算符 (^)，将其第二个参数设置为 2。）我并不是在争论Haskell的无点
　　风格总是更好，只是函数组合是我们在编程中所做的一切的基础。 尽管我们实际上是在组合函数，Haskell还是费尽心思提供了一种称为 do符号的命令式语法来进行单子组合。 我们稍后会看到它的用法。 但首先，让我解释一下为什么我们需要单子组合。

20.1 Kleisli范畴

我们之前通过对常规函数进行修饰来得到了写入器单子。 特定的修饰是通过将它们的返回值与字符串或者更一般地，与幺半群的元素配对来完成的。
我们现在可以认识到这样的修饰是一个函子：

```
newtype Writer w a = Writer (a, w)

instance Functor (Writer w) where
        fmap f (Writer (a, w)) = Writer (f a, w)
```

随后，我们找到了一种组合修饰函数或者Kleisli箭头的方法，它们是以下形式的函数：

```
a -> Writer w b
```

正是在组合中，我们实现了日志的累积。

现在我们准备给出Kleisli范畴的更一般定义。 我们从一个范畴 𝒞和一个自函子 𝑇开始。 相应的Kleisli范畴 𝒞𝑇具有与 𝒞相同的对象，但其态射是不同的。 在 𝒞中，两个对象 𝑋和 𝑌之间的态射被实现为一个态射：

$$X \to T\,Y$$

在原始范畴 𝒞中。 重要的是要记住，我们将 𝒞中的Kleisli箭头视为 𝑋和 𝑌之间的态射，而不是 𝑋和𝑇 𝑌之间的态射。

在我们的例子中，𝑇被特化为 Writer w，其中w是某个固定的幺半群。

只有当我们能为Kleisli箭头定义适当的组合时，它们才形成一个范畴。 如果存在一个关联且对每个对象都有一个恒等箭头的组合，则函子 𝑇被称为一个 monad，而结果范畴被称为Kleisli范畴。

在Haskell中，Kleisli组合是使用鱼操作符 >=>定义的，而恒等箭头是一个名为 return的多态函数。 下面是使用Kleisli组合定义monad的定义：

```
Monad类 m where
    (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
    return :: a -> m a
```

请记住，定义单子的方式有很多等价的方式，并且这不是Haskell生态系统中的主要方式。 我喜欢它的概念简洁性和提供的直觉，但在编程时还有其他更方便的定义。 我们马上会谈到它们。

在这个表述中，单子定律非常容易表达。 它们在Haskell中无法强制执行，但可以用于等式推理。

推理。 它们只是Kleisli范畴的标准组合法则：

```
(f >=> g) >=> h = f >=> (g >=> h) -- 结合律
return >=> f = f                  -- 左单位
f >=> return = f                  -- 右单位
```

这种定义也表达了一个范畴的真正含义：它是一种组合装饰函数的方式。 它不是关于副作用或状态。它是关于组合。 正如我们后面将看到的，装饰函数可以用来表达各种效果或状态，但这不是范畴的目的。范畴是将一个装饰函数的一端与另一个装饰函数的另一端连接起来的粘性胶带。

回到我们的 Writer示例：日志函数（ Writer函子的Kleisli箭头）构成一个范畴，因为 Writer是一个单子：

```
instance Monoid w => Monad (Writer w) where
    f >=> g = \a ->
        let Writer (b, s) = f a
                    Writer (c, s') = g b
        in Writer (c, s `mappend` s')
    return a = Writer (a, mempty)
```

只要满足 Writer w的单子定律，就满足单子定律（在Haskell中也无法强制执行）。

对于 Writer单子，有一个有用的Kleisli箭头定义为 tell。 它的唯一目的是将其参数添加到日志中：

```
tell :: w -> Writer w ()
tell s = Writer ((), s)
```

我们稍后将其用作其他单子函数的构建块。

20.2 鱼的解剖

当为不同的单子实现fish运算符时，您很快会意识到有很多重复的代码可以轻松提取出来。首先，两个函数的Kleisli组合必须返回一个函数，因此它的实现可以从一个以类型 a为参数的lambda开始：

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

我们唯一能做的就是将这个参数传递给 f:

```
f >=> g = \a -> let mb = f a
                    in ...
```

此时，我们需要生成类型为 m c的结果，我们可以使用类型为 m b的对象和函数 g :: b -> m c。 让我们定义一个可以帮助我们完成这个任务的函数。这个函数被称为 bind，通常以中缀运算符的形式书写:

```
(>>=) :: m a -> (a -> m b) -> m b
```

对于每个单子，我们可以定义绑定操作符，而不是定义鱼操作符。 事实上，标准的Haskell单子定义使用了绑定操作符:

```
Monad类 m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

这是 Writer monad的绑定(bind)的定义:

```
(Writer (a, w)) >>= f = let Writer (b, w') = f a
                        in Writer (b, w `mappend` w')
```

它确实比鱼算子的定义要短。

我们可以进一步解析绑定(bind)，利用以下事实 m是一个函子。 我们可以使用 fmap将函数 a -> m b应用于 m a的内容。 这将把 a转换为 m b。 应用的结果因此是 m (m b)类型的。 这不完全是我们想要的——我们需要 m b类型的结果——但我们已经接近了。 我们所需要的只是一个将 m的双重应用折叠或扁平化的函数。这个函数被称为 join:

```
join :: m (m a) -> m a
```

使用 join，我们可以将绑定(bind)重写为:

```
ma >>= f = join (fmap f ma)
```

这导致我们定义一个单子的第三个选项：

```
class Functor m => Monad m where
    join :: m (m a) -> m a
    return :: a -> m a
```

在这里，我们明确要求 m是一个 Functor。 在之前两个单子的定义中，我们不需要这样做。 那是因为任何支持fish或bind运算符的类型构造器 m自动成为一个函子。 例如，可以用bind和 return来定义 fmap：

```
fmap f ma = ma >>= \a -> return (f a)
```

为了完整起见，这里是 Writer单子的 join：

```
join :: Monoid w => Writer w (Writer w a) -> Writer w a
join (Writer ((Writer (a, w')), w)) = Writer (a, w `mappend` w')
```

20.3 do-Notation

使用单子编写代码的一种方式是使用Kleisli箭头- 使用fish运算符进行组合。 这种编程模式是点无关风格的泛化。 点无关的代码紧凑而且通常非常优雅。 总的来说，它可能很难理解，接近于晦涩。 这就是为什么大多数程序员更喜欢给函数参数和中间值命名。

当处理单子时，它意味着更喜欢绑定运算符而不是鱼运算符。 绑定操作接受一个单子值并返回一个单子值。 程序员可以选择给这些值命名。 但是这几乎没有改进。 我们真正想要的是假装我们正在处理常规值，而不是封装它们的单子容器。 这就是命令式代码的工作方式 - 副作用，例如更新全局日志，大多数情况下对用户隐藏。 这就是Haskell中的do表示法所模拟的。

你可能会想知道，为什么还要使用单子？ 如果我们想要使副作用不可见，为什么不坚持使用命令式语言？
答案是单子能更好地控制副作用。 例如，在Writer单子中的日志会从函数传递到函数，从未在全局范围内暴露。 没有破坏日志或创建数据竞争的可能性。 此外，单子代码明确地划分并与程序的其余部分隔离开来。

这种符号只是单子组合的语法糖。 表面上看起来很像命令式代码，但它直接转换为一系列的绑定和lambda表达式。

例如，我们之前用来说明 Writer单子中Kleisli箭头组合的例子。 使用我们当前的定义，它可以重写为：

```
process :: String -> Writer String [String]
process = upCase >=> toWords
```

这个函数将输入字符串中的所有字符转换为大写，并将其拆分为单词，同时生成其操作的日志。

在 do 表示法中，它看起来像这样：

```
process s = do
    upStr <- upCase s
    toWords upStr
```

在这里， upStr只是一个 String，即使 upCase产生一个 Writer：

```
upCase :: String -> Writer String String
upCase s = Writer (map toUpper s, "upCase ")
```

这是因为编译器将do块展开为：

```
处理 s =
        upCase s >>= \upStr ->
            toWords upStr
```

将upCase的单子结果绑定到一个以String为参数的lambda函数中。在do块中，这个字符串的名称会显示在这一行中：

```
upStr <- upCase s
```

我们说upStr获得了upCase s的结果。

当我们内联toWords时，伪命令式风格更加明显。 我们用tell函数替换它，tell函数记录字符串"toWords "，然后调用return函数返回使用words函数拆分字符串upStr的结果。 注意，words是一个对字符串起作用的常规函数。

```
处理 s = do
        upStr <- upCase s
        tell "toWords "
        return (words upStr)
```

在这里，do块中的每一行都引入了一个新的嵌套绑定。

```
处理 s =
        upCase s >>= \upStr ->
           tell "toWords " >>= \() ->
              return (words upStr)
```

注意 tell产生一个单位值，所以不需要传递给下一个lambda函数。 忽略单子结果的内容（但不忽略其效果 - 在这里是对日志的贡献）是非常常见的，因此有一个特殊的运算符来替换绑定：

```
(>>) :: m a -> m b -> m b
m >> k = m >>= (\_ -> k)
```

我们的代码的实际展开如下：

```
process s =
        upCase s >>= \upStr ->
           tell "toWords " >>
           return (words upStr)
```

一般来说，do块由使用左箭头引入新名称的行（或子块）组成，然后这些名称在代码的其余部分中可用，或者纯粹用于副作用。 绑定运算符在代码行之间是隐式的。 顺便说一下，在Haskell中，是可能的

用大括号和分号替换 do块中的格式。 这为将单子描述为重载分号的方式提供了理由。

请注意，在解糖时，lambda和bind运算符的嵌套会影响 do块中其余部分的执行，根据每行的结果。 这个特性可以用来引入复杂的控制结构，例如模拟异常。

有趣的是， do符号的等价物在命令式语言中也有应用，尤其是C++。 我在谈论可恢复的函数或协程。 C++的未来形式是一个单子[1]。 这是一个延续单子的例子，我们很快会讨论。 延续的问题在于它们很难组合。 在Haskell中，我们使用 do符号将“我的处理程序将调用你的处理程序”的混乱代码转化为看起来非常像顺序代码的形式。 可恢复函数使得在C++中实现相同的转换成为可能。相同的机制可以应用于将嵌套循环的混乱代码[2]转化为列表推导式或“生成器”，它们本质上是列表单子的 do符号。 在没有单子的统一抽象的情况下，每个问题通常都通过为语言提供自定义扩展来解决。 在Haskell中，所有这些都通过库来处理。

1https://bartoszmilewski.com/2014/02/26/c17-i-see-a-monad-in-your-future/
2https://bartoszmilewski.com/2014/04/21/getting-lazy-with-c/

21

单子和效果

现在我们知道单子的用途了——它允许我们组合装饰函数——真正有趣的问题是为什么在函数式编程中装饰函数如此重要。我们已经看到了一个例子，即 Writer单子，其中装饰允许我们在多个函数调用之间创建和累积日志。 原本需要使用不纯函数（例如，访问和修改某个全局状态）来解决的问题，现在可以使用纯函数来解决。

21.1 问题

这里是一份类似问题的简短列表，摘自Eugenio Moggi的经典论文[1]，所有这些问题传统上都是通过放弃函数的纯度来解决的。

- 偏函数：可能不会终止的计算

1https://core.ac.uk/download/pdf/21173011.pdf

- 非确定性：可能返回多个结果的计算
- 副作用：访问/修改状态的计算
  - 只读状态，或环境
  - 只写状态，或日志
  - 读写状态
- 异常：可能失败的偏函数
- 连续性：保存程序状态并按需恢复

- 交互输入
- 交互输出

真正令人惊叹的是，所有这些问题都可以使用同样巧妙的技巧来解决：转向装饰函数。 当然，每种情况下的装饰都会完全不同。

你必须意识到，在这个阶段，没有要求装饰是单子的。 只有当我们坚持组合时，将一个单一的装饰函数分解为更小的装饰函数时，我们才需要单子。 同样，由于每个装饰的方式都不同，单子的组合将以不同的方式实现，但总体模式是相同的。 这是一个非常简单的模式：具有结合性和身份的组合。

下一节将大量使用Haskell示例。 如果你渴望回到范畴论或者已经熟悉Haskell的单子实现，可以随意略过或者跳过这一节。

21.2 解决方案

首先，让我们分析一下我们使用了 Writermonad的方式。 我们从一个执行特定任务的纯函数开始 - 给定参数，它会产生特定的输出。

我们用另一个函数替换了这个函数，它将原始输出与一个字符串配对。 这是我们解决日志问题的方法。 那就是我们解决日志问题的解决方案。

我们不能停在那里，因为一般来说，我们不想处理庞大的解决方案。 我们需要能够将一个产生日志的函数分解为更小的产生日志的函数。 正是这些较小函数的组合使我们引入了单子的概念。

真正令人惊奇的是，同样的函数返回类型修饰模式适用于很多通常需要放弃纯度的问题。 让我们逐个遍历列表，并确定适用于每个问题的修饰。

21.2.1 部分性

我们修改了可能不会终止的每个函数的返回类型，将其转换为“提升”类型 - 一个包含所有原始类型值和特殊的“底部”值 ⊥。 例如，作为一个集合， Bool类型将包含两个元素： True和 False。 提升的Bool包含三个元素。 返回提升的 Bool的函数可能产生 True或 False，或者永远执行。

有趣的是，在像Haskell这样的惰性语言中，一个永不结束的函数实际上可能会返回一个值，并且这个值可以传递给下一个函数。 我们称这个特殊值为底部。 只要这个值不是显式需要的（例如，用于模式匹配或作为输出产生），它可以在程序的执行过程中传递而不会停顿。 因为每个Haskell函数都可能是非终止的，所以在Haskell中假定所有类型都是提升的。

这就是为什么我们经常谈论Haskell的范畴 Hask（提升）

类型和函数而不是更简单的 □ □ ⫝̸ 尽管如此，不清楚 Hask是否是一个真正的范畴（参见Andrei Bauer的帖子[2]）。

21.2.2 非确定性

如果一个函数可以返回许多不同的结果，那么它可以一次性返回所有结果。 从语义上讲，一个非确定性函数等价于返回结果列表的函数。 在懒惰的垃圾回收语言中，这是很有道理的。 例如，如果你只需要一个值，你可以直接取列表的头部，而尾部将永远不会被计算。 如果你需要一个随机值，可以使用随机数生成器从列表中选择第n个元素。 懒惰甚至允许你返回一个无限的结果列表。

在列表单子中，Haskell的非确定性计算的实现中， join被实现为 concat。 记住 join应该将一个容器的容器展平 - concat将一个列表的列表连接成一个单一的列表。return创建一个单例列表：

```
实例Monad []在这里
join = concat
return x = [x]
```

列表单子的绑定运算符由以下通用公式给出：
fmap 后跟 join ，在这种情况下得到：

```
as >>= k = concat (fmap k as)
```

在这里，函数 k本身产生一个列表，应用于列表 as的每个元素。 结果是一个列表的列表，使用以下方法展开
concat.

2http://math.andrej.com/2016/08/06/hask-is-not-a-category/

从程序员的角度来看，使用列表比例如在循环中调用非确定性函数或实现返回迭代器的函数更容易（尽管在现代C++[3]中，返回惰性范围几乎等同于返回Haskell中的列表）。

在游戏编程中，使用非确定性的创造性例子是很好的。 例如，当计算机与人类下棋时，它无法预测对手的下一步。 然而，它可以生成所有可能的移动列表并逐个分析它们。类似地，非确定性解析器可以为给定的表达式生成所有可能的解析列表。

尽管我们可能将返回列表的函数解释为非确定性的，但列表单子的应用范围要广得多。 那是因为将产生列表的计算串联起来是函数式编程中迭代结构（循环）的完美替代品。 一个循环通常可以使用fmap来将循环体应用于列表的每个元素进行重写。 列表单子中的donotation可以用来替代复杂的嵌套循环。

我最喜欢的例子是生成勾股数的程序——正整数三元组，可以构成直角三角形的边。

```
triples = do
    z <- [1..]
    x <- [1..z]
    y <- [x..z]
    guard (x^2 + y^2 == z^2)
    return (x, y, z)
```

3http://ericniebler.com/2014/04/27/range-comprehensions/

第一行告诉我们，z从一个无限的正数列表[1..]中获取一个元素。 然后，x从一个介于1和z之间的（有限的）数字列表[1..z]中获取一个元素。 最后，y从介于x和z之间的数字列表中获取一个元素。 我们有三个数字1□□□□□□可供使用。函数guard接受一个Bool表达式并返回一个单位列表：

```
guard :: Bool -> [()]
guard True = [()]
guard False = []
```

这个函数（它是一个更大类别 MonadPlus的成员）在这里用于过滤掉非勾股数三元组。 实际上，如果你看一下bind的实现（或者相关的运算符 >>），你会注意到，当给定一个空列表时，它会产生一个空列表。 另一方面，当给定一个非空列表（这里是包含单元 [()]的单例列表）时，bind将调用继续函数，这里是 return (x, y, z)，它会产生一个包含经过验证的勾股数三元组的单例列表。 所有这些单例列表将被外部的bind连接起来，产生最终的（无限）结果。 当然，调用 triples的调用者永远无法消耗整个列表，但这并不重要，因为Haskell是惰性的。

通常需要三层嵌套循环的问题在列表单子和do表示法的帮助下得到了极大简化。 如果这还不够，Haskell还允许你使用列表推导式进一步简化这段代码：

```
triples = [(x, y, z) | z <- [1..]
                     , x <- [1..z]
                     , y <- [x..z]
                     , x^2 + y^2 == z^2]
```

这只是对列表单子的进一步语法糖（严格来说，
MonadPlus）。
在其他函数式或命令式
语言中，你可能会看到类似的构造，称为生成器和协程。

21.2.3 只读状态

一个对某些外部状态或环境只读访问的函数可以总是被一个接受该环境作为额外参数的函数所替代。 一个纯函数 (a, e) -> b（其中 e 是环境的类型）乍一看不像一个Kleisli箭头。但是一旦我们将其柯里化为 a -> (e -> b)我们就能认出这种修饰是我们的老朋友读取器函子：

```
newtype Reader e a = Reader (e -> a)
```

你可以将返回 Reader的函数解释为生成一个小型可执行文件：一个给定环境产生所需结果的操作。 有一个辅助函数 runReader用于执行这样的操作：

```
runReader :: Reader e a -> e -> a
runReader (Reader f) e = f e
```

它可能对不同的环境值产生不同的结果。
请注意，返回一个 Reader的函数和 Reader
动作本身都是纯的。
要实现 Reader单子的绑定，首先注意你必须生成一个接受环境 e 并产生一个 b的函数：

```
ra >>= k = Reader (\e -> ...)
```

在lambda内部，我们可以执行动作 ra来产生一个 a：

```
ra >>= k = Reader (\e -> let a = runReader ra e
                        in ...)
```

然后，我们可以将 a 传递给延续 k 以获得一个新的动作 rb：

```
ra >>= k = Reader (\e -> let a = runReader ra e
                            rb = k a
                        in ...)
```

最后，我们可以使用环境 e运行动作 rb：

```
ra >>= k = Reader (\e -> let a = runReader ra e
                            rb = k a
                        in runReader rb e)
```

为了实现 return，我们创建一个忽略环境并返回不变值的操作。

将所有内容放在一起，经过一些简化，我们得到以下定义：

实例Monad（Reader e）的定义如下：

```
ra >>= k = Reader (\e -> runReader (k (runReader ra e)) e)
return x = Reader (\e -> x)
```

21.2.4 只写状态

这只是我们最初的日志示例。 装饰由 Writer函子提供：

```
newtype Writer w a = Writer (a, w)
```

为了完整起见，还有一个简单的辅助函数 runWriter，用于解包数据构造函数：

```
runWriter :: Writer w a -> (a, w)
runWriter (Writer (a, w)) = (a, w)
```

正如我们之前所见，在使 Writer可组合的过程中，w必须是一个幺半群。 以下是用绑定运算符写的 Writer的monad实例：

```
instance (Monoid w) => Monad (Writer w) where
    (Writer (a, w)) >>= k = let (a', w') = runWriter (k a)
                                in Writer (a', w `mappend` w')
    return a = Writer (a, mempty)
```

21.2.5 状态

具有读/写访问状态的函数结合了 Reader和 Writer的修饰。 你可以将它们看作是纯函数，它们将状态作为额外的参数，并产生一对值/状态作为结果：(a, s) -> (b, s)。 在柯里化之后，我们将它们转化为Kleisli箭头的形式 a -> (s -> (b, s))，其中修饰被抽象为 State函子：

```
newtype State s a = State (s -> (a, s))
```

同样，我们可以将Kleisli箭头视为返回一个动作，可以使用辅助函数来执行：

```
runState :: State s a -> s -> (a, s)
runState (State f) s = f s
```

不同的初始状态不仅可能产生不同的结果，还可能产生不同的最终状态。

对于 Statemonad的bind实现与 Readermonad的实现非常相似，只是在每一步中需要注意传递正确的状态：

```
  sa >>= k = State (\s -> let (a, s') = runState sa s
                                                sb = k a
                                            in runState sb s')
```

这是完整的实例：

```
instance Monad (State s) where
        sa >>= k = State (\s -> let (a, s') = runState sa s
                                                in runState (k a) s')
        return a = State (\s -> (a, s))
```

还有两个辅助的Kleisli箭头可以用来操作状态。 其中一个用于检查状态：

```
get :: State s s
get = State (\s -> (s, s))
```

另一个用于替换为全新的状态：

```
put :: s -> State s ()
put s' = State (\s -> ((), s'))
```

21.2.6 异常

抛出异常的命令式函数实际上是一个部分函数 - 它对于某些参数值没有定义。 在纯函数的术语中，使用 Maybe函子实现异常的最简单方法。 部分函数被扩展为总函数，当有意义时返回 Just a，当没有意义时返回 Nothing。 如果我们还想返回有关失败原因的一些信息，可以使用 Either 函子（第一个类型固定为 String）。这是 Maybe的 Monad实例：

```
实例Monad Maybe where
  Nothing >>= k = Nothing
  Just a >>= k = k a
  return a = Just a
```

注意，Maybe的单子组合在检测到错误时正确地短路了计算（续传k从未被调用）。 这是我们对异常的预期行为。

21.2.7 连续

这就是你在面试后可能会遇到的“别打电话给我们，我们会打电话给你！”的情况。 与其直接得到答案，你应该提供一个处理程序，一个用于处理结果的函数。 这种编程风格在调用时结果未知时特别有用，例如，它可能正在被另一个线程评估或从远程网站传递。 在这种情况下，Kleisli箭头返回一个接受处理程序的函数，该处理程序代表“剩余的计算”：

```
data Cont r a = Cont ((a -> r) -> r)
```

当处理程序最终被调用时，处理程序 a -> r 会产生类型为 r 的结果，并在最后返回该结果。延续是由结果类型参数化的。 （在实践中，这通常是某种状态指示器。）

还有一个辅助函数用于执行 Kleisli 箭头返回的操作。 它将处理程序传递给延续：

```
 runCont :: Cont r a -> (a -> r) -> r
runCont (Cont k) h = k h
```

延续的组合被公认为非常困难，因此通过一个单子和特别是 do符号来处理它是极其有利的。

让我们来看看 bind 的实现。 首先让我们看一下简化的签名：

```
(>>=) :: ((a -> r) -> r) ->
(a -> (b -> r) -> r) ->
((b -> r) -> r)
```

我们的目标是创建一个函数，它接受处理器 (b -> r)并产生结果 r。 所以这是我们的起点：

```
ka >>= kab = Cont (\hb -> ...)
```

在lambda内部，我们想要使用适当的处理器调用函数 ka，该处理器表示计算的其余部分。 我们将实现这个处理器作为一个lambda表达式：

```
runCont ka (\a -> ...)
```

在这种情况下，计算的其余部分涉及首先调用 kab与 a，然后将 hb传递给结果动作 kb：

```
runCont ka (\a -> let kb = kab a
                  in runCont kb hb)
```

正如你所看到的，延续是从内向外组合的。最终的处理器hb是从计算的最内层调用的。 这是完整的实例：

```
instance Monad (Cont r) where
  ka >>= kab = Cont (\hb -> runCont ka (\a -> runCont (kab a)
   hb))
  返回 a = Cont (\ha -> ha a)
```

21.2.8 交互输入

这是最棘手的问题，也是很多人困惑的根源。 显然，像 getChar这样的函数，如果它返回键盘上键入的字符，就不能是纯函数。 但是，如果它返回一个容器中的字符呢？ 只要没有办法从这个容器中提取字符，我们就可以说这个函数是纯函数。 每次调用 getChar时，它都会返回完全相同的容器。 从概念上讲，这个容器将包含所有可能的字符的叠加。

如果你熟悉量子力学，你应该没有问题理解这个类比。 就像盒子里有薛定谔的猫一样 - 只是没有办法打开或窥视盒子。这个盒子是使用特殊的内置 IO函子定义的。 在我们的例子中，getChar可以被声明为一个Kleisli箭头：

```
getChar :: () -> IO Char
```

（实际上，由于从单位类型到返回类型的函数等价于选择一个返回类型的值，因此 getChar的声明简化为
getChar :: IO Char。）

作为一个函子， IO允许您使用 fmap来操作其内容。 而且，作为一个函子，它可以存储任何类型的内容，而不仅仅是字符。
这种方法的真正实用性在于，在Haskell中， IO是一个单子。 这意味着您可以组合生成 IO对象的Kleisli箭头。

您可能会认为Kleisli组合允许您查看 IO对象的内容（因此，如果我们继续使用量子类比，它会“折叠波函数”）。 确实，您可以将getChar与另一个接受字符并将其转换为整数的Kleisli箭头组合。 但问题在于，这个第二个Kleisli箭头可能

只返回这个整数作为一个 (IO Int)。 同样，你最终会得到所有可能整数的叠加。等等。薛定谔的猫永远不会出来。一旦你进入 IO单子，就没有办法出来了。在 IO单子中没有类似于runState或runReader的等价物。没有 runIO！

那么你可以用Kleisli箭头的结果，即 IO对象，做些什么呢？除了将其与另一个Kleisli箭头组合起来？ 嗯，你可以从 main中返回它。 在Haskell中， main的签名是：

```
main :: IO ()
```

你可以自由地将其视为Kleisli箭头：

```
main :: () -> IO ()
```

从这个角度来看，一个Haskell程序只是一个在 IO单子中的大型Kleisli箭头。 你可以使用单子组合从较小的Kleisli箭头组合它。 运行时系统会对生成的 IO对象（也称为 IO操作）做一些处理。

请注意，箭头本身是一个纯函数 - 它一直是纯函数。 肮脏的工作被委托给系统。 当它最终执行从主函数返回的IO操作时，它会执行各种恶心的事情，比如读取用户输入，修改文件，打印令人讨厌的消息，格式化磁盘等等。Haskell程序从不弄脏它的手（嗯，除非它调用unsafePerformIO，但那是另外一回事）。

当然，因为Haskell是惰性的，主函数几乎立即返回，肮脏的工作立即开始。 在执行IO操作期间，纯计算的结果会按需求被请求和评估。 因此，实际上，程序的执行是纯（Haskell）和肮脏（系统）代码的交错。

IO单子还有一种更奇怪但作为数学模型完全合理的替代解释。 它将整个宇宙视为程序中的一个对象。 请注意，从概念上讲，命令式模型将宇宙视为一个外部全局对象，因此执行I/O的过程通过与该对象进行交互而具有副作用。 它们既可以读取也可以修改宇宙的状态。

我们已经知道如何在函数式编程中处理状态- 我们使用状态单子。 然而，与简单的状态不同，宇宙的状态不能轻易地使用标准数据结构来描述。
但是，只要我们从不直接与它交互，我们就不需要这样做。我们只需要假设存在一种类型 RealWorld，并且通过某种宇宙工程的奇迹，运行时能够提供这种类型的对象。 一个 IO操作只是一个函数：

```
类型 IO a = RealWorld -> (a, RealWorld)
```

或者，用 State单子来表示：

```
类型 IO = State RealWorld
```

然而， >==>和 return对于 IO单子必须内置到语言中。

21.2.9 交互输出

同样的 IO单子也用于封装交互式输出。RealWorld应该包含所有输出设备。 你可能想知道为什么我们不能只是从Haskell中调用输出函数并假装它们什么都不做。 例如，为什么我们有：

```
putStr :: String -> IO ()
```

而不是更简单的：

```
putStr :: String -> ()
```

有两个原因：Haskell是惰性的，所以它永远不会调用一个函数，其输出——在这里是单位对象——没有被用于任何事情。 而且，即使它不是惰性的，它仍然可以自由地改变这些调用的顺序，从而破坏输出。 在Haskell中，强制两个函数的顺序执行的唯一方法是通过数据依赖。 一个函数的输入必须依赖于另一个函数的输出。 在 IO操作之间传递 RealWorld强制顺序执行。

从概念上讲，在这个程序中：

```
main :: IO ()
main = do
    putStr "Hello "
    putStr "World!"
```

打印“World!”的操作接收到作为输入的宇宙，其中“Hello ”已经显示在屏幕上。 它输出一个新的宇宙，屏幕上显示着“Hello World!”。

21.3 结论

当然，我只是浅尝辄止地介绍了单子编程的表面。
单子不仅可以通过纯函数实现在命令式编程中通常通过副作用完成的任务，而且还可以以高度的控制和类型安全性来完成。 然而，它们并非没有缺点。 关于单子的主要抱怨是它们不容易相互组合。 当然，你可以使用单子变换器库组合大多数基本单子。这相对来说比较容易。

可以轻松创建一个将状态与异常组合的单子堆栈，但是没有一种公式可以将任意单子堆叠在一起。

22

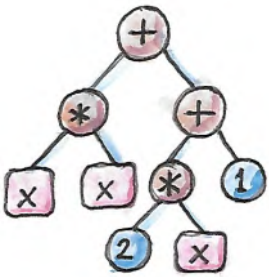
范畴论中的单子

如果你向程序员提到单子，你可能会谈到效果。 对于数学家来说，单子是关于代数的。
我们稍后会讨论代数——它们在编程中起着重要的作用——但首先我想给你一点关于它们与单子的关系的直觉。 目前来说，这是一个有点模糊的论证，但请耐心等待。

代数是关于创建、操作和评估表达式的。
表达式是使用运算符构建的。 考虑这个简单的表达式：

$$□^{2+2}□ + 1$$

这个表达式是使用变量如 □和常量如 1或2，以及加法或乘法等运算符绑定在一起形成的。 作为程序员，我们经常将表达式视为树。



树是容器，因此更一般地说，表达式是用于存储变量的容器。 在范畴论中，我们将容器表示为自函子。 如果我们将类型 A分配给变量 X，那么我们的表达式将具有类型$F\ X$，其中 F是构建表达式树的自函子。 （非平凡的分支表达式通常使用递归定义的自函子创建。）

可以对表达式执行的最常见操作是什么？ 它是替换：用表达式替换变量。 例如，在我们的示例中，我们可以用$X - 1$替换 X，得到：

$$(X - 1)^{2+2}(X - 1) + 1$$

这是发生的事情：我们拿到了一个类型为$F\ X$的表达式，并应用了一个类型为$X \to F\ X$的转换（F代表 T的类型）。 结果是一个类型为$F\ X$的表达式。让我详细解释一下：

$$F\ X \to (X \to F\ X) \to F\ X$$

是的，那就是单子绑定的签名。

这是一点动力。 现在让我们来看一下单子的数学。
数学家和程序员使用不同的符号表示法。 他们更喜欢使用字母 T表示自函子，并使用希腊字母: μ表示 join和

□表示 return。 join和 return都是多态函数，所以我们可以猜测它们对应于自然变换。

因此，在范畴论中，一个单子是指一个带有一对自然变换 □和 □的自函子□。

□是从函子 □的平方到 □的一个自然变换。 平方就是函子与自身的组合，□ ◦ □（我们只能对自函子进行这种平方操作）。

$$□ :: □^{2} \to □$$

这个自然变换在对象 □上的分量是态射：

$$□ \ □ :: □ \ (□ \ □) \to □ \ □$$

在 Hask中，它直接对应于我们对 join的定义。

□是从恒等函子 □到 □的一个自然变换：

$$□ :: □ \to □$$

考虑到 □对对象 □的作用就是 □，这个自然变换的分量由态射给出：

$$□ \ □ :: □ \to □ \ □$$

它直接对应于我们对 return的定义。

这些自然变换必须满足一些额外的法则。

从另一个角度来看，这些法则允许我们为自函子 □定义一个Kleisli范畴。 记住，一个从 □到 □的Kleisli箭头被定义为一个态射□ → □ □。 两个这样的箭头的组合（我将其写为带有下标 □的圆圈）可以使用□来实现：

$$□ \ ◦□ \ □ = □ \ □ ◦ \ (□ \ □) \ ◦ \ □$$

哪里

$$\square :: \square \to \square\ \square$$
$$\square :: \square \to \square\ \square$$

在这里 □，作为一个函子，可以应用于态射 □。 在Haskell符号中，可能更容易识别这个公式：

```
f >=> g = join . fmap g . f
```

或者，按组件表示：

```
(f >=> g) a = join (fmap g (f a))
```

从代数解释的角度来看，我们只是组合了两个连续的替换。

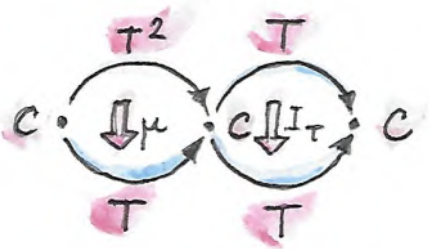
为了使Kleisli箭头形成一个范畴，我们希望它们的组合是结合的，并且 □□是 □处的单位Kleisli箭头。 这需要对 □和 □进行单子律的翻译。 但是还有另一种推导这些定律的方法，使它们看起来更像幺半群定律。
实际上， □通常被称为乘法，而 □被称为单位。

粗略地说，结合律规定了将立方 □的两种方式，即 $\square^3$，缩减为 □必须给出相同的结果。 两个单位律（左和右）规定了当 □应用于 □然后通过 □缩减时，我们得到回 □。

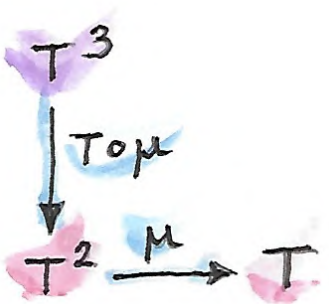
事情有点棘手，因为我们正在组合自然变换和函子。 因此，有必要对水平组合进行一点复习。 例如， $\square^3$可以看作是 □之后 $\square^2$的组合。
我们可以将两个自然变换的水平组合应用于它：

$$\square\ \square \circ \square$$

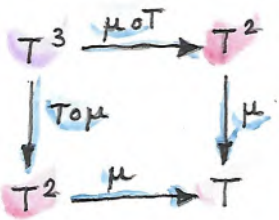
并得到□ ◦ □；然后通过应用□进一步缩减为 □。□□是从□到□的恒等自然变换。 你经常会看到这种类型的水平组合的符号表示为 □□ ◦ □缩写为□ ◦□。这种表示法是一致的。



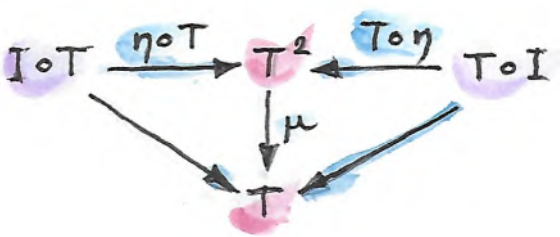
这是含糊不清的，因为在这个上下文中，将一个函子与一个自然变换进行组合没有意义，因此 □必须表示 □□。
我们还可以在（自）函子范畴 [□, □]中绘制图表：



或者，我们可以将 □3视为 □2◦ □的组合，并对其应用□ ◦ □。结果也是□ ◦ □，再次可以使用 □来简化为 □。 我们要求这两条路径产生相同的结果。



同样地，我们可以将水平组合□∘□应用于标识函子 □之后的 □的组合，从而获得 □2，然后可以使用 □来简化。 结果应该与直接将标识自然变换应用于 T相同。 类似地，对于□∘□，也应该是如此。



你可以自己验证这些法则确保Kleisli箭头的组合确实满足范畴的法则。

单子和幺半群之间的相似之处令人惊讶。 我们有乘法 □，单位□，结合律和单位律。 但是我们对幺半群的定义太狭窄，无法将单子描述为幺半群。 所以让我们将幺半群的概念泛化。

22.1 幺半范畴

让我们回到幺半群的传统定义。 它是一个带有二元操作和一个特殊元素称为单位元的集合。 在Haskell中，这可以表示为一个类型类：

```
class Monoid m where
      mappend :: m -> m -> m
      mempty :: m
```

二元操作 mappend必须是结合的和有单位元的（即，单位元 mempty的乘法是一个无操作）。

注意，在Haskell中， mappend的定义是柯里化的。 它可以被解释为将 m的每个元素映射到一个函数：

```
mappend :: m -> (m -> m)
```

正是这种解释引出了单子范畴的定义，其中自同态 (m -> m)表示单子的元素。 但是由于柯里化已经内置到Haskell中，我们也可以从不同的乘法定义开始：

```
mu :: (m, m) -> m
```

在这里，笛卡尔积 (m, m)成为要相乘的对的来源。

这个定义提出了一种不同的泛化路径：用范畴积替换笛卡尔积。 我们可以从一个全局定义了积的范畴开始，选择一个对象 m，并将乘法定义为一个态射：

$$\mu :: m \times m \to m$$

然而，我们有一个问题：在任意范畴中，我们无法查看对象的内部，那么如何选择单位元素？有一个技巧可以解决。还记得如何通过函数从单例集合中选择元素吗？ 在Haskell中，我们可以用一个函数替换 mempty的定义：

```
eta :: () -> m
```

单子集是 □ □ 中的终对象，因此自然地将这个定义推广到任何具有终对象 □的范畴：

$$□ : : □ \to □$$

这样我们就可以选择单位“元素”而不必谈论元素。

与我们之前将幺半群定义为单对象范畴不同，这里的幺半群定律不会自动满足——我们必须加以约束。 但是为了制定这些定律，我们必须先建立底层范畴积的幺半结构。 让我们首先回顾一下Haskell中的幺半结构是如何工作的。

我们从结合性开始。 在Haskell中，相应的等式定律是：

```
mu (x, mu (y, z)) = mu (mu (x, y), z)
```

在我们将其推广到其他范畴之前，我们必须将其重写为函数（态射）的等式。 我们必须将其从对个别变量的作用中抽象出来——换句话说，我们必须使用无点符号表示法。 由于笛卡尔积是一个双函子，我们可以将左边写成：

```
(mu . bimap id mu)(x, (y, z))
```

以及右手边为:

```
(mu . bimap mu id)((x, y), z)
```

这几乎是我们想要的。 不幸的是，笛卡尔积
不是严格的结合的 — `(x, (y, z))` 不同于 `((x, y), z)` —
所以我们不能只写成无点的形式:

```
mu . bimap id mu = mu . bimap mu id
```

另一方面，两个嵌套的对是同构的。 有一个
可逆函数称为结合器，可以在它们之间进行转换:

```
alpha :: ((a, b), c) -> (a, (b, c))
alpha ((x, y), z) = (x, (y, z))
```

借助结合器的帮助，我们可以写出无点形式的结合律
对于 `mu`:

```
mu . bimap id mu . alpha = mu . bimap mu id
```

我们可以对单位律应用类似的技巧，新的表示法中，
形式如下:

```
mu (eta (), x) = x
mu (x, eta ()) = x
```

它们可以被重写为：

```
(mu . bimap eta id) ((), x) = lambda((), x)
(mu . bimap id eta) (x, ()) = rho (x, ())
```

等同态 lambda和 rho被称为左单位元和右单位元，分别。 它们证明了单位元 ()是笛卡尔积的恒等元素，直到同构为止：

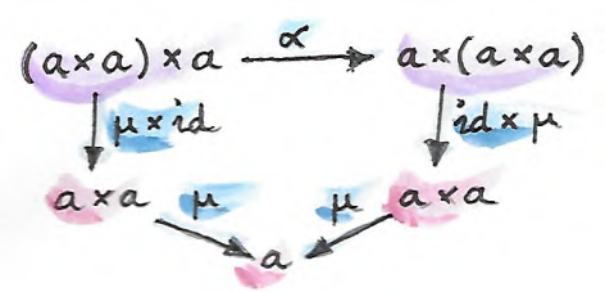
```
lambda :: ((), a) -> a
lambda ((), x) = x

rho :: (a, ()) -> a
rho (x, ()) = x
```

因此，单位元法则的无点版本是：

```
mu . bimap id eta = rho
mu . bimap eta id = lambda
```

我们已经使用底层的笛卡尔积本身在类型的范畴中充当幺半群乘法的事实，为 mu和 eta制定了无点幺半群法则。 但请记住，笛卡尔积的结合律和单位元法则只在同构的情况下有效。

事实证明，这些法则可以推广到具有积和终端对象的任何范畴。 范畴积确实是同构的结合律，而终端对象是单位元，也是同构的。 结合子和两个单位元是自然同构。 这些法则可以用交换图表示。



请注意，因为乘积是一个双函子，它可以提升一对态射 - 在Haskell中使用 bimap来实现。

我们可以在任何具有范畴乘积和终对象的范畴上定义一个幺半群，到此为止就可以了。 只要我们可以选择一个对象 □和两个态射 □和 □满足幺半群的定律，我们就有一个幺半群。 但我们可以做得更好。 我们不需要一个完整的范畴乘积来为 □和 □制定定律。 回想一下，乘积是通过使用投影来定义的一个普遍构造。 在我们的幺半群定律的公式中，我们没有使用任何投影。

一个行为类似于乘积但不是乘积的双函子被称为张量积，通常用中缀运算符 ⊗表示。 一般情况下，张量积的定义有点棘手，但我们不用担心。我们只列出它的性质 - 最重要的是同构的结合性。

同样地，我们不需要对象 □是终对象。我们从未使用过它的终对象属性——即，从任何对象到它的唯一态射的存在性。我们所要求的是它在张量积中能够很好地协同工作。 这意味着我们希望它成为张量积的单位元，同样地，这是同构的。 让我们把它们都放在一起：

一个融合范畴是一个配备了一个双函子的范畴 ，该双函子被称为张量积：

$$\otimes : : \times \to$$

以及一个不同的对象 ，被称为单位对象，还有三个分别称为结合子和左右单位子的自然同构：

$$: (\otimes)\otimes \to \otimes (\otimes)$$
$$: : \otimes \to$$
$$: : \otimes \to$$

（还有一个简化四重张量积的一致性条件。）

重要的是，张量积描述了许多熟悉的双函子。 特别地，它适用于乘积、余积，正如我们很快将看到的，它还适用于自函子的复合（以及一些更奇特的产品，如Day卷积）。 融合范畴在丰富范畴的制定中将发挥重要作用。

22.2 幺半范畴中的幺半

我们现在准备在更一般的单子范畴中定义一个幺半群。 我们从选择一个对象 开始。 使用张量积，我们可以形成 的幂。 平方 是 ⊗ 。 形成立方体 有两种方式，但它们通过结合子同构。 类似地，对于 的更高幂次（这就是我们需要的相干条件）。 要形成一个幺半群，我们需要选择两个态射-

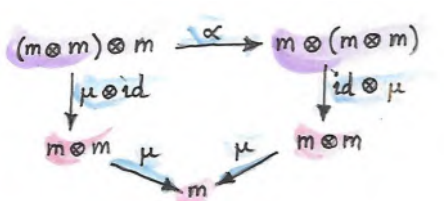
态射:

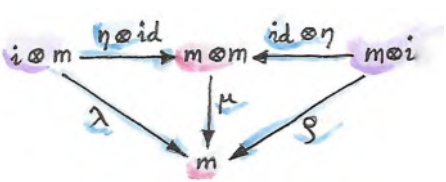
$$\square :: \square \otimes \square \to \square$$

$$\square :: \square \to \square$$

其中 $\square$是我们张量积的单位对象。

这些态射必须满足结合律和单位律，可以用以下交换图表来表示：





注意张量积必须是一个双函子，因为我们需要提升态射对来形成像□ ⊗ id或□ ⊗ id这样的积。 这些图表只是我们对范畴积的直接推广。

22.3 单子作为幺半

单调结构出现在意想不到的地方。 其中一个地方就是函子范畴。 如果你眯眼一点，你可能会看到函子组合是一种乘法形式。 问题在于不是任意两个函子都可以组合 - 一个的目标范畴必须是另一个的源范畴。 这只是态射组合的常规规则 - 如我们所知，函子确实是范畴 □ □ 中的态射。 但就像自同态（回到同一对象的态射）总是可组合一样，自函子也是可组合的。对于给定的范畴 □，从 □到 □的自函子形成了函子范畴 [□, □]。 它的对象是自函子，态射是它们之间的自然变换。 我们可以从这个范畴中取任意两个对象，比如自函子 □和 □，并产生第三个对象□ ◦ □ - 一个是它们的组合的自函子。

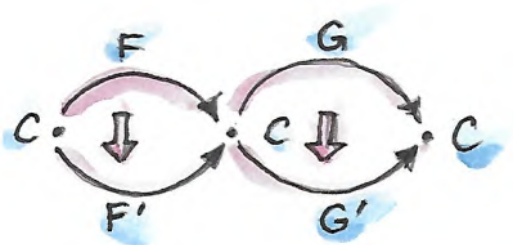
端函子组合是否是张量积的一个好选择？
首先，我们必须确定它是一个双函子。 它能用来提升一对态射吗 — 这里指的是自然变换？ 张量积的类似于bimap的签名看起来像这样：

$$□ □ □ □ : □ (□ → □) → (□ → □) → (□ ⊗ □ → □ ⊗ □)$$

如果你用端函子替换对象，用自然变换替换箭头，用组合替换张量积，你会得到：

$$(\mathcal{C} \to \mathcal{C}') \to (\mathcal{C} \to \mathcal{C}') \to (\mathcal{C} \circ \mathcal{C} \to \mathcal{C}' \circ \mathcal{C}')$$

你可能会认出这是水平组合的特殊情况。

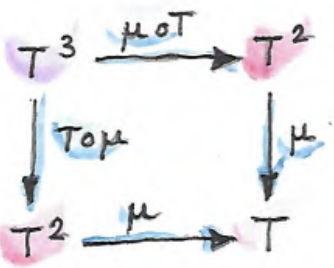


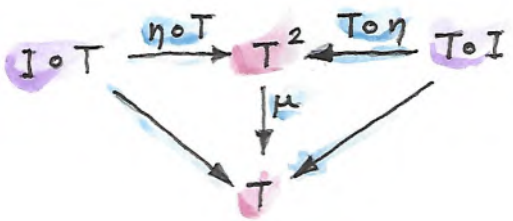
我们还可以使用恒等端函子 $\mathcal{I}$，它可以作为端函子组合的恒等元素 — 我们的新张量积。此外，函子组合是可结合的。 事实上，结合律和单位元法则是严格的 — 不需要结合子或两个单位元。 因此，端函子构成了一个严格的幺半范畴，其中函子组合作为张量积。

在这个范畴中，什么是幺半群？ 它是一个对象 — 也就是一个自函子 $\mathcal{M}$；以及两个态射 — 也就是自然变换：

$$\mathcal{M} \circ \mathcal{M} \to \mathcal{M}$$
$$\mathcal{I} \to \mathcal{M}$$

不仅如此，这里还有幺半群定律：





它们正是我们之前见过的单子定律。 现在你理解了Saunders Mac Lane的著名引语：

> 总的来说，单子就是在自函子范畴中的幺半群。

你可能在函数式编程会议上的一些T恤上见过它的标志。

22.4 从伴随构造单子

一个伴随[1]□ □ □，是在两个范畴 □和 □之间来回进行的一对函子。有两种组合它们的方式，产生了两个自函子，□ ◦ □和□ ◦ □。 根据一个伴随，这些自函子通过两个自然变换与恒等函子相关联，这两个自然变换被称为单位和余单位：

$$□ : : □ □ \rightarrow □ \circ □$$

$$□ : : □ \circ □ \rightarrow □ □$$

立即我们看到一个伴随的单位看起来就像一个单子的单位。 事实证明，自函子□ ◦ □确实是一个单子。
我们所需要的是定义适当的μ与η相配。 这是我们的自函子的平方与自函子之间的一个自然变换，或者用伴随函子的术语来说：

$$□ \circ □ \circ □ \circ □ \rightarrow □ \circ □$$

实际上，我们可以使用余单位将□ ◦ □在中间折叠。
□的确切公式由水平组合给出：

$$□ = □ \circ □ \circ □$$

单子定律来自于伴随函子的单位和余单位的满足的恒等式以及交换律。

在Haskell中，我们不经常看到从伴随函子派生的单子，因为伴随函子通常涉及两个范畴。然而，

1参见关于伴随的第18章。

指数的定义，或者函数对象，是一个例外。 这里是形成这个伴随的两个自函子：

$$□\ □ = □ \times □$$

$$□\ □ = □ \Rightarrow □$$

你可能会认出它们的组合是熟悉的状态单子：

$$□\ (□\ □) = □ \Rightarrow (□ \times □)$$

我们之前在Haskell中见过这个单子：

```
newtype State s a = State (s -> (a, s))
```

我们还要将伴随转换为Haskell。左函子是乘积函子：

```
新类型 Prod s a = Prod (a, s)
```

右函子是读取器函子：

```
新类型 Reader s a = Reader (s -> a)
```

它们形成了伴随：

```
instance Adjunction (Prod s) (Reader s) where
    counit (Prod (Reader f, s)) = f s
    unit a = Reader (\s -> Prod (a, s))
```

你可以很容易地相信，读取器函子在乘积函子之后的组合确实等同于状态函-子：

```
newtype State s a = State (s -> (a, s))
```

正如预期的那样，伴随的 unit等同于状态单子的 return函数。 伴随的 counit通过对其参数求值来起作用。 这可以识别为函数 runState的非柯里化版本：

```
runState :: State s a -> s -> (a, s)
runState (State f) s = f s
```

（非柯里化，因为在 counit 中它作用于一对）。

现在我们可以将状态单子的 join定义为自然变换 □的一个组成部分。 为此，我们需要三个自然变换的水平组合：

$$\square = \square \circ \square \circ \square$$

换句话说，我们需要将counit □悄悄地穿过一个层次的reader函子。 我们不能直接调用 fmap，因为编译器会选择 State函子的版本，而不是 Reader函子的版本。但是请记住，reader函子的 fmap只是左函数组合。 所以我们将直接使用函数组合。

我们必须首先剥离数据构造器 State，以暴露 State函子内部的函数。 这可以通过 runState来完成：

```
 ssa :: State s (State s a)
runState ssa :: s -> (State s a, s)
```

然后我们将其与counit进行左组合，counit由 uncurry runState定义。最后，我们将其重新包装在 State 数据构造器中：

```
join :: State s (State s a) -> State s a
join ssa = State (uncurry runState . runState ssa)
```

这确实是 State monad的 join的实现。

事实证明，不仅每个伴随给出一个monad，而且反过来也成立：每个monad都可以分解为两个伴随函子的组合。 但是这种分解并不唯一。

我们将在下一节讨论另一个自函子□ ∘ □。

23

余单子

现在我们已经讲解了单子，我们可以通过反转箭头并在相反的范畴中工作来免费获得余单子的好处。

回想一下，在最基本的层面上，单子是关于组合Kleisli箭头的：

```
a -> m b
```

其中 m是一个作为单子的函子。 如果我们使用字母 w （倒过来对于共函子的m)，我们可以将co-Kleisli箭头定义为类型的态射：

```
w a -> b
```

对于共Kleisli箭头的鱼算子的类比定义如下：

```
(=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
```

为了使共Kleisli箭头形成一个范畴，我们还必须有一个身份共Kleisli箭头，称为提取

```
提取 :: w a -> a
```

这是 return的对偶。 我们还必须强加结合性法则以及左右身份法则。 将所有这些放在一起，我们可以在Haskell中定义一个共函子：

```
class Functor w => Comonad w where
    (=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
    extract :: w a -> a
```

在实践中，我们使用稍微不同的原语，稍后我们将看到。

问题是，共函子在编程中有什么用途?

23.1 使用余单子进行编程

让我们比较单子与余单子。 单子通过 return提供了将值放入容器的方式。 它不提供访问存储在内部的值的方式。 当然，实现单子的数据结构可能提供访问其内容的方式，但这被视为额外的功能。 没有提取单子中值的通用接口。 我们已经看到了 IO单子的例子，它以不暴露其内容为傲。

另一方面，余单子提供了从中提取单个值的方法。它不提供插入值的方法。 因此，如果你想将余单子视为容器，它总是预先填充了内容，并允许你窥视它。

正如Kleisli箭头接受一个值并产生一些装饰的结果 - 它用上下文装饰它 - co-Kleisli箭头接受一个值

与整个上下文一起产生结果。 它是上下文计算的体现。

23.2 乘积余单子

还记得读者单子吗？ 我们引入它来解决需要访问一些只读环境的计算实现问题。 这样的计算可以表示为纯函数的形式：

```
(a, e) -> b
```

我们使用柯里化将它们转换为Kleisli箭头：

```
a -> (e -> b)
```

但请注意，这些函数已经具有共Kleisli箭头的形式。 让我们将它们的参数转换为更方便的函子形式：

```
data Product e a = P e a deriving Functor
```

我们可以通过使相同的环境对箭头可用来定义组合运算符：

```
(=>=) :: (Product e a -> b) -> (Product e b -> c) -> (Product e
 □          a -> c)
f =>= g = \(P e a) -> let b = f (P e a)
                                        c = g (P e b)
                                in c
```

提取的实现简单地忽略了环境：

提取 (P e a) = a

毫不奇怪，乘积余子范畴可以用于执行与读者单子完全相同的计算。 在某种程度上，环境的余子范畴实现更自然 - 它遵循“上下文中的计算”的精神。另一方面，单子带有方便的语法糖，即 do 符号。

读者单子和乘积余子范畴之间的联系更深入，与读者函子是乘积函子的右伴随有关。 总的来说，余子范畴涵盖的计算概念与单子不同。 我们稍后会看到更多的例子。

很容易将 Product余子范畴推广到包括元组和记录在内的任意乘积类型。

23.3 解剖组合

继续对偶化过程，我们可以继续对偶化单子的绑定和合并。 或者，我们可以重复我们在单子中使用的过程，其中我们研究了鱼算子的解剖学。 这种方法似乎更加启发人。

起点是意识到组合运算符必须产生一个接受 w a并产生一个 c的共Kleisli箭头。 产生 c的唯一方法是将第二个函数应用于类型为 w b的参数:

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

但是我们如何产生一个类型为 w b的值，可以传递给 g呢？ 我们可以使用类型为 w a的参数和函数

f :: w a -> b来解决这个问题。 解决方案是定义bind的对偶，称为extend:

```
extend :: (w a -> b) -> w a -> w b
```

使用 extend我们可以实现组合:

```
f =>= g = g . extend f
```

接下来我们可以解剖 extend吗？ 你可能会说，为什么不只是将函数 w a -> b应用于参数 w a，但是你很快意识到你没有办法将结果 b转换为 w b。 记住，共函子没有提供提升值的方法。 在这一点上，对于单子的类似构造，我们使用了 fmap。 我们唯一能在这里使用 fmap的方法是如果我们有类型为 w (w a)的东西。如果我们只能将 w a转换为

w (w a). 而且，方便地，那将恰好是 join的对偶。我们
我们称之为 duplicate：

```
duplicate :: w a -> w (w a)
```

所以，就像单子的定义一样，我们有三个等价的共单子定义：使用共Kleisli箭头， extend，或者duplicate。 这是直接从Haskell定义中获取的
Control.Comonad库：

```
class Functor w => Comonad w where
    extract :: w a -> a
    duplicate :: w a -> w (w a)
    duplicate = extend id
    extend :: (w a -> b) -> w a -> w b
    extend f = fmap f . duplicate
```

提供了 extend的默认实现，以及 duplicate与之相反，所以你只需要覆盖其中一个。

这些函数背后的直觉是基于这样一个想法：一般来说，共单子可以被看作是一个填充有类型为 a的值的容器（乘积共单子只是一个值的特殊情况）。 有一个“当前”值的概念，可以通过extract轻松访问。 共Kleisli箭头执行一些计算，专注于当前值，但它可以访问所有周围的值。

想象康威的生命游戏。 每个细胞包含一个值（通常是 True或 False）。 与生命游戏对应的共函子是一个以“当前”细胞为焦点的细胞网格。

那么duplicate函数是做什么的？ 它接受一个共函子容器 w a并生成一个容器的容器 w (w a)。 这些容器的每一个都以不同的 a为焦点在 w a内。 在生命游戏中，你会得到一个网格的网格，外部网格的每个细胞都包含一个以不同细胞为焦点的内部网格。

现在看一下 extend。 它接受一个共Kleisli箭头和一个共函子容器 w a，其中填充了 aS。它将计算应用于所有这些 aS，用 bS替换它们。结果是一个填充了 bS的共函子容器。 extend通过将焦点从一个 a转移到另一个a，并依次对每个焦点应用共Kleisli箭头来实现。 在生命游戏中，共Kleisli箭头将计算当前细胞的新状态。 为了实现这一点，它会查看其上下文 - 可能是最近的邻居。
默认实现 extend说明了这个过程。 首先，我们调用 duplicate来生成所有可能的焦点，然后我们对每个焦点应用 f。

23.4 流余单子

将焦点从容器的一个元素转移到另一个元素的过程最好通过无限流的例子来说明。
这样的流就像一个列表，只是它没有空的构造函数：

```
data Stream a = Cons a (Stream a)
```

它是一个简单的 Functor：

```
instance Functor Stream where
        fmap f (Cons a as) = Cons (f a) (fmap f as)
```

流的焦点是它的第一个元素，所以这是实现的 extract：

```
extract (Cons a _) = a
```

duplicate生成一个流的流，每个流都以不同的元素为焦点。

```
duplicate (Cons a as) = Cons (Cons a as) (duplicate as)
```

第一个元素是原始流，第二个元素是原始流的尾部，第三个元素是它的尾部，依此类推，无限循环。
这是完整的实例：

```
实例 Comonad Stream where
  extract (Cons a _) = a
  duplicate (Cons a as) = Cons (Cons a as) (duplicate as)
```

这是一种非常函数式的看待流的方式。 在命令式语言中，我们可能会从一个方法 advance开始，将流向前移动一个位置。 在这里，duplicate一下就可以生成所有的移动后的流。 Haskell的惰性使这成为可能，甚至是可取的。 当然，为了使 Stream实用，我们还需要实现 advance的类似物：

```
tail :: Stream a -> Stream a
tail (Cons a as) = as
```

但它从来不是共范畴接口的一部分。

如果你有数字信号处理的经验，你会立即看到流的共Kleisli箭头只是一个数字滤波器，并且 extend生成一个经过滤波的流。

作为一个简单的例子，让我们实现移动平均滤波器。
这是一个将流的 n 个元素求和的函数：

```
sumS :: Num a => Int -> Stream a -> a
sumS n (Cons a as) = if n <= 0 then 0 else a + sumS (n - 1) as
```

这是一个计算流的前n个元素的平均值的函数：

```
average :: Fractional a => Int -> Stream a -> a
average n stm = (sumS n stm) / (fromIntegral n)
```

部分应用的 average n是一个共Kleisli箭头，所以我们可以将其扩展到整个流上：

```
movingAvg :: Fractional a => Int -> Stream a -> Stream a
movingAvg n = extend (average n)
```

结果是运行平均值的流。

流是单向、一维共单子的一个例子。
它可以很容易地变成双向的，或者扩展到两个或更多维度。

23.5 范畴论中的余单子

在范畴论中定义一个共单子是一个简单的对偶练习。 与单子一样，我们从一个自函子 T开始。 定义单子的两个自然变换，η和μ，在共单子中只需反转：

$$\epsilon :: F \to I$$
$$\delta :: F \to F^2$$

这些变换的组成部分对应于提取和复制。 余单子定律是单子定律的镜像。这并不令人意外。

然后从一个伴随导出单子。 对偶将一个伴随反转：左伴随变为右伴随，反之亦然。 而且，由于组合$F \circ G$定义了一个单子，$G \circ F$必须定义一个余单子。 伴随的余单位：

$$\epsilon :: F \circ G \to I$$

确实是我们在余单子的定义中看到的相同的ε — 或者，以Haskell的提取的组成部分形式。 我们还可以使用伴随的单位：

$$\eta :: I \to G \circ F$$

在$G \circ F$中间插入$F \circ G$并产生$G \circ F \circ F \circ G$。 从$F$定义$F^2$，这样就完成了余单子的定义。

我们还看到了单子是一个幺半群。 这个语句的对偶需要使用一个余幺半群，那么什么是余幺半群？ 作为一个单一对象范畴的原始定义并不能推导出任何有趣的结果。 当你反转所有自同态的方向时，你得到另一个幺半群。 然而，请记住，在我们对单子的方法中，我们使用了一个更一般的幺半群定义，即幺半群是幺半群范畴中的一个对象。 这个构造是基于两个态射：

$$\square :: \square \otimes \square \to \square$$
$$\square :: \square \to \square$$

这些态射的反转在幺半群范畴中产生了一个余幺半群：

$$\square :: \square \to \square \otimes \square$$
$$\square :: \square \to \square$$

人们可以用Haskell写一个余幺半群的定义：

```
class Comonoid m where
    split :: m -> (m, m)
    destroy :: m -> ()
```

但这相当琐碎。显然 destroy忽略了它的参数。

```
destroy _ = ()
```

split只是一对函数：

```
split x = (f x, g x)
```

现在考虑对偶于幺半群单位律的余幺半群律。

```
lambda . bimap destroy id . split = id
rho . bimap id destroy . split = id
```

在这里， lambda和 rho分别是左单位器和右单位器（参见幺半范畴的定义）。 将定义代入，我们得到：

```
lambda (bimap destroy id (split x))
= lambda (bimap destroy id (f x, g x))
= lambda ((), g x)
= g x
```

这证明了 g = id。 类似地，第二个律展开为 f = id。
总结起来：

```
split x = (x, x)
```

这表明在Haskell中（以及一般情况下，在范畴 □ □ 中），每个对象都是一个平凡的余幺半群。

幸运的是，还有其他更有趣的幺半范畴可以定义余幺半群。 其中之一是自函子范畴。 而事实证明，就像单子是自函子范畴中的幺半群一样，

> 共单子是自函-
> tors范畴中的共单子

23.6 存储余单子

另一个重要的共单子的例子是状态单子的对偶
它被称为余态共单子或者存储共单子

我们之前已经看到状态单子是由定义指数的伴随-
tion生成的

$$\square\ \square = \square \times \square$$

$$\square\ \square = \square \Rightarrow \square$$

我们将使用相同的伴随定义余态共单子 一个共单子由组合□。□定义：

$$\square\ (\square\ \square) = (\square \Rightarrow \square) \times \square$$

将其翻译成Haskell，我们从左边的Prod函子和右边的Reader函子之间的伴随开始 在 Reader之后组合Prod等同于以下定义：

```
数据类型 Store s a = Store (s -> a) s
```

在对象 □处取伴随的余单位是态射：

$$\square\ \square :: ((\square \Rightarrow \square) \times \square) \to \square$$

或者，用Haskell表示：

```
counit (Prod (Reader f, s)) = f s
```

这变成了我们的 提取 ：

```
extract (Store f s) = f s
```

伴随的单位:

```
unit a = Reader (\s -> Prod (a, s))
```

可以重写为部分应用的数据构造函数:

```
Store f :: s -> Store f s
```

我们构造 □, 或者 duplicate, 作为水平组合:

$$□ :: □ ∘ □ → □ ∘ □ ∘ □ ∘ □$$
$$□ = □ ∘ □ ∘ □$$

我们必须通过最左边的 □偷偷地传递 □, 这是 Prod函子.
这意味着在对偶对的左分量上使用 □, 或者 Store f, (这就是 fmap对 Pr od的作用). 我们得到:

```
duplicate (Store f s) = Store (Store f) s
```

(记住，在 □的公式中， □和 □代表的是分量为恒等态射的恒等自然变换.)这是 Store余单子的完整定义:

```
实例 Comonad (Store s) where
      extract (Store f s) = f s
duplicate (Store f s) = Store (Store f) s
```

你可以将 Reader部分视为通用容器其中的 a是使用类型 s的元素作为键。 例如，如果 s是Int， Reader Int a是一个无限的双向流，其中包含 a的无限个s。 Store将此容器与键类型的值配对。 例如， Reade r Int a与 Int配对。 在这种情况下， extract使用此整数索引到无限流中。 你可以将 Store的第二个组件视为当前位置。

继续使用这个例子， duplicate创建一个新的无限流通过一个 Int进行索引。 这个流包含流作为其元素。 特别地，在当前位置，它包含原始流。但是如果

你使用其他的 Int(正数或负数)作为键，你将获得一个
位移后的流，位于新的索引位置。

一般来说，你可以让自己相信当提取作用于duplicated Store时，它会产生原始的 Store（实际上，共单子的恒等律规定 extract . duplicate = id）。 从概念上讲， Store共单子封装了“聚焦”（就像
镜头）在特定子结构上的思想，使用类型 s作为索引。 特别地，类型为的函数：

```
a -> Store s a
```

等价于一对函数：

```
 set :: a -> s -> a
get :: a -> s
```

如果 a是一个乘积类型， □ □ 可以被实现为设置type s的字段，同时返回修改后的版本 a。 类似地，get可以被实现为从 a中读取 s字段的值。 我们将在下一节中更详细地探讨这些想法。

23.7 挑战

1. 使用 Store共范畴实现康威生命游戏。
   提示：你会选择什么类型作为 s？

24

F-代数

我们已经看到了几种单子的表述方式：作为一个集合，作为一个单一对象的范畴，作为一个范畴中的对象。 我们能从这个简单的概念中挤出多少更多的精华呢？

让我们试试。拿这个作为一个集合 □的单子定义和一对函数：

$$□ :: □ \times □ \to □$$
$$□ :: 1 \to □$$

这里，1是 □ □ 中的终结对象——单元素集合。第一个函数定义了乘法（它接受一对元素并返回它们的乘积），第二个函数从 □中选择单位元素。 并不是每个具有这些签名的两个函数的选择都会得到一个单子。 为此，我们需要加上额外的条件：结合律和单位律。

但是让我们暂时忘记这些，只考虑“潜在的单子”。一对函数是笛卡尔积的一个元素

两组函数。 我们知道这些集合可以表示为指数对象：

$$\square \in \square^{\square \times \square}$$

$$\square \in \square^{1}$$

这两组集合的笛卡尔积是：

$$\square^{\square \times \square} \times \square^{1}$$

使用一些高中代数（在每个笛卡尔闭范畴中都适用），我们可以将其重写为：

$$\square^{\square \times \square + 1}$$

在 □ □中， +符号代表余积。 我们刚刚用一个函数替换了一对函数 - 这是集合的一个元素：

$$\square \times \square + 1 \to \square$$

这组函数的任何元素都是一个潜在的幺半群。

这种表述的美妙之处在于它引导出有趣的一般化。 例如，我们如何用这种语言描述一个群？ 群是一个带有额外函数的幺半群，该函数为每个元素分配其逆元。 后者是类型为$\square \to \square$的函数。 作为一个例子，整数形成一个群，其中加法是二元运算，零是单位元，取反是逆元。 要定义一个群，我们将从三个函数开始：

$$\square \times \square \to \square$$

$$\square \to \square$$

$$1 \to \square$$

与之前一样，我们可以将所有这些三元组合并为一个函数集：

$$\square \times \square + \square + 1 \to \square$$

我们从一个二元运算符（加法）、一个一元运算符（取反）和一个零元运算符（恒等 — 这里是零）开始。 我们将它们合并为一个函数。 具有这个签名的所有函数定义了潜在的群。

我们可以继续这样下去。 例如，要定义一个环，我们需要添加一个二元运算符和一个零元运算符，依此类推。每次我们最终得到的函数类型的左侧是幂的和（可能包括零次幂 — 终结对象），右侧是集合本身。

现在我们可以对泛化进行疯狂的探索。 首先，我们可以用对象替换集合，用态射替换函数。 我们可以将n元运算符定义为从n元积到态射的映射。 这意味着我们需要支持有限积的范畴。 对于零元运算符，我们需要终结对象的存在。 所以我们需要一个笛卡尔范畴。 为了组合这些运算符，我们需要指数，因此这是一个笛卡尔闭范畴。 最后，我们需要余积来完成我们的代数把戏。

或者，我们可以忘记我们推导公式的方式，只专注于最终产品。 我们态射左手边的乘积之和定义了一个自函子。 如果我们选择一个任意的自函子 $\square$呢？ 在这种情况下，我们不需要对我们的范畴施加任何约束。 我们得到的被称为F-代数。

F-代数是由一个自函子 $\square$，一个对象 $\square$和一个态射组成的三元组。

$$\square \ \square \to \square$$

这个对象通常被称为载体，底层对象或者在编程上下文中被称为载体类型。 这个态射通常被称为评估函数或者结构映射。将自函子 □看作形成表达式，将态射看作对它们进行评估。

下面是F-代数的Haskell定义：

```
type Algebra f a = f a -> a
```

它将代数与其评估函数进行了标识。

在幺半群的例子中，所讨论的函子是：

```
数据类型 MonF a = MEmpty | MAppend a a
```

这是 Haskell 中的1 + □ × □（记住代数数据结构）。

使用以下函子来定义一个环：

```
数据类型 RingF a = RZero
                 | ROne
                 | RAdd a a
                 | RMul a a
                 | RNeg a
```

这是 Haskell 中的1 + 1 + □ × □ + □ × □ + □。

环的一个例子是整数集。 我们可以选择整数作为载体类型，并将评估函数定义为：

```
evalZ :: 代数 RingF 整数
evalZ RZero          = 0
evalZ ROne           = 1
evalZ (RAdd m n) = m + n
evalZ (RMul m n) = m * n
evalZ (RNeg n)       = -n
```

基于相同函子的F-代数有更多 RingF。 例如，多项式形成一个环，方阵也是如此。

正如你所看到的，函子的作用是生成可以使用代数的求值器进行求值的表达式。 到目前为止，我们只看到了非常简单的表达式。 我们通常对使用递归定义的更复杂的表达式感兴趣。

24.1 递归

生成任意表达式树的一种方法是用递归替换函子定义中的变量 a。 例如，环中的任意表达式由这个类似树状数据结构生成：

```
data Expr = RZero
          | ROne
          | RAdd Expr Expr
          | RMul Expr Expr
          | RNeg Expr
```

我们可以用递归版本替换原始的环求值器：

```
evalZ :: Expr -> Integer
evalZ RZero          = 0
evalZ ROne           = 1
evalZ (RAdd e1 e2) = evalZ e1 + evalZ e2
evalZ (RMul e1 e2) = evalZ e1 * evalZ e2
evalZ (RNeg e)       = -(evalZ e)
```

这仍然不太实用，因为我们被迫将所有整数表示为一的和，但在紧急情况下可以使用。

但是我们如何用F-代数的语言描述表达式树？ 我们必须以某种方式形式化替换过程

在我们的函子定义中，递归地用替换的结果替换自由类型变量。想象一下分步骤完成这个过程。 首先，定义一个深度为一的树：

```
type RingF1 a = RingF (RingF a)
```

我们正在用由RingF a生成的深度为零的树填充RingF的定义中的空白部分。 深度为2的树可以类似地获得：

```
type RingF2 a = RingF (RingF (RingF a))
```

我们也可以将其写为：

```
type RingF2 a = RingF (RingF1 a)
```

继续这个过程，我们可以写出一个符号方程：

```
type RingFn+1 a = RingF (RingFn a)
```

从概念上讲，在无限次重复这个过程之后，我们最终得到了我们的 Expr。 注意 Expr不依赖于 a。 我们旅程的起点并不重要，我们总是最终到达同一个地方。
对于任意范畴中的任意自函子，这并不总是成立，但在范畴 □ □ □ 中是成立的。

当然，这只是一个模糊的论证，我稍后会更加严谨地阐述。

无限次应用一个自函子会产生一个不动点，即一个被定义为：

□ □ □□ = □ (□ □ □□)

这个定义背后的直觉是，由于我们无限次应用 □来得到□ □ □□，再应用一次不会改变任何东西。 在Haskell中，不动点的定义是：

```
newtype Fix f = Fix (f (Fix f))
```

可以说，如果构造函数的名称与定义的类型名称不同，那么这将更易读，如下所示：

```
newtype Fix f = In (f (Fix f))
```

但我将坚持接受的符号表示法。 构造函数 `Fix`（或 `In`，如果你喜欢）可以看作是一个函数：

```
Fix :: f (Fix f) -> Fix f
```

还有一个函数可以剥离一个函子应用的层级：

```
unFix :: Fix f -> f (Fix f)
unFix (Fix x) = x
```

这两个函数是彼此的逆函数。 我们稍后会用到这些函数。

24.2 F-代数范畴

这是书中最古老的技巧：每当你想出一种构造新对象的方法时，看看它们是否构成一个范畴。 毫不奇怪，给定的自函子 F上的代数构成一个范畴。 该范畴中的对象是代数 —— 由原始范畴 C中的载体对象 A和态射$F A \to A$组成的对。

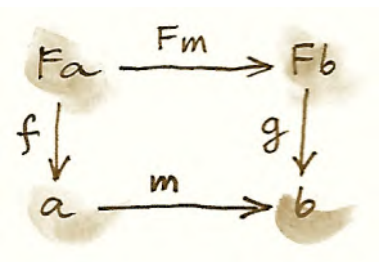
为了完整地描述，我们必须在F-代数的范畴中定义态射。一个态射必须将一个代数 $(A, F A)$映射到另一个代数 $(B, F B)$。 我们将其定义为一个映射 h，它将原始范畴中的载体从 A映射到 B。并不是任何态射都可以

满足要求：我们希望它与两个评估器兼容。 （我们将这样的保结构态射称为同态。）下面是如何定义F-代数的同态。首先，注意我们可以将 □提升到映射：

□ □ : : □ □ → □ □

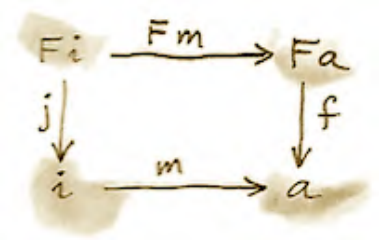
然后我们可以跟随 □以达到 □。 或者，我们可以使用 □从□ □到 □，然后跟随 □。 我们希望这两条路径相等：

□ ◦ □ □ = □ ◦ □



很容易让自己相信这确实是一个范畴（提示：从 □工作的恒等态射很好，同态的组合是同态）。

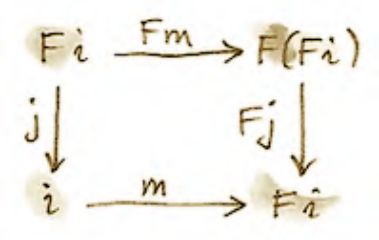
在F-代数范畴中，如果存在一个初始对象，则称之为初始代数。 让我们称这个初始代数的载体为 □，它的求值器为□ : : □ □ → □。 事实证明，初始代数的求值器 □是一个同构。 这个结果被称为Lambek定理。
证明依赖于初始对象的定义，它要求存在一个唯一的同态 □从它到任何其他F-代数。由于 □是一个同态，以下图表必须交换：



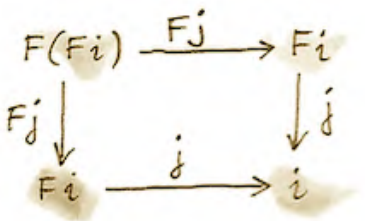
现在让我们构造一个载体为□ □的代数。 这样一个代数的求值器必须是从 □ (□ □)到□ □的态射。 我们可以通过简单地提升 □来轻松构造这样一个求值器。

$$□\ □ :: □\ \ (□\ □) \to □\ □$$

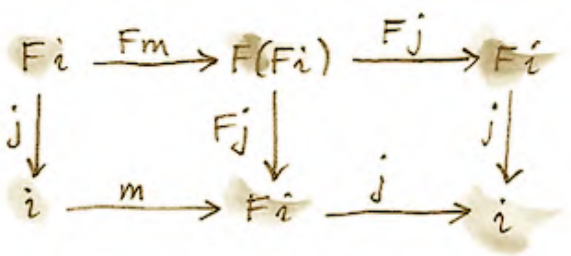
因为 (□, □)是初始代数，所以必须存在一个唯一的同态映射 □从它到 (□ □, □ □)。 下面的图表必须满足交换性：



但我们也有这个显然满足交换性的图表（两条路径是相同的！）：



这可以解释为 □是代数同态映射，将 (□ □, □ □)映射到 (□, □)。 我们可以将这两个图表拼接在一起得到：



这个图表可以被解释为□ ◦ □是代数同态映射。 只有在这种情况下，两个代数结构才是相同的。 此外，因为 (□, □)是初始的，所以只能存在一个从它到自身的同态映射，那就是恒同态映射 $\mathrm{id}_{□}$—我们知道它是代数同态映射。 因此□ ◦ □ = $\mathrm{id}_{□}$。 利用这个事实和左图表的交换性质，我们可以证明□ ◦ □ = $\mathrm{id}_{□\,□}$。 这表明 □是 □的逆映射，因此 □是□ □和 □之间的同构映射：

□ □ ≅ □

但这只是在说 □是 □的一个不动点。 这是原始手势论证的形式证明。

回到Haskell：我们将 □识别为我们的 Fix f，将 □识别为我们的构造函数Fix，并将其逆转为 unFix。 Lambek定理中的同构告诉我们，为了得到初始代数，我们将函子 □的参数 □替换为 Fix f。 我们还可以看到为什么不动点不依赖于 □。

24.3 自然数

自然数也可以定义为一个F-代数。起点是一对态射：

$$□ □ □ :□ \ 1 \to □$$

$$□ □ □ :□ \ □ \to □$$

第一个选择零，第二个将所有数字映射到它们的后继。 与之前一样，我们可以将两者合并为一个：

$$1 + □ \to □$$

左边定义了一个函子，在Haskell中可以写成这样：

```
data NatF a = ZeroF | SuccF a
```

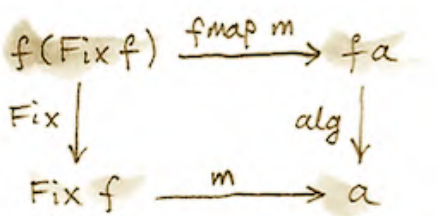
这个函子的不动点（它生成的初始代数）可以用Haskell编码为：

```
data Nat = Zero | Succ Nat
```

自然数要么是零，要么是另一个数的后继。 这被称为自然数的Peano表示。

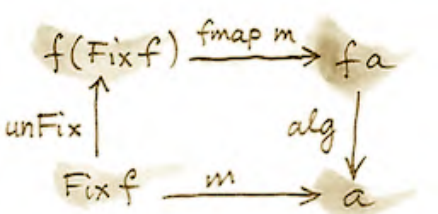
24.4 卡塔莫菲斯

让我们使用Haskell符号重新编写初始条件。 我们将初始代数称为Fix f。 它的评估器是构造函数Fix。 从初始代数到同一函子上的任何其他代数存在唯一的态射m。 让我们选择一个以a为载体和以alg为评估器的代数。



顺便提一下，注意m是什么：它是不动点的评估器，整个递归表达式树的评估器。 让我们找到一种通用的实现方式。

Lambek定理告诉我们构造函数Fix是一个同构。 我们称其逆为unFix。 因此，我们可以在这个图中翻转一个箭头来得到：



让我们写下这个图表的交换条件：

```
m = alg . fmap m . unFix
```

我们可以将这个方程解释为 m的递归定义。 对于使用函子创建的任何有限树，递归都将终止。
f。 我们可以通过注意到 fmap m在函子 f的顶层下操作来看出这一点。 换句话说，它在原始树的子节点上工作。 子节点始终比原始树的层级浅一级。

当我们将 m应用于使用Fix f构建的树时会发生什么呢？ unFix的作用是剥离构造器，暴露出树的顶层。 然后，我们将 m应用于顶节点的所有子节点。 这将产生类型为 a的结果。 最后，我们通过应用非递归评估器 alg来组合这些结果。 关键点是我们的评估器 alg是一个简单的非递归函数。

由于我们可以对任何代数 alg进行这样的操作，因此定义一个以代数为参数并给出我们所称之为 m的函数的高阶函数是有意义的。 这个高阶函数被称为一个范畴论：

```
cata :: 函子 f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix
```

让我们看一个例子。 以定义自然数的函子为例：

```
data NatF a = ZeroF | SuccF a
```

让我们选择 (Int, Int)作为载体类型，并将我们的代数定义为：

```
fib :: NatF (Int, Int) -> (Int, Int)
fib ZeroF = (1, 1)
fib (SuccF (m, n)) = (n, m + n)
```

你可以很容易地说服自己，这个代数的范畴论，
cata fib，计算斐波那契数。

一般来说，一个 NatF的代数定义了一个递归关系：当前元素的值与前一个元素有关。 然后，一个范畴论计算该序列的第n个元素。

24.5 折叠

一个列表的 e是以下函子的初始代数：

```
data ListF e a = NilF | ConsF e a
```

事实上，将变量 a替换为递归的结果，我们将其称为 List e，我们得到：

```
data List e = Nil | Cons e (List e)
```

列表函子的代数选择一个特定的载体类型，并定义一个在两个构造函数上进行模式匹配的函数。 对于 NilF的值告诉我们如何评估一个空列表，对于 ConsF的值告诉我们如何将当前元素与先前累积的值组合起来。

例如，这是一个可以用来计算列表长度的代数（载体类型为 Int）：

```
lenAlg :: ListF e Int -> Int
lenAlg (ConsF e n) = n + 1
lenAlg NilF = 0
```

事实上，得到的范畴折叠 cata lenAlg计算列表的长度。 请注意，评估器是由（1）接受列表元素和累加器并返回新累加器的函数和（2）起始值（这里是零）组合而成的。 值的类型和累加器的类型由载体类型给出。

将其与传统的Haskell定义进行比较：

```
length = foldr (\e n -> n + 1) 0
```

foldr的两个参数正好是代数的两个组成部分。

让我们再举一个例子：

```
sumAlg :: ListF Double Double -> Double
 sumAlg (ConsF e s) = e + s
sumAlg NilF = 0.0
```

再次与以下进行比较：

```
 sum = foldr (\e s -> e + s) 0.0
```

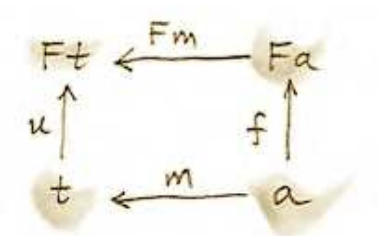
正如你所看到的， foldr只是列表的一个方便的特化。

24.6 余代数

通常情况下，我们有一个F-余代数的对偶构造，其中态射的方向被颠倒：

□ → □ □

给定一个函子的余代数也构成一个范畴，同态保持余代数结构。在该范畴中，终对象 (□, □)被称为终（或最终）余代数。 对于每个其他代数 (□, □)存在一个唯一的同态 □使得以下图表成立：



终端余代数是函子的不动点，即
态射□ : : □ → □ □是一个同构（Lambek定理对于
余代数）：

□ □ ≅ □

终端余代数通常在编程中被解释为生成（可能是无限的）数据结构或转换系统的方法。

就像一个折叠函数可以用来评估一个初始代数一样，一个展开函数可以用来共评估一个终端余代数：

```
 ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg
```

一个余代数的一个典型例子是基于一个函子，其不动点是类型为 e 的无限流。这是该函子：

```
data StreamF e a = StreamF e a
   deriving Functor
```

这是它的不动点：

```
data Stream e = Stream e (Stream e)
```

对于StreamF e的余代数是一个函数，它接受类型为a的种子，并生成一个由元素和下一个种子组成的对（StreamF是对的一个花哨的名字）。

你可以轻松地生成产生无限序列的余代数的简单示例，比如平方列表或倒数。

一个更有趣的例子是产生一个质数列表的余代数。 关键是使用一个无限列表作为载体。 我们的起始种子将是列表 [2..]。 下一个种子将是这个列表的尾部，其中移除了所有的2的倍数。 这是一个以3为起始的奇数列表。 在下一步中，我们将取出这个列表的尾部，并移除所有的3的倍数，以此类推。你可能会认出埃拉托斯特尼筛法的构造过程。"This coalgebra is implemented by the following function:

```
era :: [Int] -> StreamF Int [Int]
era (p : ns) = StreamF p (filter (notdiv p) ns)
    where notdiv p n = n `mod` p /= 0
```

这个余代数的anamorphism生成了质数列表：

```
primes = ana era [2..]
```

流是一个无限列表，所以应该可以将其转换为一个Haskell列表。 为了做到这一点，我们可以使用相同的函子 StreamF 形成一个代数，然后我们可以对其运行一个catamorphism。例如，这是一个将流转换为列表的catamorphism：

```
toListC :: Fix (StreamF e) -> [e]
toListC = cata al
    where al :: StreamF e [e] -> [e]
                        al (StreamF e a) = e : a
```

在这里，相同的不动点同时是一个初始代数和一个终端余代数，对于相同的自函子。 在任意范畴中，情况并不总是如此。 一般来说，自函子可能有许多（或者没有）不动点。 初始代数被称为最小不动点，终端余代数被称为最大不动点。 然而，在Haskell中，它们都由相同的公式定义，并且它们重合。

列表的反范畴是称为展开的。 为了创建有限列表，函子被修改为生成一个可能的对：

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

值为无将终止列表的生成。

一个有趣的余代数案例与透镜相关。 透镜可以表示为一个获取器和一个设置器的对：

```
 set :: a -> s -> a
get :: a -> s
```

在这里，一个通常是一个具有类型为 s的字段的乘积数据类型。 获取器检索该字段的值，设置器用新值替换该字段。 这两个函数可以合并为一个：

```
a -> (s, s -> a)
```

我们可以进一步将这个函数重写为：

```
a -> Store s a
```

其中我们定义了一个函子：

```
数据类型 Store s a = Store (s -> a) s
```

注意，这不是一个简单的由和积构造的代数函子。 它涉及到一个指数 $□^{□}$。

透镜是这个函子的一个余代数，其载体类型为 a。 我们之前已经看到 Store s也是一个余单子。 事实证明，一个行为良好的透镜对应于与余单子结构兼容的余代数。 我们将在下一节中讨论这个问题。

24.7 挑战

1. 为一个变量的多项式环实现求值函数。 你可以将多项式表示为 □的幂前的系数列表。 例如， $4□^{2}$– 1可以表示为（从零次幂开始） [-1, 0, 4]。

2. 将前面的构造推广到多个独立变量的多项式，例如 $□^{2}□ - 3□^{3}□$。
3. 实现2 × 2矩阵的代数运算。
4. 定义一个余代数，其anamorphism生成自然数的平方列表。
5. 使用 unfoldr生成前 □个质数的列表。

25

Monad的代数

如果我们将自函子解释为定义表达式的方式，那么代数可以让我们对其进行求值，而单子则可以让我们形成和操作它们。 通过将代数与单子结合，我们不仅获得了很多功能，还可以回答一些有趣的问题。

其中一个问题涉及到单子和伴随的关系。 正如我们所见，每个伴随都定义了一个单子（和一个余单子）。 问题是：每个单子（余单子）都可以从一个伴随中派生出来吗？ 答案是肯定的。 有一个完整的伴随家族可以生成给定的单子。 我将展示给你两个这样的伴随。让我们回顾一下定义。

定义。 单子是一个自函子 □配备有两个满足一些连贯性条件的自然变换。这些变换的分量在□处为：

同样的自函子的代数是选择一个特定的对象 — 承载者 □— 以及态射：

$$\square\ \square :: \square \to \square\ \square$$
$$\square\ \square :: \square\ (\square\ \square) \to \square\ \square$$

首先要注意的是，代数的方向与□ □□相反。

$$\square\ \square\ \square : \square\ \square \to \square$$

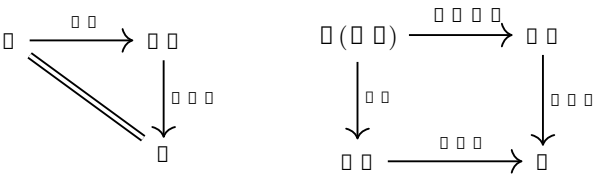
直观上，□□从类型□的值创建一个平凡的表达式。 使代数与单子兼容的第一个连贯性条件确保使用承载者为□的代数来评估这个表达式会得到原始值： 这个表达式使用承载者为 □的代数来评估会得到原始值。

$$\square\ \square\ \square\square\ \square = \mathrm{id}\,\mathrm{id}$$

第二个条件是由于有两种方式来评估嵌套表达式 □ (□ □)而产生的。 我们可以先应用 □□来展开表达式，然后使用代数的求值器；或者我们可以应用提升的求值器来评估内部表达式，然后再应用求值器来得到结果。 我们希望这两种策略是等价的：

$$\square\ \square\ \square\square\ \square = \square\ \square\ \square\square\ \square\ \square$$

在这里，□ □ 是使用函子□提升得到的态射。 下面的交换图描述了这两个条件（我用 □替换了 □，预示着接下来的内容）：



我们也可以用Haskell来表达这些条件：

```
alg . return = id
alg . join = alg . fmap alg
```

让我们看一个小例子。 列表自函子的代数包括某种类型□和一个从□列表生成□的函数。 我们可以使用 foldr来表达这个函数，通过选择元素类型和累加器类型都等于□：

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

这个特定的代数是由一个双参数函数指定的，让我们称之为 f，并且一个值 z。 列表函子也是一个单子，其中 return将一个值转换为一个单元素列表。 代数的组合，这里是 foldr f z，在 return之后将 x 转换为：

```
 foldr f z [x] = x `f` z
```

其中 f的操作以中缀表示法书写。 如果对于每个 x都满足以下一致性条件，那么代数与单子是兼容的：

```
x `f` z = x
```

如果我们将 f视为二元运算符，这个条件告诉我们 z是右单位元。

第二个一致性条件适用于列表的列表。 操作 join将各个列表连接起来。 然后我们可以对结果列表进行折叠。 另一方面，我们可以先对各个列表进行折叠，然后再对结果列表进行折叠。 同样，如果我们将 f解释为二元运算符，这个条件告诉我们这个二元运算符是可结合的。 当 (a, f, z)是一个幺半群时，这些条件肯定是满足的。

25.1 T-代数

由于数学家更喜欢将他们的单子称为 $\mathbb{T}$，他们称与之兼容的代数为T-代数。给定范畴 $\mathcal{C}$中的单子 $\mathbb{T}$的T-代数形成了一个称为Eilenberg-Moore范畴的范畴，通常用 $\mathcal{C}^{\mathbb{T}}$表示。 该范畴中的态射是代数的同态。 这些同态与我们之前定义的F-代数的同态是相同的。

T-代数是由一个载体对象和一个评估器组成的一对，(A, h)。从 $\mathcal{C}^{\mathbb{T}}$到 $\mathcal{C}$有一个明显的遗忘函子 $U^{\mathbb{T}}$，它将(A, h)映射到 A。 它还将T-代数的同态映射到 $\mathcal{C}$中的对应态射。 你可能还记得我们在关于伴随函子的讨论中提到过，遗忘函子的左伴随被称为自由函子。

左伴随到 $U^{\mathbb{T}}$被称为 $F^{\mathbb{T}}$。 它将一个对象 A在 $\mathcal{C}$映射到 $F^{\mathbb{T}}$中的自由代数。 这个自由代数的载体是$T A$。 它的评估器是从 $T (T A)$回到$T A$的态射。 由于 T是一个单子，我们可以使用单子的 μ_A（在Haskell中是join）作为评估器。

我们仍然需要证明这是一个T-代数。为此，必须满足两个一致性条件：

□ □ □□ □□ = id id □

□ □ □□ □= □ □ □□ □ □ □

但这只是单子定律，如果你将 □插入到代数中。

正如你可能记得的，每个伴随定义了一个单子。 事实证明， □□和 □□之间的伴随定义了构造Eilenberg-Moore范畴中使用的单子□。 由于我们可以对每个单子执行这个构造，我们得出结论，每个单子都可以从一个伴随生成。 稍后我会向你展示，还有另一个伴随可以生成相同的单子。

这是计划：首先，我将向你展示 □□确实是 □□的左伴随。 我将通过定义这个伴随的单位和余单位，并证明相应的三角恒等式得到满足来完成。 然后，我将向你展示由这个伴随生成的单子是我们的原始单子。

伴随的单位是自然变换：

□ : : □ → □ □◦ □ □

让我们计算这个变换的 □分量。 恒等函子给出了 □。 自由函子产生了自由代数 (□ □, □□)，而遗忘函子将其减少为□ □。 总而言之，我们得到了从 □到□ □的映射。 我们将简单地使用单子 □的单位作为这个伴随的单位。

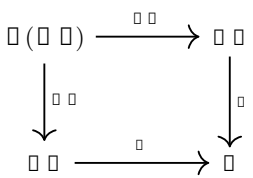
让我们来看看余单位：

□ : : □ □◦ □ □→ □

让我们计算一下它在某个T-代数 (A, a)的分量。 遗忘函子遗忘了 A，而自由函子产生了一对 $(T A, \mu_A)$。 因此，为了定义余单位 ε在 (A, a)的分量，我们需要Eilenberg-Moore范畴中的右态射，或者是T-代数的同态：

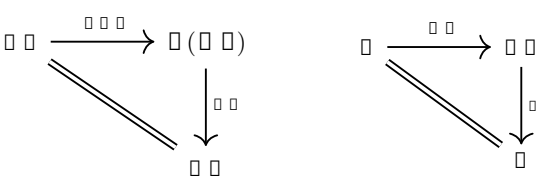
$$(T A, \mu_A) \to (A, a)$$

这样的同态应该将载体 $T A$映射到 A。 让我们恢复被遗忘的求值器 a。 这次我们将它用作T-代数的同态。实际上，使ε成为T-代数的同态的同一个交换图可以重新解释为它是T-代数的同态：



因此，我们定义了余单位自然变换的分量 ε在范畴T-代数中的对象 (A, a)为 a。

为了完成伴随，我们还需要证明单位和余单位满足三角恒等式。它们分别是：



第一个恒等式成立是因为单子 □的单位法则。 第二个恒等式就是T-代数 (□, □)的法则。

我们已经证明了这两个函子构成了一个伴随：

□ □□ □ □

每个伴随都会产生一个单子。这个往返

□ □◦ □ □

是在C中产生相应单子的自函子。 让我们看看它对一个对象 □的作用。由 □□创建的自由代数是 遗忘函子 □□去掉了求值器。所以，确实，我们有：

□ □◦ □ □= □

正如预期的那样，伴随的单位是单子的单位 □.

你可能记得，伴随的共单位通过以下公式产生单子乘法:

□ = □ ◦ □ ◦ □

这是三个自然变换的水平组合，其中两个分别是将这是映射到映射和将和将映射到映射的恒等自然变换。 中间的那个，共单位，是一个自然变换，其在代数 (□, □)处的分量是 □.

让我们计算分量 □□. 我们首先水平组合 □在 □□之后，这将得到 □在 □□ □处的分量. 由于 □□将 □映射到代数 (□ □, □□)，而 □选择了评估器，我们最终得到 □□.

在左侧进行水平组合与 □□不会改变任何东西，因为 □□对态射的作用是平凡的。因此，确实，从伴随得到的 □与原始单子 □的 □是相同的。

25.2 Kleisli范畴

我们之前见过Kleisli范畴。 它是从另一个范畴 $\mathcal{C}$和一个单子 T构造出来的范畴。 我们将这个范畴称为 $\mathcal{C}_T$。 Kleisli范畴 $\mathcal{C}_T$中的对象是 $\mathcal{C}$的对象，但态射是不同的。 Kleisli范畴中从 A到 B的态射 $A \to_T B$对应于原始范畴中从 A到$T B$的态射 $A \to T B$。 我们称这个态射为从 A到 B的Kleisli箭头。

在Kleisli范畴中，态射的复合是通过Kleisli箭头的单子复合来定义的。 例如，让我们在 $A \to_T B$之后组合 $B \to_T C$。 在Kleisli范畴中，我们有：

$$f :: A \to T\ B$$
$$g :: B \to T\ C$$

在范畴 $\mathcal{C}$中，它对应于：

$$f :: A \to T\ B$$
$$g :: B \to T\ C$$

我们定义复合：

$$h f\ g = \mu \circ T\ g \circ f$$

作为范畴 $\mathcal{C}$中的Kleisli箭头

$$h :: A \to T\ C$$
$$h = \mu \circ (T\ g) \circ f$$

在Haskell中，我们将其写为：

```
h = join . fmap g . f
```

存在一个从范畴 □到范畴 □□的函子 □，它在对象上的作用是平凡的。 对于态射，它将 □在范畴 □中映射到范畴 □□中的一个态射，通过创建一个Kleisli箭头来装饰 □的返回值。 给定一个态射

$$: \square : : \square \rightarrow \square$$

它在 □□中创建了一个态射，对应的Kleisli箭头为:

$$\square \circ \square$$

在Haskell中，我们会这样写:

```
return . f
```

我们还可以定义一个从 □□回到 □的函子 □. 它接受一个来自Kleisli范畴的对象 □，并将其映射到 □中的一个对象□ □. 它对于一个对应于Kleisli箭头的态射 □□的作用是一个在 □中的态射:

$$\square \ \square \rightarrow \square \ \square$$通过首先提升 □然后应用

□给出:

$$\square \ \square\square \circ \square \ \square$$

在Haskell符号中，这将被表示为:

```
G fT = join . fmap f
```

你可能会将其认作以...定义的单子绑定
连接.

很容易看出这两个函子形成了一个伴随：

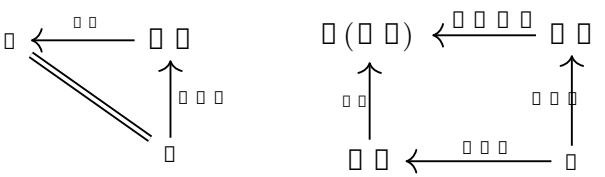
$$\square \dashv \square$$

它们的组合$\square \circ \square$重现了原始单子$\square$.

所以这是产生相同单子的第二个伴随。事实上，存在一整个伴随的范畴$\square^{\square}(\square\square, \square)$，其结果是相同的单子$\square$在$\square$上。我们刚刚看到的Kleisli伴随是这个范畴中的初始对象，而Eilenberg-Moore伴随是终结对象。

25.3 余代数

类似的构造可以用于任何余单子$\square$。我们可以定义与余单子兼容的余代数的范畴。它们使以下图表交换：



在$\square\ \square$的余代数中，余代数的载体是$\square$，那么$\square\ \square$是余代数的共伴态射：

$$\square\ \square\ \square : \square \to \square\ \square$$

而$\square$和$\square$是定义余函子的两个自然变换（在Haskell中，它们的分量被称为 extract和 duplicate）。

从这些余代数的范畴到$\square$，存在一个明显的遗忘函子$\square^{\square}$。它只是忽略了共伴态射。我们将考虑它的右

共伴函子 $\square^{\square}$。

$$\square \ \square\square \ \square \ \square$$

遗忘函子的右共伴被称为自由函子。遗忘$^{\square}$生成自由余代数。 对于范畴 □中的一个对象 □，它分配了一个余代数 ($\square \ \square, \square_{\square}$)。 这个共伴关系将原始余函子复制为复合 $\square^{\square} \circ \square^{\square}$。

同样地，我们可以构建一个具有共Kleisli箭头的共Kleisli范畴，并从相应的伴随中重新生成共函子。

25.4 镜头

让我们回到关于镜头的讨论。 镜头可以被写成一个余代数：

$$\square \ \square \ \square \ \square \ \square : \square \rightarrow \square \ \square \ \square \ \square\square \ \square\square$$

对于函子 $\square \ \square \ \square \ \square\square \ \square$

数据类型 Store s a = Store (s -> a) s

这个余代数也可以表示为一对函数：

$$\square \ \square \ \square : \square \rightarrow \square \rightarrow \square$$

$$\square \ \square \ \square : \square \rightarrow \square$$

（将 □视为“全部”，将 □视为其中的“一小部分”。）根据这对函数，我们有：

$$\square \ \square \ \square \ \square \ \square = \square \ \square \ \square \ \square(\square \ \square \ \square) \ (\square \ \square \ \square)$$

在这里， □是类型为 □的值。 注意，部分应用的 □ □是一个函数 $\square \rightarrow \square$。

我们还知道□ □ □ □□是一个余单子：

```
实例 Comonad (Store s) where
    extract (Store f s) = f s
duplicate (Store f s) = Store (Store f) s
```

问题是：在什么条件下，镜头是这个余单子的代数？ 第一个相干条件：

翻译为：

这是镜头法则，它表达了如果将结构的一个字段 □设置为其先前值，那么什么都不会改变。

第二个条件：

需要更多的工作。 首先，回顾一下 fmap对于 Store函子的定义：

```
fmap g (Store f s) = Store (g . f) s
```

将 fmap coalg应用于 coalgg的结果，我们得到：

```
Store (coalg . set a) (get a)
```

另一方面，将 duplicate应用于 coalg的结果，我们得到：

```
Store (Store (set a)) (get a)
```

为了使这两个表达式相等，在 Store下的两个函数在作用于任意的 s 时必须相等：

```
coalg (set a s) = Store (set a) s
```

展开 coalg，我们得到：

```
Store (set (set a s)) (get (set a s)) = Store (set a) s
```

这等同于剩下的两个镜头定律。第一个定律：

```
set (set a s) = set a
```

告诉我们将字段的值设置两次与设置一次是相同的。第二个定律：

```
get (set a s) = s
```

告诉我们获取已设置为 □的字段的值会返回 □。

换句话说，一个行为良好的镜头确实是 Store函子的余代数。

25.5 挑战

1. 自由函子□ ∷ □ → □ □在态射上的作用是什么？
   提示：使用单子 □的自然性条件。
2. 定义伴随：

   □ □□ □ □

3. 证明上述伴随重现了原始余单子。

26

终态和余终态

在范畴中，我们可以将许多直觉附加到态射上，但我们都可以同意，如果存在从对象 □到对象 □的态射，那么这两个对象在某种程度上是“相关的”。态射在某种意义上是这种关系的证明。 这在任何偏序范畴中都是显而易见的，其中一个态射是一个关系。 一般来说，可能存在许多关于两个对象之间相同关系的“证明”。

这些证明构成了我们称之为同态集合的集合。 当我们改变对象时，我们得到了从对象对到“证明”集合的映射。这个映射是函子性的 - 对第一个参数逆变，对第二个参数协变。 我们可以将其视为在范畴中建立全局关系。 这种关系由同态函子描述：

$$\square(-,=) :: \square^{\square} \times \square \to \square_{\square}$$

一般来说，像这样的任何函子都可以被解释为在一个范畴中建立对象之间的关系。 一个关系也可以涉及两个不同的范畴 C和 D。 描述这样一个关系的函子具有以下签名，并被称为一个半函子：

□ : : □ □ □× □ → □ □ □

数学家说它是从 □到 □的一个半函子（注意反转），并使用一个斜线箭头作为其符号：

□ □ □

你可以将半函子看作是证明相关的关系，它在 □的对象和 □的对象之间，其中集合的元素象征着关系的证明。 当□ □ □为空时， □和 □之间没有关系。 请记住，关系不一定是对称的。

另一个有用的直觉是将一个自函子是一个容器的思想进行概括。 类型为□ □ □的半函子值可以被认为是由类型为□的元素作为键的 □的容器。 特别地，一个同态半函子的元素是从 □到 □的函数。

在Haskell中，profunctor被定义为一个带有名为dimap的方法的二元类型构造器 pequipped，它将一对函数提升，第一个函数以“错误”的方向进行：

```
自函子 p 的类
    dimap :: (c -> a) -> (b -> d) -> p a b -> p c d
```

profunctor的函子性告诉我们，如果我们有一个证明 a与 b相关，那么只要存在从 c到 a的态射和从 b到 d的另一个态射，我们就可以得到证明 c与 d相关。或者，我们可以

将第一个函数视为将新键转换为旧键，将第二个函数视为修改容器的内容。

对于在同一范畴内起作用的协函子，我们可以从类型□ □ □的对角元素中提取很多信息。 只要我们有一对态射□ → □和□ → □，我们可以证明 □与 □相关。 更好的是，我们可以使用单个态射来达到非对角值。 例如，如果我们有一个态射□ : : □ → □，我们可以将对□ □, $\mathrm{id}_{□}$ □的提升从□ □ □转换为□ □ □：

```
dimap f id (p b b) :: p a b
```

或者我们可以将对偶 □$\mathrm{id}_{□}$, □ □提升为从□ □ □到□ □ □的映射：

```
dimap id f (p a a) :: p a b
```

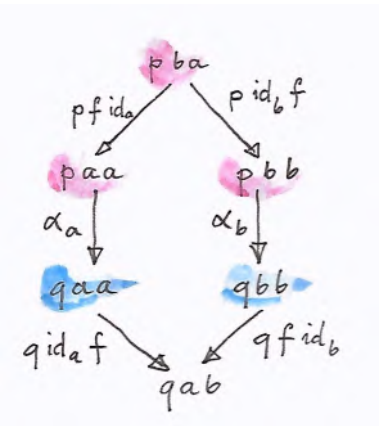
26.1 双自然变换

由于协函子是函子，我们可以以标准方式定义它们之间的自然变换。然而，在许多情况下，仅定义两个协函子的对角元素之间的映射就足够了。

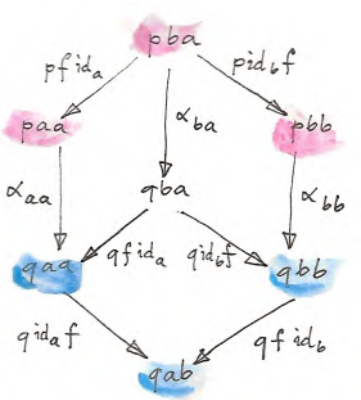
这样的变换被称为双自然变换，前提是它满足反映将对角元素连接到非对角元素的两种方式的交换条件。 在函子范畴 |□$^{□}$ □× □, □ □ □|中，两个协函子 □和 □之间的双自然变换是一个族的态射：

□ □: : □ □ □ → □ □ □

对于任何□ : : □ → □，以下图表都是可交换的：



请注意，这比自然性条件要弱得多。 如果 □是 [□ □× □, □ □]中的一个自然变换，上述图表可以由两个自然性正方形和一个函子性条件（保持复合的半函子 □）构建：



请注意，自然变换 □的一个分量在 [□□ □× □, □ □]中由一对对象 □□ □索引。 另一方面，一个双自然变换只由一个对象索引，因为它只映射相应半函子的对角线元素。

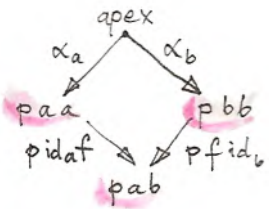
26.2 端

我们现在准备从“代数”进一步发展到范畴论的“微积分”阶段。 端（和余端）的微积分借鉴了传统微积分的思想和符号。 特别地，余端可以理解为无穷和或积分，而端类似于无穷乘积。 甚至有一些类似于狄拉克δ函数的东西。

端是极限的一种推广，将函子替换为半函子。 我们不再有锥体，而是有一个楔形。 楔形的底部由半函子 □的对角元素组成。 楔形的顶点是一个对象（在这里是一个集合，因为我们考虑的是 □ □ □值的半函子），而侧面是将顶点映射到底部集合的一族函数。 你可以将这个族看作是一个多态函数——一个在返回类型上是多态的函数：

□ ∷ ∀□ . □ □ □ ⊢ □ □ □

与锥形不同，在楔形中我们没有任何连接基点的函数。 然而，正如我们之前所见，对于范畴 □中的任何态射□ ∷ □ → □，我们可以将□ □ □和□ □ □连接到共同的集合□ □ □上。 因此，我们坚持以下图表的交换：



这被称为楔形条件。可以写成：

□ id id □ ∘ □ □ = □ □ id id ∘ □ □

或者，使用Haskell符号表示：

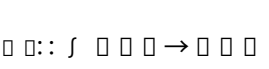
```
dimap id f . alpha = dimap f id . alpha
```

现在我们可以继续进行通用构造，并将 □的终点定义为通用楔形——一个集合 □以及一族函数□，对于任何具有顶点 □和一族 □的其他楔形，存在一个唯一的函数*h* : : □ → □使得所有三角形都交换：

□ □ ∘ *h* = □ □

结束的符号是积分符号，带有“积分变量”在下标位置：

□的组成部分被称为结束的投影映射：



请注意，如果请注是一个离散范畴（除了恒等态射之外没有其他态射），那么结束只是请注在整个范畴请注中所有对角线条目的全局乘积。 稍后我将向您展示，在更一般的情况下，通过等值器，结束与此乘积之间存在关系。

在Haskell中，结束公式直接转换为普遍量词：

```
forall a. p a a
```

严格来说，这只是严格的所有对角元素的乘积，但由于参数性，楔形条件会自动满足[1]。 对于任何函数□ : : □ → □，楔形条件如下：

```
dimap f id . pi = dimap id f . pi
```

或者，带有类型注释的：

```
dimap f idb . pib = dimap ida f . pia
```

方程两边的类型都是：

1https://bartoszmilewski.com/2017/04/11/profunctor-parametricity/

```
Profunctor p => (forall c. p c c) -> p a b
```

而且，pi是多态投影：

```
 pi :: Profunctor p => forall c. (forall a. p a a) -> p c c
pi e = e
```

在这里，类型推断会自动选择e的正确组件。就像我们能够将

一个锥的所有交换条件表示为一个自然变换一样，同样地，我们可以将所有楔条件分组为一个双自然变换。 为此，我们需要将常量函子 $\Delta_{\square}$推广为将所有对象对映射到单个对象 $\square$，并将所有对映射到该对象的恒等态射。 楔是从该函子到profunctor $\square$的双自然变换。 事实上，当我们意识到 $\Delta_{\square}$将所有态射提升为一个恒等函数时，双自然性六边形就缩小为楔形图。

除了 $\square\ \square$ 之外，目标范畴的末尾也可以定义在其他范畴上，但在这里我们只考虑 $\square\ \square$ 值的协函子和它们的末尾。

26.3 端作为等值器

在末尾的定义中，交换条件可以用等值器来表示。 首先，让我们定义两个函数（我使用Haskell表示法，因为在这种情况下数学表示法似乎不太用户友好）。 这些函数对应于楔形条件的两个收敛分支：

```
lambda :: Profunctor p => p a a -> (a -> b) -> p a b
lambda paa f = dimap id f paa
```

```
rho :: Profunctor p => p b b -> (a -> b) -> p a b
rho pbb f = dimap f id pbb
```

这两个函数将协函子 p的对角元素映射到多态函数的类型：

```
type ProdP p = forall a b. (a -> b) -> p a b
```

这些函数具有不同的类型。 然而，如果我们形成一个大的产品类型，将所有对角线元素聚集在一起，我们可以统一它们的类型，

```
newtype DiaProd p = DiaProd (forall a. p a a)
```

函数 lambda和 rho从这个产品类型中引出了两个映射：

```
lambdaP :: Profunctor p => DiaProd p -> ProdP p
lambdaP (DiaProd paa) = lambda paa

rhoP :: Profunctor p => DiaProd p -> ProdP p
rhoP (DiaProd pbb) = rho pbb
```

范畴 p的终结物是这两个函数的等值器。 请记住，等值器选择两个函数相等的最大子集。在这种情况下，它选择了所有对角线元素的产品的子集，使得楔形图是可交换的。

26.4 自然变换作为端

最重要的等值器的例子是自然变换的集合。 在两个函子 □和 □之间的自然变换是一个

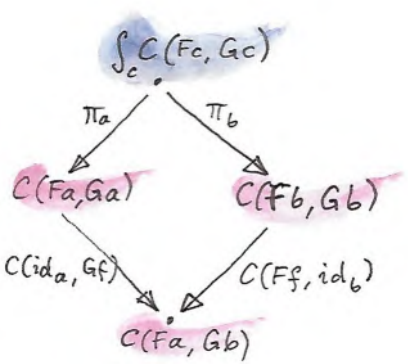
从形式为 □(□ □, □ □)的同态集中选择的同态族。 如果没有自然性条件，自然变换的集合将只是所有这些同态集的乘积。 事实上，在Haskell中，它是：

```
forall a. f a -> g a
```

它在Haskell中有效的原因是因为自然性是由参数性推导出来的。 然而，在Haskell之外，并不是所有这些同态集上的对角线截面都会产生自然变换。 但请注意，这个映射：

$$□ □, □ □ → □(□ □, □ □)$$

是一个协函子，因此研究它的终结是有意义的。这是楔形条件：



让我们从集合 ∫□ □(□ □, □ □)中选择一个元素。 这两个投影将这个元素映射到一个特定变换的两个分量上。

mation, 让我们称之为：

$$\square\ \square : : \square\ \square \to \square\ \square$$
$$\square\ \square : : \square\ \square \to \square\ \square$$

在左分支中，我们使用同态函子提升一对态射 $\square \mathrm{id}_{\square}, \square\ \square\ \square$。 你可能还记得这种提升是通过同时进行预合成和后合成来实现的。 当作用于 $\square_{\square}$时，提升的一对态射给出：

$$\square\ \square \circ \square\ \square \circ \mathrm{id} \circ \mathrm{i}$$

图表的另一个分支给出：

$$\mathrm{id}_{\mathrm{id}} \circ \square\ \square \circ \square\ \square$$

它们的相等性，由楔条件要求，实际上就是 $\square$的自然性条件。

26.5 余端

正如预期的那样，一个末端的对偶被称为余末端。 它是由一个称为余楔的对偶构造而来（发音为co-wedge，而不是cow-edge）。

余末端的符号是带有“积分变量”的积分号，位于上标位置：

$$\int^{\square} \square\ \square\ \square$$

就像末端与积的关系一样，余末端与余积（或和）有关（在这一点上，它类似于积分，它是一个".

和求和的极限). 我们没有投影，而是从协函子的对角元素到余终点的注入。如果没有余边条件，我们可以说协函子 □的余终点是□ □ □，或□ □ □，或□ □ □，等等。或者我们可以说存在这样的 □，使得余终点只是集合□ □ □。

我们在定义余终点时使用的普遍量词变成了存在量词。

这就是为什么在伪Haskell中，我们会将余终点定义为：

```
存在一个 a使得 p a a
```

在Haskell中，编码存在量词的标准方法是使用全称量化的数据构造函数。因此，我们可以定义：

一个时髦的奶牛?

```
数据 Coend p = forall a. Coend (p a a)
```

背后的逻辑是，应该可以使用任何类型族□ □ □的值来构造一个共端，无论我们选择什么 □。

就像可以使用等值器定义一个末端一样，可以使用一个共等化器来描述一个共端。 所有的共边条件可以通过将所有可能的函数□ → □的□ □ □的巨大余积来总结。 在Haskell中，这将被表示为一种存在类型：

```
数据 SumP p = forall a b. SumP (b -> a) (p a b)
```

评估这个和类型有两种方法，一种是使用 dimap提升函数并将其应用于 □的仿函。

```
lambda, rho :: Profunctor p => SumP p -> DiagSum p
lambda (SumP f pab) = DiagSum (dimap f id pab)
rho                 (SumP f pab) = DiagSum (dimap id f pab)
```

其中 DiagSum是 □的对角元素之和：

```
data DiagSum p = forall a. DiagSum (p a a)
```

这两个函数的共等化器是共端。 通过将由 lambda或 rho应用于相同参数的值进行标识，可以从 DiagSum p获得一个共等化器。 在这里，参数是一个由函数□ → □和□ □ □的元素组成的对。 应用 lambda和 rho会产生两个可能不同的 DiagSum p类型的值。 在共端中，这两个值被标识为相同，从而自动满足了共楔条件。

在集合中识别相关元素的过程被正式称为取商。 要定义商，我们需要一个等价关系 □，这个关系是自反的、对称的和传递的：

□ □ □

如果□ □ □则□ □ □

如果□ □ □且□ □ □则□ □ □

这样的关系将集合分割成等价类。 每个类包含彼此相关的元素。 我们通过从每个类中选择一个代表来形成一个商集。 一个经典的例子是将有理数定义为具有以下等价关系的整数对：

(□, □) □ (□, □) iff □ □ □ = □ □ □

很容易验证这是一个等价关系。 一对 (m, n)被解释为一个分数 $\frac{m}{n}$，并且具有公共除数的分数被识别出来。有理数是这些分数的等价类。

你可能还记得我们之前讨论过极限和余极限的时候，同态函子是连续的，也就是说，它保持极限。 相反地，逆变同态函子将余极限转化为极限。 这些性质可以推广到端和余端，它们分别是极限和余极限的推广。 特别地，我们得到了一个非常有用的将余端转化为端的等式：

$$\mathbf{Set}\left(\int^{x} p\, x\, x, c\right) \cong \int_{x} \mathbf{Set}(p\, x\, x, c)$$

让我们用伪Haskell来看一下它：

(存在 x. p x x) -> c ≅ 对于所有的 x. p x x -> c

它告诉我们，接受存在类型的函数等价于多态函数。 这是很有道理的，因为这样的函数必须准备好处理可能编码在存在类型中的任何类型之一。 这是同样的原理告诉我们，接受和类型的函数必须实现为一个case语句，其中包含一个处理程序的元组，每个处理程序对应和类型中的每个类型。在这里，和类型被余端取代，一族处理程序变成了端，或者说是多态函数。

26.6 忍者米田引理

在Yoneda引理中出现的自然变换的集合可以使用一个end进行编码，得到以下公式：

$$\int_{z} \mathbf{Set}(\mathcal{C}(a, z), F\, z) \cong F\, a$$

还有一个对偶公式：

$$\int^{a} C(a, a) \times F a \cong F a$$

这个恒等式强烈地让人想起了Dirac delta函数的公式（一个函数 $\delta(x - a)$，或者更准确地说是一个分布，在 $x = a$ 处有一个无限高的峰值）。 在这里，同态函子扮演了delta函数的角色。这两个恒等式有时被称为忍者Yoneda引理。

为了证明第二个公式，我们将使用Yoneda嵌入的结果，该结果指出如果两个对象的同态函子是同构的，则这两个对象也是同构的。 换句话说，如果存在以下类型的自然变换：

$$[C, \mathbf{Set}](C(a, -), C(b, -)) \cong C(a, b)$$

那么它是一个同构。

我们首先将要证明的恒等式的左边插入到一个同态函子中，该同态函子将指向任意对象 x：

$$\mathbf{Set}\left(\int^{a} C(a, a) \times F a, x\right)$$

使用连续性论证，我们可以用末尾替换余末尾：

$$\int_{a} \mathbf{Set}(C(a, a) \times F a, x)$$

我们现在可以利用积与指数之间的伴随关系：

$$\int_{a} \mathbf{Set}(C(a, a), \mathbf{Set}(F a, x))$$

我们可以通过使用Yoneda引理来“执行积分”得到：

$$s^{(p\ a)}$$

这个指数对象同构于同态集合：

$$\mathbf{Set}(p\ a, s)$$

最后，我们利用Yoneda嵌入来得到同构：

$$\int^{z} C(a, z) \times p\ z \cong p\ a$$

26.7 仿函合成

让我们进一步探讨一个profunctor描述一个关系的想法 - 更准确地说，是一个证明相关的关系，意味着集合$p\ d\ c$表示了 d与 c相关的证明集合。 如果我们有两个关系p和 q，我们可以尝试将它们组合起来。 我们将说 d通过 q之后的 p与 c相关，如果存在一个中间对象e，使得$q\ d\ e$和$p\ e\ c$都不为空。 这个新关系的证明是所有个别关系的证明对。 因此，根据存在量词对应于余末尾，两个集合的笛卡尔积对应于“证明对”，我们可以使用以下公式定义profunctors的组合：

$$(q \circ p)\ d\ c = \int^{e} q\ d\ e \times p\ e\ c$$

这是等价的Haskell定义

Data.Profunctor.Composition，在一些重命名之后：

```
data Procompose q p a b where
    Procompose :: q a c -> p c b -> Procompose q p a b
```

这是使用广义代数数据类型（generalized algebraic data type）orgadtsyntax，其中一个自由类型变量（这里是 c）会自动存在量化。 （非柯里化的）数据构造函数 Procompose因此等价于：

```
exists c. (q a c, p c b)
```

因此，这样定义的组合的单位是同态函子（hom-functor）——这是由忍者米田引理（Ninja Yoneda lemma）立即得出的。 因此，很有意义地问是否存在一个范畴，其中profunctors作为态射。 答案是肯定的，但要注意，profunctor组合的结合性和单位元律只在自然同构下成立。 这样一个范畴，在同构下律仍然成立，被称为双范畴（bicategory）（比 -范畴更一般）。 因此，我们有一个双范畴 , 其中对象是范畴，态射是profunctors，而态射之间（也称为二胞胎）是自然变换。 实际上，我们还可以更进一步，因为除了profunctors之外，我们还有常规函子作为范畴之间的态射。 一个具有两种类型的态射的范畴被称为双范畴。

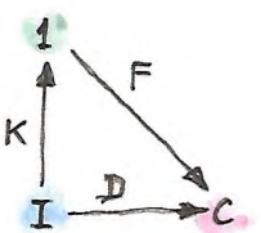
Profunctors在Haskell镜头库和箭头库中起着重要作用。

27

Kan扩展

到目前为止，我们主要在一个单一的范畴或一对范畴中工作。在某些情况下，这有点太限制了。

例如，在一个范畴 $\mathcal{C}$中定义极限时，我们引入了一个索引范畴 $\mathcal{I}$作为模式的模板，这将成为我们锥体的基础。

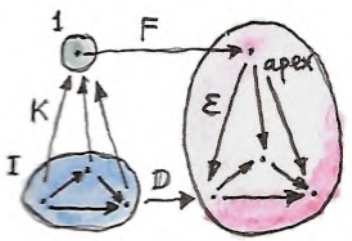


引入另一个范畴，一个平凡的范畴，作为锥体的顶点，这是有意义的。

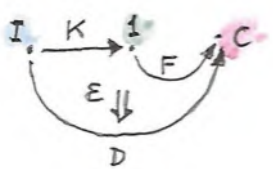
相反，我们使用了常量函子 $\Delta_{\mathcal{C}}$从 $\mathcal{C}$到 $\mathcal{I}$。

是时候解决这个尴尬了。 让我们使用三个范畴来定义一个极限。 让我们从索引范畴 $\mathcal{I}$到范畴$\mathcal{C}$定义函子 $\mathcal{F}$。 这是选择锥体基底的函子 - 图表函子。

新添加的范畴是包含一个对象（和一个恒等态射）的范畴 □。 从□到这个范畴只有一个可能的函子 □。 它将所有对象映射到 □中的唯一对象，并将所有态射映射到恒等态射。 从 □到 □的任何函子 □都选择了我们锥体的一个潜在顶点。

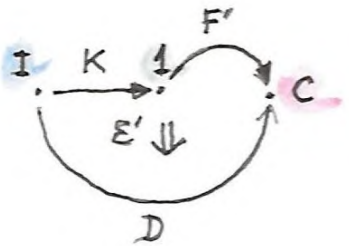


锥体是从□ ∘ □到 □的一个自然变换 □。 注意□ ∘ □与我们原始的 Δ□ 完全相同。 下图显示了这个变换。



现在我们可以定义一个选择“最佳”这样的函子 □的普适性质。 这个 □将 □映射到 □中 □的极限对象，并且从□ ∘ □到 □的自然变换 □ 将提供相应的投影。 这个普适函子被称为沿着 □的右Kan扩展，并用 $\mathrm{Ran}_{□}□$表示。

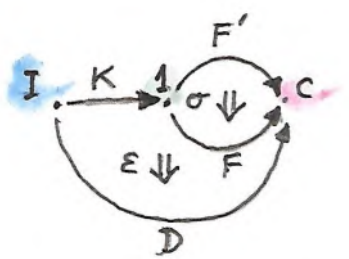
让我们来制定普适性质。 假设我们有另一个锥体 - 也就是另一个函子 □′连同一个自然变换□′从 □′∘□到 □。



如果存在Kan扩展□ = □ □ □□，那么必须存在一个唯一的自然变换 □从 □′到它，使得 □′通过 □因子分解，即：

$$□' = □ . (□ ∘ □)$$

在这里，□∘□是两个自然变换的水平组合（其中一个是 □上的恒等自然变换）。 然后，这个变换与 □垂直组合。



在分量上，当作用于对象 □在 □中时，我们得到：

□ □′ = □ □∘ □ □□

在我们的情况下， □只有一个组件对应于 □的单个对象。 因此，确实，这是从由 □′定义的锥形的顶点到由 $\mathrm{Ran}_{□}$ □定义的通用锥形的顶点的唯一态射。
交换条件恰好是限制定义的条件。

但是，重要的是，我们可以用任意范畴 □替换平凡范畴 □，并且右Kan扩展的定义仍然有效。

27.1 右Kan扩展

沿着函子□ : : □ → □的右Kan扩展，沿着函子□ : : □ → □的右Kan扩展是一个函子□ : : □ → □（表示为 $\mathrm{Ran}_{□}$ □），以及一个自然变换。

□ : : □ ∘ □ → □

这样，对于任何其他函子 □′: : □ → □和一个自然变换

□′ : : □′ ∘ □ → □

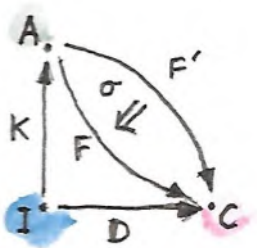
存在唯一的自然变换

□ : : □′ → □

使得分解 □′.

□′ = □ . (□ ∘ □)

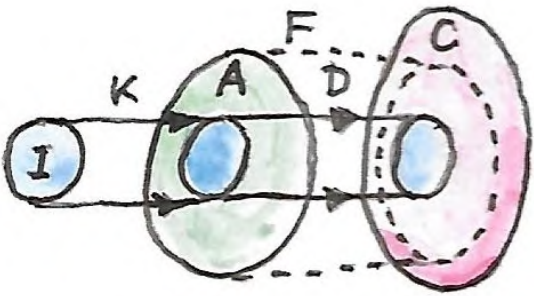
这听起来很复杂，但可以用这个漂亮的图表来可视化：



一个有趣的观点是注意到，在某种意义上，Kan扩展的作用类似于“函子乘法”的逆过程。一些作者甚至使用符号 □/□来表示 $\mathrm{Ran}_{\square}$ □。 实际上，在这个表示法中，右Kan扩展的单位 □的定义看起来像是简单的消除：

$$\square :: \square/\square \circ \square \to \square$$

Kan扩展还有另一种解释。 考虑到函子 □将范畴 □嵌入到范畴 □中。 在最简单的情况下， □可以只是 □的一个子范畴。 我们有一个将 □映射到 □的函子 □。 我们能否将 □扩展为在整个 □上定义的函子 □? 理想情况下，这样的扩展将使得组合□ ∘ □与□同构。 换句话说， □将 □的定义域扩展到 □。 但通常要求一个完全同构可能太过分了，我们可以只要求其中一半，即从□ ∘ □到□的单向自然变换 □。 （左Kan扩展选择另一个方向。）



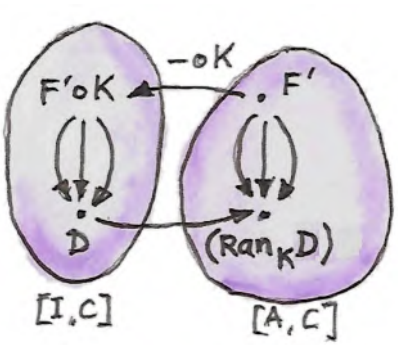
当函子 □在对象上不是单射或在同态集上不是忠实的时候，嵌入图就会崩溃。 在这种情况下，Kan扩展尽力推断出丢失的信息。

27.2 Kan扩展作为伴随

现在假设对于任何 □（和一个固定的□），右Kan扩展存在。 在这种情况下，$\mathrm{Ran}_{□}-$（用破折号替代 □）是从函子范畴 [□, □]到函子范畴 [□, □]的函子。 事实证明，这个函子是前合成函子−∘□的右伴随。 后者将 [□, □]中的函子映射到 [□, □]中的函子。这个伴随关系是：

$$[□, □](□' \circ □, □) \cong [□, □](□', \mathrm{Ran}\mathsf{RaRa})$$

这只是一个事实的重新陈述，即对于每个自然变换
我们称之为 □′对应一个唯一的自然变换，我们称之为 □。



此外，如果我们选择的范畴 □与 □相同，我们可以用恒等函子 □□替换 □。 我们得到以下等式：

$$[□, □](□' ∘ □, □ □) ≅ [□, □](□', \mathrm{Ran}\mathsf{RaRaRa})$$

我们现在可以选择 □′与 $\mathrm{Ran}_{□}$ □□相同。 在这种情况下，右边包含了单位自然变换，并且对应地，左边给出了以下自然变换：

$$□ ∷ \mathrm{Ran}\mathsf{RaRaRa} ∘ □ → □ □$$

这看起来非常像一个伴随的余单位：

$$\mathrm{Ran}_{\mathsf{RaRaRa}} □ □$$

事实上，沿着一个函子□的单位函子的右Kan扩展可以用来计算 □的左伴随。 为此，还需要满足一个条件：函子 □必须保持右Kan扩展。 扩展的保持意味着，如果我们计算函子与 □的复合的Kan扩展，我们应该得到

与 □的原始Kan扩展预合成得到相同的结果。 在我们的情况下，这个条件简化为：

$$□ \circ \mathrm{Ran}_{R} R \cdot R \cong \mathrm{Ran} R a R a$$

请注意，使用除以□的符号，这个伴随可以写成：

$$□ / □ \dashv □$$

这证实了我们的直觉，即伴随描述了某种逆。 保持条件变为：

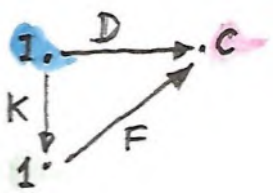
$$□ \circ □ / □ \cong □ / □$$

沿着自身的函子的右Kan扩展，□ /□，被称为一个密度单子。

伴随公式是一个重要的结果，因为正如我们很快将看到的，我们可以使用ends（coends）来计算Kan扩展，从而为我们提供找到右（和左）伴随的实际手段。

27.3 左Kan扩展

有一个对偶的构造给出了左Kan扩展。 为了建立一些直觉，我们可以从余极限的定义开始，并重构它以使用单子范畴 □。 我们通过使用函子□ : : □ → □构建一个余锥，以形成其基础，并使用函子□ : : □ → □选择其顶点。

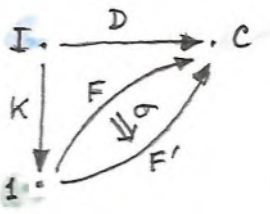


共锥体的边，即注入，是从 □到□ ∘ □的一个自然变换的组成部分。

余极限是普遍共锥体。 因此，对于任何其他函子 □′和一个自然变换

$$□' :: □ \to □' \circ □$$

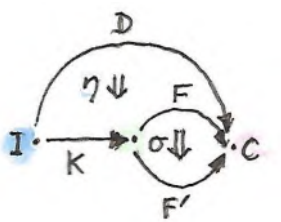
存在一个唯一的自然变换 □从 □到 □′



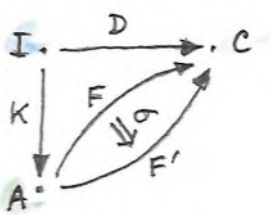
满足以下条件：

$$\square' = (\square \circ \square).\square$$

这在下面的图表中有所说明：



将单子范畴 $\square$ 替换为 $\square$，这个定义自然地推广到左Kan扩张的定义，记为 $\mathrm{Lan}_{\square}\,\square$。



自然变换：

$$\eta :: \mathrm{Id} \to \mathrm{Lan}\,\mathrm{La}\,\mathrm{La} \circ \mathrm{La}$$

被称为左Kan扩张的单位。

与之前一样，我们可以重新表述自然变换之间的一一对应关系：

$$\alpha' = (\varepsilon \circ \mathrm{La}).\ \alpha$$

在自然变换的条件下：

$$[\mathcal{C}, \mathcal{D}](\mathrm{Lan}\,\mathrm{La}, \alpha') \cong [\mathcal{C}, \mathcal{D}](\mathrm{Id}, \alpha' \circ \eta)$$

换句话说，左Kan扩展是左伴随，而右Kan扩展是与 La的后合成的右伴随。

就像单位函子的右Kan扩展可以用来计算 La的左伴随一样，单位函子的左Kan扩展也是 La的右伴随（其中 ε是伴随的单位）：

$$\varepsilon \dashv \mathrm{Lan}\,\mathrm{La}\,\mathrm{La}\,\mathrm{La}$$

结合这两个结果，我们得到：

$$\mathrm{Ran}_{\mathrm{La}}\,\mathrm{La}_{\mathrm{La}} \dashv \mathrm{La} \dashv \mathrm{Lan}_{\mathrm{La}}\,\mathrm{La}_{\mathrm{La}}$$

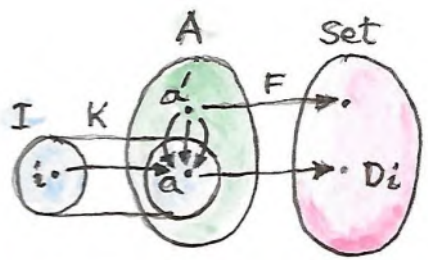
27.4 Kan扩展作为端

Kan扩展的真正威力来自于它们可以使用ends（和coends）进行计算。 为了简单起见，我们将限制我们的注意力在目标范畴 $\mathcal{D}$是 $\mathbf{Set}$的情况下，但是这些公式可以扩展到任何范畴。

让我们重新审视一个Kan扩展可以用于扩展函子在其原始定义域之外的操作的想法。 假设 □嵌入 □到 □中。 函子 □将 □映射到 □ □ □。 我们可以简单地说，对于图像中的任何对象 □，即□ = □ □，扩展的函子将 □映射到□ □。 问题是，对于那些在 □的图像之外的 □中的对象该怎么办？ 这个想法是每个这样的对象潜在地通过许多态射与 □的图像中的每个对象相连接。

一个函子必须保持这些态射。 从对象 □到 □的图像的态射的总体由同态函子来刻画：

$$□(□, □ -)$$



注意这个同态函子是两个函子的组合：

$$□(□, □ -) = □(□, -) \circ □$$

右Kan扩张是函子组合的右伴随：

$$[□, □ □]([□□' \circ □, □) \cong [□, □ □]([□□', \mathrm{Ran}\mathrm{RaRa})$$

让我们看看当我们用同态函子替换 □′ 时会发生什么：

$$[□, □ □]([□□(□, -) \circ □, □) \cong [□, □ □]([□□(□, -), \mathrm{Ran}\mathrm{RaRa})$$

然后内联组合：

$$[\mathbf{I}, \mathbf{Set}](\mathcal{C}(a, K -), D) \cong [\mathcal{A}, \mathbf{Set}](\mathcal{A}(a, -), \mathrm{Ran}_K D)$$

右边可以使用Yoneda引理简化：

$$[\mathbf{I}, \mathbf{Set}](\mathcal{C}(a, K -), D) \cong (\mathrm{Ran}_K D)\, a$$

现在我们可以将自然变换的集合重写为末尾，得到这个非常方便的右Kan扩展公式：

$$\mathrm{Ran}_K D\, a \cong \int_i \mathbf{Set}(\mathcal{C}(a, K i), D i)$$

对于左Kan扩张，存在一个类似的公式，用一个
余合表示：

$$\mathrm{Lan}_K D\, a = \int^i \mathcal{C}(K i, a) \times D i$$

为了证明这一点，我们将展示这确实是左伴随
to函子组合：

$$[\mathcal{A}, \mathbf{Set}](\mathrm{Lan}_K D, F) \cong [\mathbf{I}, \mathbf{Set}](D, F \circ K)$$

让我们在左手边替换我们的公式：

$$[\mathcal{A}, \mathbf{Set}](\int^i \mathcal{A}(K i, -) \times D i, F)$$

这是一组自然变换，因此可以重写为一个末尾：

$$\int_a \mathbf{Set}(\int^i \mathcal{A}(K i, a) \times D i, F a)$$

利用同态函子的连续性，我们可以用末尾替换
余合：

$$\int_{\square}\int_{\square} \square\ \square(\square\square(\square\ \square, \square) \times \square\ \square, \square'\ \square)$$

我们可以使用乘积-指数伴随：

$$\int_{\square}\int_{\square} \square\ \square(\square\square(\square\ \square, \square), (\square'\ \square)^{\square\ \square})$$

指数同构于相应的同态集合：

$$\int_{\square}\int_{\square} \square\ \square(\square\square(\square\ \square, \square), \square(\square\ \square, \square'\ \square))$$

有一个被称为Fubini定理的定理，它允许我们交换两端：

$$\int_{\square}\int_{\square} \square\ \square(\square\square(\square\ \square, \square), \square(\square\ \square, \square'\ \square))$$

内部端表示两个函子之间的自然变换的集合，因此我们可以使用Yoneda引理：

$$\int_{\square} \square(\square\ \square, \square'(\square\ \square))$$

这确实是我们要证明的伴随的右侧的自然变换的集合：

$$[\square, \square\ \square](\square\square, \square' \circ \square)$$

使用端、余端和Yoneda引理进行这种计算在“端的微积分”中相当典型。

27.5 Haskell中的Kan扩展

Kan扩展的端/余端公式可以很容易地转换为Haskell。让我们从右扩展开始：

$$\mathrm{Ran}_{RaRa\,\square} \cong \int \square\,\square\,(\square\square(\square,\square\,\square),\square\,\square)$$

$$\square$$

我们用全称量词替换端，用函数类型替换同态集：

```
newtype Ran k d a = Ran (forall i. (a -> k i) -> d i)
```

从这个定义来看，很明显Ran必须包含一个类型为a的值，该函数可以应用于两个函子k和d之间的自然变换。 例如，假设k是树函子，d是列表函子，你得到了一个Ran Tree [] String。 如果你传递给它一个函数：

```
f :: 字符串 -> 树 Int
```

然后你会得到一个 Int列表，以此类推。右Kan扩展将使用你的函数生成一棵树，然后重新打包成一个列表。 例如，你可以传递一个从字符串生成解析树的解析器，然后你会得到一个对应于该树的深度优先遍历的列表。

通过将函子 d替换为恒等函子，可以使用右Kan扩展来计算给定函子的左伴随。 这导致了函子 k的左伴随由以下类型的多态函数集表示：

```
forall i. (a -> k i) -> i
```

假设 k是从幺半群范畴的遗忘函子。
普遍量词然后遍历所有幺半群。 当然，在Haskell中我们无法表达幺半群定律，但以下是结果自由函子（遗忘函子 k在对象上是恒等的）的一个不错的近似：

```
type Lst a = forall i. Monoid i => (a -> i) -> i
```

正如预期的那样，它生成自由幺半群，或者说 Haskell 列表：

```
toLst :: [a] -> Lst a
toLst as = \f -> foldMap f as

fromLst :: Lst a -> [a]
 fromLst f = f (\a -> [a])
```

左 Kan 扩展是一个余共端：

$$\mathrm{Lan}_{\mathrm{LaLa}\ \square} = \int^{\square} \square(\square\ \square, \square) \times \square\ \square$$

因此，它可以转换为存在量词。符号上：

```
Lan k d a = 存在 i. (k i -> a, d i)
```

这可以使用 gadts 在 Haskell 中进行编码，或者使用一个全称量化的数据构造器：

```
 data Lan k d a = forall i. Lan (k i -> a) (d i)
```

这个数据结构的解释是它包含一个函数，该函数接受一些未指定的 i的容器并产生一个 a。 它还有一个包含这些 i的容器。由于你不知道 i是什么，所以唯一的

你可以使用这个数据结构做的一件事是检索包含is的容器，使用自然变换将其重新打包到由函子 k定义的容器中，并调用函数以获得 a。 例如，如果 d是一棵树， k是一个列表，你可以将树序列化，使用生成的列表调用函数，并获得一个 a。

左Kan扩展可以用来计算一个函子的右伴随。 我们知道乘积函子的右伴随是指数函子，所以让我们尝试使用Kan扩展来实现它：

```
type Exp a b = Lan ((,) a) I b
```

事实上，这与函数类型是同构的，可以通过以下一对函数来证明：

```
toExp :: (a -> b) -> Exp a b
toExp f = Lan (f . fst) (I ())

fromExp :: Exp a b -> (a -> b)
 fromExp (Lan f (I x)) = \a -> f (a, x)
```

请注意，在前面的一般情况中，我们执行了以下步骤：

1. 检索了容器 x（这里只是一个平凡的恒等容器）和函数 f。
2. 使用恒等函子和对偶函子之间的自然变换重新打包了容器。
3. 调用了函数 f。

27.6 自由函子

Kan扩展的一个有趣应用是构建一个自由函子。 它是以下实际问题的解决方案：假设你

有一个类型构造器 - 即对象的映射。 是否可能基于这个类型构造器定义一个函子？ 换句话说，我们是否可以定义一个映射的态射，将这个类型构造器扩展为一个完整的自函子？

关键观察是，类型构造器可以被描述为一个定义域为离散范畴的函子。 离散范畴除了恒等态射之外没有其他态射。 给定一个范畴 $\mathcal{C}$，我们总是可以通过简单地丢弃所有非恒等态射来构造一个离散范畴 $|\mathcal{C}|$。 从 $|\mathcal{C}|$到 $\mathcal{C}$的函子 F然后是一个简单的对象映射，或者我们在Haskell中称之为类型构造器。还有一个将 $|\mathcal{C}|$嵌入 $\mathcal{C}$的规范函子 J：它在对象上是一个恒等函子（在恒等态射上也是如此）。 如果存在，沿着J的左Kan扩展则是一个从 $\mathcal{C}$到 $\mathcal{C}$的函子：

$$\mathrm{Lan}_{\mathrm{LaLa}\ J}\ F = \int^{c} \mathcal{C}(J\ c, -) \times F\ c$$

它被称为基于 F的自由函子。

在Haskell中，我们会这样写：

```
data FreeF f a = forall i. FMap (i -> a) (f i)
```

事实上，对于任何类型构造器 f， FreeF f都是一个函子：

```
instance Functor (FreeF f) where
    fmap g (FMap h fi) = FMap (g . h) fi
```

正如你所看到的，自由函子通过记录函数和其参数来模拟函数的提升。 它通过记录它们的组合来累积提升的函数。 函子规则会自动满足。

满足。 这个构造在一篇名为《自由单子，更多可扩展效果》的论文中使用。

或者，我们可以使用右Kan扩展来实现相同的目的：

```
新类型FreeF f a = FreeF (forall i. (a -> i) -> f i)
```

很容易检查这确实是一个函子：

```
实例函子（FreeF f）where
    fmap g (FreeF r) = FreeF (\bi -> r (bi . g))
```

1http://okmij.org/ftp/Haskell/extensible/more.pdf

28

丰富的范畴

如果一个范畴的对象形成一个集合，那么它是小的。但我们一个知道集合之外还有更大的东西。 众所周知，在标准集合论（泽尔梅洛-弗兰克尔理论，可选地加上选择公理）中无法构建所有集合的集合。因此，所有集合的范畴必须是大的。 有一些数学技巧，如Grothendieck宇宙，可以用来定义超越集合的集合。 这些技巧使我们能够讨论大范畴。

如果任意两个对象之间的态射形成一个集合，则范畴是局部小的。如果它们不形成一个集合，我们必须重新思考一些定义。特别是，如果我们甚至不能从一个集合中选择它们，那么组合态射意味着什么？ 解决方法是通过用来自其他范畴 的对象替换在 中的同态集合。 不同之处在于，一般情况下，对象没有元素，因此我们不再允许讨论单个态射。我们有

以整体的方式，用关于同态对象的操作来定义富化范畴的所有属性。 为了做到这一点，提供同态对象的范畴必须具有额外的结构，即它必须是一个幺半范畴。 如果我们将这个幺半范畴称为 $\mathcal{V}$，我们可以谈论一个在 $\mathcal{V}$上富化的范畴 $\mathcal{C}$。

除了大小的原因，我们可能对将同态集合概括为比纯粹的集合更具结构的东西感兴趣。 例如，传统的范畴没有对象之间的距离概念。
两个对象要么通过态射连接，要么不连接。与现实生活不同，在范畴中，与给定对象连接的所有对象都是它的邻居。与现实生活不同，在范畴中，一个朋友的朋友的朋友与我亲密的朋友一样亲近。 在适当丰富的范畴中，我们可以定义对象之间的距离。

还有一个非常实际的原因是要获得一些关于丰富范畴的经验，那就是一个非常有用的在线范畴知识来源，nLab[1]，主要是用丰富范畴的术语编写的。

28.1 为什么是幺半范畴?

在构建一个丰富范畴时，我们必须记住，当我们用 $\otimes$ $\otimes$ 替换单调范畴并用同态对象替换同态集时，我们应该能够恢复常规定义。 实现这一点的最佳方法是从常规定义开始，并以无点方式重新表述它们-即，不命名集合的元素。

让我们从组合的定义开始。 通常，它需要一对态射，一个来自 $\mathcal{C}(A, B)$，一个来自 $\mathcal{C}(B, C)$，并将其映射

1https://ncatlab.org/

到一个从 $\square(\square, \square)$ 的态射。 换句话说，这是一个映射：

$$\square(\square, \square) \times \square(\square, \square) \to \square(\square, \square)$$

这是两个集合之间的函数 — 其中一个是两个同态集合的笛卡尔积。 这个公式可以通过用更一般的东西替换笛卡尔积来轻松推广。 范畴积可以工作，但我们甚至可以更进一步，使用一个完全通用的张量积。

接下来是恒等态射。 我们可以使用来自单元集 $\square$ 的函数来定义它们，而不是从同态集中选择单个元素：

$$\square\ \square :: \square \to \square(\square, \square)$$

同样，我们可以用终结对象替换单元集，但我们甚至可以更进一步，用张量积的单位元 $\square$o来替换它。

正如你所看到的，从某个单子范畴 $\square$ 中取出的对象是替代同态集的好候选者。

28.2 幺半范畴

我们之前已经讨论过单子范畴，但值得重申一下定义。 单子范畴定义了一个张量积，它是一个双函子：

$$\otimes :: \square \times \square \to \square$$

我们希望张量积是结合的，但只需要满足自然同构的结合性即可。 这个同构被称为结合子，它的组成部分是：

$$\square\ \square\square\ \square :: (\square \otimes \square) \otimes \square \to \square \otimes (\square \otimes \square)$$

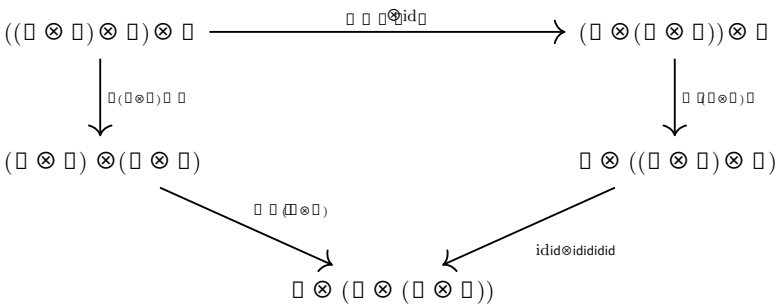
它必须对所有三个参数都是自然的。

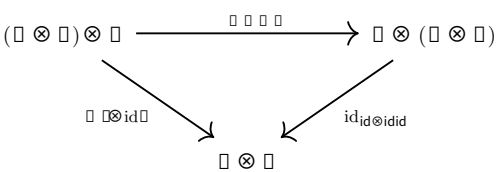
单子范畴还必须定义一个特殊的单位对象 I，它作为张量积的单位；同样，满足自然同构。 这两个同构分别被称为左单位元和右单位元，它们的组成部分是：

$$\lambda_A :: I \otimes A \to A$$

$$\rho_A :: A \otimes I \to A$$

结合子和单位元必须满足一致性条件：





如果存在一个具有以下组成部分的自然同构，则单子范畴被称为对称的：

$$\gamma_{A,B} :: A \otimes B \to B \otimes A$$

其“平方为一”：

$$\square \square \circ \square \square = \mathrm{id}\mathrm{id} \otimes \mathrm{id}$$

并且与幺半结构一致。

关于幺半范畴的一个有趣之处是，你可以将内部同态（函数对象）定义为张量积的右伴随。 你可能还记得，函数对象或指数通过范畴积的右伴随来定义。 对于任意一对对象存在这样一个对象的范畴被称为笛卡尔闭。 这是在幺半范畴中定义内部同态的伴随：

$$\square(\square \otimes \square, \square) \square \square(\square, [\square, \square])$$

根据G. M. Kelly[2]的说法，我使用记号 $[\square, \square]$表示内部同态。这个伴随的余单位是自然变换，其分量被称为评估态射：

$$\square \square \square :: ([\square, \square] \otimes \square) \to \square$$

请注意，如果张量积不对称，我们可以使用以下伴随来定义另一个内部同态，记作 $[[\square, \square]]$：

$$\square(\square \otimes \square, \square) \square \square(\square, [[\square, \square]])$$

在其中两者都被定义的幺半范畴被称为双闭。 一个不是双闭的范畴的例子是在 $\square \square$ 中的自函子范畴，其中函子组合作为张量积。 那就是我们用来定义单子的范畴。

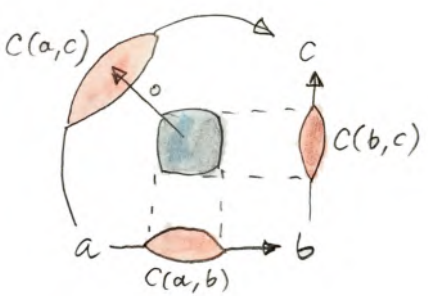
2http://www.tac.mta.ca/tac/reprints/articles/10/tr10.pdf

28.3 富集范畴

在一个幺半范畴 $\mathcal{V}$上富化的范畴 $\mathcal{C}$用富化的同态对象替代了同态集合。 对于范畴 $\mathcal{C}$中的每一对对象 a和 b，我们关联一个对象 $\mathcal{C}(a, b)$ in $\mathcal{V}$。 我们对同态对象使用与同态集合相同的符号，理解它们不包含同态。 另一方面， $\mathcal{V}$是一个常规（非富化）的范畴，具有同态集合和同态。 所以我们并没有完全摆脱集合 - 我们只是把它们藏起来了。

由于我们不能在 $\mathcal{V}$中讨论单个态射，态射的组合被替换为 $\mathcal{V}$中的一组态射：

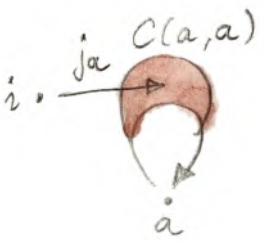
$$\circ :: \mathcal{C}(b, c) \otimes \mathcal{C}(a, b) \to \mathcal{C}(a, c)$$



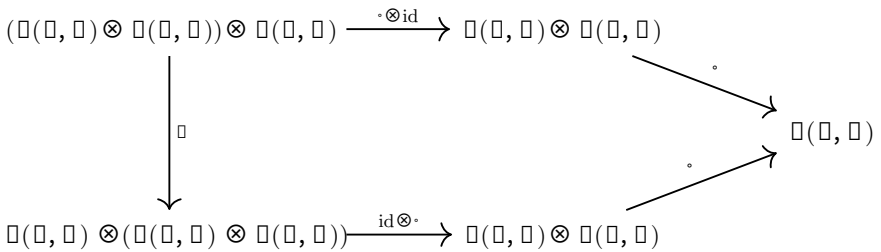
同样，单位态射被替换为$\mathcal{V}$中的一组态射：

$$id :: I \to \mathcal{C}(a, a)$$

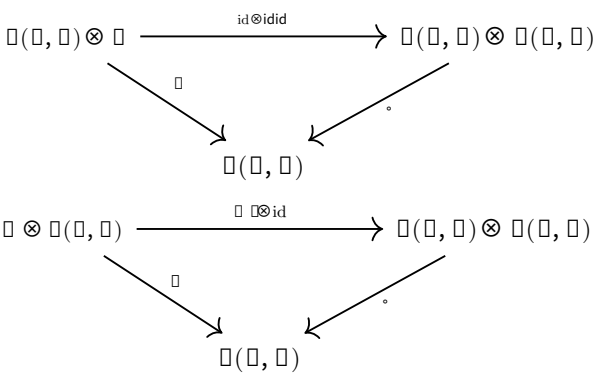
其中 I是 $\mathcal{V}$中的张量单位。



组合的结合性是通过 □中的结合子来定义的：



单位律也可以用单位子来表达：



28.4 偏序集

偏序集被定义为一个薄范畴，其中每个同态集要么为空，要么是一个单元素集。 我们将非空集合 □(□, □)解释为 □小于或等于 □的证明。 这样的范畴可以被解释为富集在一个非常简单的单调范畴上，该范畴只包含两个对象 0和 1（有时称为□ □ □ □和□ □ □ □）。 除了强制的单位态射外，该范畴还有一个从 0到 1的单一态射，我们称之为0 → 1。 可以在其中建立一个简单的单调结构，张量积模拟了 0和 1的简单算术（即，唯一的非零乘积是1 ⊗ 1）。 该范畴中的单位对象是 1。 这是一个严格的单调范畴，即结合子和单位子都是单位态射。

由于在预序中，同态集合要么为空，要么是一个单元素集合，我们可以轻松地用我们的小范畴中的同态对象来替换它。 富集预序 □对于任意一对对象 □和□，都有一个同态对象 □(□, □)。 如果 □小于或等于 □，则该对象为 1；否则为 0。

让我们来看看组合。 任意两个对象的张量积为 0，除非它们都为 1，在这种情况下为 1。 如果为 0，则对于组合态射有两个选项：可以是id_0或0 → 1。 但如果为 1，则唯一的选项是 id_1。 将这个转化回关系，这意味着如果□ □ □且□ □ □，则□ □ □，这正是我们需要的传递性法则。

那么身份呢？ 它是从 1到 □(□, □)的态射。 只有一个从 1出发的态射，那就是身份 id_1，所以 □(□, □)必须是 1。 这意味着□ □ □，这是一个前序关系的自反性法则。 因此，如果我们将一个前序关系实现为一个丰富的范畴，那么传递性和自反性都会自动得到强制执行。

28.5 度量空间

一个有趣的例子是由William Lawvere提出的[3]。 他注意到度量空间可以使用丰富的范畴来定义。 度量空间定义了任意两个对象之间的距离。 这个距离是一个非负实数。 将无穷大作为可能的值包括进来是很方便的。 如果距离是无穷大，那么从起始对象到目标对象就没有办法到达。

有一些明显的性质必须由距离满足。 其中之一是对象到自身的距离必须为零。 另一个是三角不等式：直接距离不大于经过中间站点的距离之和。 我们不要求距离是对称的，这一点起初可能看起来很奇怪，但正如劳-维尔所解释的，你可以想象在一个方向上你在爬坡，而在另一个方向上你在下坡。 无论如何，对称性可以作为一个附加约束后来加上。

那么如何将度量空间转化为范畴语言呢？ 我们必须构建一个范畴，其中的同态对象是距离。 请注意，距离不是态射，而是同态对象。 同态对象怎么可能是一个数字呢？ 只有当我们能够构建一个蒙德尔范畴 ，其中这些数字是对象时，它才能成为一个数字。 非负实数（加上无穷大）构成一个全序，因此它们可以被视为一个薄范畴。 两个这样的数字 和 之间存在一个态射，当且仅当 （注意：这与传统上在偏序定义中使用的方向相反）。 蒙德尔结构由加法给出，零作为单位对象。 换句话说，两个数字的张量积是它们的和。

3http://www.tac.mta.ca/tac/reprints/articles/1/tr1.pdf

度量空间是一个在这样的幺半范畴上富集的范畴。
从对象 □到 □的同态对象 □(□, □)是一个非负（可能是无穷大）的数，我们称之为从 □到 □的距离。 让我们看看在这样一个范畴中的恒等和复合是什么。

根据我们的定义，从张量单位（即数字零）到同态对象 □（□ , □）的态射是这样的关系：

$$0 \square \square(\square, \square)$$

由于 □（□， □）是一个非负数，这个条件告诉我们从 □到 □的距离始终为零。检查一下！

现在让我们来谈谈组合。 我们从两个相邻的同态对象 □（□，□）⊗ □（□， □）开始。 我们将张量积定义为这两个距离的和。 组合是从这个积到 □（□， □）的 □中的态射。 在 □中，态射被定义为大于或等于的关系。 换句话说，从□到 □的距离和从 □到 □的距离大于或等于从 □到□的距离。 但这只是标准的三角不等式。 检查一下！通过将度量空间重新表述为一个富范畴，我们可以免费获得三角不等式和零自距离。

28.6 富有趣味的函子

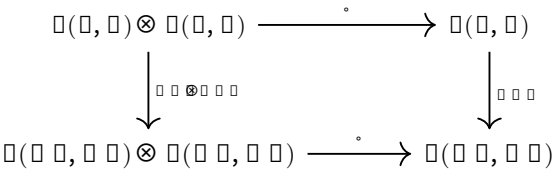
函子的定义涉及到态射的映射。 在丰富的环境中，我们没有个别态射的概念，所以我们必须批量处理同态对象。 同态对象是一个融合范畴 □中的对象，我们可以在其间进行态射。因此，在范畴在相同的融合范畴 □上进行丰富时，定义丰富函子是有意义的。

然后，我们可以使用融合范畴 中的态射来映射两个丰富范畴之间的同态对象。

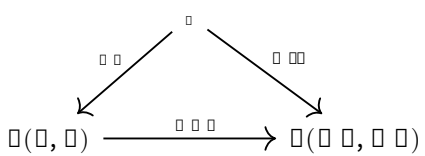
在两个范畴 和 之间的丰富函子 除了将对象映射到对象之外，还为 中的每对对象分配了一个 中的态射：

$$\square\ \square\square :: \square(\square, \square) \to \square(\square\ \square, \square\ \square)$$

函子是一种保持结构的映射。 对于常规函子，这意味着保持组合和恒等性。 在丰富的环境中，组合的保持意味着以下图表是可交换的：



在中，保留恒等性的方式被保留了mor-phisms，这些mor-phisms“选择”了恒等性:



28.7 自我丰富

一个闭合的对称幺半范畴可以通过用内部幺半代替同态集合来自丰富（见上面的定义）。为了实现这一点

，我们必须为内部幺半定义组合法则。 换句话说，我们必须实现一个具有以下签名的morphism:

$$[□, □] ⊗ [□, □] → [□, □]$$

这与任何其他编程任务并没有太大的区别，只是在范畴论中，我们通常使用无点实现。 我们从指定其元素的集合开始。在这种情况下，它是同态集合的成员:

$$□([□, □] ⊗ [□, □], [□, □])$$

这个同态集合是同构的:

$$□(([□, □] ⊗ [□, □]) ⊗ □, □)$$

我刚刚使用了定义了内部幺半 $[□, □]$的伴随。 如果我们可以在这个新集合中构建一个morphism，那么伴随将指向原始集合中的morphism，然后我们可以将其用作组合。
我们通过组合几个可用的态射来构造这个态射。首先，我们可以使用结合子 $□_{[□,□]\,[□,□]\,□}$来重新组合左边的表达式：

$$([□, □] ⊗ [□, □]) ⊗ □ → [□, □] ⊗([□, □] ⊗ □)$$

我们可以跟随伴隶属的共单位 $□_{□\,□}$：

$$[□, □] ⊗ ([□, □] ⊗ □) → [□, □] ⊗ □$$

然后再次使用共单位 $□_{□\,□}$来得到 □。 因此，我们构造了一个态射：

$$□ \,□□ . (\mathrm{id}_{[□,□]} ⊗ □ \,□□) . □_{[□,□]\,[□,□]} □$$

它是同态集合的一个元素：

$$\square((\square, \square] \otimes [\square, \square]) \otimes \square, \square)$$

伴隨将给我们所寻找的组合法则。同样地，单位元：

$$\square\ \square :: \square \to [\square, \square]$$

是以下同态集合的一个成员：

$$\square(\square, [\square, \square])$$

通过伴隨，它与以下同态集合同构：

$$\square(\square \otimes \square, \square)$$

我们知道这个同态集合包含左单位元 $\square_{\square}$。 我们可以定义$\square_{\square}$为它在伴隨下的像。

自我丰富的一个实际例子是范畴 $\square\ \square$，它作为编程语言中类型的原型。 我们之前已经看到它是关于笛卡尔积的闭合幺半范畴。在 $\square\ \square$ 中，任意两个集合之间的同态集合本身也是一个集合，因此它是一个对象在 $\square\ \square$ 中。 我们知道它与指数集是同构的，因此外部同态和内部同态是等价的。 现在我们还知道，通过自我丰富，我们可以使用指数集作为同态对象，并用指数对象的笛卡尔积来表示组合。

28.8 与 $\square$-范畴的关系

我在 $\square\ \square$ 的上下文中讨论了 $\square$-范畴，即（小）范畴的范畴。 范畴之间的态射是函子，但是还有

是一种附加结构：函子之间的自然变换。
在一个 □-范畴中，对象通常被称为零胞体；态射， 1-胞体；而态射之间的态射， 2-胞体。 在 □ □ 中， 0-胞体是范畴， 1-胞体是函子，而 2-胞体是自然变换。

但请注意，两个范畴之间的函子也形成一个范畴；因此，在 □ □ □中，我们实际上有一个同态范畴而不是同态集合。 事实证明，就像 □ □ 可以被视为在□ □ 上富集的范畴一样， □ □ 可以被视为在 □ □ 上富集的范畴。 更一般地说，就像每个范畴都可以被视为在 □ □ 上富集一样，每个□-范畴都可以被视为在 □ □ 上富集。

29

拓扑学

意识到我们可能正在远离编程并深入到硬核数学中。 但是你永远不知道下一个大的编程革命会带来什么样的数学，以及了解它所需的数学。有一些非常有趣的想法正在流行，比如具有连续时间的函数响应式编程，使用依赖类型扩展Haskell的类型系统，或者在编程中探索同伦类型论。

到目前为止，我一直随意地将类型与值的集合等同起来。 这并不严格正确，因为这种方法没有考虑到在编程中，我们计算值，并且计算是一个需要时间的过程，在极端情况下可能不会终止。

发散的计算是每个图灵完备语言的一部分。

还有一些基础原因，为什么集合论可能不是计算机科学甚至数学本身的最佳基础。 一个很好的类比是，集合论是与特定语言绑定的汇编语言。

对于特定的架构。 如果你想在不同的
架构上运行你的数学，你必须使用更通用的工具。

一种可能性是使用空间代替集合。 空间具有更多的结构，并且可以在不使用集合的情况下定义。 通常与空间相关的一件事是拓扑学，它是定义连续性等概念所必需的。 而传统的拓扑学方法是通过集合论来进行的。 特别是，子集的概念对于拓扑学来说是核心的。 毫不奇怪，范畴论家将这个想法推广到了除了 $\mathbf{Set}$ 之外的其他范畴。 具有适当属性以作为集合论替代的范畴类型被称为拓扑学（复数形式：topoi），它提供了广义的子集概念，以及其他一些东西。

29.1 子对象分类器

让我们从尝试使用函数而不是元素来表达子集的概念开始。 任何从某个集合 A 到 B 的函数 f 都定义了 B 的一个子集——即 f 在 B 下的像。 但是有很多函数定义了相同的子集。 我们需要更具体一些。 首先，我们可以关注那些是单射的函数——即不会将多个元素压缩成一个的函数。单射函数将一个集合“注入”到另一个集合中。

对于有限集合，你可以将单射函数可视化为连接一个集合的元素到另一个集合的平行箭头。 当然，第一个集合不能比第二个集合更大，否则箭头必然会收敛。 仍然存在一些不确定性：可能存在另一个集合 A' 和另一个从该集合到 B 的单射函数 f' 选择相同的子集。 但是你可以很容易地说服自己，这样的集合必须与 A 同构。 我们可以利用这个事实来将子集定义为由其定义域的同构关系相关的一族单射函数。

更准确地说，如果存在一个同构关系： 则我们说两个单射函数等价：

$$□ :: □ \to □$$
$$□' :: □' \to □$$

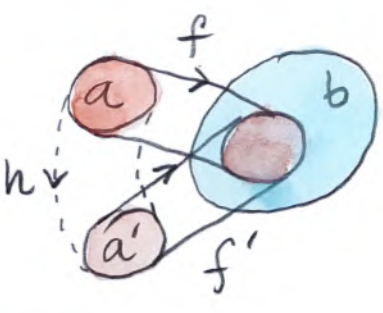
是等价的。

$$h :: □ \to □'$$

满足以下条件：

$$h$$

这样一族等价的单射函数定义了 □的一个子集。 如果我们用单态射替换单射函数，这个定义可以推广到任意范畴。



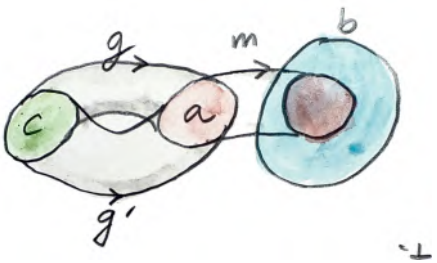
只是提醒一下，从 □到 □的单态射 □通过其普适性质来定义。 对于任何对象□和任何一对态射： 任意对象□和一对态射：

$$□ :: □ \to □$$
$$□' :: □ \to □$$

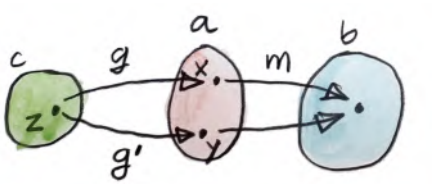
满足以下条件：

$$□ . □ = □ . □'$$

必须是 $□ = □'$.



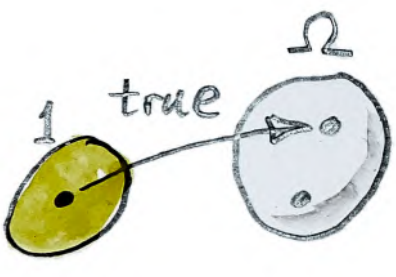
在集合上，如果我们考虑一个函数 □ not是单态射，这个定义更容易理解。 它会将 □的两个不同元素映射到 □的一个元素上。 然后我们可以找到两个函数 □和 □′，它们只在这两个元素上有所不同。 与 □的后合成将掩盖这种差异。



还有另一种定义子集的方法：使用一个称为特征函数的单一函数。 它是从集合 □到一个二元集 Ω的函数。 这个集合的一个元素被指定为“真”，另一个元素被指定为“假”。这个函数将那些属于子集的元素赋值为“真”，将不属于子集的元素赋值为“假”。

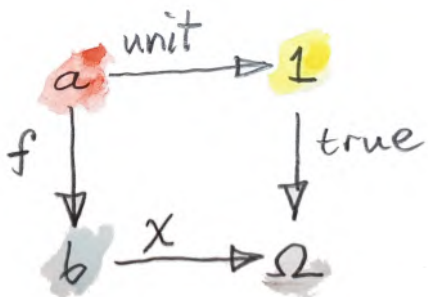
现在需要明确指定将 Ω的元素指定为“true”的含义。我们可以使用标准技巧：使用从单例集合到 Ω的函数。 我们将这个函数称为 □ □ □: □

$$\square\ \square\ \square : \square\ 1 \to \Omega$$



这些定义可以以一种方式组合，不仅定义了子对象的含义，还定义了特殊对象 Ω，而不涉及元素。 这个想法是我们希望态射 □ □ □ □ 表示一个“通用”的子对象。 在 □ □ 中，它从一个两元素集合 Ω 中选择一个单元素子集。 这是最通用的情况。 显然，它是一个真子集，因为 Ω有一个不在该子集中的额外元素。

在更一般的情况下，我们将 □ □ □定义为从终对象到分类对象 Ω的单态射。 但我们需要定义分类对象。 我们需要一个将这个对象与特征函数联系起来的普遍性质。 事实证明，在 □ □ 中，沿着特征函数 □的□ □ □的拉回定义了子集 □和将其嵌入 □的单射函数。 下面是拉回图：



让我们分析这个图表。 回拉方程是：

□ □ □.□□ □ □=□□ . □

函数□ □ □, □□ □ □将□□的每个元素映射到“true”。因此，的每必须将 □的所有元素映射到 □的那些满足 □为“true”的元素。根据定义，这些是由特征函数 □指定的子集的元素。 因此， □的图像确实是问题中的子集。 回拉的普遍性保证了 □是单射的。

这个回拉图表可以用来定义除 □ □之外的范畴中的分类对象。这样的范畴必须有一个终端对象，这将使我们能够定义单态射 □ □ □□。 它还必须有回拉 —— 实际要求是它必须具有所有有限极限（回拉是有限极限的一个例子）。 在这些假设下，我们通过以下属性定义分类对象 Ω：对于每个单态射 □，存在一个唯一的态射 □完成回拉图表。

让我们分析最后的陈述。 当我们构建一个拉回时，我们给出了三个对象 Ω, □和 1；以及两个态射 □ □ □和□。 拉回的存在意味着我们可以找到最好的对象 □，

配备有两个态射 □和 □ □ □ □(后者由终对象的定义唯一确定），使得图表成立。

在这里，我们正在解决一个不同的方程组。 我们在求解 Ω和 □ □ □ 时同时变化 □和 □。 对于给定的 □和 □，可能有也可能没有单态射□ : : □ → □。 但是如果有一个，我们希望它是某个 □的拉回。 此外，我们希望这个 □由 □唯一确定。

我们不能说单态射 □和特征函数 □之间存在一一对应关系，因为拉回只是同构意义下的唯一性。 但请记住我们之前对子集的定义，即等价的嵌入的集合。 我们可以通过将子对象定义为到 □的等价单态射的集合来进行推广。 这个单态射的集合与我们的图的等价拉回的集合一一对应。

因此，我们可以定义 □的子对象集合 □ □ (□)为一组单态射，并且可以看到它与从 □到 Ω的态射集合同构：

$$□ □ (□) ≅ □(□, Ω)$$

这恰好是两个函子的自然同构。 换句话说， □ □ (−)是一个可表示的（逆变）函子，其表示对象是 Ω。

29.2 Topos

一个拓扑是一个范畴，它具有以下特性：

1. 是笛卡尔闭的：它具有所有的积、终对象和指数（定义为积的右伴随），

2. 对于所有有限图形都有极限，
3. 具有子对象分类器 Ω。

这些特性使得拓扑在大多数应用中都是理想的选择。 它还具有从定义中得出的其他属性。
例如，拓扑具有所有有限余极限，包括初始对象。

我们可能会诱惑地将子对象分类器定义为两个终对象的余积（和）的形式——这就是在集合论中的定义——但我们希望更加一般化。 满足这一条件的拓扑被称为布尔拓扑。

29.3 Topoi和逻辑

在集合论中，特征函数可以被解释为定义集合元素的属性——一个对某些元素为真，对其他元素为假的谓词。 谓词□ □ □ □从由自然数集合中选择出偶数的子集。 在一个拓扑中，我们可以将谓词的概念推广为从对象 □到 Ω的态射。 这就是为什么Ω有时被称为真值对象。

谓词是逻辑的基本构建块。 一个拓扑包含了研究逻辑所需的所有工具。 它具有与逻辑的合取（逻辑与）相对应的积，与析取（逻辑或）相对应的余积，以及与蕴含相对应的指数。 除了排中律（或者等价地说，双重否定消除律）之外，所有标准的逻辑公理在一个拓扑中都成立。 这就是为什么拓扑的逻辑对应于构造性或直觉主义逻辑。

直觉主义逻辑一直在稳步发展，并在计算机科学中找到了意想不到的支持。 排中律的经典概念基于这样的信念：存在绝对的真理：任何陈述要么为真，要么为假，或者如古罗马人所说的，第三种选择是不存在的。

non datur（没有第三种选择）。 但是我们只有在能够证明或证伪时才能知道某个陈述是真还是假。证明是一个过程，一种计算——我们知道计算需要时间和资源。 在某些情况下，它们可能永远不会终止。 如果我们无法在有限的时间内证明一个陈述，那么声称它为真就没有意义。 一个拓扑及其更细致的真值对象为建模有趣的逻辑提供了一个更通用的框架。

29.4 挑战

1. 证明函数 □是 □ □ □的拉回函数，必须是单射的。

30

Lawvere理论

N现在谈论函数式编程时，不能不提到单子。 但是在另一个宇宙中，由于偶然，Eugenio Moggi将他的注意力转向了Lawvere理论而不是单子。 让我们探索那个宇宙。

30.1 通用代数

有许多描述各种抽象层次的代数的方法。 我们试图找到一种通用语言来描述类似于幺半群、群或环的东西。 在最简单的层次上，所有这些构造都定义了对集合元素的操作，以及一些必须满足的规则。 例如，可以通过一个满足结合律的二元操作来定义幺半群。 我们还有一个单位元和单位规则。 但是稍微想象一下，我们可以将单位元转化为一个零元操作——一个不带参数并返回结果的操作。

集合的一个特殊元素。如果我们想讨论群，我们添加一个一元运算符，它接受一个元素并返回其逆元。 与之对应的左逆和右逆定律。一个环定义了两个二元运算符以及更多的定律。 等等。大致情况是，代数由一组对于不同的 □值的 □元操作和一组等式
定义。 这些等式都是普遍量化的。 结合律等式必须对于所有可能的三个元素的组合都成立，等等。顺便说一下，这排除了域的考虑，因为零（加法单位元）在乘法方面没有逆元。 域的逆定律不能普遍量化。

如果我们用态射替换操作（函数），则可以将这个通用代数的定义扩展到除了 □ □ 之外的范畴。 我们选择一个对象 □（称为通用对象）而不是一个集合。 一元运算只是 □的自同态。 但是其他arity（arity是给定操作的参数数量）怎么办？ 二元运算（arity 2）可以定义为从积□ × □到 □的态射。 一般的 □元操作是从 □的 □次幂到 □的态射：

□ □: : □ □→ □

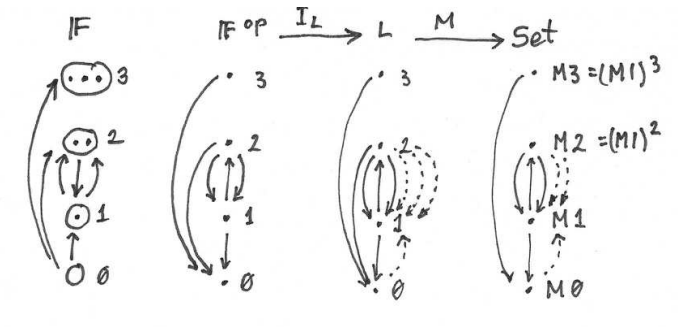
一个零元操作是从终端对象（零次幂 □）到的态射。 因此，为了定义任何代数，我们只需要一个对象是特殊对象 □的幂的范畴。 具体的代数编码在该范畴的同态集中。 这是一个Lawvere理论的简介。

Lawvere理论的推导经过了许多步骤，所以这是一个路线图：

1. 有限集合的范畴□ □ □ □。□ □

2. 它的骨架 □。
3. 它的对偶 □□ □。
4. Lawvere理论 □：范畴 □ □ 中的一个对象。
5. Lawvere范畴的模型 □：范畴

□ □ (□ □ ,□ □)中的一个对象。



30.2 Lawvere理论

所有的劳维尔理论都有一个共同的基础。 劳维尔理论中的所有对象都是使用乘积（实际上是幂）从一个对象生成的。 但是我们如何在一个普通的范畴中定义这些乘积？ 事实证明，我们可以使用一个简单范畴的映射来定义乘积。 实际上，这个简单范畴可能定义的是余乘积而不是乘积，我们将使用一个逆变函子将它们嵌入到目标范畴中。 一个逆变函子将余乘积转化为乘积，并将注入映射为投影映射。

作为一个Lawvere范畴的骨干，自然的选择是有限集合的范畴，□ □ □ □。□ 它包含空集 ∅，一个单元素集 1，一个两元素集 2，以此类推。这个范畴中的所有对象都可以

使用余积（将空集视为零元余积的特殊情况）从单例集生成。 例如，一个有两个元素的集合是两个单例集的和，2 = 1 + 1，如Haskell中所表示的：

```
类型 Two = 要么 () ()
```

然而，尽管自然地认为只有一个空集，但可能存在许多不同的单元素集合。 特别地，集合 1+∅与集合∅ + 1不同，并且与 1不同——尽管它们都是同构的。 在集合范畴中，余积不是结合的。 我们可以通过构建一个将所有同构集合等同起来的范畴来解决这个问题。 这样的范畴被称为骨架。 换句话说，任何 Lawvere 理论的骨架 □ of□ □ □ □是其支撑。 这个范畴中的对象可以与自然数（包括零）对应，这些自然数对应于□ □ □ □中的元素计数。 余积扮演了加法的角色。 在 □中，态射对应于有限集合之间的函数。 例如，从 ∅到 □有一个唯一的态射（空集是初始对象），从 □到 ∅没有态射（除了∅ → ∅），从 1到□有 □个态射（即插入），从 □到 1有一个态射，等等。这里，□表示 □中对应于所有 □元素集合的对象，这些集合通过同构被等同起来。

使用范畴 □我们可以正式地定义一个Lawvere理论作为一个范畴 □配备有一个特殊的函子

$$□ □ : : □ □ ⇸ □$$

这个函子必须在对象上是一个双射，并且它必须保持有限的积（在 □中的积与在 □中的余积相同）：

$$□ □ (□ × □) = □ □□ × □ □□$$

有时候你可能会看到这个函子被描述为对象上的恒等函子，这意味着 □和 □中的对象是相同的。 因此，我们将使用相同的名称来表示它们 —— 我们将用自然数来表示它们。
但请记住， □中的对象与集合不同（它们是同构集合的类）。

在 □中的同态集合一般比在 □中的同态集合更丰富。 它们可能包含除了对应于函数在□ □ □ □中的那些之外的态射（后者有时被称为基本积操作）。 Lawvere理论的等式定律被编码在这些态射中。

关键观察是，单例集合 1在 □中被映射到一个我们也称之为 1的对象，并且 □中的所有其他对象都是这个对象的幂。 例如，两个元素的集合 2在 □中是余积1 + 1，因此它必须被映射到一个积1 × 1（或
1^2）在 □中。从这个意义上说，范畴 □的行为类似于 □的对数。

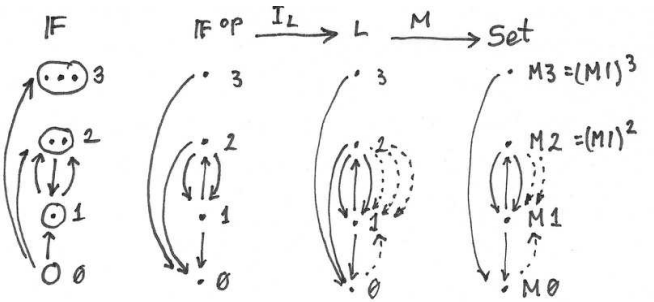
在 □中的态射中，我们有那些由函子 □□从 □转移的态射。它们在 □中起着结构性的作用。特别是余积的注入□□变成了积的投影 □□。一个有用的直觉是想象一下投影：

$$\square\ \square : : 1\square \to 1$$

作为一个忽略除了第 □个变量之外的 □个变量的函数的原型。 相反，在 □中，常数态射□ → 1变成了对角线态射$1 \to 1^{\square}$在 □中。 它们对应于变量的复制。

在 □中，有趣的态射是定义 □元操作的态射，而不仅仅是投影。 正是这些态射区分了一个Lawvere理论和另一个。 这些是乘法、加法、单位元素的选择等，它们定义了代数。 但是为了使 □成为一个完整的范畴，我们还需要复合操作□ → □（或者等价地， $1^{\square} \to 1^{\square}$）。 由于范畴的简单结构，它们实际上是更简单的态射的乘积。

类型□ → 1的 这是一个对函数返回乘积的一般化陈述的推广（或者，正如我们之前看到的，同态函子是连续的）。



Lawvere理论 □基于 □□，从中继承了定义产品的“无聊”态射。 它添加了描述 □元操作（带点的箭头）的“有趣”态射。

Lawvere理论形成了一个范畴 □ □ 𝔇 在这个范畴中，态射是保持有限积并与函子 □交换的函子。
给定两个这样的理论， $(\square, \square_{\square})$和 $(\square', \square_{\square}')$，它们之间的态射是一个函子□ : : □ → □′，满足以下条件：

□ (□ × □) = □ □ × □ □

□ ◦ □ □= □ □′ ′

Lawvere理论之间的态射封装了一个理论在另一个理论内部的解释的思想。 例如，如果我们忽略逆元，群乘法可以被解释为幺半群乘法。

最简单的Lawvere范畴的平凡例子是 □□ □本身（对应于选择恒等函子为 $\square_{\square}$）。 这个没有操作或规则的Lawvere理论恰好是这个这个这个中的初始对象。

在这一点上，展示一个非平凡的Lawvere理论将非常有帮助，但是在理解模型之前很难解释它。

30.3 Lawvere理论的模型

理解Lawvere理论的关键是意识到这样一个理论概括了许多共享相同结构的个体代数。
例如，幺半群的Lawvere理论描述了成为幺半群的本质。 它必须对所有幺半群都有效。 一个特定的幺半群成为这样一个理论的模型。 模型被定义为从Lawvere理论 □到集合范畴 □ □ 的函子。 （Lawvere理论的一般化使用其他范畴作为模型，但在这里我只关注 □ □ ）由于 □的结构严重依赖于积，我们要求这样一个函子保持有限积。 一个 □的模型，也称为Lawvere理论上的代数，因此由一个函子定义：

$$□ : : □ → □ □ □$$

$$□ \ (□ × □) ≅ □ □ × □ □$$

请注意，我们只需要保留乘积直到同构为止. 这非常重要，因为严格保留乘积会消除大部分有趣的理论.

模型对乘积的保留意味着 □在 □ □ 中的图像是由集合□ 1的幂生成的序列— 这是从对象 1的图像 □中生成的. 让我们称这个集合为 □. (有时这个集合被称为排序，这样的代数被称为单排序. 存在将Lawvere理论推广到多排序代数的泛化.) 特别地,

从 □到函数的映射:

$$□ \times □ \to □$$

与任何函子一样，可能会将 □中的多个态射折叠成 □ □ 中的同一个函数.

顺便说一下，所有的法则都是普遍量化的等式，这意味着每个劳维尔理论都有一个平凡的模型：一个将所有对象映射到单例集合，将所有态射映射到其上的恒等函数的常量函子。

在 □中，一般的态射形式□ → □被映射为一个函数：

$$□ \; □ \to □ \; □$$

如果我们有两个不同的模型 □和 □，它们之间的自然变换是一个由 □索引的函数族：

$$□ \; □ : : □ \; □ \to □ \; □$$

或者，等价地说：

$$□ \; □ : : □ \; □ \to □ \; □$$

其中□ = □ 1。

注意，自然性条件保证了对 □元操作的保持：

$$□ \; □ \circ □ \; □ = □_1 \circ □ \; □$$

其中□ : : □ → 1是 □元操作在 □中的一个操作。

定义模型的函子构成了一个模型的范畴，□ □ (□, □ □)，自然变换作为态射。

考虑一个平凡的Lawvere范畴的模型 □□ □。这样的模型完全由其在 1，□ 1的值决定。由于□ 1可以是任意的

集合，这些模型的数量与 □ □ 中的集合数量相同。 此外， □ □ (□□ □ □, □ □)中的每个态射（即函子 □和 □之间的自然变换）都由其在 □ 1的分量唯一确定。 反过来，每个函数□ 1 → □ 1都会引起两个模型 □和 □之间的自然变换。 因此， □ □ (□□ □ □, □ □)等价于 □ □ 。□

30.4 幺半群理论

最简单的非平凡示例是描述幺半群结构的Lawvere理论。 它是一个单一的理论，提炼了所有可能幺半群的结构，因为该理论的模型涵盖了幺半群范畴 □ □ 的整个范围。 我们已经看到了一个通用构造，它表明每个幺半群都可以通过识别一组态射来从适当的自由幺半群获得。 因此，一个单一的自由幺半群已经泛化了很多幺半群。 然而，自由幺半群有无穷多个。 幺半群的Lawvere理论 □□ □ □将它们全部结合在一个优雅的构造中。

每个幺半群必须有一个单位元，所以我们必须在 □□ □ □中有一个特殊的态射 □从 0到 1。 注意，在 □中不能有相应的态射。 这样的态射将会朝相反的方向，从 1到 0，在□ □ □ □中，□这将是从单元集合到空集合的函数。不存在这样的函数。

接下来，考虑态射2 → 1，即 □□ □ (2, 1)的成员，它们必须包含所有二元操作的原型。 在构建 □ □ (□□ □ □, □ □)中的模型时，这些态射将被映射到从笛卡尔积□ 1 × □ 1到□ 1的函数。 换句话说，这是两个参数的函数。

问题是：只使用幺半群运算符，能实现多少个两个参数的函数。 我们将这两个参数称为□和 □。 有一个函数忽略两个参数并返回幺半群单位。 然后有两个投影函数分别返回 □和 □。 接下来是返回 □ □、 □ □、 □ □、 □ □、 □ □等函数... 实际上，有与自由幺半群生成元 □和□中的元素数量相同的这样的两个参数函数。 注意 □□ □(2, 1)必须包含所有这些态射，因为其中一个模型是自由幺半群。 在自由幺半群中，它们对应于不同的函数。 其他模型可能会将多个态射合并为□□ □(2, 1)中的单个函数，但自由幺半群不会。

如果我们用 □个生成元来表示自由幺半群 □□，我们可以将同态集合 □(2, 1)与同态集合 □ □(1□, 2□)在 □ □中等同起来，即幺半群的范畴。 一般来说，我们选择 □□ □(□, □)为 □ □(□□□, □□)。 换句话说，范畴 □□ □是自由幺半群范畴的对偶。

幺半群的Lawvere理论的模型范畴□ □(□□□□, □□ □)与所有幺半群的范畴 □ □等价。

30.5 Lawvere理论和Monad

正如你可能记得的那样，代数理论可以用单子范畴来描述。 因此，不足为奇的是，Lawvere理论和单子之间存在联系。

首先，让我们看看Lawvere理论如何引出一个单子。 它通过一个遗忘函子和一个自由函子之间的伴随关系来实现。
遗忘函子 □将每个模型映射到一个集合。 这个集合通过对函子 □从 □ □(□□□, □□ □)在 □中的对象 1进行求值得到。

另一种导出 □的方法是利用 □[□]是 □ □ 中的初始对象这一事实。这意味着，对于任何Lawvere理论 □，都存在一个唯一的函子 □ □[□]→ □。这个函子在模型上诱导出相反的函子（因为模型是从理论到集合的函子）：

□ □ (□□, □ □)□→ □ □ (□□ □ ,□□ □)□

但是，正如我们讨论过的那样，模型的范畴 □[□]等价于 □ □ □，所以我们得到了遗忘函子：

□ : : □ □ (□□, □ □)□→ □ □ □

可以证明，这样定义的 □总是有一个左伴随，即自由函子 □。

对于有限集合来说，这很容易看出。自由函子 □产生自由代数。自由代数是 □ □ (□□, □ □)中的一个特定模型，它是由一组有限生成元 □生成的。我们可以将 □实现为可表示函子：

□(□, −) : : □ → □ □ □

为了证明它确实是自由的，我们只需要证明它是遗忘函子的左伴随：

□ □ (□□(□, −), □) ≅ □ □ (□□, □(□))

让我们简化右边的表达式：

□ □ (□□, □(□)) ≅ □ □ (□□, □ 1) ≅ (□ 1)□ ≅ □ □

（我使用了一个事实，即态射的集合同态于指数
在这种情况下，它只是迭代乘积。）伴随是
Yoneda引理的结果：

[□, □ □]□□(□, −), □) ≅ □ □

忘却函子和自由函子共同定义了一个单子 $T = U \circ F$
在 $\mathcal{C}$ 上。 因此，每个Lawvere理论都生成一个单子。
事实证明，这个单子的代数范畴等价于模型范畴。

你可能还记得，单子代数定义了使用单子形成的表达式的求值方法。 Lawvere理论定义了可以用来生成表达式的n元操作。 模型提供了评估这些表达式的方法。

单子和Lawvere理论之间的联系并不是双向的。 只有有限单子才能导致Lawvere理论。 有限单子基于有限函子。 在 $\mathcal{C}$ 上的有限函子完全由其对有限集的作用确定。 它对任意集合 A的作用可以使用以下余积来评估：

$$F A = \int^{n} F n \times A^{n}$$

由于余端广义化了余积，或者说和，这个公式是幂级数展开的广义化。 或者我们可以使用直觉，一个函子是一个广义的容器。 在这种情况下，一个有限容器of As可以被描述为形状和内容的和。 在这里，$F n$是一个存储 n个元素的形状集合，内容是一个 n元组的元素，本身是 A^{n}的一个元素。 例如，一个列表（作为一个函子）是有限的，每个arity都有一个形状。 一棵树每个arity有更多的形状，等等。

首先，从Lawvere理论生成的所有单子都是有限的，它们可以表示为余端：

$$T A = \int^{n} F n \times A(n, 1)$$

反过来，给定任何有限单子 □在 □ □ 止，我们可以构造一个Law-vere理论。 我们首先构造一个 □的Kleisli范畴。 正如你可能记得的那样，在Kleisli范畴中，从 □到 □的态射由底层范畴中的态射给出：

$$\square \to \square \square$$

当限制在有限集合上时，这变成了：

$$\square \to \square \square$$

与这个Kleisli范畴相对的范畴， □ □ □ □ □，限制在有限集合上，就是所讨论的Lawvere理论。 特别地，描述 □中n元操作的同态集合 □(□, 1)由同态集合 □ □(1, □)给出。

事实证明，我们在编程中遇到的大多数单子都是有限的，唯一的例外是延续单子。 可以将Lawvere理论的概念扩展到有限操作之外。

30.6 Monad作为共端

让我们更详细地探讨共端公式。

$$\square \square \square = \int^{\square} \square \square \times \square(\square, 1)$$

首先，这个共端公式是在一个定义为 □中的profunctor □上进行的。

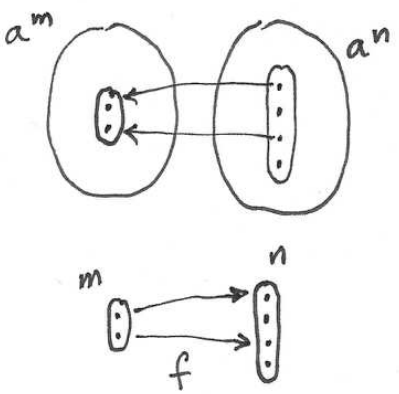
$$\square \square \square = \square \square \times \square(\square, 1)$$

这个profunctor对第一个参数 □是逆变的。 考虑它如何提升态射。 在□ □ □ □中,□态射是有限集合的映射。

□ : : □ → □. 这样的映射描述了从一个 □元素集合中选择 □个元素的情况（允许重复）。 它可以被提升为 □的幂的映射，即（注意方向）：

□ □→ □ □

提升简单地从一个元组中选择 □个元素（可能重复），该元组包含了 □个元素 $(□_1, □_2, ...□_□)$。



例如，让我们考虑 $□_□$: : 1 → □— 从一个 □元素集合中选择第 □个元素。它被提升为一个函数，该函数接受一个 □元素的元组并返回第 □个元素。

或者让我们来看一个将所有 □元素映射到一个元素的常数函数 □ : : □ → 1—它的提升是一个函数，它接受一个 □元素并将其复制 □次：

□ □→ (□, □, … , □ □ □ □ □ □ □ □) □ □ □ □
□

你可能注意到，很明显这个问题中的协变函子不是第二个参数的 。 同态函子□(□, 1)实际上是 □的逆变函子。 然而，我们在范畴 □ 而不是范畴 □中取共端。 共端变量 □遍历有限集合（或其骨架） 。 范畴 □包含 □的对偶，因此范畴 □中的态射□ → □是 □(□, □)中的成员（嵌入由函子 □□给出）。

让我们检查从 □到 □ □ 的函子性，其中 □(□, 1)是一个函子。 我们想要提升一个函数□ ∶∶ □ → □，所以我们的目标是实现一个从 □(□, 1)到 □(□, 1)的函数。 对应于函数 □，在 □中有一个从 □到 □ 的态射（注意方向）。 将这个态射与 □(□, 1)预合成，得到的是 □(□, 1)的一个子集。

$$\square(\square, 1) \longrightarrow \square(\square, 1)$$

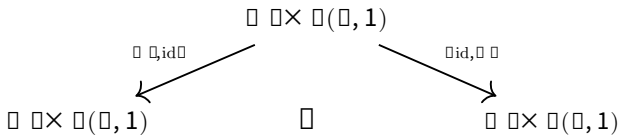
$$\square \cdot \xrightarrow[\square]{} \cdot \square$$

注意，通过提升一个函数1 → □，我们可以从 □(1, 1)到 □(□, 1)。 我们稍后会用到这个事实。

逆变函子 □□和协变函子□(□, 1)的乘积是一个半函子 □□ □× □ → □ □ ⫝̸ 记住，一个余端可以被定义为一个半函子的所有对角线成员的余和（不相交的和），其中一些元素被识别。 这些识别对应于余边条件。

在这里，余端从所有□上的集合 □ □× □(□, 1)的不相交和开始 。这些识别可以通过将余端表示为一个余等化器来生成。 我们从一个非对角线项 □□× □(□, 1)开始。 为了到达对角线，我们可以将一个态射 □ ∶∶ □ → □应用到

产品的第一个或第二个组成部分。 然后将这两个结果进行对应。



$$\square :: \square \to \square$$

我之前已经证明了将$\square :: 1 \to \square$的提升结果为这两个变换：

$$\square\ \square \to \square$$

和：

$$\square(1,1) \to \square(\square,1)$$

因此，从 $\square^{\square} \times \square(1,1)$开始，我们可以同时到达以下两个结果：

$$\square \times \square(1,1)$$

当我们提升$\square\ \square, \mathrm{id}\square$时和：

$$\square\ \square \times \square(\square,1)$$

当我们提升 $\square \mathrm{id}, \square\ \square$时。 然而，这并不意味着$\square^{\square} \times \square(\square,1)$的所有元素都可以与$\square \times \square(1,1)$进行对应。 这是因为并非所有 $\square(\square,1)$的元素都可以从 $\square(1,1)$到达。 请记住，我们只能从 $\square$中提升态射。 在 $\square$中，无法通过提升一个态射$\square :: 1 \to \square$来构造一个非平凡的 $\square$元操作。

换句话说，我们只能通过应用基本态射从 $\square(1,1)$到达 $\square(\square,1)$的余和公式中识别所有加数。 它们都等价于$\square \times \square(1,1)$。 基本态射是那些在 $\square$中的态射的图像。

让我们看看这在最简单的劳维尔理论的情况下是如何工作的，即 $\mathbb{N}^{\mathbb{N}}$本身。 在这样的理论中，每个 $\mathbb{N}(\mathbb{N}, 1)$都可以从$\mathbb{N}(1, 1)$到达。 这是因为 $\mathbb{N}(1, 1)$是一个只包含恒等态射的单子集，而 $\mathbb{N}(\mathbb{N}, 1)$只包含对应于在 $\mathbb{N}$中的插入$1 \to \mathbb{N}$的态射，即基本态射。 因此，余积中的所有加数是等价的，我们得到：

$$\mathbb{N}\ \mathbb{N} = \mathbb{N} \times \mathbb{N}(1, 1) = \mathbb{N}$$

这是身份单子。

30.7 副作用的Lawvere理论

由于单子和劳维尔理论之间有如此强烈的联系，自然而然地会问，劳维尔理论是否可以作为单子的替代品在编程中使用。 单子的主要问题是它们不能很好地组合。 构建单子变换器没有通用的方法。 劳维尔理论在这个领域有一个优势：它们可以使用余积和张量积进行组合。 另一方面，只有有限单子可以轻松转换为劳维尔理论。 这里的例外是延续单子。

这个领域正在进行研究（见参考文献）。

为了让你了解劳维尔理论如何用于描述副作用，我将讨论传统上使用 Maybe单子实现的异常的简单情况。

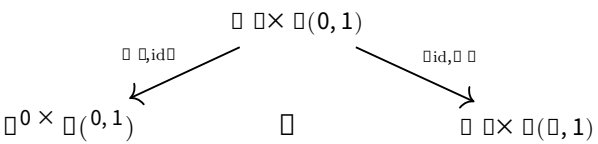
Maybe单子由具有单一零元操作$0 \to 1$的劳维尔理论生成。 这个理论的一个模型是将其映射为
1到某个集合 $\mathbb{N}$，并将零元操作映射为一个函数：

```
raise :: () -> a
```

我们可以使用余端公式恢复 Maybemonad。 让我们考虑一下空元操作对于同态集合 □(□, 1)的影响。
除了创建一个新的 □(0, 1)（在 □□ 中不存在），它还向 □(□, 1)中添加了新的态射。 这些是将类型为□ → 0的态射与我们的0 → 1组合的结果。 这些贡献在余端公式中都可以用 □0× □(0, 1)来表示，因为它们可以从以下方式获得：

$$\square\ \square\times \square(0,1)$$

通过两种不同的方式将0 → □提升。



$$\square :: 0 \to \square$$

共端减少为：

$$\square\ \square\square = \square^0 + \square^1$$

或者，使用Haskell符号表示：

类型Maybe a = Either () a

等价于：

data Maybe a = 无 | 有 a

请注意，这个Lawvere理论只支持异常的引发，
而不支持异常的处理。

30.8 挑战

1. 枚举在枚举中 2和 3之间的所有态射（之间是 □ □ □ □的骨架）。
2. 证明Lawvere理论的模型范畴等价于列表单子的单子代数范畴。
3. Lawvere理论生成了列表单子。 证明其二元操作可以使用相应的Kleisli箭头生成。

4. FinSet是 □ □的子范畴，并且存在一个函子将其嵌入到 □ □ □中。. 任何在 □ □止的函子都可以限制到□ □ □ □上。□ 证明有限函子是其自身限制的左Kan扩展。

30.9 进一步阅读

1. 代数理论的函子语义[1], F. William Lawvere
2. 计算概念决定单子[2], Gordon Plotkin and John Power

1http://www.tac.mta.ca/tac/reprints/articles/5/tr5.pdf
2http://homepages.inf.ed.ac.uk/gdp/publications/Comp_Eff_Monads.pdf

31

单子、幺半群和范畴

T没有一个好的地方来结束一本关于范畴论的书。 总是有更多东西可以学习。 范畴论是一个广阔的主题。 与此同时，很明显相同的主题、概念和模式一次又一次地出现。 有一句话说所有的概念都是Kan扩展，事实上，你可以使用Kan扩展来推导极限、余极限、伴随、单子、Yoneda引理等等。

范畴本身的概念在所有抽象层次上都出现，幺半群和单子的概念也是如此。 哪一个是最基础的？ 事实证明它们都是相互关联的，一个引导着另一个在一个永无止境的抽象循环中。 我决定展示这些相互关联可能是结束这本书的好方法。

31.1 双范畴

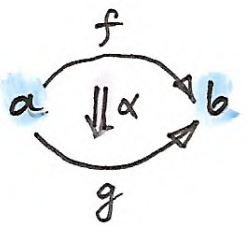
范畴论最困难的一个方面是不断切换视角。 以集合范畴为例。 我们习惯用元素来定义集合。 空集合没有元素。 单元素集合有一个元素。 两个集合的笛卡尔积是一个由有序对组成的集合，等等。 但是在谈论范畴两个两个两个时，我要求你忘记集合的内容，而是专注于它们之间的态射（箭头）。 你可以偶尔窥探一下，看看你可你可你可中特定的普遍构造是如何用元素来描述的。 终对象原来是一个有一个元素的集合，等等。但这些只是一些合理性检查。

函子被定义为范畴之间的映射。 将映射视为范畴中的态射是很自然的。 函子原来是范畴范畴中的态射（如果我们想避免关于大小的问题，则是小范畴）。 将函子视为箭头时，我们放弃了关于它对范畴内部（对象和态射）的作用的信息，就像我们将函数视为将函将函将函中的箭头时放弃了它对集合元素的作用的信息一样。 但是任意两个范畴之间的函子也构成一个范畴。

这次你被要求考虑在一个范畴中是箭头的东西在另一个范畴中是对象。 在一个函子范畴中，函子是对象，自然变换是态射。 我们发现同一件事可以在一个范畴中是箭头，在另一个范畴中是对象。 将对象视为名词，箭头视为动词的天真观点是不成立的。

我们可以尝试将两种观点合并成一个，而不是在两种观点之间切换。这就是我们得到一个 □-范畴的概念，其中ob-

对象被称为 0-细胞，态射被称为 1-细胞，态射之间的态射被称为 2-细胞。



0-细胞□，□；1-细胞□，□；和一个 2-细胞 □。

范畴的范畴 □ □ 是一个直接的例子。 我们将范畴作为 0-细胞，函子作为 1-细胞，自然变换作为2-细胞。 一个 □-范畴的定律告诉我们，任意两个 0-细胞之间的 1-细胞形成一个范畴（换句话说， □（□，□）是一个同态范畴而不是一个同态集合）。 这与我们之前的断言相吻合，即任意两个范畴之间的函子形成一个函子范畴。

特别是，从任何 0-细胞返回到自身的 1-细胞也形成一个范畴，即同态范畴 □(□, □)；但是这个范畴具有更多的结构。 属于 □(□, □)的成员可以被看作是 □中的箭头或者对象。 作为箭头，它们可以相互组合。 但是当我们将它们看作对象时，组合就变成了从一对对象到一个对象的映射。 实际上，它看起来非常像一个乘积——确切地说是一个张量乘积。 这个张量乘积有一个单位元：恒等 1-细胞。 事实证明，在任何2-范畴中，同态范畴□(□, □)自动成为一个带有以组合 1-细胞定义的张量乘积的幺半范畴。 结合律和单位元法则只是从相应的范畴法则中推导出来的。

让我们来看看这在我们的经典示例中意味着什么，一个 □-范畴 □ □ ⫰□ 同态范畴 □ □ (□□, □)是以 □为对象的自函子的范畴。 自函子的组合在其中扮演着张量积的角色。恒等函子是关于此积的单位元。 我们之前已经看到自函子构成了一个幺半范畴（我们在单子的定义中使用了这个事实），但现在我们看到这是一个更一般的现象：在任何 □-范畴中，自函子构成了一个幺半范畴。

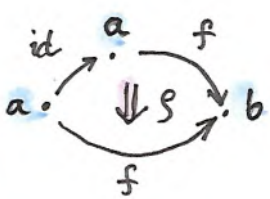
我们稍后会回到这个问题，当我们推广单子时。

你可能还记得，在一般的幺半范畴中，我们并没有坚持严格满足幺半律。 通常，满足单位律和结合律的同构就足够了。 在 □-范畴中， □(□, □)中的幺半律可以从 1-胞胎的组合律中推导出来。 这些律法是严格的，所以我们总是得到一个严格的幺半范畴。 然而，也有可能放宽这些律法。

我们可以说，例如，一个恒等 1-细胞 $\mathrm{id}_{□}$ 与另一个 1-细胞的组合，□ ∶∶ □ → □，是同构的，而不是相等的 □。 同构的 1-细胞是使用 2-细胞定义的。 换句话说，有一个 2-细胞：

$$□ ∶∶ □ ◦ \mathrm{id} ◦ \mathrm{i} → □$$

它有一个逆元。

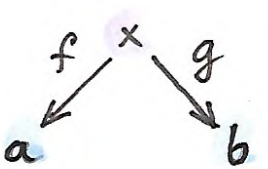


在一个双范畴中，恒等律成立，直到同构（一个可逆的 2-细胞）。

我们可以对左恒等律和结合律做同样的事情。 这种放松的 □-范畴被称为双范畴（这里还有一些额外的一致性法则，我将在这里省略）。

如预期的那样，在双范畴中，自身1-细胞形成一个具有非严格法则的一般幺半范畴。

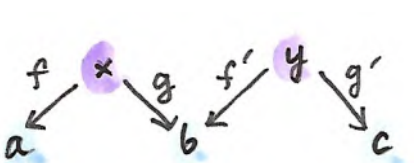
一个有趣的双范畴的例子是跨度范畴。 两个对象 □和 □之间的跨度是一个对象 □和一对态射：□ : : □ → □□ : : □ → □



你可能还记得，在范畴论的定义中我们使用了跨度。 在这里，我们想把跨度看作是一个双范畴中的1-胞腔。 第一步是定义跨度的组合。 假设我们有一个相邻的跨度：

□ ′ : : □ → □

□ ′ : : □ → □



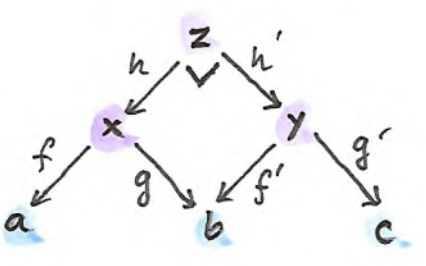
组合将是第三个跨度，带有一些顶点 □。 对于它来说，最自然的选择是 □沿着 □′的拉回。 记住，拉回是对象 □和两个态射的组合：

$$h :: □ \to □$$
$$h' :: □ \to □$$

满足以下条件：

$$□ \circ h = □' \circ h'$$

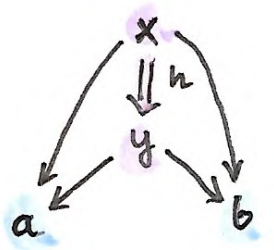
它在所有这样的对象之间是普遍的。



现在，让我们专注于在集合范畴上的跨度。 在这种情况下，拉回只是由笛卡尔积□ × □中的一对 (□, □)组成的集合，满足以下条件：

$$□\,□ = □'\,□$$

两个共享相同端点的跨度之间的态射被定义为它们顶点之间的态射 h，使得适当的三角形可交换。



在 □ □ □中的一个 2-细胞。

总结一下，在双范畴 □ □ □中: 0-细胞是集合， 1-细胞是跨度，2-细胞是跨度态射。 一个恒等 1-细胞是一个退化的跨度，其中三个对象都相同，两个态射都是恒等态射。

我们之前见过另一个双范畴的例子：范畴□ □ □的双范畴，其中0-细胞是范畴，1-细胞是双函子，2-细胞是自然变换。 双函子的合成由一个余积给出。

31.2 Monad

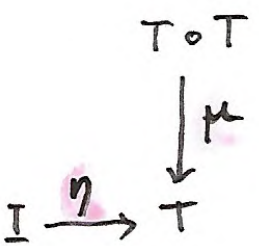
到现在为止，你应该对在自函子范畴中定义的单子范畴的定义非常熟悉。 让我们重新审视这个定义，理解到自函子范畴只是双范畴 □ □ □中的一个小的同态范畴的一部分。 我们知道它是一个幺半范畴：张量积来自自函子的合成。 幺半范畴的定义是一个对象在幺半范畴中的一个对象——在这里它将是一个自函子 □——以及两个态射。 自函子之间的态射是自然变换。

一个态射将单子范畴的单位元——恒等自函子——映射到 □:

$$\square :: \square \to \square$$

第二个态射将□ ⊗□的张量积映射到 □。 张量积由自函子的复合给出，因此我们得到:

$$\square :: \square \circ \square \to \square$$

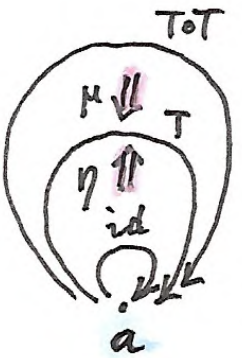


我们将其识别为定义单子的两个操作（在Haskell中称为 return和 join），并且我们知道幺半群定律转化为单子定律。

现在让我们从这个定义中删除所有关于自函子的提及。
我们从一个双范畴 □开始，并选择其中的一个 0-细胞 □。正如我们之前看到的，同态范畴 □(□, □)是一个幺半范畴。 因此，我们可以通过选择一个 1-细胞 □和两个 2-细胞来在 □(□, □)中定义一个幺半

$$\square :: \square \to \square$$
$$\square :: \square \circ \square \to \square$$

满足幺半群定律。我们称之为单子。

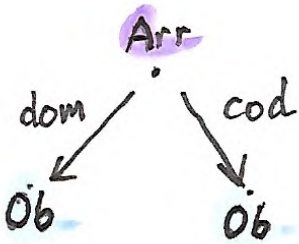


这是一个更一般的单子定义，只使用 0-细胞，
1-细胞和 2-细胞。 当应用于范畴 □ □ 时，它简化为常规单子。 但是让我们看看在其他范畴中会发生什么。

让我们在 □ □ □中构建一个单子。 我们选择一个 0-细胞，它是一个集合， 接下来，我们选择一个从 □ □返回 □ □的内部-1-细胞。 它在顶点处有一个集合，我将其称为 □ □， 并配备两个函数：

$$\square\ \square\ \square: \square\ \square \to \square\ \square$$

$$\square\ \square\ \square: \square\ \square \to \square\ \square$$

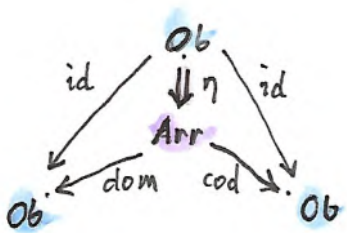


让我们将集合 □ □的元素称为“箭头”。如果我还告诉你将 □ □的元素称为“对象”，你可能会猜到这将导致什么。这两个函数 □ □ □和 □ □将定义域和共域分配给一个“箭头”。

为了将我们的跨度变成一个单子，我们需要两个 2-细胞， □和□。 在这种情况下，单子单位是从 □ □到 □ □的平凡跨度，其顶点在 □ □止，并且有两个恒等函数。 2-细胞 □是一个函数，它在顶点 □ □和 □ □之间。 换句话说， □将每个“对象”分配给一个“箭头”。在 □ □ □中，一个 2-细胞必须满足交换条件 - 在这种情况下：

$$\square \square \square \square = \mathrm{id}$$

$$\square \square \square \square = \mathrm{id}$$



在分量中，这变成了：

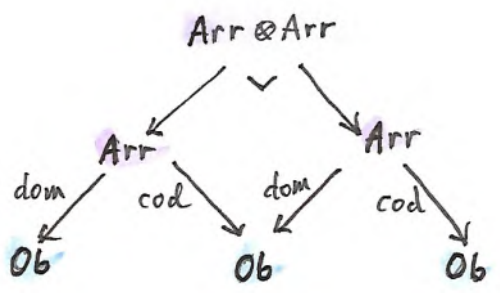
$$\mathrm{dom}(\mathrm{dom}\,f) = \mathrm{dom}\,f = \mathrm{dom}(\mathrm{cod}\,f)$$

其中 $\mathrm{dom}\,f$ 是 dom 中的一个“对象”。换句话说，dom 为每个“对象”和其定义域和值域为该“对象”的“箭头”分配一个特殊的“箭头”。我们将其称为“恒等箭头”。

第二 2-细胞 dom 作用于跨度 dom 与自身的组合。组合被定义为一个拉回，因此它的元素是来自 dom 的元素对——“箭头”对 (f_1, f_2)。拉回条件是：

$$\mathrm{dom}\,f_1 = \mathrm{dom}\,f_2$$

我们说 f_1 和 f_2 是“可组合”的，因为一个的定义域是另一个的共域。



2-细胞 dom 是一个将可组合箭头对 (f_1, f_2) 映射到单个箭头 f_3 的函数。换句话说，dom 定义了箭头的组合。

很容易验证单子律对应于箭头的恒等律和结合律。我们刚刚定义了一个范畴（一个小范畴，注意，其中对象和箭头形成集合）。

所以，总的来说，一个范畴只是一个双范畴中的单子。

令人惊奇的是，这个结果将范畴与其他代数结构（如单子和幺半群）放在了同一水平上。

成为一个范畴并没有什么特别之处。 它只是两个集合和四个函数。 实际上，我们甚至不需要为对象单独设置一个集合，因为对象可以与恒等箭头一一对应。 所以，它实际上只是一个集合和几个函数。考虑到范畴论在数学中的关键作用，这是一个非常令人谦卑的认识。

31.3 挑战

1. 推导在双范畴中定义的张量积作为内自然变换组合的单位和结合律。
2. 检查单子在 □ □ □中的单子定律是否对应于所得范畴中的恒等和结合律。
3. 证明在 □ □ □中的单子是一个对象上的恒等函子。
4. 在 □ □ □中的单子代数是什么？

31.4 参考文献

1. Paweł Sobociński的博客[1]。

1https://graphicallinearalgebra.net/2017/04/16/

一个幺半群是一个范畴，一个范畴是一个单子，一个单子是一个幺半群。

索引

> 此索引中的任何不准确之处可能是因为它是通过计算机的帮助准备的。
>
> —Donald E. Knuth，基本算法
> （《计算机程序设计艺术》第1卷）

致谢

我要感谢Edward Kmett和Gershom Bazerman检查我的数学和逻辑，以及André van Meulebrouck，在这一系列帖子中一直自愿提供编辑帮助。

我想感谢安德鲁·萨顿根据他和Bjarne Stroustrup的最新提案重写了我的C++单子概念代码。

我感谢Eric Niebler阅读了草稿，并提供了使用C++14的高级特性来推断类型的巧妙实现。 我能够删除了使用类型特征进行相同操作的整个旧式模板魔法部分。 太好了我也感谢Gershom Bazerman对我澄清一些重要观点的有用评论。

后记

这本书是由Igal Tabachnik[2]编译的，通过将Bartosz Milewski的原始文本转换为LaTeX格式，首先使用Mercury Web Parser[3]从原始WordPress博客文章中获取干净的HTML内容，使用Beautiful Soup[4]进行修改和调整，最后使用Pandoc[5]转换为LaTeX。

正文使用的字体是Linux Libertine，标题使用的字体是Linux Biolinum，两者都是由Philipp H. Poll设计的。打字机字体是Inconsolata，由Raph Levien设计，并由Dimosthenis Kaponis和Takashi Tanigawa补充了Inconsolatalgc版本。封面字体是Alegreya，由Juan Pablo del Peral设计。

原始书籍的布局设计和排版由Andres Raba完成。

2https://hmemcpy.com
3https://mercury.postlight.com/web-parser/
4https://www.crummy.com/software/BeautifulSoup/
5https://pandoc.org/

版权声明

6https://www.gnu.org/philosophy/free-sw.en.html