

```
'keywordstyle=blue,  
commentstyle=green,  
stringstyle=red,  
morecomment=[1] [magenta]--,  
breaklines=true  
'keywordstyle=blue,  
commentstyle=green,  
stringstyle=red,  
breaklines=true
```

Category Theory for Programmers

By Bartosz Milewski

compiled and edited by
Igal Tabachnik

CATEGORY THEORY FOR PROGRAMMERS

Bartosz Milewski

Version
2024 年 8 月 26 日



This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0).

Converted from a series of blog posts by Bartosz Milewski.
PDF and book compiled by Igal Tabachnik.

TeX source code is available on GitHub:
<https://github.com/hmemcpy/milewski-ctfp-pdf>

目录

Preface

你好啊最近一段时间，我一直在酝酿写一本关于范畴论 (Category Theory) 的书，目标读者是程序员 (programmers)，注意，不是计算机科学家 (computer scientists)，而是程序员——工程师 (engineers)，而不是科学家 (scientists)。我知道这听起来很疯狂，而且我也确实感到害怕。我不能否认科学与工程之间存在巨大的差距，因为我在这两方面都有工作经验。但我总是有一种强烈的冲动去解释事物。我对理查德·费曼 (Richard Feynman) 有着极大的敬佩，他是简明解释的大师。我知道我不是费曼，但我会尽力而为。我首先发布这篇序言，希望能激发读者学习范畴论的兴趣，并希望借此展开讨论并征求反馈意见。¹

Haskell 代码示例 0000

```
f :: A -> B
toExp :: (a -> b) -> Exp a b
fromExp (Lan f (I x)) = \a -> f (a, x)
instance Functor (FreeF f) where
    fmap g (FMap h fi) = FMap (g . h) fi
```

Haskell 代码示例 11111

```
f :: A -> B
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)

instance Functor (FreeF f) where
    fmap g (FMap h fi) = FMap (g . h) fi
```

¹你也可以观看我向现场观众讲授这部分内容的视频，网址为 <https://goo.gl/GT2UWU>（或在 YouTube 上搜索“bartosz milewski category theory”）。

OCaml 代码示例 11111

```
module type Polymorphic_Function_F = sig
  type a
  type b

  val f : a -> b
end
module FreeFunctor (F : sig
  type 'a f
end) : Functor = struct
  module type F = FreeF with type 'a f = 'a F.f

  type 'a t = (module F with type a = 'a)

  let fmap
    (type a' b')
    (f : a' -> b')
    (module FF : F with type a = a')
    =
    (module struct
      type 'a f = 'a F.f
      type a = b'
      type i = FF.i

      let h i = f (FF.h i)
      let fi = FF.fi
    end : F
    with type a = b')
  ;;
end
```

我 将尝试在几段文字中说服你，这本书是为你而写，无论你对在“充裕的业余时间”里学习最抽象的数学分支之一有何反对意见，这些反对意见都是毫无根据的。

我的乐观基于几个观察。首先，范畴论（Category Theory）是极为有用的编程思想的宝库。Haskell 程序员已经很长时间在利用这一资源，这些思想正在慢慢渗透到其他语言中，但这一过程太慢了。我们需要加快这个过程。

其次，不同类型的数学吸引着不同的受众。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。我甚至要说，范畴论是特别适合程序员（programmers）思维的一类数学。这是因为范畴论并不处

理特定事物，而是处理结构（structure）。它处理的是使程序可组合的那类结构。

组合（Composition）是范畴论的根本所在——它是范畴（category）本身定义的一部分。而且我会强烈主张，组合是编程的本质。我们一直在进行组合，早在某位伟大的工程师提出子程序的概念之前，我们就在组合事物。早些时候，结构化编程的原则（principles of structured programming）因为使代码块可组合而彻底改变了编程。然后面向对象编程（object oriented programming）出现了，面向对象编程的核心就是对象的组合。函数式编程（functional programming）不仅涉及函数和代数数据结构的组合，它还使并发（concurrency）可组合——这是在其他编程范式中几乎不可能做到的。

第三，我有一件秘密武器，一把屠刀，我将用它来屠宰数学，使之更易于被程序员接受。作为一名专业的数学家，你必须非常小心，确保所有假设都正确，恰当地限定每一个陈述，并严谨地构造所有证明。这使得数学论文和书籍对于外行人来说极难阅读。我是受过物理学训练的，在物理学中，我们通过非正式推理取得了惊人的进展。数学家曾嘲笑狄拉克 δ 函数（Dirac delta function），这是伟大的物理学家 P. A. M. Dirac 在现场编造出来的，用以解决一些微分方程。当数学家们发现一个全新的分支——称为分布理论（distribution theory）——并将狄拉克的洞见形式化时，他们才停止了嘲笑。

当然，使用这种挥手自如的论据时，你有可能说出明显错误的东西，因此我会尽量确保书中的非正式论据背后有扎实的数学理论支持。我床头放着一本已经用旧了的桑德斯·麦克莱恩（Saunders Mac Lane）所著的《工作数学家的范畴论》（Categories for the Working Mathematician）。

由于这是面向程序员的范畴论（Category Theory for Programmers），我将使用计算机代码来说明所有主要概念。你可能知道，函数式语言（functional languages）比更流行的命令式语言（imperative languages）更接近数学。它们也提供了更强的抽象能力。因此，一个自然的诱惑是：你必须学习 Haskell，才能享受到范畴论的好处。但这意味着范畴论在函数式编程之外没有应用，这是完全不真实的。因此，我会提供很多 C++ 示例。当然，你需要克服一些丑陋的语法，这些模式可能不会从冗长的背景中脱颖而出，并且你可能需要进行一些复制粘贴，以代替更高的抽象，但这就是 C++ 程序员的命运。

但在 Haskell 方面你并非完全摆脱了困扰。你不必成为 Haskell 程序员，但你需要将其作为一种草图语言，用来记录和实现 C++ 中的想法。这正是我开始学习 Haskell 的方式。我发现其简洁的语法和强大的类型系统对理解和实现 C++ 模板、数据结构和算法大有帮助。但由于我不能指望读者已经了解 Haskell，我会慢慢介绍，并逐步解释一切。

如果你是有经验的程序员，你可能会问自己：我编程这么久了，从未关心过范畴论或函数式方法，那又有什么改变呢？你肯定注意到，新的函数式特性（functional features）正在持续不断地入侵命令式语言。即使是面向对象编程的堡垒 Java 也引入了 lambda 表达式。C++ 最近一

一直在以狂热的速度发展——每隔几年就会有一个新标准——试图跟上变化的世界。所有这些活动都是为一种颠覆性的变化做准备，或者像我们物理学家所说的那样，是一种相变（phase transition）。如果你不断加热水，它最终会开始沸腾。我们现在就像一只青蛙，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。



推动这一重大变化的力量之一是多核革命（multicore revolution）。现行的编程范式——面向对象编程——在并发性和并行性方面毫无用处，反而鼓励危险且漏洞百出的设计。数据隐藏（data hiding），作为面向对象的基本前提，当与共享和变异相结合时，就成为数据竞争的配方。将一个互斥量（mutex）与它所保护的数据结合的想法很好，但不幸的是，锁（locks）不能组合，而锁的隐藏使得死锁（deadlocks）更有可能发生，也更难调试。

但即使在没有并发性的情况下，日益复杂的软件系统也在考验命令式编程范式的可扩展性。简单地说，副作用（side effects）正在失控。诚然，具有副作用的函数往往方便且易于编写。它们的效果原则上可以通过它们的名字和注释来编码。一个名为 SetPassword 或 WriteFile 的函数显然在改变某种状态并产生副作用，我们习惯于处理这种情况。只有当我们开始在具有副作用的函数之上组合另一个具有副作用的函数，如此反复，问题才会变得复杂起来。问题不在于副作用本身，而在于它们被隐藏，使得在更大规模上无法管理。副作用不具备可扩展性，而命令式编程完全依赖副作用。

硬件的变化和软件复杂性的增加迫使我们重新思考编程的基础。就像欧洲伟大哥特式教堂的建造者一样，我们已经将手艺磨练到了材料和结构的极限。在法国博韦（Beauvais），有一座未完工的哥特式大教堂，见证了这种深深的人类对抗限制的斗争。它本来打算打破所有之前的高度和轻盈记录，但经历了一系列坍塌。像铁杆和木质支撑这样的临时措施防止了它的解体，但显然很多事情出了问题。从现代的角度来看，许多哥特式结构在没有现代材料科学、计算机建模、有限元分析以及广义数学和物理学的帮助下顺利完成，是一个奇迹。我希望未来的几代人能够像钦佩哥特式建筑的建造者一样，钦佩我们在构建复杂操作系统、Web 服务器和互联网基础设施方面展示的编程技能。而且，坦率地说，



防止博市大教堂坍塌的临时措施。

他们应该钦佩，因为我们所有这些都是基于非常薄弱的理论基础。如果我们想要前进，就必须修复这些基础。

Part One

1

Category: The Essence of Composition

范畴

CATEGORY 是一个令人难以置信的简单概念。一个范畴由对象 (objects) 和在它们之间的箭头 (arrows) 组成。因此，范畴非常容易用图形表示。一个对象可以画成一个圆或一个点，而一个箭头... 就是一个箭头。（为了增加一些变化，我偶尔会把对象画成小猪，把箭头画成烟火。）但范畴的本质是组合 (composition)。或者，如果你愿意，组合的本质就是一个范畴。箭头是可以组合的，所以如果你有一个从对象 A 到对象 B 的箭头，以及另一个从对象 B 到对象 C 的箭头，那么一定存在一个箭头——它们的组合——从 A 到 C。

1.1 箭头作为函数 (Arrows as Functions)

这是不是已经太过抽象了？别灰心。我们来谈些具体的内容。将箭头 (arrows)，也称为态射 (morphisms)，视为函数。你有一个函数 f ，它接受一个类型为 A 的参数，并返回一个类型为 B 的值。你还有另一个函数 g ，它接受一个 B 并返回一个 C 。你可以通过将 f 的结果传递给



在一个范畴中，如果有一个箭头从 A 到 B ，并且有另一个箭头从 B 到 C ，那么必然存在一个直接从 A 到 C 的箭头，这就是它们的组合。这个图示并不是一个完整的范畴，因为它缺少了恒等态射 (identity morphisms)，稍后会提到。

g 来组合它们。这样你就定义了一个新函数，它接受一个 A 并返回一个 C 。

在数学中，这样的组合用函数之间的一个小圆圈来表示： $g \circ f$ 。请注意，组合的顺序是从右到左的。对于某些人来说，这可能有些令人困惑。你可能熟悉 Unix 中的管道符号，如：

或者在 F# 中的尖括号，它们都是从左到右进行操作的。但是在数学和 Haskell 中，函数的组合是从右到左的。如果你将 $g \circ f$ 理解为“ f 之后的 g ”，这会有所帮助。

让我们通过写一些 C 代码来使这一点更加明确。我们有一个函数，它接受一个类型为 的参数，并返回一个类型为 的值：

还有另一个函数：

它们的组合是：

这里，你再次看到从右到左的组合：；这次是在 C 语言中。

我希望我能告诉你，在 C++ 标准库中有一个模板可以接受两个函数并返回它们的组合，但实际上没有。所以我们来试试 Haskell。以下是从 A 到 B 的函数声明：

snippet01 类似地：

snippet02 它们的组合是：

snippet03 一旦你看到 Haskell 中的这些东西有多么简单，无法在 C++ 中表达简单的函数概念就有点尴尬了。事实上，Haskell 允许你使用 Unicode 字符，所以你可以这样写组合：

你甚至可以使用 Unicode 的双冒号和箭头：

::

所以这是第一课 Haskell：双冒号表示“类型为...”。一个函数类型是通过在两个类型之间插入一个箭头创建的。你通过在两个函数之间插入一个点（或 Unicode 圆圈）来组合它们。

1.2 组合的性质 (Properties of Composition)

在任何范畴中，组合必须满足两个极为重要的性质。

- 组合是结合的。如果你有三个可以组合的态射， f , g 和 h , (即它们的对象端对端匹配)，你在组合它们时不需要括号。在数学符号中，这表示为：

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

在（伪）Haskell 中：

snippet04[b] （我说“伪”，因为函数的等号并没有定义。）

当处理函数时，结合性是显而易见的，但在其他范畴中可能并不明显。

- 对于每个对象 A , 都有一个箭头，它是组合的单位。这个箭头从对象自身回到自身。作为组合的单位意味着，当与任何从 A 开始或结束于 A 的箭头组合时，它分别返回相同的箭头。对象 A 的单位箭头被称为 id_A (A 的恒等 (identity))。在数学符号中，如果 f 从 A 到 B , 那么

$$f \circ \text{id}_A = f$$

并且

$$\text{id}_B \circ f = f$$

在处理函数时，恒等箭头实现为恒等函数 (identity function)，它仅返回其参数。这个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：

当然，在 C++ 中，事情没有那么简单，因为你不仅要考虑你传递的是什么，还要考虑如何传递（即按值、按引用、按 `const` 引用、按移动等）。

在 Haskell 中，恒等函数是标准库 (Prelude) 的一部分。以下是其声明和定义：

snippet05 正如你所见，在 Haskell 中，多态函数非常简单。在声明中，你只需用类型变量替换类型。诀窍在于：具体类型的名称总是以大写字母开头，类型变量的名称以小写字母开头。所以这里的 `a` 代表所有类型。

Haskell 函数定义由函数名和形式参数组成——这里只是一个。函数体在等号后面。对于新手来说，这种简洁性常常令人震惊，但你很快就会发现它是非常合理的。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小。不仅参数列表没有括号，参数之间也没有逗号（稍后你会看到，当我们定义多参数函数时）。

函数体总是一个表达式——函数中没有语句。函数的结果就是这个表达式——这里只是。

这就是我们的第二课 Haskell。

恒等条件可以写成（再次在伪 Haskell 中）：

snippet06 你可能会问自己：为什么有人会在意恒等函数——一个什么都不做的函数？再想想，为什么我们要在意数字零？零是无的符号。古罗马人的数字系统中没有零，他们依然能够建造出卓越的道路和水渠，其中一些至今仍在使用。

像零或 `id` 这样的中立值在处理符号变量时非常有用。这就是为什么古罗马人在代数方面并不擅长，而阿拉伯人和波斯人，由于熟悉零的概念，则擅长代数。所以恒等函数在作为高阶函数的参数或返回值时非常方便。高阶函数使得函数的符号操作成为可能。它们是函数的代数。

总结：一个范畴由对象和箭头（态射）组成。箭头可以组合，组合是结合的。每个对象都有一个恒等箭头，作为组合下的单位。

1.3 组合是编程的本质 (Composition is the Essence of Programming)

函数式程序员有一种独特的方式来处理问题。他们首先会问一些非常禅宗式的问题。例如，当设计一个交互式程序时，他们会问：什么是交互？当实现康威的生命游戏时，他们可能会思考生命的意义。在这种精神下，我要问：什么是编程？在最基本的层面上，编程是告诉计算机该做什么。“取内存地址 `x` 的内容，并将其添加到寄存器 `EAX` 的内容中。”但即使我们用汇编语言编程，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（如果是平凡的问题，我们就不需要计算机的帮助了）。我们如何解决问题？我们将大问题分解为小问题。如果小问题仍然太大，我们进一步分解它们，依此类推。最后，我们编写代码来解决所有的小问题。然后，编程的本质就来了：我们将这些代码片段组合起来，形成解决更大问题的方案。如果不能将这些部分重新组合在一起，分解就毫无意义。

这种分层分解和重组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。心理学中被引用最多的论文之一，《神奇的七，正负二》¹，假设我们的大脑只能容纳 7 ± 2 个“块”信息。我们对人类短期记忆的理解细节可能在变化，但我们可以肯定的是，它是有限的。关键是我们无法处理一锅杂物或代码拼盘。我们需要结构，不是因为结构良好的程序看起来赏心悦目，而是因为否则我们的大脑无法有效地处理它们。我们经常形容某段代码优雅或美丽，但我们真正的意思是它易于我们有限的人类大脑处理。优雅的代码创造了正好大小的块，数量正好适合我们的精神消化系统来吸收它们。

那么，程序组合的合适块是什么？它们的表面积必须比它们的体积增长得慢。（我喜欢这个类比，因为几何物体的表面积随着其大小的平方增长——比体积增长得慢，体积随着其大小的立方增长。）表面积是我们需要的信息，用以组合块。体积是我们需要的信息，用以实现它

¹

们。这个想法是，一旦实现了一个块，我们就可以忘记其实现的细节，并专注于它如何与其他块交互。在面向对象编程中，表面就是对象的类声明或其抽象接口。在函数式编程中，它就是函数的声明。(我稍微简化了一些，但这就是要点。)

范畴论在这一点上是极端的，因为它积极地阻止我们查看对象的内部。范畴论中的一个对象是一个抽象的朦胧实体。你唯一能知道的就是它如何与其他对象关联——如何用箭头连接它们。这就是互联网搜索引擎通过分析进出链接来对网站进行排名的方式（除非它们作弊）。在面向对象编程中，一个理想化的对象只能通过其抽象接口可见（纯表面，无体积），方法扮演箭头的角色。当你必须深入研究对象的实现以理解如何将其与其他对象组合时，你就失去了编程范式的优势。

1.4 挑战 (Challenges)

1. 尽你所能，在你最喜欢的語言中实现恒等函数 (identity function) (如果你最喜欢的語言是 Haskell，请在你第二喜欢的語言中实现)。
2. 在你最喜欢的語言中实现组合函数 (composition function)。它接受两个函数作为参数，并返回它们的组合。
3. 编写一个程序，尝试测试你的组合函数是否遵守恒等性。
4. 在任何意义上，万维网 (world-wide web) 是一个范畴吗？链接是态射吗？
5. Facebook 是一个范畴吗？人是对象，友谊是态射？
6. 何时一个有向图是一个范畴？

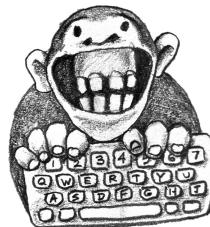
2

Types and Functions

类型与函数的范畴在编程中起着重要作用，所以让我们讨论一下什么是类型以及为什么我们需要它们。

2.1 谁需要类型? (Who Needs Types?)

关于静态 vs 动态，以及强类型 vs 弱类型的优点似乎存在一些争议。让我用一个思想实验来说明这些选择。想象一下，数百万只猴子在计算机键盘上开心地敲打随机键，编写程序、编译并运行它们。



在机器语言中，猴子产生的任何字节组合都会被接受并运行。但在更高级的语言中，我们确实感谢编译器能够检测到词法和语法错误。许多猴子将得不到香蕉，但剩下的程序将有更大的机会有用。类型检查提供了另一道屏障，以防止无意义的程序。此外，在动态类型语言中，类型不匹配将在运行时被发现，而在强类型静态检查语言中，类型不匹配在编译时就会被发现，从而在程序有机会运行之前消除许多错误的程序。

所以问题是，我们是想让猴子开心，还是想产生正确的程序？

在打字猴子思想实验中的通常目标是生产出莎士比亚的全集。让拼写检查器和语法检查器参与其中将大大提高成功的几率。类型检查器的类似物甚至会更进一步，确保一旦罗密欧被宣告为人类，他不会长出叶子或在他强大的引力场中捕获光子。

2.2 类型与组合性 (Types Are About Composability)

范畴论是关于箭头的组合的。但并不是任何两个箭头都可以组合。一个箭头的目标对象必须与下一个箭头的源对象相同。在编程中，我们将一个函数的结果传递给另一个函数。如果目标函数不能正确解释源函数生成的数据，程序将无法运行。为了使组合起作用，两端必须匹配。语言的类型系统越强，这种匹配就越容易描述和机械验证。

我听到的唯一反对强静态类型检查的严肃论点是，它可能会排除一些语义上正确的程序。实际上，这种情况极为罕见，而且无论如何，每种语言都提供了某种绕过类型系统的后门，当确实有必要时。即使是 Haskell 也有。但这种手段应谨慎使用。弗朗茨·卡夫卡 (Franz Kafka) 的角色，格雷戈尔·萨姆萨 (Gregor Samsa)，当他变形为一只巨大的虫子时，打破了类型系统，我们都知道这是如何结束的。

我经常听到的另一个论点是，处理类型给程序员带来了太多的负担。在自己写了几个 C++ 中迭代器的声明之后，我可以理解这种情绪，除了有一种叫做类型推断 (type inference) 的技术，它让编译器可以从使用它们的上下文中推断出大多数类型。在 C++ 中，你现在可以声明一个变量为，让编译器确定它的类型。

在 Haskell 中，除了极少数情况下，类型注释是纯粹可选的。程序员通常还是会使用它们，因为它们可以传达大量的代码语义，并使编译错误更易于理解。在 Haskell 中，一个常见的做法是先设计类型，然后，类型注释驱动实现，并成为编译器强制的注释。

强静态类型通常被用作不测试代码的借口。你有时会听到 Haskell 程序员说，“如果它编译了，那它就一定是正确的。”当然，没有任何保证类型正确的程序在产生正确输出的意义上也是正确的。结果是，在几项研究中，Haskell 在代码质量方面的表现并没有像预期的那样遥遥领先。看来，在商业环境中，修复错误的压力只会施加到某个质量水平，这与软件开发的经济性和最终用户的容忍度息息相关，而与编程语言或方法论几乎无关。更好的标准应该是衡量有多少项目落后于进度或以大幅减少的功能交付。

至于单元测试可以取代强类型的论点，请考虑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。在弱类型语言中，函数现在期望不同数据的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，但测试几乎总是一个概率过程，而不是确定性的。测试是证明的糟糕替代品。

2.3 什么是类型? (What Are Types?)

对类型最简单的直觉是它们是值的集合。类型（记住，在 Haskell 中，具体类型以大写字母开头）是一个包含 和 的两元素集合。类型是所有 Unicode 字符的集合，如 或 。

集合可以是有限的或无限的。类型 是 列表的同义词，它是一个无限集合的例子。

当我们声明 是 时：

snippet01 我们是在说它是整数集的一个元素。Haskell 中的 是一个无限集合，可以用于任意精度的算术运算。还有一个有限集合 对应于机器类型，就像 C++ 中的 。

有些微妙之处使得将类型与集合等同变得棘手。涉及循环定义的多态函数有问题，而且你不能拥有所有集合的集合；但正如我所承诺的，我不会拘泥于数学。好消息是，有一个集合的范畴，称为 **Set**，我们将仅与它打交道。在 **Set** 中，对象是集合，态射（箭头）是函数。

Set 是一个非常特殊的范畴，因为我们实际上可以窥视其内部对象并从中获得大量直觉。例如，我们知道空集没有元素。我们知道有特殊的单元素集。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，但不能将一个元素映射到两个。我们知道恒等函数将集合的每个元素映射到其自身，等等。我们的计划是逐渐忘记所有这些信息，而是用纯范畴术语（即对象和箭头的术语）表达所有这些概念。

在理想世界中，我们会说 Haskell 类型是集合，Haskell 函数是集合之间的数学函数。只是有一个小问题：数学函数不执行任何代码——它只是知道答案。Haskell 函数必须计算答案。如果答案可以在有限步数内获得——无论这个数字多大——这都不是问题。但有些计算涉及递归，而这些计算可能永远不会终止。我们不能仅仅因为区分终止和不终止的函数是不可判定的（著名的停机问题）而禁止 Haskell 中的不终止函数。这就是为什么计算机科学家提出了一个聪明的主意，或者从某种观点来看是一个重要的黑客，来扩展每种类型再加上一个特殊值，称为 底 (bottom)，表示为 或 Unicode \perp 。这个“值”对应于一个不终止的计算。所以一个声明为：

snippet02 的函数可以返回 , , 或 ；后者意味着它将永远不会终止。

有趣的是，一旦你接受底作为类型系统的一部分，将每个运行时错误视为底也是很方便的，甚至允许函数显式地返回底。后者通常使用表达式 来完成，例如：

snippet03 这个定义通过类型检查，因为 计算为底，底是任何类型的成员，包括 。你甚至可以写：

snippet04 (没有)，因为底也是类型 的成员。

可能返回底的函数称为部分函数，与此相对的是全函数，它们对每个可能的参数返回有效结果。

由于底的存在，你会看到 Haskell 类型和函数的范畴称为 **Hask** 而不是 **Set**。从理论的角度来看，这是无尽复杂的源头，所以在这一点上，我会用我的屠刀切断这种推理。从实际的角度来看，忽略不终止函数和底是可以的，并将 **Hask** 视为真正的 **Set**。¹

2.4 我们为什么需要数学模型? (Why Do We Need a Mathematical Model?)

作为一个程序员，你对编程语言的语法和文法非常熟悉。这些语言的方面通常在语言规范的开头使用正式的符号进行描述。但语言的含义或语义要难得多，需要更多的页面，通常不够正式，几乎从未完整。因此，语言律师之间的讨论永无止境，专门用于解释语言标准细节的书籍也应运而生。

有一些用于描述语言语义的正式工具，但由于它们的复杂性，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。其中一种工具称为操作语义 (*operational semantics*)，它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，通常以“抽象机”的形式进行。

问题在于，使用操作语义来证明程序的性质非常困难。为了展示一个程序的性质，你基本上必须通过理想化解释器“运行”它。

程序员从来不会进行形式化的正确性证明并不重要。我们总是“认为”我们编写的程序是正确的。没有人会坐在键盘前说，“哦，我就随便写几行代码，看看会发生什么。”我们认为我们写的代码会执行某些操作并产生预期的结果。当它不这样做时，我们通常会感到非常惊讶。这意味着我们确实在推理我们编写的程序，通常是通过在脑海中运行一个解释器。只是很难跟踪所有的变量。计算机擅长运行程序——人类则不擅长！如果我们擅长，我们就不需要计算机了。

但有另一种选择。它称为指称语义 (*denotational semantics*)，它基于数学。在指称语义中，每个编程构造都有其数学解释。如果你想证明一个程序的性质，你只需证明一个数学定理。你可能会认为证明定理很难，但事实是，我们人类已经建立了数千年的数学方法，因此有丰富的知识可供利用。而且，与职业数学家证明的那种定理相比，我们在编程中遇到的问题通常相当简单，甚至是琐碎的。

考虑一下 Haskell 中的阶乘函数定义，这是一个非常适合指称语义的语言：

snippet05 表达式 是从 到 的整数列表。函数 乘以列表中的所有元素。这就像从数学文本中取出的阶乘定义。将其与 C 进行比较：

¹Nils Anders Danielsson, John Hughes, Patrik Jansson, Jeremy Gibbons, 《快速而松散的推理在道德上是正确的》. 这篇论文为在大多数情况下忽略底提供了理由。

我还需要说什么吗？

好吧，我会第一个承认这是一个廉价的伎俩！阶乘函数有一个明显的数学指称。一个敏锐的读者可能会问：从键盘读取一个字符或通过网络发送一个数据包的数学模型是什么？很长一段时间，这将是一个尴尬的问题，导致一个相当复杂的解释。看起来指称语义并不是为大量重要任务（这些任务对于编写有用的程序至关重要）提供最佳支持，而这些任务可以通过操作语义轻松处理。突破来自范畴论。Eugenio Moggi 发现计算效应可以映射到单子（monads）。事实证明，这是一个重要的观察，不仅为指称语义带来了新的生机，使纯函数式程序更加可用，还为传统编程提供了新的视角。我会在稍后，当我们开发更多范畴工具时讨论单子。

拥有编程数学模型的一个重要优点是可以对软件的正确性进行形式证明。当你编写消费者软件时，这可能看起来不那么重要，但在某些编程领域，失败的代价可能是极其昂贵的，甚至关乎人类生命。但即使是在为健康系统编写 Web 应用程序时，你也会欣赏 Haskell 标准库中的函数和算法附带正确性证明这一事实。

2.5 纯函数与脏函数 (Pure and Dirty Functions)

我们在 C++ 或任何其他命令式语言中称为函数的东西，与数学家称为函数的东西并不相同。数学函数只是值到值的映射。

我们可以在编程语言中实现一个数学函数：给定一个输入值，这样的函数将计算输出值。生成一个数的平方的函数可能会将输入值乘以自身。每次调用它时，它都会这样做，并且保证每次以相同的输入调用它都会产生相同的输出。一个数的平方不会随着月相的变化而变化。

此外，计算一个数的平方不应该有副作用，例如为你的狗分发美味的零食。一个“函数”如果这样做，就无法轻松地建模为数学函数。

在编程语言中，给定相同的输入总是产生相同的结果且没有副作用的函数称为纯函数（pure functions）。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，这些语言更容易赋予它们指称语义并使用范畴论对其建模。至于其他语言，你总是可以将自己限制在一个纯子集，或者单独推理副作用。稍后我们将看到单子如何让我们只使用纯函数来建模各种效应。因此，通过将自己限制在数学函数内，我们实际上并没有失去任何东西。

2.6 类型的例子 (Examples of Types)

一旦你意识到类型是集合，你就可以想到一些相当奇特的类型。例如，什么类型对应于空集？不，这不是 C++ 中的，尽管这个类型在 Haskell 中确实被称为 。它是一种没有任何值的类型。你可以定义一个

接受 的函数，但你永远无法调用它。要调用它，你必须提供一个类型为 的值，但根本没有这样的值。至于这个函数可以返回什么，没有任何限制。它可以返回任何类型（尽管它永远不会返回，因为它无法被调用）。换句话说，它是一个在返回类型上是多态的函数。Haskellers 为它起了一个名字：

snippet06 （记住，是一个可以代表任何类型的类型变量。）这个名字并非巧合。类型和函数在逻辑方面有一个更深的解释，称为 Curry-Howard 对应。类型 表示虚假，而函数 的类型对应于从虚假中可以得出任何结论的声明，正如拉丁格言“*ex falso sequitur quodlibet*”所言。

接下来是对应于单例集的类型。这是一个只有一个可能值的类型。这个值只是“存在”。你可能不会立即认出它，但这就是 C++ 中的 。想想从这个类型到另一个类型的函数。一个从 到另一个类型的函数总是可以被调用。如果它是一个纯函数，它将总是返回相同的结果。以下是一个这样的函数的例子：

你可能认为这个函数不接受任何输入，但正如我们刚才看到的，一个不接受任何输入的函数永远无法被调用，因为没有表示“无”的值。那么这个函数接受什么呢？概念上，它接受一个虚拟值，只有一个实例存在，所以我们不必明确提及它。然而，在 Haskell 中，有一个符号表示这个值：一个空的括号对， . 因此，巧合的是（或者这真的是巧合吗？），在 C++ 和 Haskell 中调用一个 void 函数的方式看起来相同。此外，由于 Haskell 对简洁的喜爱，同样的符号 用于类型、构造函数和唯一对应于单例集的值。所以这里是这个函数在 Haskell 中的定义：

snippet07 第一行声明 接受 类型，读作“unit”，返回 类型。第二行通过模式匹配，即 unit 的唯一构造函数，生成数字 44。你通过提供 unit 值 来调用这个函数：

请注意，任何 unit 的函数都相当于从目标类型中选择一个元素（这里是选择 44）。事实上，你可以将 视为数字 44 的另一种表示形式。这是我们如何用函数（箭头）代替集合的明确元素的一个例子。将函数从 unit 到任何类型 A 视为与该集合 A 的元素之间存在一一对应关系。

那么具有 返回类型或在 Haskell 中具有 unit 返回类型的函数呢？在 C++ 中，这些函数用于副作用，但我们知道这些在数学意义上不是真正的函数。返回 unit 的纯函数什么都不做：它丢弃其参数。

从集合 A 到单例集的函数在数学上将 A 的每个元素映射到该单例集的单个元素。对于每个 A，这样的函数恰好只有一个。以下是 的这个函数：

snippet08 你给它任何整数，它都会返回一个 unit。秉持简洁的精神，Haskell 允许你使用通配符模式，即下划线，来表示一个被丢弃的参数。这样你就不必为它发明一个名字。所以上面的代码可以重写为：

snippet09 请注意，这个函数的实现不仅不依赖于传递给它的值，它甚至不依赖于参数的类型。

可以使用相同公式为任何类型实现的函数称为参数化多态函数。你可以使用一个类型参数而不是具体类型来实现一整个家族的此类函数。我们应该如何称呼从任何类型到 `unit` 类型的多态函数？当然，我们称它为：

snippet10 在 C++ 中你会这样编写这个函数：

接下来是类型学中的两元素集。在 C++ 中，它被称为 `Bool`，而在 Haskell 中，预料之中的是 `Bool`。不同之处在于，在 C++ 中，`Bool` 是一种内置类型，而在 Haskell 中，它可以定义如下：

snippet11 （读这个定义的方式是“不是就是”。）原则上，在 C++ 中也应该能够将布尔类型定义为枚举：

但 C++ 中的 `Bool` 秘密地是一个整数。C++11 的“”可以代替使用，但你必须用类名限定它的值，如 `bool`，更不用说在使用的每个文件中都必须包含适当的头文件。

从 `bool` 到目标类型的纯函数只需选择两个值，一个对应 `true`，另一个对应 `false`。

到 `Bool` 的函数称为谓词（predicates）。例如，Haskell 库充满了谓词，如 `odd` 或 `even`。在 C++ 中有一个类似的库，它定义了 `isOdd` 和 `isEven` 等函数，但这些返回的是一个 `Bool`，而不是布尔值。实际的谓词定义在 `<i> Prelude </i>` 中，并具有 `Pred` 等形式。

2.7 挑战 (Challenges)

1. 在你最喜欢的语言中定义一个高阶函数（或函数对象）。该函数接受一个纯函数作为参数，并返回一个函数，该函数的行为几乎与 `id` 相同，只不过它只对每个参数调用原始函数一次，内部存储结果，并在每次以相同参数调用时返回该存储的结果。你可以通过观察它的性能来区分记忆化函数与原始函数。例如，尝试记忆化一个需要很长时间才能计算出结果的函数。第一次调用它时，你需要等待结果，但在随后的调用中，只要使用相同的参数，你应该能立即得到结果。
2. 尝试记忆化你通常用来产生随机数的标准库中的函数。它能正常工作吗？
3. 大多数随机数生成器都可以用种子进行初始化。实现一个函数，该函数接受一个种子，调用带有该种子的随机数生成器，并返回结果。记忆化该函数。它能正常工作吗？
4. 以下哪些 C++ 函数是纯函数？尝试记忆化它们，并观察在多次调用时（记忆化与否）会发生什么：

- (a) 本文中的阶乘函数。
 - (b)
 - (c)
 - (d)
5. 从 to 的函数有多少种不同的实现方式？你能实现它们吗？
6. 绘制一个范畴的图示，其唯一的对象是 、 (unit) 和 ； 箭头对应于这些类型之间的所有可能的函数。用函数的名字标注箭头。

3

Categories Great and Small

你 可以通过学习各种例子来真正理解范畴。范畴有各种形状和大小，常常出现在意想不到的地方。我们将从一些非常简单的例子开始。

3.1 没有对象 (No Objects)

最简单的范畴是没有对象的范畴，因此也没有态射 (morphisms)。它本身是一个非常无趣的范畴，但在其他范畴的上下文中可能很重要，例如，在所有范畴的范畴中（是的，确实有这样的一个范畴）。如果你认为空集有意义，那么为什么空范畴没有意义呢？

3.2 简单图 (Simple Graphs)

你可以通过用箭头连接对象来构建范畴。你可以想象从任何有向图开始，通过简单地添加更多箭头使其成为一个范畴。首先，在每个节点添加一个恒等箭头。然后，对于任何两个箭头，如果一个箭头的终点与另一个箭头的起点重合（换句话说，任何两个可组合的 (composable) 箭头），则添加一个新的箭头作为它们的组合。每次添加一个新箭头时，你还必须考虑它与任何其他箭头（除了恒等箭头）和它自身的组合。通常你最终会得到无穷多个箭头，但这没问题。

另一种看待这个过程的方法是，你正在创建一个范畴，它为图中的每个节点提供一个对象，并将所有可能的链 (chains) ——即可组合的图边——作为态射。（你甚至可以将恒等态射视为长度为零的特殊链。）

这样的范畴被称为由给定图生成的自由范畴 (free category)。它是自由构造的一个例子，这一过程通过扩展最少数量的项来完成给定结构以满足其定律（在这里是范畴的定律）。我们将来会看到更多的例子。

3.3 序 (Orders)

现在来看一些完全不同的东西！在一个范畴中，态射是对象之间的一种关系：小于或等于的关系。让我们检查一下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 $a \leq b$ 且 $b \leq c$ ，那么 $a \leq c$ ：检查！组合是结合的吗？检查！具有这种关系的集合称为预序（preorder），所以预序确实是一个范畴。

你还可以有更强的关系，满足额外的条件，即如果 $a \leq b$ 且 $b \leq a$ ，那么 a 必须与 b 相同。这被称为偏序（partial order）。

最后，你可以施加一个条件，使得任何两个对象之间都存在某种关系，这就得到了线性序（linear order）或全序（total order）。

让我们将这些有序集描述为范畴。预序是一个范畴，其中从任意对象 a 到任意对象 b 的态射至多有一个。这种范畴的另一个名字是“薄”（thin）。预序是一个薄范畴。

范畴 \mathbf{C} 中从对象 a 到对象 b 的态射集合被称为态射集（hom-set），记作 $\mathbf{C}(a, b)$ （有时也写作 $\mathbf{Hom}_{\mathbf{C}}(a, b)$ ）。所以在预序中，所有的态射集要么为空，要么是单元素集。这包括预序中的态射集 $\mathbf{C}(a, a)$ ，即从 a 到 a 的态射集，它必须是单元素集，且仅包含恒等态射。然而，在预序中可能会有循环。在偏序中，循环是被禁止的。

能够识别预序、偏序和全序非常重要，因为排序算法，如快速排序（quicksort）、冒泡排序（bubble sort）、合并排序（merge sort）等，只有在全序上才能正确工作。偏序可以使用拓扑排序（topological sort）进行排序。

3.4 作为集合的幺半群 (Monoid as Set)

幺半群是一个令人难以置信的简单但惊人强大的概念。它是基本算术的背后概念：加法和乘法都形成幺半群。幺半群在编程中无处不在。它们以字符串、列表、可折叠的数据结构、并发编程中的期望值、函数响应式编程中的事件等形式出现。

传统上，幺半群被定义为一个带有二元运算的集合。这个运算所需的全部条件是它是结合的，并且存在一个特殊的元素在此运算下表现为单位元。

例如，带有零的自然数在加法下形成一个幺半群。结合性意味着：

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

（换句话说，我们可以在加法时忽略括号。）

中性元素是零，因为：

$$0 + a = a$$

并且

$$a + 0 = a$$

第二个方程式是多余的，因为加法是交换的 ($a + b = b + a$)，但交换性不是幺半群定义的一部分。例如，字符串连接不是交换的，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空字符串，它可以附加在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——一种有中性元素 和二元运算 的类型：

snippet01 这个二元函数的类型签名乍一看可能有些奇怪，但在我 们讨论柯里化 (currying) 之后，它将变得完全合理。你可以用两种基本方式解释带有多个箭头的类型签名：作为一个具有多个参数的函数，最右边的类型是返回类型；或者作为一个单参数函数（最左边的一个），返回一个函数。后者的解释可以通过添加括号来强调（虽然它们是多余的，因为箭头是右结合的），如：。我们稍后会回到这种解释。

注意，在 Haskell 中，没有办法表达 和 的幺半群性质（即 是中性元素 和 是结合的事实）。程序员有责任确保它们得到满足。

Haskell 类不像 C++ 类那么具侵入性。当你定义一个新类型时，你不必预先指定它的类。你可以自由地拖延，并在以后再声明给定类型是某个类的实例。例如，让我们通过提供 和 的实现来声明 是一个幺半群（实际上，这在标准 Prelude 中已经为你做了）：

snippet02 这里，我们重用了列表连接运算符，因为 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将其括在括号中变成一个二元函数。给定两个字符串，你可以在它们之间插入 来连接它们：

或通过将它们作为两个参数传递给括号中的：

注意，函数的参数之间没有逗号，也没有被括号包围。（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，如：

从概念上讲，这与表达函数产生的值的相等性不同，如：

前者翻译成 **Hask**（或忽略底的情况下 的 **Set**）范畴中的态射相等性。这种方程不仅更简洁，而且通常可以推广到其他范畴。后者称为外延 (extensional) 相等，表示对于任何两个输入字符串， 和 的输出是相同的。由于参数的值有时被称为点 (points)（如： f 在点 x 的值），这被称为逐点相等。未指定参数的函数相等被描述为无点 (point-free)。（顺便提一下，无点方程通常涉及函数的组合，它用一个点符号 表示，所以这对初学者来说可能有点混淆。）

在 C++ 中最接近声明一个幺半群的方法是使用 C++20 标准的 concept 特性。

第一个定义是一个结构，旨在为每个专门化类型保存中性元素。

关键字 意味着没有定义默认值：它必须在具体情况下指定。同样，对于 也没有默认值。

`concept` 检查是否存在适当的 和 定义，用于给定的类型。

可以通过提供适当的专业化和重载来实例化 Monoid 概念：

3.5 作为范畴的幺半群 (Monoid as Category)

那是关于幺半群的“熟悉”的定义，即集合的元素。但是，正如你所知，在范畴论中，我们试图摆脱集合及其元素，转而讨论对象和态射。所以让我们稍微改变一下视角，将二元运算的应用视为在集合中“移动”或“转移”事物。

例如，有一个将每个自然数加 5 的操作。它将 0 映射到 5，1 映射到 6，2 映射到 7，依此类推。这是一个定义在自然数集合上的函数。这很好：我们有一个函数和一个集合。一般来说，对于任意数 n ，都有一个将 n 加上的函数—— n 的“加法器”。

加法器如何组合？将加 5 的函数与加 7 的函数组合，结果是加 12 的函数。因此，加法器的组合可以与加法规则等同。这也很好：我们可以用函数组合替换加法。

但等等，还有更多：还有一个对应于中性元素的加法器，零。加零不会移动任何事物，所以它是自然数集合中的恒等函数。

我可以不告诉你传统的加法规则，而是给你加法器组合的规则，不会丢失任何信息。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。

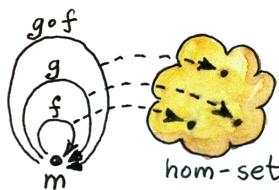
一个聪明的读者可能已经注意到，从整数到加法器的映射源于 类型签名的第二种解释，即 。它告诉我们 将一个幺半群集合的元素映射到作用于该集合的函数。

现在我希望你忘记你正在处理的是自然数集合，而只将其视为一个单一的对象，一个带有一堆态射的 blob——加法器。幺半群是一个单对象范畴。事实上，幺半群这个名字来源于希腊语 *mono*，意思是单一的。每个幺半群都可以描述为一个单对象范畴，其中有一组符合组合规则的态射。



字符串连接是一个有趣的例子，因为我们可以选择定义右连接器和左连接器（或前置器，如果你愿意的话）。两种模型的组合表是镜像反转的。你可以轻松地让自己相信在“foo”之后附加“bar”对应于在附加“bar”之后前置“foo”。

你可能会问，是否每个范畴论上的幺半群——一个单对象范畴——都定义了一个唯一的带有二元运算符的集合幺半群。事实证明，我们总是可以从一个单对象范畴中提取一个集合。这个集合是态射的集合——在我们的例子中是加法器。换句话说，我们有单对象 m 在范畴 \mathbf{M} 中的态射集 $\mathbf{M}(m, m)$ 。我们可以很容易地在这个集合中定义一个二元运算符：两个集合元素的幺半群乘积是对应态射组合对应的元素。如果你给我两个对应于 f 和 g 的 $\mathbf{M}(m, m)$ 元素，它们的乘积将对应于组合 $f \circ g$ 。这个组合总是存在的，因为这些态射的源和目标是相同的对象。根据范畴的规则，它是结合的。恒等态射是这个乘积的中性元素。因此，我们总是可以从一个范畴幺半群中恢复一个集合幺半群。对于所有意图和目的来说，它们是相同的。



作为态射的幺半群态射集和作为集合中的点。

数学家可以挑剔的唯一一点是：态射不一定要形成一个集合。在范畴世界中，有些东西比集合更大。在任意两个对象之间的态射形成集合的范畴称为局部小范畴（locally small）。正如承诺的那样，我将主要忽略这些细微差别，但我觉得我应该提到它们以备记录。

在范畴论中，许多有趣的现象都根源于这样的事实：态射集的元素既可以视为符合组合规则的态射，也可以视为集合中的点。在这里， \mathbf{M} 中态射的组合转化为集合 $\mathbf{M}(m, m)$ 中的幺半群乘积。

3.6 挑战 (Challenges)

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) 一个有一个节点和一条（有向）边的图（提示：这条边可以与其自身组合）
 - (c) 一个有两个节点和一个箭头的图
 - (d) 一个有一个节点和 26 条以字母标记的箭头的图：a, b, c … z.
2. 这种序是什么类型的？
 - (a) 一个带有包含关系的集合的集合：如果 A 的每个元素也是 B 的元素，则 A 包含于 B 。
 - (b) C++ 类型及其以下子类型关系：如果可以将 的指针传递给期望 指针的函数而不触发编译错误，则 是 的子类型。
3. 考虑到 是一个包含 和 两个值的集合，证明它在运算符 (AND) 和 (OR) 下分别形成了两个 (集合论) 幺半群。
4. 用 AND 运算符表示 幺半群作为一个范畴：列出态射及其组合规则。
5. 用一个幺半群范畴表示模 3 加法。

4

Kleisli Categories

你已经看到了如何将类型和纯函数建模为范畴。我还提到过在范畴论中有一种方法可以建模副作用（side effects）或非纯函数。让我们看看这样一个例子：记录或跟踪执行过程的函数。在命令式语言中，这可能会通过修改某些全局状态来实现，例如：

你知道这不是一个纯函数，因为它的记忆化版本将无法生成日志。此函数具有副作用。

在现代编程中，我们尽量避免全局可变状态——仅仅是因为并发的复杂性。而且你永远不会在库中编写这样的代码。

幸运的是，我们可以让这个函数变成纯函数。你只需显式传递日志，输入和输出日志。让我们通过添加一个字符串参数，并将常规输出与包含更新日志的字符串配对来实现这一点：

这个函数是纯函数，没有副作用，每次用相同的参数调用时都会返回相同的对（pair），并且如果需要可以将其记忆化。然而，考虑到日志的累积性质，你将不得不记忆化所有可能导致给定调用的历史记录。将会有单独的记忆条目用于：

以及

等等。

这对于库函数来说也不是一个很好的接口。调用者可以选择忽略返回类型中的字符串，因此这不是一个巨大的负担；但它们被迫传递一个字符串作为输入，这可能会带来不便。

有没有一种更不显眼的方式来实现相同的效果？有没有办法分离关注点？在这个简单的例子中，函数 的主要目的是将一个布尔值转换为另一个布尔值。记录日志是次要的。当然，记录的消息是函数特有的，但将消息聚合成一个连续日志的任务是一个单独的关注点。我们仍然希望函数生成一个字符串，但我们希望解除它生成日志的负担。因此，这里有一个折衷的解决方案：

这个想法是，日志将在函数调用之间被聚合。

为了看看如何做到这一点，让我们切换到一个稍微现实一点的例子。我们有一个从字符串到字符串的函数，它将小写字符转换为大写字符：

还有另一个函数，它将字符串拆分为字符串向量，按空白边界拆分：

实际工作在辅助函数 中完成：

我们希望修改函数 和 ，使它们能够在常规返回值的基础上附加一条消息字符串。



我们将“修饰”这些函数的返回值。让我们通过定义一个模板 来以通用的方式做到这一点，该模板封装了一个对，其第一个组件是任意类型的值，第二个组件是一个字符串：

以下是修饰后的函数：

我们希望将这两个函数组合成另一个修饰过的函数，该函数将字符串转换为大写并将其拆分为单词，同时生成这些操作的日志。以下是我们可能的实现方式：

我们已经达到了目标：日志的聚合不再是各个函数的关注点。它们生成自己的消息，然后在外部将其连接成更大的日志。

现在想象一个完全以这种方式编写的程序。这是一个充满重复性、易出错的代码的噩梦。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，这不是你一般的抽象——我们必须抽象函数组合本身。但组合是范畴论的本质，因此在编写更多代码之前，让我们从范畴的角度分析这个问题。

4.1 Writer 范畴 (The Writer Category)

通过修饰一组函数的返回类型以附加一些额外的功能的想法证明是非常有用的。我们将看到更多的例子。起点是我们常规的类型和函数的范畴。我们将类型保留为对象，但将态射重新定义为修饰后的函数。

例如，假设我们要修饰从 A 到 B 的函数。我们将其转换为一个态射，该态射由修饰后的函数表示。重要的一点是，这个态射仍然被认为是对象 A 和 B 之间的箭头，尽管修饰后的函数返回的是一个对：

根据范畴的定律，我们应该能够将这个态射与另一个从对象 A 到任意对象的态射组合。特别是，我们应该能够将其与我们之前的组合：

显然，我们不能以组合常规函数的方式组合这两个态射，因为输入/输出不匹配。它们的组合应该看起来更像这样：

所以这是我们正在构建的新范畴中两个态射的组合的配方：

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组件，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组件（字符串）和第二个结果的第二个组件（字符串）
4. 返回一个新对，将最终结果的第一个组件与连接后的字符串组合在一起。

如果我们希望将这种组合抽象为 C++ 中的高阶函数，则必须使用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。它应接受根据我们的规则可以组合的两个修饰函数，并返回第三个修饰函数：

现在我们可以回到前面的例子，使用这个新模板来实现 和 的组合：

在向 模板传递类型时仍有很多噪音。如果你有一个支持返回类型推导的泛型 lambda 函数的 C++14 兼容编译器，这种噪音可以避免（此代码的贡献归于 Eric Niebler）：

在这个新定义中， 的实现简化为：

但我们还没有完成。我们已经在新范畴中定义了组合，但是恒等态射是什么？它们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，这意味着它们是如下形式的修饰函数：

它们必须像组合的单位元一样运作。如果你看一下我们定义的组合，你会看到一个恒等态射应该传递其参数而不做任何更改，并且只向日志中贡献一个空字符串：

你可以很容易地说服自己，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。如果你跟踪每个对的第一个组件所发生的事情，它只是一个常规函数组合，这是结合的。第二个组件正在连接，而连接也是结合的。

一个聪明的读者可能会注意到，很容易将此构造推广到任何幺半群，而不仅仅是字符串幺半群。我们可以在 中使用，在 中使用（代替 和）。实际上，我们没有理由仅限于记录字符串。一个好的库编写者应该能够识别出使库工作的最小约束——这里记录库的唯一要求是日志具有幺半群属性。

4.2 Haskell 中的 Writer 范畴 (Writer in Haskell)

在 Haskell 中做同样的事情更简洁一些，并且我们还得到了编译器的更多帮助。让我们从定义 类型开始：

snippet01 这里我只是在定义一个类型别名，相当于 C++ 中的 （或）。类型由类型变量 参数化，相当于 和 的对。对的语法是最简的：只是两个用逗号分隔的项目放在括号中。

我们的态射是从任意类型到某个 类型的函数：

snippet02 我们将声明组合为一个有趣的中缀运算符，有时称为“鱼”：

snippet03 这是一个有两个参数的函数，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，第二个参数是，结果是。

以下是这个中缀运算符的定义——两个参数 和 出现在鱼符号的两边：

snippet04 结果是一个参数为 的 lambda 函数。lambda 写作反斜杠 —— 将其视为希腊字母 λ 的一条截肢腿。

表达式允许你声明辅助变量。这里调用 的结果与一对变量 进行模式匹配；调用 的结果，使用第一个模式中的参数，与 进行匹配。

在 Haskell 中，通常匹配对而不是使用访问器，就像我们在 C++ 中所做的那样。除此之外，两种实现之间有一个相当直接的对应关系。

表达式的整体值在其 子句中指定：这里是一个对，其第一个组件是，第二个组件是两个字符串 的连接。

我还将在我们的范畴中定义恒等态射，但由于某些原因，这些原因将在后面详细解释，我将其称为。

snippet05 为了完整性，我们来看一下 Haskell 版本的修饰函数 和：

snippet06 函数 对应于 C++ 中的 。它将字符函数 应用于字符串。辅助函数 定义在标准 Prelude 库中。

最后，两个函数的组合使用鱼运算符完成：

snippet07

4.3 Kleisli 范畴 (Kleisli Categories)

你可能已经猜到我没有临时发明这个范畴。它是所谓的 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，但我想让你了解它们可以做什么。对于我们的有限目的，Kleisli 范畴的对象是底层编程语言的类型。从类型 A 到类型 B 的态射是从 A 到使用特定修饰从 B 派生出的类型的函数。每个 Kleisli 范畴定义了它自己的组合这些态射的方法，以及相对于该组合的恒等态射。（稍后我们将看到，不精确的术语“修饰”对应于范畴中的端函子概念。）

我在本章中用作范畴基础的特定单子称为 *Writer* 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中的更一般机制的一个例子。你之前已经看到，我们可以在集合范畴中建模编程语言的类型和函数（像往常一样，忽略底层）。在这里，我们将此模型扩展到一个略有不同的范畴，其中态射由修饰函数表示，并且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。我们有了一个额外的自由度：组合本身。事实证明，这正是使得能够在命令式语言中传统上使用副作用实现的程序提供简单指称语义的自由度。

4.4 挑战 (Challenge)

一个未定义其参数所有可能值的函数称为部分函数 (partial function)。从数学意义上讲，它并不是真正的函数，因此它不符合标准的范畴模型。然而，它可以通过返回一个修饰类型 的函数来表示：

例如，下面是修饰函数 的实现：

挑战如下：

1. 构造部分函数的 Kleisli 范畴（定义组合和恒等态射）。
2. 实现修饰函数，该函数返回其参数的有效倒数（reciprocal），如果它不为零。
3. 组合函数 和 以实现，该函数在可能的情况下计算。

5

Products and Coproducts

 希腊的剧作家欧里庇得斯曾说过：“每个人都像他惯常交往的朋友。”我们的身份是由我们的关系定义的。这在范畴理论 (category theory) 中尤为真实。如果我们想要在一个范畴中单独挑选出某个对象，我们只能通过描述它与其他对象（以及它自己）的关系来做到这一点。这些关系是由态射 (morphisms) 定义的。

在范畴理论中，有一种常见的结构称为泛结构 universal construction，用于通过对象之间的关系来定义对象。实现这种结构的一种方法是选择一个模式，即由对象和态射构成的特定形状，然后在范畴中寻找所有满足这一形状的情况。如果这是一个足够常见的模式，并且范畴足够大，那么你可能会找到许多这样的情况。诀窍在于在这些情况中建立某种排序，并选择最适合的那个。

这个过程类似于我们进行网络搜索的方式。查询就像一个模式。一个非常通用的查询会给你带来大量的召回率 recall：大量的结果。其中一些可能相关，另一些可能无关。为了消除无关的结果，你可以精炼查询。这将提高它的精确度 precision。最后，搜索引擎将对结果进行排序，并希望你感兴趣的那個结果能排在最前面。

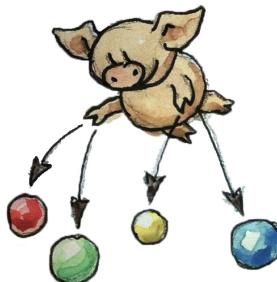
5.1 初始对象

Initial Object

最简单的形状是一个单一对象。显然，在给定的范畴中，这种形状的实例与对象的数量相同。我们需要建立某种排序，并试图找到在这个层次结构中处于顶端的对象。我们唯一的工具是态射。如果你把态射看作箭头，那么可能存在从范畴的一端到另一端的箭头流动。这在有序范畴 (ordered categories) 中是存在的，例如在偏序 (partial orders) 中。我们可以通过说对象 a “更初始”于对象 b ，如果有一个箭头（态射）从

a 指向 b 。然后我们可以定义初始对象 initial object 为具有指向所有其他对象的箭头的对象。显然，不能保证这样的对象一定存在，这也没关系。一个更大的问题是可能有太多这样的对象：召回率很好，但精确度不足。解决方案是借鉴有序范畴的提示——它们允许在任意两个对象之间最多只有一个箭头：只有一种方式是“少于或等于”另一个对象。这引导我们得出初始对象的定义：

初始对象 initial object 是指在范畴中具有唯一且只有一个态射指向任意对象的对象。



然而，这并不保证初始对象的唯一性（如果它存在）。但它保证了另一种最佳情况：同构 isomorphism 意义上的唯一性。同构在范畴理论中非常重要，我稍后会讨论它。现在，我们只需同意，在定义初始对象时，使用“the”是有道理的。

以下是一些例子：在偏序集（通常称为偏序集 poset）中，初始对象是其最小元素。有些偏序集没有初始对象——比如所有整数的集合（包括正数和负数），它们的态射是“小于或等于”关系。

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 中的类型（在 C++ 中没有对应的类型），并且从 到任何其他类型的唯一多态函数称为：

snippet01 正是这一系列态射使得 成为类型范畴中的初始对象。

5.2 终端对象

Terminal Object

让我们继续讨论单一对象模式，但这次改变我们对对象排序的方式。我们将说对象 a “更终端”于对象 b ，如果存在从 b 指向 a 的态射（注意方向的反转）。我们将寻找一个比范畴中任何其他对象都更终端的对象。同样，我们将坚持唯一性：

终端对象 terminal object 是指在范畴中具有唯一且只有一个态射从任何对象指向它的对象。



同样，终端对象是唯一的，同构意义上的唯一性，我稍后会展示。但首先让我们来看一些例子。在一个偏序集中，终端对象（如果存在）是最大的对象。在集合范畴中，终端对象是一个单元素集合。我们已经讨论过单元素集合——它们对应于 C++ 中的类型和 Haskell 中的单位类型。这是一种只有一个值的类型——在 C++ 中是隐含的，在 Haskell 中是显式的，表示为。我们还确定，从任何类型到单位类型的纯函数只有一个：

snippet02 因此，满足终端对象的所有条件。

注意，在这个例子中，唯一性条件至关重要，因为存在其他集合（实际上是所有集合，除了空集）具有从每个集合指向它们的态射。例如，定义在每个类型上的布尔值函数（谓词）：

snippet03 但不是终端对象。对于每种类型（除了，对于它来说，两个函数都等于），至少还有一个值的函数：

snippet04 坚持唯一性使我们能够将终端对象的定义精确到只有一种类型。

5.3 对偶性

Duality

你可能已经注意到，我们定义初始对象和终端对象的方式是对称的。两者之间唯一的区别是态射的方向。事实证明，对于任何范畴 \mathbf{C} ，我们可以通过反转所有箭头来定义对偶范畴 opposite category \mathbf{C}^{op} 。只要我们同时重新定义组合，对偶范畴自动满足范畴的所有要求。如果原始态射 $f :: a \rightarrow b$ 和 $g :: b \rightarrow c$ 组合为 $h :: a \rightarrow c$ ，使得 $h = g \circ f$ ，那么反转后的态射 $f^{op} :: b \rightarrow a$ 和 $g^{op} :: c \rightarrow b$ 将组合为 $h^{op} :: c \rightarrow a$ ，并且 $h^{op} = f^{op} \circ g^{op}$ 。反转恒等箭头是一个（双关警告！）无操作。

对偶性是范畴的一个非常重要的性质，因为它将每个研究范畴理论的数学家的生产力翻倍。对于你提出的每个构造，都会有它的对偶；

对于你证明的每个定理，你会免费得到一个。在对偶范畴中的构造通常以“co”开头，所以你有积和余积（products and coproducts）、单子和共单子（monads and comonads）、锥和余锥（cones and cocones）、极限和余极限（limits and colimits）等等。然而，没有共单子（cocomonads），因为反转箭头两次会使我们回到原始状态。

因此，终端对象就是对偶范畴中的初始对象。

5.4 同构

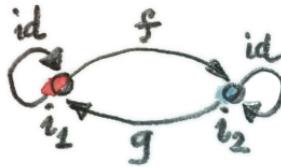
Isomorphisms

作为程序员，我们深知定义相等性是一个非平凡的任务。对于两个对象来说，相等是什么意思？它们是否必须占据内存中的同一位置（指针相等性）？还是说它们的所有组件的值相等就足够了？如果一个复数是用实部和虚部表示的，而另一个是用模和角度表示的，那么这两个复数是否相等？你可能会认为数学家已经弄清楚了相等的含义，但他们并没有。他们面临着相同的问题，即相等性的多种竞争性定义。存在命题相等性（propositional equality）、意图相等性（intensional equality）、外延相等性（extensional equality）和同伦类型论中的路径相等性（path equality）。然后还有较弱的同构（isomorphism）和更弱的等价（equivalence）的概念。

直观上，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与另一个对象的某个部分一一对应。从我们的仪器来看，这两个对象是彼此的完美副本。在数学上，这意味着存在一个从对象 a 到对象 b 的映射，并且存在一个从对象 b 回到对象 a 的映射，它们是彼此的逆映射。在范畴理论中，我们用态射来代替映射。同构是可逆的态射；或者说是一对态射，其中一个是另一个的逆。

我们通过组合和恒等性来理解逆态射：态射 g 是态射 f 的逆态射，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：

snippet05 当我说初始对象（终端对象）在同构意义上是唯一的时，我的意思是任何两个初始对象（终端对象）都是同构的。这实际上很容易看出。假设我们有两个初始对象 i_1 和 i_2 。由于 i_1 是初始对象，因此存在一个从 i_1 到 i_2 的唯一态射 f 。同样，由于 i_2 是初始对象，因此存在一个从 i_2 到 i_1 的唯一态射 g 。这两个态射的组合是什么？



该图中的所有态射都是唯一的。

组合 $g \circ f$ 必须是从 i_1 到 i_1 的态射。但是 i_1 是初始对象，因此只能有一个从 i_1 到 i_1 的态射。由于我们处于一个范畴中，我们知道存在一个从 i_1 到 i_1 的恒等态射，并且由于只有一个位置，这必须是它。因此 $g \circ f$ 等于恒等态射。同样， $f \circ g$ 也必须等于恒等态射，因为只能有一个从 i_2 回到 i_2 的态射。这证明了 f 和 g 必须是彼此的逆态射。因此，任何两个初始对象都是同构的。

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。没有它，我们无法证明“同构意义上的唯一性”这一部分。但为什么我们需要 f 和 g 的唯一性呢？因为不仅初始对象在同构意义上是唯一的，它还在唯一同构意义上是唯一的。原则上，在两个对象之间可能存在多个同构，但在这里不是这种情况。这种“唯一同构意义上的唯一性”是所有泛结构的一个重要属性。

5.5 积

Products

下一个泛结构是积的构造。我们知道两个集合的笛卡尔积是什么：它是一个有序对的集合。但是连接积集与其组成集合的模式是什么呢？如果我们能弄清楚这一点，我们就能将其推广到其他范畴。

我们只能说，有两个函数（projections）从积到它们的组成部分。在 Haskell 中，这两个函数分别被称为 和，它们分别选取有序对的第一个和第二个组件：

`snippet06`

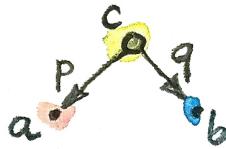
`snippet07` 在这里，这些函数是通过对其参数进行模式匹配来定义的：匹配任何对的模式是，它将其组件提取到变量 和 中。

通过使用通配符，这些定义可以进一步简化：

`snippet08` 在 C++ 中，我们将使用模板函数，例如：

有了这些看似非常有限的知识，让我们尝试在集合的范畴中定义一个对象和态射的模式，这将引导我们构造两个集合的积。这个模式由一个对象 c 和两个态射 p 和 q 组成，它们分别连接到 a 和 b ：

`snippet09`



所有符合这一模式的 c 都将被视为积的候选者。可能会有很多这样的对象。



例如，让我们选择两个 Haskell 类型 和，并获取它们积的候选样本。

这是一个 $:.$ 。可以被视为和的积的候选者吗？是的，可以——这里是它的投影：

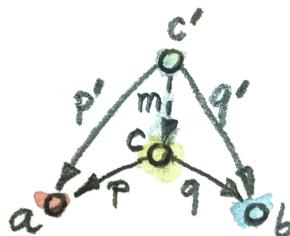
`snippet10` 这很弱，但它符合标准。

这是另一个 $:.$ 。这是一个三元素的元组或三元组。这里有两个态射，使其成为一个合法的候选者（我们使用的是对三元组的模式匹配）：

`snippet11` 你可能已经注意到，虽然我们的第一个候选者太小了——它只涵盖了积的维度；第二个则太大了——它错误地复制了维度。

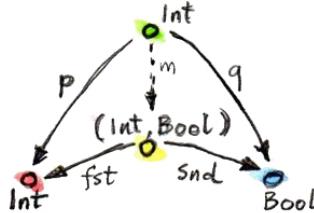
但我们还没有探索泛结构的另一部分：排序。我们希望能够比较我们的模式的两个实例。我们希望比较一个候选对象 c 及其两个投影 p 和 q 与另一个候选对象 c' 及其两个投影 p' 和 q' 。我们希望说 c “优于” c' ，如果存在一个从 c' 到 c 的态射 m ——但这太弱了。我们还希望它的投影“优于”或“更普遍” c' 的投影。这意味着可以通过 m 从 p 和 q 重新构建 p' 和 q' ：

`snippet12`



另一种看待这些方程的方法是， m 因式分解 p' 和 q' 。假装这些方程是在自然数中，并且点是乘法： m 是 p' 和 q' 的公因子。

为了建立一些直觉，让我向你展示，具有两个标准投影 和 的对 确实比我之前展示的两个候选者更好。



第一个候选者的映射 是：

snippet13 确实，这两个投影 和 可以通过以下方式重建：

snippet14 第二个示例的 也类似地唯一确定：

snippet15 我们能够证明 比前两个候选者更好。让我们看看为什么反过来是不成立的。我们能找到一些 来帮助我们从 和 中重建 和 吗？

snippet16 在我们的第一个示例中， 总是返回，而我们知道有些对的第二个组件是。我们无法从中重建。

第二个例子不同：运行 或 后，我们保留了足够的信息，但有不止一种方式来因式分解 和 。因为 和 都忽略了三元组的第二个组件，所以我们的 可以在其中放置任何东西。我们可以有：

snippet17

或者 snippet18 等等。

综上所述，给定任何具有两个投影 和 的类型，存在唯一的 从 到笛卡尔积，它们会因式分解这些投影。实际上，它只是将 和 组合成一对。

snippet19 这使得笛卡尔积 成为我们的最佳匹配，这意味着这个泛结构在集合范畴中有效。它选择了任何两个集合的积。

现在，让我们忘记集合，并使用相同的泛结构在任何范畴中定义两个对象的积。这种积不总是存在，但当它存在时，它是唯一的，同构意义上的唯一性。

两个对象 a 和 b 的积 product 是对象 c ，配备两个投影，使得对于任何其他配备两个投影的对象 c' ，存在唯一的态射 m 从 c' 到 c ，因式分解这些投影。

产生因式分解函数 的（高阶）函数有时称为因式分解器 factorizer。在我们的例子中，它将是函数：

snippet20

5.6 余积

Coproduct

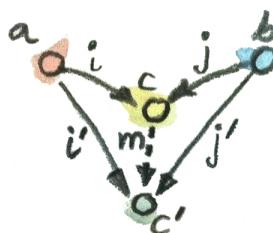
像范畴理论中的每个构造一样，积有一个对偶，称为余积。当我们反转积模式中的箭头时，我们最终得到一个对象 c ，它配备了两个嵌入 injections，和：从 a 和 b 到 c 的态射。

snippet21



排序也被反转了：对象 c “优于”配备了嵌入 i' 和 j' 的对象 c' ，如果存在一个态射 m 从 c 到 c' ，它因式分解了这些嵌入：

snippet22



这样的“最佳”对象，具有唯一的态射将其连接到任何其他模式，称为余积，如果它存在，则在唯一同构意义上是唯一的。

两个对象 a 和 b 的余积 coproduct 是对象 c ，配备两个嵌入，使得对于任何其他配备两个嵌入的对象 c' ，存在唯一的态射 m 从 c 到 c' ，因式分解这些嵌入。

在集合范畴中，余积是两个集合的不交并 disjoint union。集合 a 和 b 的不交并的元素要么是 a 的元素，要么是 b 的元素。如果这两个集合重叠，不交并包含公共部分的两个副本。你可以认为不交并的元素是带有标识符的，它指定了其来源。

对于程序员来说，更容易理解余积的类型意义：它是两个类型的标记并集 (tagged union)。C++ 支持并集，但它们不是标记的。这意味着在你的程序中，你必须以某种方式跟踪并集的哪个成员是有效的。要创建一个标记并集，你必须定义一个标记——枚举——并将其与并集结合。例如，一个 和 的标记并集可以实现为：

这两个嵌入可以实现为构造函数或函数。例如，下面是第一个嵌入作为函数的实现：

它将一个整数嵌入到 中。

标记并集也称为变体 variant，并且在 Boost 库中有一个非常通用的变体实现，。

在 Haskell 中，你可以通过使用竖线分隔数据构造函数，将任何数据类型组合成一个标记并集。上面的示例转换为如下声明：

snippet23 在这里，和既充当了构造函数（嵌入），又充当了模式匹配的标记（稍后将详细讨论）。例如，以下是如何使用电话号码构造一个联系人的方法：

snippet24 与 Haskell 中作为原始对内置的积的规范实现不同，余积的规范实现是一个称为 的数据类型，它在标准 Prelude 中定义为：

snippet25 它是一个由两个类型 和 参数化的数据类型，并且有两个构造函数：接收类型的值，接收类型的值。

正如我们为积定义了因式分解器一样，我们也可以为余积定义一个。给定一个候选类型 和两个候选嵌入 和，的因式分解器会生成因式分解函数：

snippet26

5.7 不对称性

Asymmetry

我们已经看到了两组对偶定义：可以通过反转箭头的方向，从初始对象的定义中得到终端对象的定义；同样地，可以从积的定义中得到余积的定义。然而，在集合的范畴中，初始对象与终端对象非常不同，余积与积也非常不同。稍后我们将看到，积表现得像乘法，终端对象扮演“一”的角色；而余积更像是加法，初始对象扮演“零”的角色。特别是，对于有限集合，积的大小是各个集合大小的乘积，而余积的大小是各个集合大小的总和。

这表明，集合范畴对于箭头反转是不对称的。

注意，虽然空集有唯一的态射指向任何集合（函数），但它没有返回的态射。单元素集合有一个唯一的态射从任何集合指向它，但它也有指向每个集合的态射（除了空集）。正如我们之前看到的那样，这些从终端对象指向其他集合的态射在选择其他集合的元素方面扮演着非常重要的角色（空集没有元素，因此没有东西可以选择）。

这就是单元素集合与积的关系，使其与余积不同。考虑使用单元素集合，表示为单位类型，作为积模式的另一个——大大逊色的——候选者。为其配备两个投影 和：从单元素集合到每个组成集合的函数。每个函数选择组成集合中的一个具体元素。因为积是普遍的，所以还存在

一个（唯一的）态射，将我们的候选对象单元素集合映射到积上。这个态射选择了积集合中的一个元素——它选择了一个具体的对。它也因式分解了这两个投影：

snippet27 当作用于单元素集合的唯一值时，这两个方程变为：

snippet28 由于是由选择的积的元素，这些方程告诉我们，从第一个集合中由选择的元素是选择的对的第一个组成部分。同样，等于第二个组成部分。这与我们对积的元素是组成集合的对的理解完全一致。

对于余积，没有如此简单的解释。我们可以尝试将单元素集合作为余积的候选者，试图从中提取元素，但我们将有两个嵌入而不是两个投影。它们不会告诉我们关于其来源的任何信息（事实上，我们已经看到它们忽略了输入参数）。余积到单元素集合的唯一态射也不会。集合的范畴看起来在初始对象方向和终端方向上非常不同。

这不是集合的内在性质，而是函数的性质，我们在 **Set** 中将其用作态射。一般来说，函数是不对称的。让我解释一下。

一个函数必须为其定义域集合的每个元素定义（在编程中，我们称其为全函数 **total function**），但它不必覆盖整个值域。我们已经看到了它的一些极端情况：从单元素集合到值域的函数——这些函数只选择值域中的单个元素。（实际上，从空集到值域的函数是极端情况。）当定义域的大小远小于值域的大小时，我们经常认为这些函数将定义域嵌入到值域中。例如，我们可以认为从单元素集合到值域的函数是将其单个元素嵌入到值域中。我称它们为嵌入函数 **embedding functions**，但数学家更喜欢给相反的函数命名：那些紧密填充其值域的函数被称为满射 **surjective** 或上的 **onto**。

另一种不对称性的来源是，函数被允许将定义域集合中的多个元素映射到值域的一个元素上。它们可以压缩它们。最极端的情况是，将整个集合映射到单元素集合的函数。你已经看到了多态的函数，它就是这么做的。压缩只能通过组合来复合。两个压缩函数的组合比单个函数更具压缩性。数学家给非压缩函数命名：他们称它们为单射 **injective** 或一对一 **one-to-one**。

当然，有些函数既不是嵌入也不是压缩的。它们被称为双射 **bijections**，它们是真正对称的，因为它们是可逆的。在集合范畴中，同构就是双射。

5.8 挑战

Challenges

1. 证明终端对象在唯一同构意义上是唯一的。
2. 偏序集中两个对象的积是什么？提示：使用泛结构。
3. 偏序集中两个对象的余积是什么？
4. 在你喜欢的语言（除了 Haskell）中实现 Haskell 的等效泛型类型。
5. 证明是比配备两个嵌入的“更好”的余积：

提示：定义一个函数

因式分解 和。

6. 继续前一个问题：你如何证明带有两个嵌入 和 的不能“优于”？

7. 继续：这些嵌入怎么样？

8. 提出一个劣质的 和 余积的候选者，因为它允许从其到 的多个可接受的态射，故它不能优于。

5.9 参考文献

Bibliography

1. The Catsters, Products and Coproducts¹ 视频。

¹

6

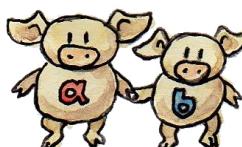
Simple Algebraic Data Types

我们已经看到了两种基本的类型组合方式：乘积（product）和余积（coproduct）。事实证明，在日常编程中，很多数据结构可以仅通过这两种机制来构建。这一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类型和余积类型，你就可以自动推导复合类型的相等运算符。在 Haskell 中，你可以为大量的复合类型自动派生相等性、比较、字符串转换等操作。

让我们仔细看看编程中出现的乘积类型和对类型。

6.1 乘积类型 (Product Types)

在编程语言中，两个类型的乘积的典型实现是对 (pair)。在 Haskell 中，对是一个原始类型构造器；在 C++ 中，它是一个相对复杂的标准库模板。



对不完全是可交换的：对不能替换为 对，尽管它们携带相同的信息。然而，它们是同构 (isomorphism) 可交换的，所谓同构是指通过 函数来实现的（该函数是其自身的逆）：

snippet01 你可以认为这两个对只是以不同的格式存储了相同的数据。这就像大端和小端之间的区别。

你可以通过在对中嵌套对来组合任意数量的类型，但有一个更简单的方法：嵌套的对等价于元组（tuples）。这是因为嵌套对的不同方式是同构的。如果你想按顺序组合三种类型、和 的乘积，你可以用两种方式来实现：

snippet02

或 **snippet03** 这些类型是不同的——你不能将一种类型传递给期望另一种类型的函数——但它们的元素是一一对应的。有一个函数可以将一种类型映射到另一种类型：

snippet04 这个函数是可逆的：

snippet05 所以它们是同构的。这些只是重新包装相同数据的不同方式。

你可以将创建乘积类型解释为类型上的二元运算。从这个角度来看，上述同构非常类似于我们在幺半群（monoid）中看到的结合律：

$$(a * b) * c = a * (b * c)$$

不同的是，在幺半群的情况下，两个乘积的组合是相等的，而在这里它们只是“同构”的相等。

如果我们可以接受同构，而不坚持严格的相等，我们可以更进一步，证明单位类型 就像 1 是乘法的单位元一样，是乘积的单位元。实际上，将某个类型 的值与一个单位配对并不会增加任何信息。类型：

snippet06 是同构于 的。下面是这个同构：

snippet07

snippet08 这些观察可以形式化为：**Set**（集合范畴）是一个幺半群范畴（monoidal category）。这是一种也是幺半群的范畴，意思是你可以对对象进行乘法（这里，取它们的笛卡尔积）。我将在未来详细讨论幺半群范畴，并给出完整的定义。

在 Haskell 中有一种更通用的方式来定义乘积类型，尤其是在它们与和类型结合使用时。它使用了带有多个参数的命名构造器。例如，一对可以定义为：

snippet09 在这里，是一个由两个其他类型 和 参数化的类型的名称；是数据构造器的名称。你可以通过向 类型构造器传递两个类型来定义一个对类型。你可以通过向构造器 传递两个适当类型的值来构造一个对值。例如，定义一个 值为 和 的对：

snippet10 第一行是类型声明。它使用类型构造器，其中 和 替代了 和泛型定义 中的 。第二行通过将一个具体的字符串和一个具体的布尔值传递给数据构造器 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。

由于 Haskell 中类型构造器和数据构造器的命名空间是分开的，你会经常看到同样的名字用于这两个部分，如：

snippet11 如果你仔细观察，你甚至可以将内置的对类型视为这种声明的一种变体，其中 的名称被替换为二元运算符。实际上，你可以像其他命名构造器一样使用 并用前缀表示法创建对：

snippet12 类似地，你可以使用 来创建三元组，等等。

除了使用泛型对或元组，你还可以定义特定的命名乘积类型，如：

snippet13 它只是 和 的乘积，但它被赋予了自己的名字和构造器。这种声明方式的优势在于，你可以定义许多具有相同内容但不同意义和功能的类型，并且它们不能相互替换。

用元组和多参数构造器进行编程可能会变得混乱且容易出错——你需要跟踪每个组件代表什么。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（record），在 C 中称为。

6.2 记录 (Records)

让我们看看一个简单的例子。我们想通过组合两个字符串（名称和符号）和一个整数（原子序数）来描述化学元素。我们可以使用一个元组 并记住每个组件代表什么。我们可以通过模式匹配来提取组件，例如这个检查元素符号是否是其名称前缀的函数（就像氦（He）是氦气（Helium）的前缀）：

snippet14 这段代码容易出错，难以阅读和维护。更好的方法是定义一个记录：

snippet15 这两种表示是同构的，这一点可以通过这两个相互逆转的转换函数来证明：

snippet16

snippet17 注意，记录字段的名称也充当了访问这些字段的函数。例如，从 中检索 字段。我们将 用作类型为：

snippet18 使用 的记录语法后，我们的 函数变得更加易读：

snippet19 我们甚至可以使用 Haskell 的技巧，通过将函数 用反引号包围将其转换为中缀运算符，并使其看起来几乎像一个句子：

snippet20 在这种情况下可以省略括号，因为中缀运算符的优先级低于函数调用。

6.3 和类型 (Sum Types)

正如集合范畴中的乘积产生了乘积类型，余积产生了和类型。Haskell 中和类型的典型实现是：

snippet21 像对一样，是（同构的）可交换的，可以嵌套，并且嵌套顺序无关紧要（同构）。因此，我们可以定义一个三元和类型的等价物：

snippet22 等等。

事实证明，**Set** 也是一个关于余积的（对称的）幺半群范畴。二元运算的作用由不相交的和（disjoint sum）来扮演，单位元素的作用由初

始对象来扮演。在类型方面，我们有 作为幺半群运算符，，即无值类型，作为其中性元素。你可以将 看作加法，将 看作零。实际上，向和类型中添加 并不会改变它的内容。例如：

snippet23 与 是同构的。这是因为没有办法构造此类型的 版本——没有 类型的值。的唯一元素是使用 构造器构造的，它们只是封装了类型的值。因此，象征性地， $a + 0 = a$ 。

和类型在 Haskell 中非常常见，但它们在 C++ 中的等价物，联合 (union) 或变体 (variant)，则要少得多。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 实现。Haskell 和类型的等价物是：

snippet24 而 C++ 的实现是：

一个更简单的和类型是：

snippet25 在 C++ 中的原始类型。

编码值的存在或不存在的简单和类型通常在 C++ 中使用特殊技巧和“无效”值来实现，例如空字符串、负数、空指针等。这种可选性 (optionality)，如果是故意的，则在 Haskell 中使用 类型表达：

snippet26 类型是两个类型的和。你可以看到，如果将两个构造器分成单独的类型，第一个类型看起来像这样：

snippet27 它是一个只有一个值的枚举，称为 。换句话说，它是一个单例，与单位类型 等价。第二部分是：

snippet28 这只是 类型的封装。我们可以将 编码为：

snippet29 在 C++ 中，复杂的和类型通常用指针伪装。指针可以是空的，或者指向特定类型的值。例如，一个 Haskell 列表类型，它可以定义为一个（递归的）和类型：

snippet30 可以翻译为 C++ 使用空指针技巧来实现空列表：

注意，两个 Haskell 构造器 和 被翻译为两个重载的 构造器，具有类似的参数（对于，没有；对于，是一个值和一个列表）。类不需要标签来区分和类型的两个组件。相反，它使用 的特殊值 来编码。

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。如果你使用一个特定的构造器创建了一个对象，该对象将永远记住使用了哪个构造器以及传递给它的参数。因此，创建为 的 对象永远不会变成 。同样，空列表将永远是空的，包含三个元素的列表将永远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，你可以始终将其分解为用于构造的部分。这种分解是通过模式匹配完成的，它重用了构造器作为模式。如果说有的话，构造器参数会被变量（或其他模式）替换。

数据类型有两个构造器，因此对任意 的分解使用了两个对应于这些构造器的模式。例如，下面是一个关于 的简单函数的定义：

`snippet31` 的定义的第一部分使用 构造器作为模式并返回。第二部分使用 构造器作为模式。它用通配符替换第一个构造器参数，因为我们对它不感兴趣。将 的第二个参数绑定到变量（尽管严格来说，变量永远不会变，因为一旦绑定到表达式，变量就永远不会改变，我还是会称这些东西为变量）。返回值是。现在，根据你的 是如何创建的，它将匹配其中一个子句。如果它是使用 创建的，那么传递给它的两个参数将被检索（第一个被丢弃）。

在 C++ 中，使用多态类层次结构实现的更复杂的和类型。一组具有共同祖先的类可以被理解为一个变体类型，其中 vtable 充当一个隐藏的标签。在 Haskell 中，通过对构造器进行模式匹配并调用专用代码来完成的事情，在 C++ 中通过基于 vtable 指针调用虚函数来实现。

你很少会看到 在 C++ 中作为 和类型使用，因为对可以放入 union 中的内容有严格的限制。你甚至不能将 放入 union 中，因为它有一个复制构造函数。

6.4 类型的代数 (Algebra of Types)

分别来看，乘积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使用。我们再次借助了组合的力量。

让我们总结一下到目前为止的发现。我们已经看到两个底层的可交换幺半群结构：我们有 和类型，其中性元素为；我们有乘积类型，其中性元素为单位类型。我们希望将它们类比为加法和乘法。在这个类比中，对应于零，单位类型 对应于一。

让我们看看这个类比能延伸到多远。例如，乘以零是否等于零？换句话说，是否有一个组件为 的乘积类型是同构于 的？例如，是否有可能创建一个 和 的对？

要创建一个对，你需要两个值。尽管你可以轻松生成一个整数，但没有 类型的值。因此，对于任何类型，类型 是无效的——没有值——因此等价于。换句话说， $a \times 0 = 0$ 。

另一个将加法和乘法联系在一起的东西是分配律：

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

这个性质对于乘积类型和 和类型是否成立？是的，它成立——和往常一样，最多是同构。左边对应的类型是：

`snippet32` 而右边对应的类型是：

`snippet33` 下面是将它们转换为一个方向的函数：

`snippet34` 而这是另一个方向的函数：

`snippet35` 语句用于函数内部的模式匹配。每个模式后面跟随一个箭头和当模式匹配时要评估的表达式。例如，如果你用值调用：

`snippet36` 在 中的 将等于。它将匹配模式，将 替换为。由于 已经匹配到，因此 子句的结果，以及整个函数的结果，将是，如预期的那样。

我不打算证明这两个函数是彼此的逆，但如果你仔细考虑，它们必须是的！它们只是简单地重新打包了两个数据结构的内容。数据是相同的，只是格式不同。

数学家有一个术语来形容这样的交错幺半群：这叫做半环（semiring）。这不是一个完整的环（ring），因为我们不能定义类型的减法。这就是为什么半环有时被称为rig，这是一种双关语，表示“没有n（负数）的环”。但即使如此，我们仍然可以通过将关于自然数的陈述（它们形成一个rig）转换为关于类型的陈述，来获得很多启示。以下是一些有趣条目的对照表：

数字	类型
0	
1	
$a + b$	
$a \times b$	or
$2 = 1 + 1$	
$1 + a$	

列表类型非常有趣，因为它是方程的解。我们正在定义的类型出现在方程的两边：

snippet37 如果我们进行通常的替换，并且还用 替换，我们会得到方程：

我们不能使用传统的代数方法来解决它，因为我们不能对类型进行减法或除法。但我们可以尝试一系列替换，不断用 替换右边的，并使用分配律。这会导致以下系列：

我们最终得到一个无限的乘积和（元组），可以解释为：一个列表要么是空的，；要么是单例，；要么是一对，；要么是三元组，；等等……嗯，这正是列表的定义——一个由 组成的字符串！

关于列表还有更多的内容，我们将在学习函子（functors）和不动点（fixed points）之后回到列表和其他递归数据结构。

用符号变量求解方程——这就是代数！这就是给这些类型命名为代数数据类型的原因。

最后，我应该提到类型代数的一个非常重要的解释。注意，两个类型 和 的乘积必须同时包含 类型的值 和 类型的值，这意味着这两种

类型都必须是可占有的（inhabited）。另一方面，两个类型的和包含了类型的值 或 类型的值，所以只要其中一个是可占有的就足够了。逻辑的 *and* 和 *or* 也形成了一个半环，它也可以映射到类型理论中：

逻辑	类型
<i>false</i>	
<i>true</i>	
$a \parallel b$	
$a \&\& b$	

这种类比更深入，是 Curry-Howard 同构（isomorphism）在逻辑与类型理论之间的基础。当我们讨论函数类型时，我们将再次讨论这个问题。

6.5 挑战 (Challenges)

1. 展示 和 之间的同构性。
2. 这里有一个在 Haskell 中定义的和类型：

当我们想要定义一个作用于 的函数如 时，我们通过对这两个构造器进行模式匹配来完成：

在 C++ 或 Java 中将 实现为一个接口，并创建两个类： 和 。将 实现为一个虚函数。

3. 继续前一个示例：我们可以轻松地添加一个新函数，它计算 的周长。我们可以在不触及 定义的情况下完成：

将 添加到你的 C++ 或 Java 实现中。你必须触及原始代码的哪些部分？

4. 继续进一步：向 添加一个新形状，并进行所有必要的更新。在 Haskell 和 C++ 或 Java 中，你需要修改哪些代码？（即使你不是 Haskell 程序员，这些修改也应该相当明显。）
5. 证明 $a + a = 2 \times a$ 在类型上成立（最多同构）。记住 2 对应于 ，根据我们的对照表。

7

Functors

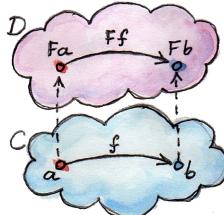
尽管可能会让人觉得我像在重复自己, 我还是要说关于函子(functor)的事情: 函子是一个非常简单但非常强大的概念。范畴论(category theory)中充满了这些简单而强大的思想。函子是范畴之间的映射。给定两个范畴, **C** 和 **D**, 一个函子 F 将 **C** 中的对象映射到 **D** 中的对象——它是一个对象上的函数。如果 a 是 **C** 中的一个对象, 我们将它在 **D** 中的像写作 Fa (不加括号)。但范畴不仅仅是对象——它还包括连接它们的态射(morphisms)。函子也映射态射——它是一个态射上的函数。但它不会随意映射态射——它保留了连接性。因此, 如果在范畴 **C** 中有一个态射 f 连接对象 a 和对象 b ,

$$f :: a \rightarrow b$$

则 f 在范畴 **D** 中的像 Ff 将连接 a 的像和 b 的像:

$$Ff :: Fa \rightarrow Fb$$

(这是一种数学和 Haskell 符号的混合, 希望现在能够理解。我不会在将函子应用于对象或态射时使用括号。)



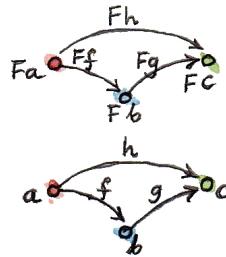
如你所见, 函子保留了范畴的结构: 一个范畴中连接的东西将在另一个范畴中保持连接。但范畴的结构还有其他方面: 还包括态射的复合。如

果 h 是 f 和 g 的复合:

$$h = g \circ f$$

我们希望 F 下的 h 的像是 f 和 g 的像的复合:

$$Fh = Fg \circ Ff$$



最后，我们希望 \mathbf{C} 中的所有恒等态射 (identity morphisms) 都被映射为 \mathbf{D} 中的恒等态射:

$$F\mathbf{id}_a = \mathbf{id}_{Fa}$$

$$F\mathbf{id}_a = \mathbf{id}_{Fa}$$

$$\mathbf{id}_a$$

注意，这些条件使得函子比普通的函数更加严格。函子必须保留范畴的结构。如果你将范畴想象为由一张态射网络连接在一起的一组对象，函子不允许在这个网络中引入任何裂缝。它可以将对象合并在一起，它可以将多个态射粘合为一个，但它不能将它们分开。这种不撕裂的约束类似于你在微积分中可能知道的连续性条件。从这个意义上说，函子是“连续的”（虽然实际上存在更严格的连续性概念）。就像函数一样，函子可以进行折叠和嵌入。当源范畴比目标范畴小得多时，嵌入的方面更为突出。在极端情况下，源可以是平凡的单对象范畴 (singleton category) ——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选择该范畴中的一个对象。这完全类似于从单元素集到目标集合中选择元素的态射的性质。最大程度折叠的函子

称为常量函子（constant functor） Δ_c 。它将源范畴中的每个对象映射到目标范畴中选定的某个对象 c 。它还将源范畴中的每个态射映射到恒等态射 id_c 。它的作用就像一个黑洞，将所有事物压缩成一个奇点。当我们讨论极限和余极限时，我们会看到更多的这种函子。

7.1 编程中的函子 (Functors in Programming)

让我们回到实际，谈谈编程。我们有一个类型和函数的范畴。我们可以讨论将这个范畴映射到自身的函子——这样的函子称为自函子 (endofunctors)。那么，在类型范畴中的自函子是什么？首先，它将类型映射到类型。我们已经看到了一些这样的映射的例子，也许我们还没有意识到它们就是这种映射。我指的是那些由其他类型参数化的类型定义。让我们来看几个例子。

7.1.1 Maybe 函子

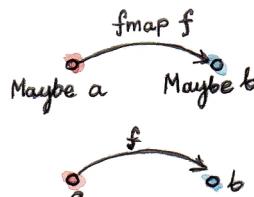
的定义是将类型 映射到类型：

snippet01 这里有一个重要的细微差别：本身不是一种类型，它是一个类型构造器 (type constructor)。你必须给它一个类型参数，比如或，才能将其转换为一种类型。没有任何参数表示它是一个作用于类型的函数。但是我们能将 转换成一个函子吗？（从现在起，当我在编程的上下文中谈到函子时，我几乎总是指自函子。）一个函子不仅是一个对象（在这里是类型）的映射，而且也是一个态射（在这里是函数）的映射。对于从 到 的任意函数：

snippet02 我们希望生成一个从 到 的函数。为了定义这样的函数，我们将要考虑 的两个构造器对应的两种情况。的情况很简单：我们只需返回。如果参数是，我们将把函数 应用于其内容。因此，下的 f 的像是函数：

snippet03 （顺便说一句，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高阶函数，称为。对于，它具有以下签名：

snippet04



我们通常会说“提升”了一个函数。提升的函数作用于 值上。与往常一样，由于柯里化，这个签名可以从两种方式解释：作为一个参数的函数

——它本身是一个函数——返回一个函数；或者作为两个参数的函数，返回：

snippet05 基于我们之前的讨论，这就是我们如何为 实现：

snippet06 要证明类型构造器 和函数 形成了一个函子，我们必须证明 保持恒等性和复合性。这些被称为“函子定律”，它们只是确保了范畴结构的保存。

7.1.2 等式推理 (Equational Reasoning)

为了证明函子定律，我将使用等式推理，这是一种在 Haskell 中常见的证明技巧。它利用了 Haskell 函数被定义为等式这一事实：左边等于右边。你总是可以将一个替换为另一个，可能会重命名变量以避免名称冲突。把这想象成内联一个函数，或者反过来，重构一个表达式为一个函数。让我们以恒等函数为例：

snippet07 例如，如果你在某个表达式中看到，你可以将它替换为（内联）。进一步说，如果你看到 应用于一个表达式，例如，你可以将它替换为表达式本身。而且这种替换是双向的：你可以将任意表达式替换为（重构）。如果一个函数通过模式匹配定义，你可以独立使用每个子定义。例如，给定上面 的定义，你可以将 替换为，反之亦然。让我们看看这在实际中的应用。让我们从保持恒等性开始：

snippet08 有两种情况需要考虑： 和 。这是第一种情况（我使用 Haskell 伪代码将左边转换为右边）：

注意，在最后一步我反向使用了 的定义。我用 替换了表达式。实际上，你通过“从两端烧蜡烛”来进行这样的证明，直到在中间遇到相同的表达式——在这里是 。第二种情况也很简单：

现在，让我们展示 保持复合性：

snippet09 首先是 情况：

然后是 情况：

值得强调的是，等式推理不适用于具有副作用的 C++ 风格的“函数”。考虑以下代码：

使用等式推理，你会将 内联为：

这显然不是一个有效的转换，也不会产生相同的结果。尽管如此，如果你将 `实现为宏`，C++ 编译器仍然会尝试使用等式推理，结果可能会灾难性地错误。

7.1.3 Optional

函子在 Haskell 中很容易表达，但它们可以在任何支持泛型编程和高阶函数的语言中定义。让我们考虑 C++ 中 的类似物，模板类型。这是实现的一个简要示意图（实际实现要复杂得多，涉及参数传递的各种方式，复制语义，以及 C++ 特有的资源管理问题）：

此模板提供了函子定义的一部分：类型映射。它将任何类型 映射到新的类型。让我们定义它在函数上的作用：

这是一个高阶函数，将一个函数作为参数并返回一个函数。下面是它的非柯里化版本：

还有一种选择是将 作为 的模板方法。这种选择的尴尬之处在于，在 C++ 中抽象出函子模式是一个问题。函子是否应该是一个接口供继承（不幸的是，你不能有模板虚函数）？它应该是一个柯里化还是非柯里化的自由模板函数？C++ 编译器是否能够正确推断出缺失的类型，还是应该明确指定它们？考虑一种情况，输入函数 将 转换为 。编译器如何确定 的类型：

特别是，如果将来有多个函子重载？

7.1.4 类型类 (Typeclasses)

那么 Haskell 是如何抽象出函子的呢？它使用了类型类 (typeclass) 机制。类型类定义了支持共同接口的类型族。例如，支持相等运算的对象类定义如下：

snippet10 此定义说明类型 是 类的一部分，如果它支持操作符，该操作符接受两个类型为 的参数并返回。如果你想告诉 Haskell 某个特定类型是 ，你必须声明它是这个类的一个实例并提供 的实现。例如，给定二维（两个 的积类型）的定义：

snippet11 你可以定义点的相等性：

snippet12 在这里，我使用了操作符（即我正在定义的那个）在两个模式 和 之间的中缀位置。函数体位于单个等号之后。一旦 被声明

为 的实例，你就可以直接比较点的相等性。注意，与 C++ 或 Java 不同，你不必在定义 时指定 类（或接口）——你可以稍后在客户端代码中进行指定。类型类也是 Haskell 唯一的函数（和操作符）重载机制。我们将需要它来重载不同函子的。不过这里有一个复杂的情况：函子并不是定义为一种类型，而是定义为类型的映射，即类型构造器。我们需要一个类型类，它不是类型的族，而是类型构造器的族。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，这里是 类的定义：

snippet13 它规定 是一个，如果存在一个具有指定类型签名的函数 。小写的 是一个类型变量，类似于类型变量 和 。不过，编译器能够通过查看其用法推断出它表示的是类型构造器而不是类型：作用于其他类型上，如 和 。因此，当声明 的实例时，你必须给出一个类型构造器，就像 一样：

snippet14 顺便说一下，类以及很多简单数据类型（包括 ）的实例定义都是标准 Prelude 库的一部分。

7.1.5 C++ 中的函子 (Functor in C++)

我们可以在 C++ 中尝试相同的方法吗？类型构造器对应于模板类，如，因此通过类比，我们可以用模板模板参数 来参数化。这是它的语法：

我们希望能够为不同的函子专门化这个模板。不幸的是，C++ 中禁止部分专门化模板函数。你不能写：

相反，我们不得不依赖函数重载，这使我们回到了原始的非柯里化 定义：

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

7.1.6 列表函子 (The List Functor)

为了对编程中函子的作用有一些直观认识，我们需要看更多的例子。任何由另一种类型参数化的类型都是函子的候选对象。泛型容器是由它们存储的元素类型参数化的，所以让我们看看一个非常简单的容器——列表：

snippet15 我们有类型构造器，它将任何类型 映射到类型。要证明 是一个函子，我们必须定义函数的提升：给定一个函数，定义一个函数：

snippet16 作用于 的函数必须考虑两种情况，对于列表的两个构造器。的情况很简单——只需返回，因为对一个空列表你无能为力。的情况有点复杂，因为它涉及递归。所以让我们退一步，考虑我们想要做什么。我们有一个 列表，一个将 转换为 的函数，我们想要生成一个 列表。显而易见的做法是使用 将列表中的每个元素从 转换为 。在实践中，如何实现这一点，给定一个（非空的）列表定义为头部和尾部的？我们将 应用于头部，并将提升后的（后的）应用于尾部。这是一个递归定义，因为我们正在用提升后的 定义提升后的：

snippet17 注意，在右边 应用于的列表比我们定义的列表要短——它作用于尾部。我们递归地处理越来越短的列表，因此我们必然最终到达空列表或。但正如我们之前决定的那样，作用于 返回，从而终止递归。要得到最终结果，我们将新的头部 和新的尾部 用 构造器组合起来。将这一切整合在一起，这是列表函子的实例声明：

snippet18 如果你对 C++ 更熟悉，可以考虑，它可能是最通用的 C++ 容器。的 实现只是 的一个薄包装：

我们可以用它来对一系列数字的元素进行平方：

大多数 C++ 容器通过实现可传递给 的迭代器而成为函子，这些迭代器是 的较原始版本。不幸的是，函子的简单性在通常的迭代器和临时变量的杂乱中丢失了（参见上面的 实现）。我很高兴地说，新的提议的 C++ range 库使得范围的函子性质更加明显。

7.1.7 Reader 函子 (The Reader Functor)

现在你可能已经形成了一些直觉——例如，函子是某种类型的容器——让我向你展示一个看起来完全不同的例子。考虑将类型 映射到 返回 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——但我们对这些作为程序员有一些理解。在 Haskell 中，函数类型是使用箭头类型构造器 构造的，它接受两个类型：参数类型和结果类型。你已经看到它的中缀形式，但它同样可以在前缀形式中使用，当它被括号包围时：

snippet19 就像普通函数一样，多参数类型函数也可以部分应用。因此，当我们只提供一个类型参数给箭头时，它仍然期待另一个参数。这就是为什么：

snippet20 是一个类型构造器。它还需要另一个类型 才能生成一个完整的类型。就其本质而言，它定义了一整套类型构造器，参数化于。让我们看看这是否也是一套函子。处理两个类型参数可能有点混乱，所以让我们重命名一下。我们称参数类型为，结果类型为，与我们之前的函子定义一致。因此，我们的类型构造器接受任何类型 并将其映射为类型。要证明它是一个函子，我们想要将一个函数 提升为一个接

受 并返回 的函数。这些是分别使用类型构造器 作用于 和 时形成的类型。以下是应用于这种情况的 的类型签名：

snippet21 我们需要解决以下难题：给定一个函数 和一个函数，创建一个函数。我们只能以一种方式将这两个函数组合起来，结果恰好是我们需要的。因此，这是我们 的实现：

snippet22 它就是有效的！如果你喜欢简洁的符号，这个定义可以进一步简化，注意到组合可以用前缀形式重写：

snippet23 并且可以省略参数，得到两个函数的直接等式：

snippet24 这种类型构造器 和上面 的实现组合在一起，称为 reader 函数。

7.2 函子作为容器 (Functors as Containers)

我们已经看到了一些在编程语言中作为通用容器定义的函子的例子，或者至少是包含它们所参数化的类型的值的对象。reader 函数似乎是一个特例，因为我们不认为函数是数据。但我们已经看到，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。反过来，由于 Haskell 的惰性，传统的容器，比如列表，实际上可能是作为函数实现的。考虑一个自然数的无限列表，它可以简洁地定义为：

snippet25 在第一行中，方括号是 Haskell 内置的列表类型构造器。在第二行中，方括号用于创建列表字面量。显然，这样的无限列表不能存储在内存中。编译器将其实现为一个按需生成 的函数。Haskell 有效地模糊了数据和代码之间的界限。列表可以被视为一个函数，而函数可以被视为一个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，这甚至可能是可行的。然而，将 实现为表查找是不现实的，因为有无限多种不同的字符串。作为程序员，我们不喜欢无限，但在范畴论中，你会学会处理无限。无论是所有字符串的集合还是宇宙的所有可能状态，过去、现在和未来——我们都可以应对！所以，我喜欢将函子对象（一个由自函子生成的类型的对象）视为包含它所参数化的类型的值，即使这些值并不实际存在于其中。函子对象的一个例子是 C++ 中的，它可能在某个时候包含一个值，但不能保证它一定会；如果你想访问它，可能会阻塞等待另一个线程完成执行。另一个例子是 Haskell 中的 对象，它可能包含用户输入，或者显示“Hello World!”的未来宇宙版本。根据这种解释，函子对象可能包含它所参数化的类型的一个或多个值。或者它可能包含生成这些值的配方。我们完全不关心是否能够访问这些值——这是完全可选的，并且超出了函子的范围。我们所关心的只是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，我们关心的只是这些操作是否能够正确组合，并且使用恒等函数的操作不会改变任何东西。为了向你展示我们有多么不关心是否能够访问函子对象中的值，下面是一个完全忽略其参数 的类型构造器：

`snippet26` 类型构造器接受两个类型，`和`。就像我们之前处理箭头构造器一样，我们将部分应用它来创建一个函子。数据构造器（也称为）只接受一个类型为 的值。它与 没有任何依赖关系。对于这个类型构造器， 的类型是：

`snippet27` 由于函子忽略了其类型参数， 的实现可以自由地忽略其函数参数——该函数没有作用的对象：

`snippet28` 在 C++ 中这可能会更清楚（我从未想过我会说出这些话！），在那里类型参数和值之间有更强的区分——前者是编译时的，后者是运行时的：

C++ 中的 实现同样忽略了函数参数，并且本质上重新转换了 参数而不改变其值：

尽管它看起来很奇怪，但 函子在许多结构中起着重要作用。在范畴论中，它是我之前提到的常量函子 Δ_c 的特例——黑洞的自函子情况。我们将在未来看到更多它的应用。

7.3 函子组合 (Functor Composition)

很容易让自己相信范畴之间的函子是可以组合的，就像集合之间的函数可以组合一样。两个函子的组合，在作用于对象时，只是它们各自的对象映射的组合；在作用于态射时，也是类似的。当穿过两个函子时，恒等态射最终仍然是恒等态射，态射的组合最终仍然是态射的组合。这真的没什么特别的。特别是，自函子的组合很容易。还记得 函数吗？我将用 Haskell 内置的列表实现重写它：

`snippet29` （我们之前称为 的空列表构造器被替换为一对空方括号。`构造器`被替换为中缀操作符（冒号）。）的结果是 和 两个函子的组合，作用于 。每个函子都有其自己的 版本，但如果我们要将某个函数应用于组合：一个 列表怎么办？我们必须突破两个函子的层次。我们可以使用 突破外部的。但我们不能直接将 发送到 内部，因为 不能作用于列表。我们必须发送 来操作内部列表。例如，让我们看看如何对一个 整数列表的元素进行平方：

`snippet30` 编译器在分析类型后，会发现对于外部的，它应该使用实例中的实现，而对于内部的，则使用列表函子的实现。可能并不立即明显，上面的代码可以重写为：

`snippet31` 但记住， 可以被视为只有一个参数的函数：

`snippet32` 在我们的例子中， 中的第二个 的参数是：

`snippet33` 并返回一个类型为：

`snippet34` 的函数。第一个 然后接受该函数并返回一个函数：

`snippet35` 最后，该函数被应用于 。因此，两个函子的组合是一个函子，其 是相应的 组合。回到范畴论：很明显，函子组合是结合的

(对象的映射是结合的，态射的映射也是结合的)。而且每个范畴中都有一个平凡的恒等函子：它将每个对象映射到自身，并将每个态射映射到自身。因此，函子具有与某些范畴中的态射相同的属性。但那个范畴是什么呢？它必须是一个对象是范畴，态射是函子的范畴。这是范畴的范畴。但是所有范畴的范畴必须包含它自身，这会导致与使得所有集合的集合不可能的悖论相同的悖论。然而，存在一个所有小范畴的范畴，称为 **Cat**（它是大的，所以不能是它自身的成员）。一个小范畴是其中对象形成一个集合的范畴，而不是比集合更大的东西。请注意，在范畴论中，即使是无限不可数的集合也被认为是“小的”。我之所以提到这些事情，是因为我觉得能够在许多抽象层次上识别出相同的结构是相当令人惊奇的。稍后我们将看到函子也形成了范畴。

7.4 挑战 (Challenges)

1. 我们能否通过定义以下内容将类型构造器变成一个函子：

它忽略了两个参数？（提示：检查函子定律。）

2. 证明 `reader` 函子的函子定律。提示：这非常简单。
3. 在你第二喜欢的语言中实现 `reader` 函子（当然，第一喜欢的语言是 Haskell）。
4. 证明列表函子的函子定律。假设该定律对你应用它的列表的尾部部分是成立的（换句话说，使用归纳法）。

8

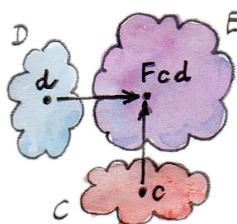
Functoriality

现

在你已经了解了什么是函子，并且看过了一些例子，现在让我们看看如何从更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（对应于范畴中的对象之间的映射）可以扩展为函子（包括态射之间的映射）。

8.1 双函子 (Bifunctors)

由于函子是 **Cat** (范畴的范畴) 中的态射，许多关于态射的直觉——特别是关于函数的直觉——同样适用于函子。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，或称为双函子。在对象上，双函子将每对对象，一个来自范畴 **C**，另一个来自范畴 **D**，映射到范畴 **E** 中的一个对象。注意，这只是说它是从范畴 $\mathbf{C} \times \mathbf{D}$ 的笛卡尔积到 **E** 的映射。



这相当直接。但是函子性意味着双函子也必须映射态射。不过这次，它必须将一对态射，一个来自 **C**，另一个来自 **D**，映射到 **E** 中的一个态射。

同样，一对态射只是在乘积范畴 $\mathbf{C} \times \mathbf{D}$ 中的一个态射。我们将笛卡尔积范畴中的态射定义为从一对对象到另一对对象的态射对。这些态

射对可以以显而易见的方式进行组合：

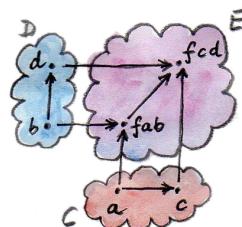
$$(f, g) \circ (f', g') = (f \circ f', g \circ g')$$

这种组合是结合的，并且它有一个恒等元——一对恒等态射 (**id**, **id**)。所以笛卡尔积范畴确实是一个范畴。

思考双函子的一种更简单的方式是将它们视为每个参数单独的函子。因此，不必将函子性法则——结合性和恒等性——从函子翻译到双函子，只需分别为每个参数单独检查它们。然而，一般来说，单独的函子性不足以证明联合函子性。在联合函子性失败的范畴中，这些范畴被称为准单元（premonoidal）。

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。双函子是一个接受两个类型参数的类型构造器。以下是来自库 的 类型类的定义：

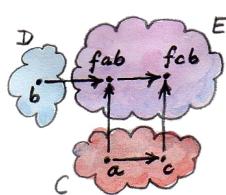
snippet01



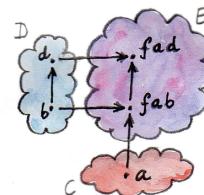
bimap

类型变量 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了：一次映射两个函数。结果是一个提升后的函数，作用于由双函子类型构造器生成的类型。还有一个基于 和 的默认实现。（如前所述，这并不总是有效，因为两个映射可能不交换，也就是说 可能与 不相同。）

另外两个类型签名 和 是两个，分别见证 在第一个和第二个参数上的函子性。



first



second

类型类定义提供了它们两者的默认实现，基于。

在声明 的实例时，你可以选择实现 并接受 和 的默认值，或者实现 和 并接受 的默认值（当然，你也可以实现这三者，但此时确保它们彼此之间相关联的责任在你）。

8.2 积函子和余积函子 (Product and Coproduct Bi-functors)

双函子的重要例子之一是范畴积——两个对象的积，它由一个通用构造定义。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。这在一般情况下成立，特别是在 Haskell 中也是如此。以下是配对构造器的 实例——最简单的积类型：

snippet02[b] 没有太多选择：简单地将第一个函数应用于第一个组件，将第二个函数应用于配对的第二个组件。考虑到类型，代码几乎是自动生成的：

snippet03 这里双函子的作用是生成类型的对，例如：

通过对偶，余积，如果它定义在范畴中的每对对象上，也将是一个双函子。在 Haskell 中，这由 类型构造器作为 实例体现：

snippet04[b] 这段代码也几乎是自动生成的。

现在，记得我们讨论过的单元范畴吗？单元范畴定义了一个作用于对象的二元操作符，以及一个单位对象。我提到过，相对于笛卡尔积，**Set** 是一个单元范畴，单例集合作为单位。而且它也是一个相对于不相交并集的单元范畴，空集合作为单位。我没有提到的是，单元范畴的一个要求是二元操作符是一个双函子。这是一个非常重要的要求——我们希望单元积与范畴的结构兼容，该结构由态射定义。现在我们离单元范畴的完整定义又近了一步（我们还需要学习自然性，然后我们才能到达那里）。

8.3 函子的代数数据类型 (Functorial Algebraic Data Types)

我们已经看到了一些参数化数据类型的例子，它们最终被证明是函子——我们能够为它们定义。复杂的数据类型是由更简单的数据类型构造的。特别是，代数数据类型 (ADTs) 是使用和与积构造的。我们刚刚看到和与积是函子的。我们也知道函子是可以组合的。因此，如果我们能证明 ADTs 的基本构建块是函子的，我们就能知道参数化的 ADTs 也是函子的。

那么什么是参数化代数数据类型的构建块？首先，有些项与函子的类型参数无关，比如 中的，或者 中的。它们等价于 函子。记住，函

子忽略了它的类型参数（实际上，它忽略了第二个类型参数，即我们感兴趣的那个，第一个保持不变）。

然后，有些元素只是简单地封装了类型参数本身，比如 中的。它们等价于恒等函子。我之前提到过恒等函子，作为 *Cat* 中的恒等态射，但没有给出它在 Haskell 中的定义。这里是：

snippet05

snippet06 你可以将 视为始终存储一个类型的（不可变的）值的最简单的容器。

代数数据结构中的其他所有内容都是使用这些两个原语通过积与和构造的。

有了这个新知识，让我们重新审视 类型构造器：

snippet07 它是两个类型的和，而我们现在知道和是函子的。第一部分，可以表示为作用于 的（的第一个类型参数设置为单位——稍后我们将看到 的更有趣的用途）。第二部分只是恒等函子的另一个名字。我们可以将 定义为，直到同构为止：

snippet08 因此 是双函子 与两个函子 和 的组合。（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——我们可以很容易地说服自己，同样的道理适用于双函子。我们需要做的就是弄清楚双函子与两个函子的组合如何在态射上起作用。给定两个态射，我们简单地用一个函子提升其中一个，用另一个函子提升另一个态射。然后我们用双函子提升得到的态射对。

我们可以在 Haskell 中表达这个组合。让我们定义一个数据类型，它由双函子 参数化（它是一个接受两个类型作为参数的类型构造器），两个函子 和 （每个都接受一个类型变量的类型构造器），以及两个常规类型 和 。我们将 应用于 ，将 应用于 ，然后将 应用于结果的两个类型：

snippet09 这就是对象或类型上的组合。注意在 Haskell 中我们如何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，不妨尝试将 应用于 、 、 和 ，以此顺序。你会恢复我们精简版的（被忽略）。

新的数据类型 是 和 的双函子，但前提是 本身是一个，而 和 是 s。编译器必须知道在 使用时，将有一个 的定义可用，以及 和 的 的定义。在 Haskell 中，这在实例声明中通过类约束集和双箭头来表达：

snippet10[b] 的 的实现是基于 的 和两个（对于 和 ）的实现。每当使用 时，编译器会自动推断出所有类型，并选择正确的重载函数。

定义中的 具有如下类型：

snippet11 这真是有点绕口。外层 通过外层 层次，两个 分别深入到 和 内部。如果 和 的类型是：

snippet12 那么最终结果的类型是：

`snippet13[b]` 如果你喜欢拼图，这种类型操作可以提供数小时的娱乐。

因此，事实证明，我们不必证明 是一个函子——这个事实是由于它被构造为两个函子性原语的和而得出的。

敏锐的读者可能会问一个问题：如果代数数据类型的 实例的推导如此机械化，难道不能由编译器自动化执行吗？确实如此，并且实际上就是这样。你只需要通过在源文件顶部包含以下行来启用特定的 Haskell 扩展：

然后将 添加到你的数据结构中：

并且相应的 将由编译器为你实现。

代数数据结构的规则性使得不仅可以推导出 实例，还可以推导出其他几个类型类的实例，包括我之前提到的 类型类。还可以选择教授编译器派生你自己的类型类实例，但这有点高级。然而，思路是相同的：你为基本构建块、和与积提供行为，然后让编译器找出其余的内容。

8.4 C++ 中的函子 (Functors in C++)

如果你是一名 C++ 程序员，那么在实现函子方面显然只能靠自己了。然而，你应该能够在 C++ 中识别出一些代数数据结构类型。如果这种数据结构被制作成通用模板，你应该能够快速为其实现。

让我们看看一个树形数据结构，我们在 Haskell 中将其定义为递归和类型：

`snippet14` 正如我之前提到的，在 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 实现为基类 的虚函数然后在所有子类中重载它是很自然的。不幸的是，这是不可能的，因为 是一个模板，不仅由它作用的对象类型（指针）参数化，还由应用于它的函数的返回类型参数化。在 C++ 中，虚函数不能模板化。我们将 实现为一个通用的自由函数，并用 替换模式匹配。

基类必须至少定义一个虚函数以支持动态转换，因此我们将析构函数设置为虚函数（这在任何情况下都是一个好主意）：

只是一个伪装成 的函子：

是一个积类型：

在实现时，我们利用了类型的动态调度。情况下应用了版本的，而情况下则被视为一个与两个函子副本组合的双函子。作为C++程序员，你可能不习惯以这些术语分析代码，但这是一个很好的范畴思考练习。

为了简单起见，我决定忽略内存和资源管理问题，但在生产代码中，你可能会使用智能指针（独占或共享，具体取决于你的策略）。

将其与Haskell中的实现进行比较：

snippet15这个实现也可以由编译器自动派生。

8.5 写入函子 (The Writer Functor)

我承诺过会回到我之前描述的Kleisli范畴。该范畴中的态射表示为返回数据结构的“装饰”函数。

snippet16我说过这种装饰与内函子(endofunctors)有某种关系。实际上，类型构造器在中是函子。我们甚至不需要为其实现，因为它只是一个简单的积类型。

但Kleisli范畴与函子之间的一般关系是什么呢？作为一个范畴，Kleisli范畴定义了组合和恒等性。让我提醒你，组合由鱼运算符给出：

snippet17恒等态射由称为的函数给出：

snippet18事实证明，如果你花足够长的时间（我指的是足够长的时间）看这些两个函数的类型，你可以找到一种方法将它们组合起来，以产生一个具有合适类型签名的函数作为。像这样：

snippet19这里，鱼运算符结合了两个函数：一个是熟悉的，另一个是将作用于lambda的参数并对其结果应用的lambda。最难理解的部分可能是的使用。鱼运算符的参数不是应该是一个接受“正常”类型并返回装饰类型的函数吗？实际上不是。没有人说在中必须是一个“正常”的类型。它是一个类型变量，因此它可以是任何东西，特别是它可以是一个装饰类型，比如。

所以会将转换为。鱼运算符会取出的值并作为传递给lambda。在那里，会将它转换为，会将其装饰，使其成为。综合起来，我们最终得到一个接受并返回的函数，这正是应该产生的。

注意，这个论点非常通用：你可以用任何类型构造器替换。只要它支持鱼运算符和，你就可以定义。因此，Kleisli范畴中的装饰总是`一个函子`。（但并不是每个函子都会产生一个Kleisli范畴。）

你可能想知道我们刚才定义的是否与编译器为我们派生的相同。有趣的是，确实如此。这是因为Haskell实现多态函数的方式。它被称为参数多态性(parametric polymorphism)，这是所谓的免费定理(theorems for free)的来源。其中特别一个定理说，如果对于给定的类型构造器有一个实现，并且它保持恒等性，那么它一定是唯一的。

8.6 协变和逆变函子 (Covariant and Contravariant Functors)

现在我们已经回顾了写入函子，让我们回到读取函子。它基于部分应用的函数箭头类型构造器：

snippet20 我们可以将其重写为类型同义词：

snippet21 我们之前见过的 实例如下所示：

snippet22 但就像配对类型构造器或 类型构造器一样，函数类型构造器接受两个类型参数。配对和 在两个参数中都是函子性——它们是双函子。函数构造器也是双函子吗？

让我们尝试使它在第一个参数中函子性。我们从一个类型同义词开始——它就像 但参数顺序相反：

snippet23 这次我们固定返回类型，并改变参数类型。让我们看看我们是否可以以某种方式匹配类型来实现，它将具有以下类型签名：

snippet24 仅凭两个接受 并分别返回 和 的函数，没有任何方法可以构建一个接受 并返回 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 并返回，情况就会有所不同。我们不能反转一个任意的函数，但我们可以进入对偶范畴。

简单回顾一下：对于每个范畴 C ，都有一个对偶范畴 C^{op} 。这是一个与 C 具有相同对象的范畴，但所有箭头都反转了。

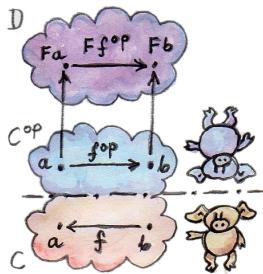
考虑一个从 C^{op} 到另一个范畴 D 的函子：

$$F :: C^{op} \rightarrow D$$

这样的函子将 C^{op} 中的态射 $f^{op} :: a \rightarrow b$ 映射到 D 中的态射 $Ff^{op} :: Fa \rightarrow Fb$ 。但态射 f^{op} 实际上对应于原范畴 C 中的某个态射 $f :: b \rightarrow a$ 。注意到这个反转。

现在， F 是一个常规的函子，但我们可以基于 F 定义另一个映射，它不是一个函子——我们称之为 G 。这是从 C 到 D 的映射。它以与 F 相同的方式映射对象，但当涉及映射态射时，它会反转它们。它接受 C 中的态射 $f :: b \rightarrow a$ ，首先将其映射到对偶态射 $f^{op} :: a \rightarrow b$ ，然后在其上使用函子 F ，得到 $Ff^{op} :: Fa \rightarrow Fb$ 。

考虑到 Fa 与 Ga 相同， Fb 与 Gb 相同，整个过程可以描述为： $Gf :: (b \rightarrow a) \rightarrow (Ga \rightarrow Gb)$ 。这是一种“带有扭曲”的函子。以这种方式反转态射方向的范畴映射称为逆变函子。注意，逆变函子只是来自对偶范畴的常规函子。而我们迄今为止研究的常规函子，称为协变函子。



以下是 Haskell 中定义逆变函子的类型类（确实是逆变端函子）：

snippet25 我们的类型构造器 是其实例：

snippet26 注意到函数 插入在 之前（即，位于右侧）的内容——函数。

的 的定义可能更加简洁，如果你注意到它只是带有翻转参数的函数组合运算符。有一个名为 的特殊函数：

snippet27 有了它，我们得到：

snippet28

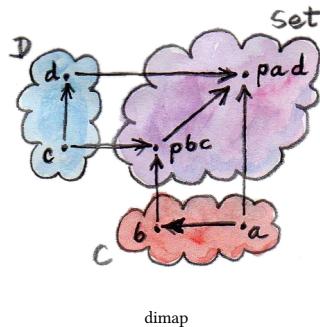
8.7 前函子 (Profunctors)

我们已经看到，函数箭头运算符在第一个参数中是逆变的，在第二个参数中是协变的。这种怪物有名字吗？事实证明，如果目标范畴是 **Set**，这种怪物被称为前函子。因为逆变函子等同于来自对偶范畴的协变函子，所以前函子定义为：

$$C^{op} \times D \rightarrow \text{Set}$$

由于 Haskell 类型在一定程度上可以看作是集合，因此我们将 名称应用于一个双参数的类型构造器，它在第一个参数中是逆变的，在第二个参数中是函子的。以下是取自 库的适当类型类：

snippet29[b] 所有三个函数都有默认实现。就像 一样，当声明 的实例时，你可以选择实现 并接受 和 的默认值，或者实现 和 并接受的默认值。



现在我们可以断言函数箭头运算符是 的一个实例：

snippet30[b] 前函子在 Haskell 的 lens 库中有其应用。我们在讨论端和余端时还会再次见到它们。

8.8 同态函子 (The Hom-Functor)

上面的例子反映了一个更一般的陈述，即将一对对象 a 和 b 映射到它们之间的态射集 $\mathbf{C}(a, b)$ 的映射是一个函子。它是从乘积范畴 $\mathbf{C}^{op} \times \mathbf{C}$ 到集合范畴 \mathbf{Set} 的一个函子。

让我们定义其对态射的作用。 $\mathbf{C}^{op} \times \mathbf{C}$ 中的一个态射是一对来自 \mathbf{C} 的态射：

$$\begin{aligned} f &:: a' \rightarrow a \\ g &:: b \rightarrow b' \end{aligned}$$

这对的提升必须是从集合 $\mathbf{C}(a, b)$ 到集合 $\mathbf{C}(a', b')$ 的态射（一个函数）。只需选择 $\mathbf{C}(a, b)$ 的任何元素 h （这是从 a 到 b 的态射），并将其映射为：

$$g \circ h \circ f$$

这是 $\mathbf{C}(a', b')$ 的一个元素。

如你所见，同态函子是前函子的一个特例。

8.9 挑战

1. 证明数据类型：

是一个双函子。为额外加分，实现 的所有三种方法，并使用等式推理证明这些定义在可以应用的情况下与默认实现兼容。

2. 证明标准定义的 与以下展开的同构性：

提示：定义两者之间的两个映射。为额外加分，使用等式推理证明它们是彼此的逆映射。

3. 让我们尝试另一个数据结构。我称之为 F ，因为它是 G 的前体。它用一个类型参数 替换了递归。

你可以通过递归应用 F 来恢复我们之前定义的 G （我们会在讨论不动点时看到它是如何完成的）。

证明 F 是 G 的一个实例。

4. 证明以下数据类型在 Haskell 和 OCaml 中定义了双函子：

为额外加分，将你的解决方案与 Conor McBride 的论文 *Clowns to the Left of me, Jokers to the Right*¹ 进行对比。

5. 用 Haskell 以外的语言定义一个双函子。在该语言中为通用对实现 \circ 。
6. 应该在两个模板参数 A 和 B 中被视为双函子还是前函子？你会如何重新设计此数据类型以使其成为这样？

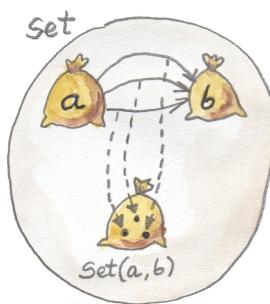
¹

9

Function Types

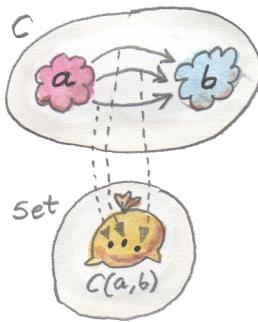
到目前为止，我一直在略过函数类型的含义。函数类型与其他类型不同。

以 $a \rightarrow b$ 为例：它只是一个整数集合。是一个包含两个元素的集合。但函数类型 $a \rightarrow b$ 不止于此：它是对象 a 和 b 之间的态射集。任何范畴中两个对象之间的态射集被称为同态集（hom-set）。恰好在范畴 Set 中，每个同态集本身就是同一范畴中的对象——因为它毕竟是一个集合。



在集合范畴（Set）中，同态集（hom-set）本质上只是一个集合

对于其他范畴来说，情况则不同，它们的同态集是外部于范畴的。这些同态集甚至被称为外部同态集。



在范畴 **C** 中，同态集是一个外部集合

正是集合范畴 **Set** 的这种自我引用特性使得函数类型如此特别。但在某些范畴中，有一种方法可以构造表示同态集的对象。这种对象被称为内部同态集。

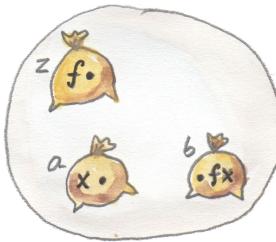
9.1 泛在构造

让我们暂时忘记函数类型是集合，尝试从头开始构造一个函数类型，或者更一般地，构造一个内部同态集。像往常一样，我们将从集合范畴 **Set** 中获得启发，但要小心避免使用任何集合的特性，这样该构造就能自动适用于其他范畴。

函数类型可以被认为是一种复合类型，因为它与参数类型和结果类型有关系。我们已经见过复合类型的构造——那些涉及对象之间关系的类型。我们使用泛在构造来定义乘积类型和余积类型。我们可以使用同样的技巧来定义函数类型。我们需要一个涉及三个对象的模式：我们正在构造的函数类型、参数类型和结果类型。

连接这三种类型的显而易见的模式被称为函数应用或求值。给定一个函数类型的候选者，我们称之为 z （注意，如果我们不在集合范畴 **Set** 中，它只是一个普通的对象），以及参数类型 a （一个对象），求值将这一对映射到结果类型 b （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）。

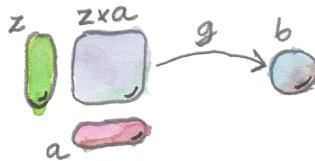
我们还具有应用关系，这是一个映射。如何将此映射纳入我们的模式？如果我们被允许查看对象的内部，我们可以将一个函数 f （ z 的一个元素）与一个参数 x （ a 的一个元素）配对，并将其映射到 fx （ f 对 x 的应用，其为 b 的一个元素）。



在集合范畴中，我们可以从函数集合 z 中选择一个函数 f ，并从集合（类型） a 中选择一个参数 x ，得到集合（类型） b 中的一个元素 fx 。

但是，与其处理单个对 (f, x) ，我们不妨讨论函数类型 z 和参数类型 a 的整个乘积。乘积 $z \times a$ 是一个对象，我们可以选择将其映射到 b 的箭头 g 作为应用态射。在集合范畴 **Set** 中， g 是将每个对 (f, x) 映射到 fx 的函数。

所以这就是模式：两个对象 z 和 a 的乘积通过一个态射 g 连接到另一个对象 b 。

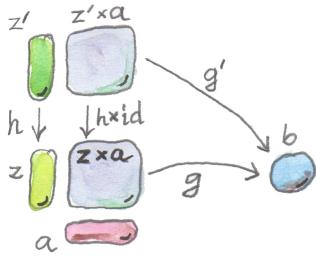


对象和态射的模式，这是泛在构造的起点

这个模式是否足够具体以通过泛在构造挑选出函数类型？并非在每个范畴中都是如此。但在我们感兴趣的范畴中确实如此。还有一个问题：是否有可能在没有首先定义乘积的情况下定义函数对象？有些范畴中没有乘积，或者某些对象对没有乘积。答案是否定的：如果没有乘积类型，就没有函数类型。稍后我们讨论指数时会回到这个问题。

让我们回顾一下泛在构造。我们从对象和态射的模式开始。这是我们不精确的查询，通常会产生大量的结果。特别是在集合范畴 **Set** 中，几乎所有东西都相互连接。我们可以取任何对象 z ，将其与 a 组成乘积，并且总会有一个从它到 b 的函数（除非 b 是一个空集）。

这时我们应用我们的秘密武器：排名。通常通过要求候选对象之间存在唯一的映射来完成排名——一个以某种方式对我们的构造进行因式分解的映射。在我们的例子中，如果 z 与从 $z \times a$ 到 b 的态射 g 比某个其他对象 z' 及其自己的应用 g' 更好，那么当且仅当存在一个从 z' 到 z 的唯一映射 h ，使得 g' 的应用通过 g 的应用进行因式分解时， z 被认为更好。（提示：在看图时阅读这句话。）



在函数对象的候选者之间建立排名

现在到了关键部分，也是我推迟讨论这个特定泛在构造的主要原因。给定从 z' 到 z 的态射 h ，我们希望关闭图表，其中 z' 和 z 都与 a 交叉。我们真正需要的是一个从 $z' \times a$ 到 $z \times a$ 的映射。现在，讨论了乘积的函子性后，我们知道如何做到这一点。由于乘积本身是一个函子（更准确地说是一个自双函子），因此可以提升态射对。换句话说，我们不仅可以定义对象的乘积，还可以定义态射的乘积。

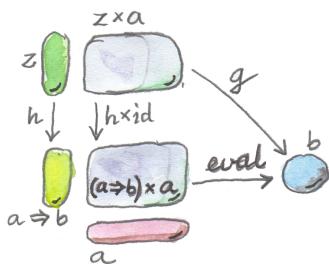
由于我们没有触及乘积 $z' \times a$ 的第二个分量，我们将提升态射对 (h, \mathbf{id}) ，其中 \mathbf{id} 是 a 上的恒等态射。

因此，这就是我们如何将一个应用 g 从另一个应用 g' 中因式分解出来的：

$$g' = g \circ (h \times \mathbf{id})$$

这里的关键是乘积对态射的作用。

泛在构造的第三部分是选择那个普遍最优的对象。我们将这个对象称为 $a \Rightarrow b$ （将其视为一个符号名称，而不是 Haskell 类型类约束——稍后我会讨论不同的命名方法）。这个对象带有它自己的应用态射——从 $(a \Rightarrow b) \times a$ 到 b 的一个态射——我们将其称为 *eval*。如果其他任何函数对象的候选者都可以唯一地映射到它，并且其应用态射 g 可以通过 *eval* 因式分解，则该对象是最佳的。这个对象比任何其他对象都好，根据我们的排名。



普遍函数对象的定义。这与上面的图是相同的，但现在对象 $a \Rightarrow b$ 是普遍的。

正式定义：

从 a 到 b 的一个函数对象是一个对象 $a \Rightarrow b$ 以及态射

$$\text{eval} :: ((a \Rightarrow b) \times a) \rightarrow b$$

使得对于任何其他对象 z 及其态射

$$g :: z \times a \rightarrow b$$

都存在唯一的态射

$$h :: z \rightarrow (a \Rightarrow b)$$

将 g 因式分解通过 eval :

$$g = \text{eval} \circ (h \times \text{id})$$

当然，不能保证在给定的范畴中，任何一对对象 a 和 b 都存在这样的对象 $a \Rightarrow b$ 。但在集合范畴 **Set** 中总是存在。而且，在 **Set** 中，这个对象同构于同态集 $\text{Set}(a, b)$ 。

这就是为什么在 Haskell 中，我们将函数类型 解释为范畴的函数对象 $a \Rightarrow b$ 。

9.2 柯里化

让我们再看看所有函数对象的候选者。不过这次，让我们将态射 g 视为一个有两个变量的函数， z 和 a 。

$$g :: z \times a \rightarrow b$$

作为一个来自乘积的态射，几乎等同于一个有两个变量的函数。特别地，在集合范畴 **Set** 中， g 是一个从对（来自集合 z 和集合 a 的值组成的对）到集合 b 的函数。

另一方面，普遍性质告诉我们，对于每个这样的 g ，都存在唯一的态射 h 将 z 映射到一个函数对象 $a \Rightarrow b$ 。

$$h :: z \rightarrow (a \Rightarrow b)$$

在集合范畴 **Set** 中，这意味着 h 是一个高阶函数，它接受一个类型为 z 的变量，并返回一个从 a 到 b 的函数。因此，普遍构造建立了两个变量函数与返回函数的一个变量函数之间的一对一对应关系。这个对应关系称为柯里化，而 h 被称为 g 的柯里化版本。

这个对应关系是一对一的，因为对于任何给定的 g 都存在唯一的 h ，并且对于任何给定的 h ，你都可以使用公式重新创建两个变量的函数 g :

$$g = \text{eval} \circ (h \times \text{id})$$

函数 g 可以称为 h 的反柯里化版本。

柯里化本质上内嵌于 Haskell 的语法中。返回一个函数的函数：

snippet01 通常被认为是一个两个变量的函数。这就是我们阅读未加括号的签名的方式：

snippet02 这种解释在我们定义多参数函数的方式中显而易见。例如：

snippet03 同样的函数可以写成一个返回函数的单参数函数——一个 lambda 表达式：

snippet04 这两个定义是等价的，并且都可以部分应用于仅一个参数，生成一个单参数函数，如：

snippet05 严格来说，两个变量的函数是一个接受一个对（一个乘积类型）的函数：

snippet06 在这两种表示之间进行转换是微不足道的，并且执行这些转换的两个高阶函数被称为 和：

snippet07 以及

snippet08 请注意，是普遍构造的因式分解器。这一点在它以这种形式重写时尤其明显：

snippet09 （作为提醒：因式分解器从候选者中生成因式分解函数。）

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对应于 Haskell 中接受元组的函数（尽管在 C++ 中你可以定义接受显式 的函数、可变参数函数以及接受初始化列表的函数，从而使情况变得更加混乱）。

你可以使用模板 部分应用一个 C++ 函数。例如，给定一个有两个字符串参数的函数：

你可以定义一个接受一个字符串参数的函数：

Scala 比 C++ 或 Java 更具函数性，处于中间位置。如果你预见到你定义的函数将被部分应用，你可以使用多个参数列表来定义它：

当然，这需要库编写者具有一定的预见性或预知能力。

9.3 指数对象

在数学文献中，函数对象或两个对象 a 和 b 之间的内部同态对象通常被称为指数对象，表示为 b^a 。注意参数类型在指数上。这个表示法乍一看可能有些奇怪，但如果你考虑到函数与乘积之间的关系，它就非常合乎逻辑。我们已经看到，在内部同态对象的泛在构造中我们必须使用乘积，但这种联系远不止于此。

这在考虑有限类型之间的函数时最为明显——有限类型是具有有限数量值的类型，如、甚至或。这样的函数，至少在原则上，可以完全备忘化或转换为可查阅的数据结构。这正是函数（态射）与函数类型（对象）之间等价性的本质。

例如，一个从到的纯函数完全由一对值（分别对应于和）指定。从到的所有可能函数的集合等同于的所有对的集合。这与乘积 \times 是一样的，或者稍微创造性地使用符号表示为 2 。

再举一个例子，让我们看看C++类型，它包含256个值（Haskell的更大，因为Haskell使用Unicode）。C++标准库的一些函数通常使用查找表实现，如或，这些函数使用的表等价于256个布尔值的元组。一个元组是一个乘积类型，所以我们正在处理256个布尔值的乘积：。我们从算术中知道，迭代的乘积定义了幂。如果你将自乘256（或）次，你就得到了的次幂，或。

在定义为256元组的类型中有多少值？恰好是 2^{256} 。这也是从到的不同函数的数量，每个函数对应于一个唯一的256元组。你可以类似地计算，从到的函数的数量是 256^2 ，等等。在这些情况下，函数类型的指数表示法非常合理。

我们可能不希望完全备忘从或到其他类型的函数。但函数与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。急切地备忘从这些类型到其他类型的函数将需要无限存储。但Haskell是一门惰性语言，所以懒惰计算的（无限）数据结构与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的函数类型与范畴指数对象的识别——这更符合我们对数据的概念。

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，有一个更大的范畴家族可以用于该目的。这些范畴被称为笛卡尔闭，而**Set**只是其中的一个例子。

一个笛卡尔闭范畴必须包含：

1. 终对象，
2. 任意一对对象的乘积，以及
3. 任意一对对象的指数对象。

如果你将指数视为迭代乘积（可能是无限多次），那么你可以将笛卡尔闭范畴视为支持任意参数乘积的范畴。特别地，终对象可以视为零个对象的乘积——或对象的零次幂。

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 演算是所有类型化编程语言的基础。

终对象和乘积有它们的对偶：初对象和余积。一个笛卡尔闭范畴如果也支持这两个，并且在其中乘积可以分布到余积上

$$\begin{aligned} a \times (b + c) &= a \times b + a \times c \\ (b + c) \times a &= b \times a + c \times a \end{aligned}$$

就被称为一个双笛卡尔闭范畴。我们将在下一节看到，双笛卡尔闭范畴（其中 **Set** 是一个主要例子）具有一些有趣的特性。

9.5 指数对象与代数数据类型

将函数类型解释为指数对象非常适合代数数据类型的方案。事实证明，所有高中代数中关于数字 0 和 1、和、乘积以及指数的基本恒等式在任何双笛卡尔闭范畴中，对于分别的初对象和终对象、余积、乘积和指数，几乎保持不变。我们还没有足够的工具来证明它们（如伴随函子或约内达引理），但我还是会在此列出它们，以作为宝贵的直觉来源。

9.5.1 零次幂

$$a^0 = 1$$

在范畴解释中，我们将 0 替换为初对象，将 1 替换为终对象，将等号替换为同构。指数对象是内部同态对象。这个特定的指数对象表示从初对象到任意对象 a 的态射集。根据初对象的定义，只有一个这样的态射，因此同态集 $C(0, a)$ 是一个单元素集。在集合范畴 **Set** 中，一个单元素集就是终对象，因此这个恒等式在 **Set** 中是显然成立的。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 `1` 替换 0，用单位类型 `Unit` 替换 1，并用函数类型替换指数对象。这个主张是，从 `Unit` 到任意类型的函数集等价于单位类型——这是一个单元素集。换句话说，只有一个函数。我们之前见过这个函数：它叫做。

这有点棘手，原因有两个。一个原因是在 Haskell 中，我们实际上没有未被占用的类型——每个类型都包含“永不终结的计算结果”或底层值。第二个原因是，所有 实现是等价的，因为无论它们做什么，都没有人可以执行它们。没有值可以传递给 。（如果你设法传递给它一个永无止境的计算，它将永远不会返回！）

9.5.2 一的幂

$$1^a = 1$$

这个恒等式在集合范畴 **Set** 中重申了终对象的定义：从任何对象到终对象都有一个唯一的态射。一般来说，从 a 到终对象的内部同态对象同构于终对象本身。

在 Haskell 中，从任何类型 到单位类型只有一个函数。我们之前见过这个函数——它叫做 。你也可以将其视为函数 部分应用于 。

9.5.3 一次幂

$$a^1 = a$$

这是对从终对象的态射可以用来选择对象 的“元素”的观察的重述。从 终对象到 的态射集与对象本身同构。在集合范畴 **Set** 和 Haskell 中，这种同构关系存在于集合 的元素与选择这些元素的函数 之间。

9.5.4 和的指数

$$a^{b+c} = a^b \times a^c$$

从范畴的角度来看，这意味着从两个对象的余积到另一个对象的指数同构于两个指数的乘积。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，从两个类型的和到另一个类型的函数等价于从每个单独类型到该类型的两个函数。这正是我们在定义关于和的函数时使用的情况分析。我们通常将一个函数定义拆分为两个（或更多）函数，分别处理每个类型构造子，而不是编写一个带有 语句的函数定义。例如，从 和类型 到另一个类型的函数：

snippet10 它可以定义为一对从 和 到目标类型的函数：

snippet11 这里， 是 类型， 是 类型。

9.5.5 指数的指数

$$(a^b)^c = a^{b \times c}$$

这是纯粹用指数对象表示柯里化的一种方式。一个返回函数的函数等价于一个从乘积（两个参数的函数）到目标类型的函数。

9.5.6 乘积的指数

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

在 Haskell 中：一个返回对的函数等价于一对函数，每个函数产生该对的一个元素。

令人惊讶的是，这些简单的高中代数恒等式可以提升到范畴理论，并在函数式编程中有实际应用。

9.6 柯里-霍华德同构

我已经提到过逻辑与代数数据类型之间的对应关系。类型 和单位类型 分别对应于假和真。乘积类型和和类型分别对应于逻辑合取 (\wedge) 和析取 (\vee)。在这个方案中，我们刚刚定义的函数类型对应于逻辑蕴涵 (\Rightarrow)。换句话说，类型 可以解读为“如果 a 则 b ”。

根据柯里-霍华德同构，每个类型都可以被解释为一个命题——一个可能为真或假的陈述或判断。这样的命题被认为是成立的，如果该类型有元素占据；否则认为是不成立的。特别地，一个逻辑蕴涵是成立的，如果对应的函数类型有元素占据，这意味着存在这样的一个函数。因此，函数的实现是一个定理的证明。编写程序等价于证明定理。让我们看看几个例子。

让我们来看一下我们在定义函数对象时引入的函数。它的签名是：snippet12 它接受一个包含一个函数及其参数的对，并生成相应类型的结果。这是 Haskell 实现的态射：

$$\text{eval} :: (a \Rightarrow b) \times a \rightarrow b$$

该态射定义了函数类型 $a \Rightarrow b$ （或指数对象 b^a ）。让我们使用柯里-霍华德同构将该签名翻译为一个逻辑谓词：

$$((a \Rightarrow b) \wedge a) \Rightarrow b$$

你可以这样解读这个命题：如果 b 由 a 推出，并且 a 是真的，那么 b 必须是真的。这非常直观并且自古以来就为人所知，称为演绎法则 (modus ponens)。我们可以通过实现函数来证明这一定理：

snippet13 如果你给我一个包含一个函数（它接受 并返回）和一个类型为 的具体值 的对，我可以简单地将函数 应用于 来生成一个类型为 的具体值。通过实现这个函数，我刚刚展示了类型 是被占据的。因此，在我们的逻辑中，演绎法则是成立的。

那么，一个明显错误的命题呢？例如：如果 a 或 b 是真的，那么 a 必须是真的。

$$a \vee b \Rightarrow a$$

这显然是错误的，因为你可以选择一个为假的 a 和一个为真的 b ，这就是一个反例。

使用柯里-霍华德同构将这个命题映射到一个函数签名中，我们得到：

snippet14 无论你怎么尝试，你都无法实现这个函数——如果你被传递了 值，你就无法生成一个类型为 的值。(请记住，我们谈论的是纯函数。)

最后，我们来看看 函数的含义：

snippet15 考虑到 转换为假，我们得到：

$$\textit{false} \Rightarrow a$$

任何东西都可以从谬误中推导出来 (*ex falso quodlibet*)。以下是这个命题（函数）在 Haskell 中的一个可能的证明（实现）：

其中 定义为：

如往常一样，类型是棘手的。这个定义使得它无法构造出一个值，因为为了构造一个值，你需要先提供一个值。因此，函数 永远无法被调用。

这些都是有趣的例子，但柯里-霍华德同构在实际编程中是否有实用的一面？可能在日常编程中并不多见。但确实有一些编程语言如 Agda 或 Coq，利用柯里-霍华德同构来证明定理。

计算机不仅帮助数学家完成他们的工作——它们正在彻底改变数学的基础。该领域最新的热门研究主题称为同伦类型论，这是类型论的一个分支。它充满了布尔值、整数、乘积和余积、函数类型等等。而且，为了消除任何怀疑，该理论正在 Coq 和 Agda 中被公式化。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, Reason Isomorphically!¹. 这篇论文包含了我在本章中提到的所有那些高中代数恒等式在范畴论中的证明。

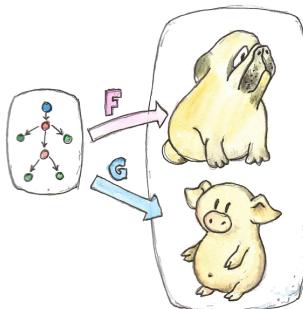
¹

10

Natural Transformations

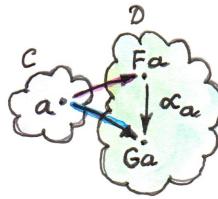
我们讨论了函子（functor）作为在范畴之间的映射，它们保留了范畴的结构。

一个函子将一个范畴“嵌入”到另一个范畴中。它可能会将多个事物折叠成一个，但它从不破坏连接。我们可以认为，使用函子时，我们是在另一个范畴内建模一个范畴。源范畴作为某种结构的模型，一个蓝图，而这个结构是目标范畴的一部分。



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，有时则非常不同。一个可能会将整个源范畴折叠成一个对象，而另一个可能将每个对象映射到不同的对象，每个态射（morphism）映射到不同的态射。同一个蓝图可以以多种不同的方式实现。自然变换（natural transformations）帮助我们比较这些实现。它们是函子的映射——一种特殊的映射，保留了函子的本质特性。

考虑在范畴 **C** 和 **D** 之间的两个函子 F 和 G 。如果你只关注 **C** 中的一个对象 a ，它将被映射为两个对象： Fa 和 Ga 。一个函子的映射应该将 Fa 映射为 Ga 。



注意到 Fa 和 Ga 是 \mathbf{D} 范畴中的对象。范畴中对象之间的映射不应违反范畴的规则。我们不想在对象之间制造人为的连接，因此使用现有的连接，即态射，是“自然的”。自然变换是态射的选择：对于每个对象 a ，它选择一个从 Fa 到 Ga 的态射。如果我们称自然变换为 α ，那么这个态射就是 α 在 a 处的分量（component），即 α_a 。

$$\alpha_a :: Fa \rightarrow Ga$$

请记住， a 是 \mathbf{C} 中的一个对象，而 α_a 是 \mathbf{D} 中的一个态射。

如果在某些 a 上， Fa 和 Ga 之间在 \mathbf{D} 中没有态射，那么 F 和 G 之间就没有自然变换。

当然，这只是故事的一半，因为函子不仅映射对象，它们也映射态射。那么，自然变换对这些映射做了什么？事实证明，在 F 和 G 之间的自然变换下， Ff 必须被转化为 Gf 。此外，两个函子对态射的映射极大地限制了定义与之兼容的自然变换的选择。考虑在 \mathbf{C} 中两个对象 a 和 b 之间的态射 f 。它被映射为在 \mathbf{D} 中的两个态射： Ff 和 Gf ：

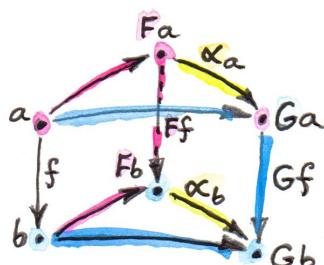
$$Ff :: Fa \rightarrow Fb$$

$$Gf :: Ga \rightarrow Gb$$

自然变换 α 提供了两个额外的态射，它们完成了 \mathbf{D} 中的图：

$$\alpha_a :: Fa \rightarrow Ga$$

$$\alpha_b :: Fb \rightarrow Gb$$

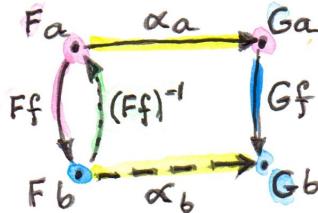


现在我们有两种从 Fa 到 Gb 的路径。为了确保它们相等，我们必须施加一个对于任何 f 都成立的自然性条件：

$$Gf \circ \alpha_a = \alpha_b \circ Ff$$

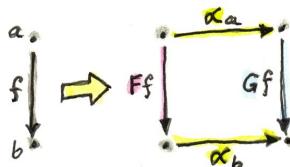
自然性条件是一个相当严格的要求。例如，如果态射 Ff 是可逆的，那么自然性就决定了 α_b 依赖于 α_a 。它传送了 α_a 沿着 f 的变化：

$$\alpha_b = (Gf) \circ \alpha_a \circ (Ff)^{-1}$$



如果在两个对象之间多于一个可逆的态射，那么所有这些传递都必须一致。然而，一般情况下，态射是不可逆的；但你可以看到，两函子之间存在自然变换的可能性远非保证。因此，函子之间由自然变换连接的稀缺性或丰富性可以告诉你关于它们所操作的范畴结构的许多信息。在讨论极限（limits）和 Yoneda 引理时，我们将看到一些例子。

从分量的角度来看待自然变换，或许可以说它将对象映射为态射。由于自然性条件的存在，你也可以说它将态射映射为交换方块——在 D 中每个 C 中的态射都有一个交换的自然性方块。



自然变换的这种特性在许多范畴论的构造中非常有用，这些构造通常包含交换图。通过慎重选择函子，许多这些交换条件可以被转化为自然性条件。当我们涉及极限、余极限（colimits）和伴随（adjunctions）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的几乎等同于说它们是相同的。自然同构被定义为所有分量都是同构（可逆态射）的自然变换。

10.1 多态函数

我们讨论了函子（更具体地说是自函子）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，这种映射是通过高阶函数（或在 C++ 中的 `f`、`g` 等）来实现的。

要构造一个自然变换，我们从一个对象开始，这里是一个类型。一个函子 将它映射到类型 Fa 。另一个函子 将它映射到 Ga 。自然变换 在处的分量是一个从 Fa 到 Ga 的函数。在伪 Haskell 中：

a

自然变换是一个对所有类型 都定义的多态函数：

snippet01 在 Haskell 中，是可选的（实际上需要开启语言扩展）。通常，你会这样写：

snippet02 请记住，它实际上是一个以 `a` 为参数的函数族。这是 Haskell 语法简洁性的另一个例子。在 C++ 中，一个类似的结构会稍微冗长一些：

在 Haskell 的多态函数和 C++ 的泛型函数之间有一个更深刻的区别，这反映在这些函数的实现和类型检查方式上。在 Haskell 中，一个多态函数必须统一地为所有类型定义。一个公式必须适用于所有类型。这被称为 参数多态性。

而在 C++ 中，默认支持的是 特定多态性，这意味着模板不必对所有类型都是良定义的。模板是否对给定类型起作用是在实例化时决定的，在此过程中，具体类型被替换为类型参数。类型检查被延迟，这不幸的是，通常导致难以理解的错误消息。

在 C++ 中，还有一个函数重载和模板特化的机制，允许为不同的类型定义相同函数的不同定义。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后果：任何类型为

snippet03 的多态函数，其中 G 和 F 是函子，自动满足自然性条件。这里是范畴符号中的表示 (f 是一个函数 $f :: a \rightarrow b$)：

$$Gf \circ \alpha_a = \alpha_b \circ Ff$$

在 Haskell 中，函子 对态射 的作用是通过 实现的。我首先会在伪 Haskell 中写出来，带有明确的类型注释：

`G`
`a`
`b`
`F`

由于类型推断，这些注释是不必要的，下面的等式成立：

这仍然不是真正的 Haskell——函数相等性在代码中是无法表达的——但它是程序员在方程推理中可以使用的等式；或者由编译器使用来实现优化。

自然性条件在 Haskell 中是自动满足的，原因在于“免费的定理”。参数多态性，在 Haskell 中用于定义自然变换，对实现施加了非常强的限制——所有类型只有一个公式。这些限制转换为关于此类函数的方程定理。在变换函子的函数的情况下，免费的定理就是自然性条件。¹

我之前提到过，将 Haskell 中的函子看作是广义容器的一种方法。我们可以继续这个类比，将自然变换看作是将一个容器的内容重新打包到另一个容器的配方。我们不触碰项目本身：我们不修改它们，也不创建新的项目。我们只是将它们（有时多次）复制到一个新容器中。

自然性条件变成了这样一种陈述：无论我们是先通过应用 修改项目，后重新打包；还是先重新打包，然后在新容器中修改项目，通过其自己的 实现，这两种行为是正交的。“一个移动鸡蛋，另一个煮它们。”

让我们来看一些 Haskell 中自然变换的例子。第一个例子是列表函子和 函子之间的自然变换。它返回列表的头部，但前提是列表非空：

snippet04 这是一个在 上多态的函数。它对任何类型 都有效，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。但为了让我们信服，让我们验证自然性条件。

snippet05 我们有两种情况需要考虑：一个空列表：

snippet06

snippet07 和一个非空列表：

snippet08

snippet09 我使用了列表的 实现：

snippet10 和 的：

snippet11 一个有趣的情况是，当其中一个函子是平凡的 函子时。从或到 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上多态的函数。

例如，可以被视为从列表函子到 函子的自然变换：

snippet12 这里使用 来剥离 构造器：

snippet13 当然，实际上 是这样定义的：

snippet14 它有效地隐藏了它是自然变换的事实。

找到一个从 函子 来的参数多态函数有点困难，因为它需要从无到有地创建一个值。我们能做的最好的是：

snippet15 另一个我们已经见过的常见函子，并将在 Yoneda 引理中发挥重要作用的，是 函子。我将重新将其定义为：

snippet16 它被两个类型参数化，但只在第二个类型上是（协变）函子化的：

¹你可以在我的博客 “Parametricity: Money for Nothing and Theorems for Free.” 中阅读更多关于免费的定理的内容。

snippet17 对于每个类型，你可以定义从 到任何其他函子 的自然变换族。稍后我们会看到，这个族的成员总是与 的元素一一对应 (Yoneda 引理)。

例如，考虑有一个带有一个元素 的稍显平凡的单位类型。函子 将任何类型 映射为函数类型。这些只是从集合 中选择单个元素的所有函数。它们与 中的元素一样多。现在我们来考虑从这个函子到 函子 的自然变换：

snippet18 只有两个： 和：

snippet19 和

snippet20 (你唯一能对 做的就是将它应用到单位值 上。)

而且，正如 Yoneda 引理所预测的那样，它们对应于 类型的两个元素，即 和 。我们稍后会回到 Yoneda 引理——这只是一个小小的预告。

10.2 超越自然性

在两个函子之间的参数多态函数（包括 函子的边界情况）总是一个自然变换。由于所有标准的代数数据类型都是函子，任何这类类型之间的多态函数都是自然变换。

我们还可以使用函数类型，而函数类型在其返回类型上是函子化的。我们可以使用它们来构建函子，并定义高阶函数形式的自然变换。

然而，函数类型在其参数类型上不是协变的。它们是逆变的。当然，逆变函子等价于从对偶范畴到 Haskell 类型的协变函子。两个逆变函子之间的多态函数在范畴意义上仍然是自然变换，只是它们在 Haskell 类型的对偶范畴中工作。

你可能还记得我们之前看过的一个逆变函子的例子：

snippet21 这个函子在 上是逆变的：

snippet22 我们可以编写一个从 到 的多态函数：

snippet23 但由于这两个函子不是协变的，这不是一个 Hask 中的自然变换。然而，由于它们都是逆变的，它们满足“相反”的自然性条件：

snippet24[b] 注意到函数 必须朝与使用 相反的方向，因为 的签名是：

snippet25 是否有不是函子的类型构造器，无论是协变的还是逆变的？这里有一个例子：

snippet26 这不是一个函子，因为相同的类型 同时用于负面 (逆变) 和正面 (协变) 的位置。你不能为这个类型实现 或 。因此，签名为

snippet27 的函数，其中 是一个任意的函子，不能是自然变换。有趣的是，有一种自然变换的推广，称为双自然变换 (dinatural transformations)，可以处理这些情况。当我们讨论端 (ends) 时，我们会涉及到它们。

10.3 函子范畴

现在我们有了函子之间的映射——自然变换——自然会问函子是否构成了一个范畴。确实如此！对于每一对范畴 **C** 和 **D**，都有一个函子的范畴。在这个范畴中的对象是从 **C** 到 **D** 的函子，态射是这些函子之间的自然变换。

我们必须定义两个自然变换的复合，但这相当容易。自然变换的分量是态射，我们知道如何复合态射。

确实，让我们考虑一个从函子 F 到 G 的自然变换 α 。它在对象 a 处的分量是某个态射：

$$\alpha_a :: Fa \rightarrow Ga$$

我们想将 α 与 β 复合， β 是从函子 G 到 H 的自然变换。 β 在 a 处的分量是一个态射：

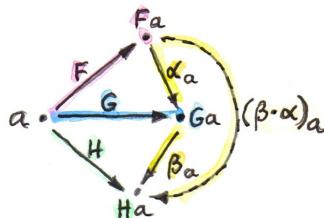
$$\beta_a :: Ga \rightarrow Ha$$

这些态射是可复合的，它们的复合是另一个态射：

$$\beta_a \circ \alpha_a :: Fa \rightarrow Ha$$

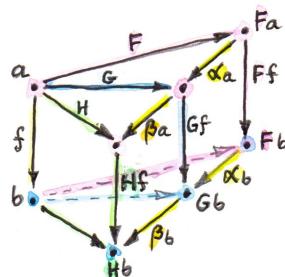
我们将使用这个态射作为自然变换 $\beta \cdot \alpha$ 的分量——两个自然变换 β 在 α 后的复合：

$$(\beta \cdot \alpha)_a = \beta_a \circ \alpha_a$$



通过观察图，我们可以确定这个复合的结果确实是从 F 到 H 的自然变换：

$$Hf \circ (\beta \cdot \alpha)_a = (\beta \cdot \alpha)_b \circ Ff$$



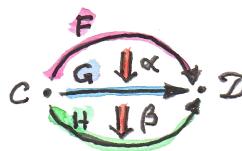
自然变换的复合是结合的，因为它们的分量是关于态射复合的结合态射。

最后，对于每个函子 F ，都有一个恒等自然变换 1_F ，其分量是恒等态射：

$$\mathbf{id}_{Fa} :: Fa \rightarrow Fa$$

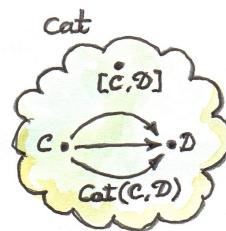
因此，函子确实构成了一个范畴。

关于符号的说明。遵循 Saunders Mac Lane 的做法，我使用圆点来表示我刚才描述的自然变换的复合。问题是自然变换有两种复合方式。这种被称为纵向复合，因为函子通常在描述它们的图中垂直堆叠。纵向复合在定义函子范畴时很重要。我会很快解释横向复合。



范畴 C 和 D 之间的函子范畴被写作 $\mathbf{Fun}(C, D)$ ，或 $[C, D]$ ，或者有时写作 D^C 。最后一种符号表明函子范畴本身可能被视为某个其他范畴中的函数对象（指数）。这确实是这样吗？

让我们看看我们迄今为止构建的抽象层次。我们从范畴开始，这是对象和态射的集合。范畴本身（或严格来说，小范畴，其对象构成集合）本身就是更高级范畴 \mathbf{Cat} 中的对象。该范畴中的态射是函子。 \mathbf{Cat} 中的一个同态集是一个函子的集合。例如， $\mathbf{Cat}(C, D)$ 是两个范畴 C 和 D 之间的函子的集合。



函子范畴 $[C, D]$ 也是两个范畴之间的函子的集合（加上自然变换作为态射）。它的对象与 $\mathbf{Cat}(C, D)$ 的成员相同。此外，函子范畴作为一个范畴，必须本身是 \mathbf{Cat} 的对象（碰巧两个小范畴之间的函子范畴本身也是小的）。我们在一个范畴中的同态集与同一个范畴中的一个对象之间有关系。情况正如我们在上一节中看到的指数对象。让我们看看如何在 \mathbf{Cat} 中构造后者。

你可能还记得，为了构造一个指数，我们需要首先定义一个积。在 \mathbf{Cat} 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，积范畴 $\mathbf{C} \times \mathbf{D}$ 中的一个对象只是一对对象 (c, d) ，一个来自 \mathbf{C} ，一个来自 \mathbf{D} 。类似地，两对 (c, d) 和 (c', d') 之间的态射是一对态射 (f, g) ，其中 $f :: c \rightarrow c'$ 和 $g :: d \rightarrow d'$ 。这些态射对分量地复合，并且总有一个恒等对，它只是一对恒等态射。简而言之，**Cat** 是一个完全的笛卡尔闭范畴，其中对于任意一对范畴都有一个指数对象 $\mathbf{D}^{\mathbf{C}}$ 。通过“对象”在 **Cat** 中，我的意思是一个范畴，所以 $\mathbf{D}^{\mathbf{C}}$ 是一个范畴，我们可以将其识别为 \mathbf{C} 和 \mathbf{D} 之间的函子范畴。

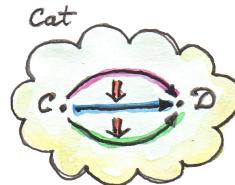
10.4 2-范畴

现在我们已经了解了 **Cat**，让我们更详细地看看它。根据定义，**Cat** 中的任何同态集都是函子的集合。但是，正如我们所见，两个对象之间的函子具有比单纯的集合更丰富的结构。它们形成一个范畴，自然变换作为态射。由于函子在 **Cat** 中被认为是态射，自然变换就是态射之间的态射。

这种更丰富的结构是一个 2-范畴的例子，它是范畴的推广，除了对象和态射（在此上下文中可能被称为 1-态射）之外，还有 2-态射，这是态射之间的态射。

在 **Cat** 作为一个 2-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。



代替两个范畴 \mathbf{C} 和 \mathbf{D} 之间的同态集，我们有一个同态范畴——函子范畴 $\mathbf{D}^{\mathbf{C}}$ 。我们有常规的函子复合：函子 F 从 $\mathbf{D}^{\mathbf{C}}$ 复合到一个函子 G 从 $\mathbf{E}^{\mathbf{D}}$ 给出 $G \circ F$ 从 $\mathbf{E}^{\mathbf{C}}$ 。但我们也有每个同态范畴内部的复合——函子之间自然变换的纵向复合，或 2-态射。

在一个 2-范畴中有两种复合方式，问题是：它们如何相互作用？

让我们选择 **Cat** 中的两个函子，或 1-态射：

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

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

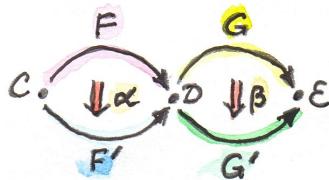
及其复合：

$$G \circ F :: \mathbf{C} \rightarrow \mathbf{E}$$

假设我们有两个自然变换 α 和 β , 它们分别作用于函子 F 和 G :

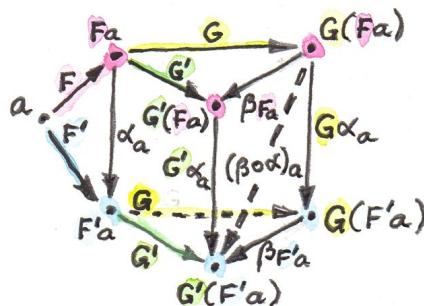
$$\alpha :: F \rightarrow F'$$

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



注意到我们不能将它们进行纵向复合, 因为 α 的目标不同于 β 的源。事实上, 它们是两个不同函子范畴的成员: $\mathbf{D}^{\mathbf{C}}$ 和 $\mathbf{E}^{\mathbf{D}}$ 。然而, 我们可以对函子 F' 和 G' 进行复合, 因为 F' 的目标是 G' 的源——它是范畴 \mathbf{D} 。 $G' \circ F'$ 和 $G \circ F$ 之间有什么关系?

有了 α 和 β 我们能定义一个从 $G \circ F$ 到 $G' \circ F'$ 的自然变换吗? 让我简略描述一下构造。



通常, 我们从 \mathbf{C} 中的一个对象 a 开始。它的图像在 \mathbf{D} 中分裂为两个对象: Fa 和 $F'a$ 。还有一个态射, 自然变换 α 的分量, 连接这两个对象:

$$\alpha_a :: Fa \rightarrow F'a$$

当从 \mathbf{D} 到 \mathbf{E} 时, 这两个对象进一步分裂为四个对象: $G(Fa)$ 、 $G'(Fa)$ 、 $G(F'a)$ 、 $G'(F'a)$ 。我们还有四个态射构成一个方块。其中两个态射是自然变换 β 的分量:

$$\beta_{Fa} :: G(Fa) \rightarrow G'(Fa)$$

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

另外两个是两个函子的 α_a 的映像 (函子映射态射):

$$G\alpha_a :: G(Fa) \rightarrow G(F'a)$$

$$G'\alpha_a :: G'(Fa) \rightarrow G'(F'a)$$

这些是很多态射。我们的目标是找到一个从 $G(Fa)$ 到 $G'(F'a)$ 的态射，这可以作为连接两个函子 $G \circ F$ 和 $G' \circ F'$ 的自然变换的分量。事实上，这里有两条路径可以从 $G(Fa)$ 到 $G'(F'a)$ ：

$$\begin{aligned} G'\alpha_a \circ \beta_{Fa} \\ \beta_{F'a} \circ G\alpha_a \end{aligned}$$

幸运的是，它们是相等的，因为我们构造的方块是 β 的自然性方块。

我们刚刚定义了一个从 $G \circ F$ 到 $G' \circ F'$ 的自然变换的分量。证明这个变换的自然性相当直接，只要你有足够的耐心。

我们称这个自然变换为 横向复合的 α 和 β ：

$$\beta \circ \alpha :: G \circ F \rightarrow G' \circ F'$$

同样，遵循 Mac Lane 的做法，我使用小圆圈表示横向复合，尽管你可能还会遇到用星号表示的。

这里有一个范畴论的经验法则：每当你有复合时，你应该寻找一个范畴。我们有自然变换的纵向复合，它是函子范畴的一部分。但横向复合呢？它属于哪个范畴？

解决这个问题的方法是横向看待 **Cat**。不要将自然变换视为函子之间的箭头，而是作为范畴之间的箭头。自然变换位于两个范畴之间，它们由它所变换的函子连接。我们可以认为它连接了这两个范畴。



让我们关注 **Cat** 的两个对象——范畴 **C** 和 **D**。存在一组自然变换，它们在连接 **C** 和 **D** 的函子之间。我们可以将这些自然变换视为从 **C** 到 **D** 的新箭头。同样地，也有自然变换连接从 **D** 到 **E** 的函子，我们可以将它们视为从 **D** 到 **E** 的新箭头。横向复合就是这些箭头的复合。

我们还可以有从 **C** 到 **C** 的恒等箭头。它是将 **C** 上的恒等函子映射到自身的恒等自然变换。注意到横向复合的恒等是纵向复合的恒等，但反之则不然。

最后，这两个复合满足交换律：

$$(\beta' \cdot \alpha') \circ (\beta \cdot \alpha) = (\beta' \circ \beta) \cdot (\alpha' \circ \alpha)$$

我会引用 Saunders Mac Lane 的话：读者可能会喜欢写出证明这个事实所需的明显图。

还有一个符号在将来可能会派上用场。在 **Cat** 的这个新横向解释中，有两种方式从对象到对象：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类型的自然变换：作用在这个函子上的恒等自然变换。因此，你经常会看到这种符号：

$$F \circ \alpha$$

其中 F 是从 **D** 到 **E** 的函子， α 是从 **C** 到 **D** 的两个函子之间的自然变换。由于你不能将一个函子与自然变换复合，这被解释为在 α 后的恒等自然变换 1_F 的横向复合。

类似地：

$$\alpha \circ F$$

是 α 后 1_F 的横向复合。

10.5 结论

这就结束了本书的第一部分。我们学习了范畴论的基本词汇。你可以将对象和范畴视为名词；将态射、函子和自然变换视为动词。态射连接对象，函子连接范畴，自然变换连接函子。

但我们也看到了，表面上在一个抽象层次上的行为，在下一个层次上变成了对象。同态集变成了函数对象。作为一个对象，它可以是另一个态射的源或目标。这就是高阶函数背后的思想。

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——。有一些简单的函子，例如、积和余积，它们可以用来生成各种代数数据类型。函数类型也是函子化的，既协变又逆变，可以用来扩展代数数据类型。

函子可以被视为函子范畴中的对象。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. 定义一个从 函子到列表函子的自然变换。证明其自然性条件。
2. 定义至少两个不同的自然变换在 和列表函子之间。有多少个不同的 列表？
3. 继续前面的练习，在 和 之间进行。
4. 证明自然变换的横向复合满足自然性条件（提示：使用分量）。这是一个很好的追图练习。
5. 写一篇关于你如何喜欢写出证明交换律所需的明显图的简短文章。
6. 为不同 函子之间的变换创建一些测试用例，验证对立的自然性条件。这里有一个选择：

和

Part Two

11

Declarative Programming

在本书的第一部分中，我提出了一个观点，即范畴论和编程都涉及组合性。在编程中，你不断地将问题分解，直到达到你可以处理的细节层次，依次解决每个子问题，并自下而上地重新组合解决方案。大致来说，有两种方式来做这件事：告诉计算机做什么，或者告诉它如何做。前者称为声明式编程，后者称为命令式编程。

你可以在最基本的层面上看到这种差异。组合本身可以以声明式的方式定义；例如，是在之后的组合：

snippet01 或者以命令式的方式；例如，先调用，记住调用的结果，然后用这个结果调用：

snippet02 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，必须在执行完成后，才能调用。至少，这是概念图——在一个懒惰语言中，使用按需调用参数传递，实际的执行可能有所不同。

事实上，取决于编译器的巧妙程度，声明式和命令式代码在执行上可能几乎没有差异。但是这两种方法在我们解决问题的方式以及由此产生的代码的可维护性和可测试性上存在很大差异。

主要问题是：当面临一个问题时，我们是否总是可以选择以声明式或命令式的方法来解决它？而且，如果有声明式的解决方案，它是否总是可以转化为计算机代码？这个问题的答案远非显而易见，如果我们能够找到它，我们可能会彻底改变我们对宇宙的理解。

让我详细说明。在物理学中也存在类似的二元性，这要么指向某些深层的基本原理，要么告诉我们一些关于我们思维方式的东西。Richard Feynman 提到了这种二元性，作为他在量子电动力学研究中的灵感来源。

表达大多数物理定律有两种形式。一种是局部的，或者说是微观的考虑。我们观察系统在一个小范围内的状态，并预测它将在下一瞬间如

何演化。这通常使用微分方程来表达，这些方程必须在一段时间内被积分，或者说被求和。

注意这种方法如何类似于命令式思维：我们通过一系列小步骤到达最终的解决方案，每一步都依赖于前一步的结果。事实上，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。这就是小行星游戏中飞船动画的实现方式。在每个时间步，飞船的位置通过添加一个小增量来改变，这个增量通过将其速度乘以时间差得到。速度反过来通过与力成比例的小增量来改变，力除以质量得到加速度。

这些是对应于牛顿运动定律的微分方程的直接编码：

$$F = m \frac{dv}{dt}$$
$$v = \frac{dx}{dt}$$



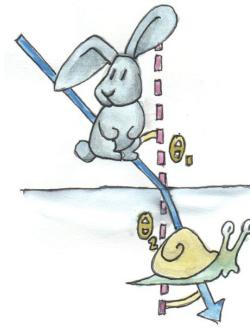
类似的方法可以应用于更复杂的问题，例如使用 Maxwell 方程组来模拟电磁场的传播，甚至使用格点量子色动力学（QCD）来模拟质子内部夸克和胶子行为。

这种局部思维与空间和时间的离散化相结合，这种离散化在使用数字计算机时被鼓励，其极端表达是在 Stephen Wolfram 通过细胞自动机系统来简化整个宇宙的复杂性的英雄尝试中表现出来。

另一种方法是全局的。我们观察系统的初始状态和最终状态，并通过最小化某种泛函来计算连接它们的轨迹。最简单的例子是费马的最短时间原理。它指出，光线沿着最小化其飞行时间的路径传播。特别地，在没有反射或折射物体的情况下，从点 A 到点 B 的光线将采取最短路径，即直线。但光在密度较大的（透明）材料中传播得更慢，例如水或玻璃。所以，如果你选择在空气中的起点和水下的终点，光在空气中行进得更长，然后通过水走捷径更为有利。最短时间的路径使光线在空气和水的边界处发生折射，从而产生了斯涅尔的折射定律：

$$\frac{\sin(\theta_1)}{\sin(\theta_2)} = \frac{v_1}{v_2}$$

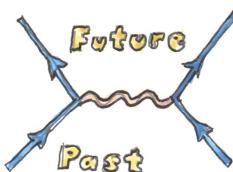
其中 v_1 是光在空气中的速度， v_2 是光在水中的速度。



所有经典力学都可以从最小作用量原理中推导出来。作用量可以通过积分拉格朗日量来计算，它是动能和势能的差值（注意：这是差值，而不是和——和将是总能量）。当你发射迫击炮来击中给定目标时，炮弹首先会上升到势能较高的地方，并在那里花费一些时间以积累负的作用量贡献。它还会在抛物线的顶端减速，以最小化动能。然后它将加速，通过低势能区域快速通过。



Feynman 的最大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，同样的问题是用初始状态和最终状态的形式提出的。Feynman 路径积分用于计算这些状态之间的转变概率。



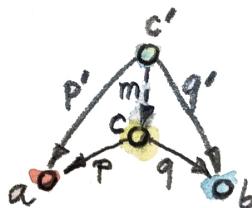
关键在于，在我们描述物理定律的方式中存在一种奇怪且未解释的二元性。我们可以使用局部图景，其中事情按顺序发生，并以小增量进行。或者我们可以使用全局图景，我们声明初始和最终条件，然后所有的中间过程都随之而来。

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。我们并不明确地最小化每条光线的飞行时间，但我们确实使用了斯涅尔定律和反射几何来达到同样的效果。

局部方法和全局方法之间的最大区别在于它们对空间的处理，尤其是时间的处理。局部方法拥抱了这里和现在的即时满足，而全局方法则采取了长期的静态视角，仿佛未来已经预定，我们只是在分析某个永恒宇宙的性质。

没有什么比在函数响应式编程（FRP）方法中更好地说明了这一点。与其为每个可能的用户操作编写单独的处理程序，并让它们都访问某些共享的可变状态，不如将外部事件视为一个无限列表，并对其应用一系列转换。从程序的角度来看，我们所有未来行为的列表就在那里，作为输入数据供我们的程序使用。对程序来说， π 的数字列表、伪随机数列表或通过计算机硬件传入的鼠标位置列表之间没有区别。在每种情况下，如果你想获得第 n 项，你必须先通过前 $n - 1$ 项。当应用于时间事件时，我们称这种性质为因果性。

那么，这与范畴论有什么关系？我将论证范畴论鼓励全局方法，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。我们拥有的只是抽象的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 A 到达 B ，你也可以一步到达。此外，范畴论的主要工具是范畴的普遍构造，这正是全局方法的典范。我们已经看到它在行动，例如，在范畴乘积的定义中。它是通过指定其性质来完成的——一种非常声明式的方法。它是一个带有两个投影的对象，并且是最优的此类对象——它优化了某种性质：其他此类对象的投影的因式分解。



将此与费马的最短时间原理或最小作用量原理进行比较。

反过来，将其与笛卡尔乘积的传统定义进行对比，这种定义更加命令式。你通过从一个集合中选择一个元素并从另一个集合中选择另一个元素来描述如何创建乘积的元素。这是创建一个对偶的配方。还有另一个用于拆解对偶的配方。

在几乎每一种编程语言中，包括 Haskell 这样的函数式语言，乘积类型、余积类型和函数类型都是内置的，而不是通过普遍构造来定义的；尽管已经有尝试创建范畴编程语言（参见例如，Tatsuya Hagino 的论文¹）。

无论直接使用与否，范畴定义证明了预先存在的编程结构，并产生了新的结构。最重要的是，范畴论提供了一种元语言，用于在声明式层

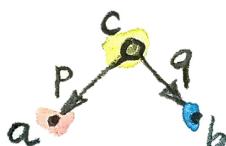
1

次上推理计算机程序。它还鼓励在将问题转换为代码之前进行问题规范化。

12

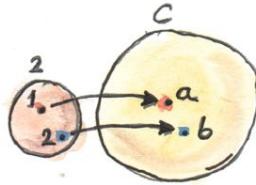
Limits and Colimits

看 起来在范畴论中，一切都与一切相关，所有事物都可以从多角度进行观察。以积的普遍构造为例。现在我们对函子和自然变换有了更多了解，我们能简化并可能推广它吗？让我们试试看。

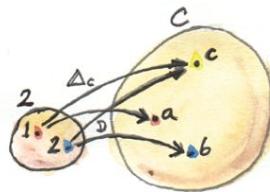


积的构造从选择两个对象 a 和 b 开始，这两个对象是我们希望构造其积的对象。但是，选择对象意味着什么？我们能否用更范畴化的术语来重新表达这一操作？两个对象形成一个模式——一个非常简单的模式。我们可以将这个模式抽象为一个范畴——一个非常简单的范畴，但仍然是一个范畴。我们称之为 $\mathbf{2}$ 。它只包含两个对象， 1 和 2 ，没有其他态射，只有两个必需的恒等态射。现在，我们可以将选择 \mathbf{C} 中的两个对象重新表述为从范畴 $\mathbf{2}$ 到 \mathbf{C} 的函子 D 的定义。一个函子将对象映射到对象，因此它的像只是两个对象（或者如果函子合并对象，这也是可以的）。它还映射态射——在这种情况下，它只是将恒等态射映射到恒等态射。

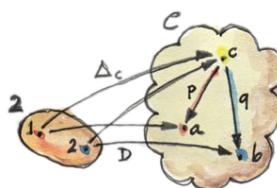
这种方法的伟大之处在于，它建立在范畴概念之上，避开了像“选择对象”这样从我们祖先的猎人-采集者词汇中直接借用的模糊描述。而且，顺便说一句，它也很容易推广，因为没有什么能阻止我们使用比 $\mathbf{2}$ 更复杂的范畴来定义我们的模式。



但让我们继续。积定义的下一步是选择候选对象 c 。在这里，我们可以再次用单一对象范畴的函子来重新表述选择的操作。实际上，如果我们在使用 Kan 扩展，那将是正确的做法。但由于我们还没有准备好 Kan 扩展，我们可以使用另一种技巧：从相同的范畴 2 到 C 的常值函子 Δ 。在 C 中选择 c 可以通过 Δ_c 来完成。记住， Δ_c 将所有对象映射到 c ，并将所有态射映射到 id_c 。

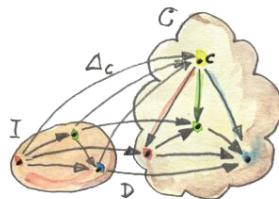


现在我们有了两个函子， Δ_c 和 D ，它们都在 2 和 C 之间，所以自然要问它们之间是否存在自然变换。由于 2 中只有两个对象，自然变换将有两个分量。 2 中的对象 1 由 Δ_c 映射到 c ，由 D 映射到 a 。因此， Δ_c 和 D 之间的自然变换在 1 处的分量是一个从 c 到 a 的态射。我们可以称之为 p 。同样，第二个分量是从 c 到 b 的态射——即 2 中对象 2 在 D 下的像。但这些与我们在最初的积定义中使用的两个投影完全相同。所以与其谈论选择对象和投影，我们可以只谈论选择函子和自然变换。在这个简单的情况下，自然变换的自然性条件是平凡的，因为在 2 中没有态射（除恒等态射外）。

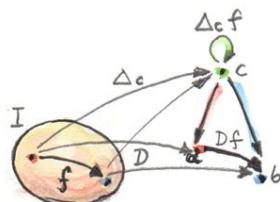


这种构造的一般化可以扩展到其他比 2 更复杂的范畴——例如，包含非平凡态射的范畴。这将对 Δ_c 和 D 之间的变换施加自然性条件。我们称这种变换为 锥，因为 Δ 的像是锥/金字塔的顶点，其侧面由自然变换的分量构成。 D 的像构成了锥的底部。

一般来说，构建一个锥，我们从一个定义模式的范畴 \mathbf{I} 开始。它是一个小的，通常是有限的范畴。我们选择一个从 \mathbf{I} 到 \mathbf{C} 的函子 D ，并称其为（或其像）一个 图。我们在 \mathbf{C} 中选择某个 c 作为我们锥的顶点。我们用它来定义从 \mathbf{I} 到 \mathbf{C} 的常值函子 Δ_c 。从 Δ_c 到 D 的自然变换就是我们的锥。对于有限的 \mathbf{I} ，它只是连接 c 和图 (\mathbf{I} 在 D 下的像) 的态射的集合。



自然性要求该图中的所有三角形（即金字塔的墙面）都交换。确实，取 \mathbf{I} 中的任意态射 f 。函子 D 将其映射到 \mathbf{C} 中的态射 Df ，这条态射构成了某个三角形的底边。常值函子 Δ_c 将 f 映射到 c 上的恒等态射。 Δ 将态射的两个端点压缩成一个对象，自然性方块变成了一个交换三角形。该三角形的两个边是自然变换的分量。

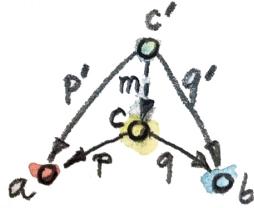


所以这是一个锥。我们感兴趣的是 普遍锥——就像我们在定义积时选择了一个普遍对象一样。

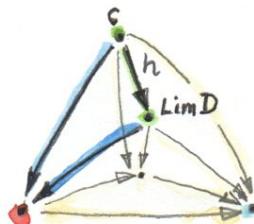
有很多方法可以实现它。例如，我们可以基于给定的函子 D 定义一个 锥的范畴。该范畴中的对象是锥。然而，并不是 \mathbf{C} 中的每个对象 c 都可以作为一个锥的顶点，因为可能不存在 Δ_c 和 D 之间的自然变换。

要构成一个范畴，我们还必须定义锥之间的态射。这些将完全由它们的顶点之间的态射决定。但并不是任何态射都可以。记住，在我们构造积时，我们规定候选对象（顶点）之间的态射必须是投影的公因子。例如：

snippet01



这一条件在一般情况下转化为这样的条件：三角形的一边是因式分解态射的因子，它们都交换。



连接两个锥的交换三角形，其中因式分解态射为 h （此处，下锥为普遍锥，顶点为 $\text{Lim } D$ ）

我们将这些因式分解态射作为锥范畴中的态射。很容易验证这些态射确实可以组合，并且恒等态射也是因式分解态射。因此锥形成了一个范畴。

现在我们可以将普遍锥定义为锥范畴中的 终对象。终对象的定义表明，从任何其他对象到该对象都有一个唯一的态射。在我们的情况下，这意味着从任何其他锥的顶点到普遍锥的顶点都有一个唯一的因式分解态射。我们将此普遍锥称为图 D 的 极限，记为 $\text{Lim } D$ （在文献中，你会经常看到 Lim 符号下方指向 I 的左箭头）。通常，作为简写，我们称该锥的顶点为极限（或极限对象）。

直觉上，极限在一个对象中体现了整个图的性质。例如，我们的两个对象图的极限就是两个对象的积。积（连同两个投影）包含了关于这两个对象的信息。作为普遍的对象，它没有多余的部分。

12.1 作为自然同构的极限

关于极限的这一定义，仍有一些不尽如人意的地方。我的意思是，它是可行的，但我们仍然有关于链接任何两个锥的三角形的交换性条件。如果我们能够用某种自然性条件来替换它，那将是多么优雅。但该怎么做呢？

我们不再仅仅处理一个锥，而是处理一个整合集合（实际上是一个范畴）的锥。如果极限存在（让我们明确一点，这没有任何保证），其

中一个锥就是普遍锥。对于每一个其他锥，我们都只有一个唯一的因式分解态射将其顶点（我们称之为 c ）映射到普遍锥的顶点，我们称之为 $\mathbf{Lim}D$ 。（实际上，我可以省略“其他”这个词，因为恒等态射将普遍锥映射到自身，并且它可以通过自身平凡地因式分解。）让我重复一下重要部分：给定任何锥，都有一种特殊类型的唯一态射。我们有一个从锥到特殊态射的一对一映射。

这个特殊态射是 $C(c, \mathbf{Lim}D)$ 中的一个元素。这个同态集的其他元素不那么幸运，因为它们不能因式分解两个锥的映射。我们想要的是能够为每个 c 从集合 $C(c, \mathbf{Lim}D)$ 中选择一个态射——满足特定交换性条件的态射。这听起来像是在定义自然变换吗？的确如此！

但是什么函子与此变换有关呢？

一个函子是将 c 映射到集合 $C(c, \mathbf{Lim}D)$ 。这是一个从 C 到 \mathbf{Set} 的函子——它将对象映射到集合。实际上，它是一个反变函子。以下是我们定义它对态射的作用的方法：假设我们有一个从 c' 到 c 的态射 f ：

$$f :: c' \rightarrow c$$

我们的函子将 c' 映射到集合 $C(c', \mathbf{Lim}D)$ 。为了定义该函子对 f 的作用（换句话说，提升 f ），我们必须定义 $C(c, \mathbf{Lim}D)$ 和 $C(c', \mathbf{Lim}D)$ 之间相应的映射。假设我们选择 $C(c, \mathbf{Lim}D)$ 的一个元素 u ，看看我们是否可以将其映射到 $C(c', \mathbf{Lim}D)$ 的某个元素。一个同态集的元素是一个态射，因此我们有：

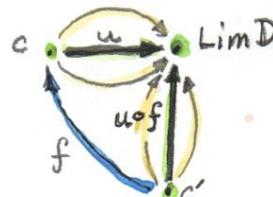
$$u :: c \rightarrow \mathbf{Lim}D$$

我们可以将 u 和 f 进行预组合得到：

$$u \circ f :: c' \rightarrow \mathbf{Lim}D$$

这就是 $C(c', \mathbf{Lim}D)$ 的一个元素——所以确实，我们找到了态射的一个映射：

snippet02 注意 c 和 c' 的顺序颠倒了，这正是 反变函子的特征。



为了定义自然变换，我们需要另一个从 C 到 \mathbf{Set} 的函子。但这次我们考虑的是一组锥。锥只是自然变换，因此我们关注的是自然变换集 $Nat(\Delta_c, D)$ 。将 c 映射到这一特定自然变换集的映射是一个（反变）函子。我们如何证明这一点呢？再次，我们定义其对态射的作用：

$$f :: c' \rightarrow c$$

对 f 的提升应该是一个自然变换之间的映射，这两个函子从 **I** 到 **C**：

$$Nat(\Delta_c, D) \rightarrow Nat(\Delta_{c'}, D)$$

我们如何映射自然变换呢？每个自然变换都是态射的选择——它的分量，每个 **I** 的元素有一个态射。某个 α ($Nat(\Delta_c, D)$ 的一个成员) 在 a (**I** 中的一个对象) 的分量是一个态射：

$$\alpha_a :: \Delta_c a \rightarrow Da$$

或者使用常值函子 Δ 的定义，

$$\alpha_a :: c \rightarrow Da$$

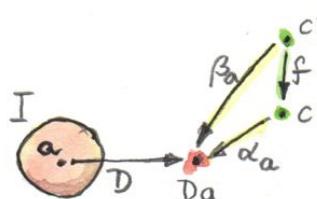
给定 f 和 α ，我们必须构造一个 β ， $Nat(\Delta_{c'}, D)$ 的成员。它在 a 处的分量应该是一个态射：

$$\beta_a :: c' \rightarrow Da$$

我们可以通过与 f 预组合来轻松地从前者 (α_a) 得到后者 (β_a)：

$$\beta_a = \alpha_a \circ f$$

很容易证明这些分量确实构成了一个自然变换。



给定我们的态射 f ，我们就这样逐个分量地构建了两个自然变换之间的映射。此映射定义了函子的：

$$c \rightarrow Nat(\Delta_c, D)$$

我刚才做的是向你展示我们有两个从 **C** 到 **Set** 的（反变）函子。我没有做任何假设——这些函子总是存在的。

顺便说一下，第一个函子在范畴论中起着重要作用，当我们谈到 Yoneda 引理时，我们会再次看到它。任何从范畴 **C** 到 **Set** 的反变函子都有一个名称：它们被称为“预层”。而这个被称为 表示预层。第二个函子也是一个预层。

现在我们有了两个函子，我们可以谈论它们之间的自然变换。所以不再赘述，结论如下：从 **I** 到 **C** 的函子 D 只有在存在两个我刚刚定义的函子之间的自然同构时才有极限：

$$C(c, \mathbf{Lim} D) \simeq Nat(\Delta_c, D)$$

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态射，也就是说是一个可逆态射。

我不会详细讨论这一声明的证明过程。这个过程相当直接，但有点繁琐。当处理自然变换时，通常关注的是态射的分量。在这种情况下，由于两个函子的目标都是 **Set**，自然同构的分量将是函数。这些是高阶函数，因为它们从同态集映射到自然变换集。同样，你可以通过考虑函数对其参数的操作来分析一个函数：在这里，参数将是一个态射—— $C(c, \text{Lim}D)$ 的成员——结果将是一个自然变换—— $Nat(\Delta_c, D)$ 的成员，或我们称之为锥的东西。这个自然变换依次有其自己的分量，这些分量也是态射。因此，自始至终都是态射，如果你能跟踪它们，你就可以证明这一声明。

最重要的结果是，此同构的自然性条件恰好是锥映射的交换性条件。

作为即将到来的预告，让我提到， $Nat(\Delta_c, D)$ 集可以看作是函子范畴中的同态集；所以我们的自然同构关系着两个同态集，这指向了一个更普遍的关系，称为伴随。

12.2 极限的示例

我们已经看到，范畴积是由我们称为 2 的简单范畴生成的图的极限。

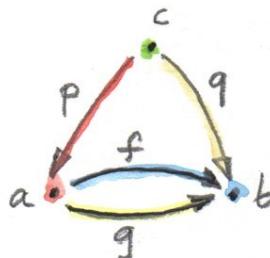
有一个更简单的极限示例：终对象。第一个冲动可能是认为单一对象范畴会导致终对象，但事实甚至比这更简洁：终对象是由空范畴生成的极限。从空范畴到某个范畴的函子不选择任何对象，因此锥缩小为仅顶点的情况。普遍锥是唯一具有从任何其他顶点到它的唯一态射的顶点。你会认出这是终对象的定义。

下一个有趣的极限称为 等化子。它是由一个两元素范畴生成的极限，该范畴中有两个平行的态射在它们之间（以及总是存在的恒等态射）。这个范畴选择了 C 中由两个对象 a 和 b 以及两个态射组成的图：

snippet03

要在此图之上构建一个锥，我们必须添加顶点 c 和两个投影：

snippet04



我们有两个必须交换的三角形：

snippet05

这告诉我们， q 由这些方程中的一个唯一决定，比如，我们可以省略它。因此我们只剩下一个条件：

snippet06

想法是，如果我们将注意力集中在 **Set** 上，函数 p 的像选择了 a 的一个子集。当限制在该子集上时，函数 f 和 g 是相等的。

例如，假设 a 是由坐标 x 和 y 参数化的二维平面。 b 是实数线，并且：

snippet07

这两个函数的等化子是实数集（顶点 c ）和函数：

snippet08

注意 $(p t)$ 定义了二维平面中的一条直线。沿着这条直线，这两个函数是相等的。

当然，还有其他的集合 c' 和函数 p' 可能会导致相等：

snippet09

但它们都唯一地因式分解通过 p 。例如，我们可以将单元素集 () 作为 c' ，并选择函数：

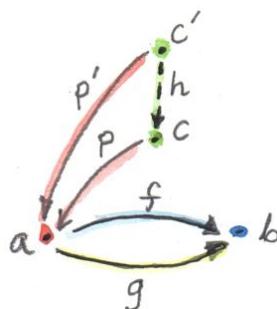
snippet10

这是一个很好的锥，因为 $f(0,0) = g(0,0)$ 。但它不是普遍的，因为它通过 h 唯一因式分解：

snippet11

其中

snippet12



因此，等化子可用于解决 $f x = g x$ 类型的方程。但它更为一般，因为它是用对象和态射定义的，而不是代数地定义的。

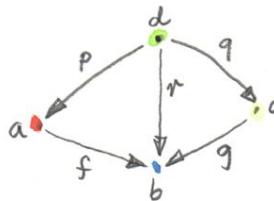
解决方程的一个更一般的想法体现在另一种极限——拉回。这里，我们仍然有两个想要等式化的态射，但这次它们的定义域不同。我们从一个形状为 $1 \rightarrow 2 \leftarrow 3$ 的三对象范畴开始。此范畴对应的图由三个对象 a 、 b 和 c 以及两个态射组成：

snippet13

此图通常称为 余弦图。

在此图之上构建的锥由顶点 d 和三个态射组成：

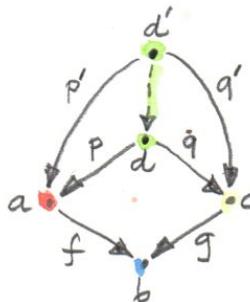
snippet14



交换性条件告诉我们 r 是完全由其他态射决定的，因此可以从图中省略。所以我们只剩下以下条件：

snippet15

拉回是这种形状的普遍锥。



再次，如果你将焦点缩小到集合，可以认为对象 d 由一对来自 a 和 c 的元素组成，其中第一个分量上的 f 等于第二个分量上的 g 。如果这仍然太过一般，考虑一个特殊情况，其中 g 是常函数，比如 $g = 1.23$ （假设 b 是实数集）。那么你实际上是在解决方程：

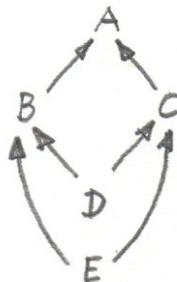
snippet16

在这种情况下， c 的选择无关紧要（只要它不是一个空集），所以我们可以将其视为一个单元素集。集合 a 可以是三维向量的集合， f 是向量的长度。然后拉回是对 (v, \emptyset) 的集合，其中 v 是长度为 1.23 的向量 ($\sqrt{x^2 + y^2 + z^2} = 1.23$ 的解)， \emptyset 是单元素集的虚拟元素。

但拉回在编程中有更广泛的应用。例如，将 C++ 类视为一个范畴，其中态射是将子类连接到超类的箭头。我们将继承视为一种传递性属性，所以如果 继承自，并且 继承自，那么我们会说 继承自（毕竟，你可以在需要指向 的地方传递指向 的指针）。此外，我们将假设 继承自，因此我们为每个类设置了恒等箭头。通过这种方式，子类化与子类型化对齐。C++ 还支持多重继承，因此你可以构建一个菱形继承图，其中两个类 和 继承自，第四个类 多重继承自 和。通常情况下，会

有两个 的副本，这通常不是我们想要的；但你可以使用虚拟继承，在中只有一个 的副本。

在这种图中，是一个拉回意味着什么？这意味着从 和 多重继承的任何类 也是 的子类。这在 C++ 中并不能直接表达，因为子类型化是名义上的（C++ 编译器不会推断这种类关系——它需要“鸭子类型”）。但我们可以超越子类型化关系，改为询问从 到 的转换是否安全。如果是 和 的裸骨组合，没有额外的数据和方法重载，那么这个转换就是安全的。当然，如果 和 的某些方法存在名称冲突，那么就不会有拉回。



在类型推断中还有一个更高级的拉回用法。通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：

它会给所有变量和子表达式分配初步类型。特别是，它将分配：

由此推导出：

它还会生成一组由函数应用规则产生的约束：

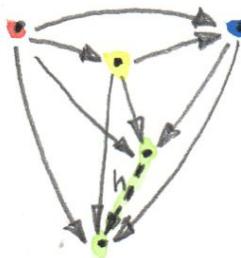
这些约束必须通过找到一组类型（或类型变量）来统一，这些类型在代入两个表达式中的未知类型时会产生相同的类型。一个这样的代入是：

但显然，这不是最一般的。最一般的代入是通过拉回获得的。我不会详细讨论其细节，因为它们超出了本书的范围，但你可以相信，结果应该是：

其中 是一个自由类型变量。

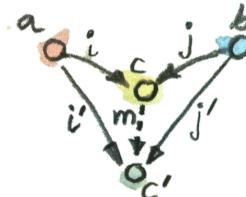
12.3 余极限

就像范畴论中的所有构造一样，极限在对偶范畴中有其对偶图像。当你反转锥中的所有箭头方向时，你得到一个余锥，而这些锥的普遍锥称为余极限。注意，反转也会影响因式分解态射，现在它从普遍余锥流向任何其他余锥。



一个余锥及连接两个顶点的因式分解态射 h 。

余积对应于由 2 生成的图，是余极限的典型示例，这是我们在定义积时使用的范畴。



积和余积都体现了一对对象的本质，但每种方式不同。

就像终对象是极限一样，初对象是基于空范畴的图的余极限。

拉回的对偶称为 推送。它基于一个图，称为跨越图，由范畴 $1 \leftarrow 2 \rightarrow 3$ 生成。

12.4 连续性

我之前说过，函子非常接近连续映射范畴的思想，因为它们永远不会破坏现有的连接（态射）。实际上，从范畴 C 到 C' 的 连续函子 F 的定义包括函子保留极限的要求。 C 中的任意图 D 可以通过简单地组合两个函子映射到 C' 中的图 $F \circ D$ 。 F 的连续性条件规定，如果图 D 有一个极限 $\text{Lim } D$ ，则图 $F \circ D$ 也有一个极限，并且它等于 $F(\text{Lim } D)$ 。



注意，由于函子将态射映射到态射，并将组合映射到组合，因此锥的像始终是一个锥。交换三角形总是被映射为交换三角形（函子保留组合）。对于因式分解态射也是如此：因式分解态射的像也是因式分解态射。因此，每个函子都几乎是连续的。可能出现的问题是唯一性条件。 \mathbf{C}' 中的因式分解态射可能不是唯一的。在 \mathbf{C} 中可能没有其他“更好的锥”。

同态函子是连续函子的一个例子。回想一下，同态函子 $\mathbf{C}(a, b)$ 在第一个变量中是反变的，而在第二个变量中是协变的。换句话说，这是一个函子：

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

当其第二个参数固定时，同态集函子（它成为表示预层）将 \mathbf{C} 中的余极限映射到 \mathbf{Set} 中的极限；当其第一个参数固定时，它将极限映射到极限。

在 Haskell 中，同态函子是将任意两个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时，例如固定为，我们得到反变函子：

`snippet17` 连续性意味着当应用于余极限（例如一个余积）时，它将生成一个极限；在本例中，是两个函数类型的积：

`snippet18` 确实，任何 的函数都实现为一个 case 语句，该语句由一对函数服务两个情况。

同样，当我们固定同态集的第一个参数时，我们得到熟悉的 reader 函子。其连续性意味着例如任何返回积的函数都等价于函数的积；特别是：

`snippet19` 我知道你在想什么：你不需要范畴论就能弄清楚这些东西。你是对的！然而，我仍然感到惊讶的是，这样的结果可以从第一个原则推导出来，而无需借助比特与字节、处理器架构、编译器技术，甚至 lambda 演算。

如果你对“极限”和“连续性”这些名称的来源感到好奇，它们是从微积分中相应概念推广而来的。在微积分中，极限和连续性是用开邻域定义的。开集定义了拓扑，形成一个范畴（一个偏序集）。

12.5 挑战

1. 你如何在 C++ 类范畴中描述推送？
2. 证明恒等函子 $\mathbf{Id} :: \mathbf{C} \rightarrow \mathbf{C}$ 的极限是初对象。

3. 给定集合的子集形成一个范畴。在该范畴中，如果第一个子集是第二个子集的子集，则定义为一个箭头。这样的范畴中两个集合的拉回是什么？推送是什么？初对象和终对象是什么？
4. 你能猜到等化子的对偶是什么吗？
5. 证明在具有终对象的范畴中，指向终对象的拉回是积。
6. 同样地，证明来自初对象的推送（如果存在）是余积。

13

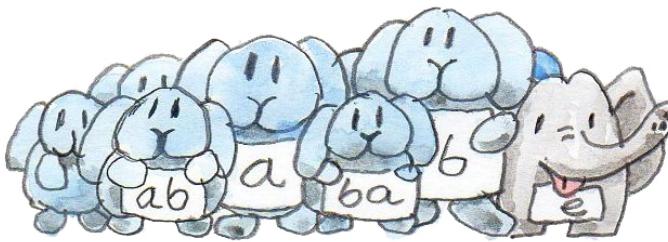
Free Monoids

单子是一个在范畴理论（CATEGORY THEORY）和编程中都非常重要的概念。范畴对应于强类型语言，而单子对应于无类型语言。这是因为在单子中你可以组合任意两个态射（arrow），就像在无类型语言中你可以组合任意两个函数一样（当然，当你执行程序时，可能会遇到运行时错误）。

我们已经看到，一个单子可以被描述为一个只有一个对象的范畴，其中所有逻辑都编码在态射组合的规则中。这种范畴模型与更传统的集合论单子定义完全等价，在集合论中我们“乘”两个元素得到第三个元素。这个“乘法”过程可以进一步拆分为首先形成一对元素，然后将这对元素与一个已有的元素——它们的“积”——相识别。

如果我们放弃乘法的第二部分——将对与现有元素相识别——会发生什么？例如，我们可以从一个任意的集合开始，形成所有可能的元素对，并称它们为新元素。然后我们将这些新元素与所有可能的元素配对，依此类推。这是一个链式反应——我们将永远添加新元素。结果，一个无限集合，将“几乎”是一个单子。但单子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，并识别一些对——仅足以支持单位和结合律。

让我们看看在一个简单的例子中这是如何工作的。我们从一个由两个元素组成的集合 $\{a, b\}$ 开始。我们称它们为自由单子的生成元。首先，我们将添加一个特殊的元素 e 作为单位。接下来我们将添加所有元素对，并称它们为“积”。 a 和 b 的积将是对 (a, b) 。 b 和 a 的积将是对 (b, a) ， a 和 a 的积将是对 (a, a) ， b 和 b 的积将是对 (b, b) 。我们还可以形成与 e 的对，例如 (a, e) 、 (e, b) 等，但我们将它们与 a 、 b 等元素相识别。因此在这一轮中我们只会添加 (a, a) 、 (a, b) 和 (b, a) 以及 (b, b) ，最终得到集合 $\{e, a, b, (a, a), (a, b), (b, a), (b, b)\}$ 。



在下一轮中，我们将继续添加元素，如 $(a, (a, b))$ 、 $((a, b), a)$ 等。这时我们必须确保结合律成立，因此我们将 $(a, (b, a))$ 与 $((a, b), a)$ 相识别。换句话说，我们不再需要内部的括号。

你可以猜到这个过程的最终结果：我们将创建所有可能的 a 和 b 列表。实际上，如果我们将 e 表示为空列表，我们可以看到我们的“乘法”实际上就是列表连接。

这种构造方法，即你不断生成所有可能的元素组合，并进行最小量的识别——刚好足以维持定律——称为自由构造。我们刚才做的就是从生成元集合 $\{a, b\}$ 构造一个自由单子。

13.1 Haskell 中的自由单子 (Free Monoid in Haskell)

Haskell 中的一个两元素集合等价于类型，由这个集合生成的自由单子等价于类型（的列表）。（我有意忽略了无限列表的问题。）

Haskell 中的单子通过类型类定义如下：

snippet01 这只是说每个 都必须有一个中性元素，称为，以及一个二元函数（乘法）称为。单位和结合律不能在 Haskell 中表达，必须由程序员在每次实例化单子时验证。

列表的任意类型形成一个单子的事实通过以下实例定义描述：

snippet02 这表明空列表 是单位元素，而列表连接 是二元操作。

正如我们所见，类型的列表对应于一个以集合 作为生成元的自由单子。以乘法为运算的自然数集合并不是一个自由单子，因为我们识别了很多积。例如，比较：

snippet03 那很容易，但问题是，我们能否在范畴理论中进行这种自由构造，在那里我们不允许查看对象的内部？我们将使用我们的常用工具：泛范式（universal construction）。

第二个有趣的问题是，是否可以通过在某些自由单子中识别超过所需最小数量的元素，得到任何单子？我将向你展示这直接来自于泛范式。

13.2 自由单子的泛范式 (Free Monoid Universal Construction)

如果你回想我们之前对泛范式的体验，你可能会注意到，这不只是关于构造某物，而是关于选择最适合给定模式的对象。因此，如果我们想使用泛范式“构造”一个自由单子，我们必须考虑一大堆单子以供选择。我们需要一个完整的单子范畴来选择。但单子能形成一个范畴吗？

首先让我们将单子看作由单位和乘法定义的集合。我们选择作为态射的那些保持单子结构的函数。这种保持结构的函数称为同态 (homomorphism)。一个单子同态必须将两个元素的积映射为这两个元素映射的积：

snippet04 并且必须将单位映射为单位。

例如，考虑一个从整数列表到整数的同态。如果我们将 $\text{映射为 } 2$ ，将 $\text{映射为 } 3$ ，我们必须将 $\text{映射为 } 6$ ，因为连接

snippet05 变成了乘法

snippet06 现在让我们忘记单个单子的内部结构，只看它们作为对象以及相应的态射。你会得到一个单子范畴 **Mon**。

好了，可能在我们忘记内部结构之前，让我们注意一个重要的属性。**Mon** 的每个对象都可以被平凡地映射到一个集合。它只是其元素的集合。这个集合称为基础集合 (underlying set)。事实上，我们不仅将 **Mon** 的对象映射到集合，还可以将 **Mon** 的态射 (同态) 映射为函数。同样，这似乎是琐碎的，但它很快就会变得有用。这个从 **Mon** 到 **Set** 的对象和态射映射实际上是一个函子 (functor)。由于这个函子“遗忘”了单子结构——一旦我们进入普通集合，我们就不再区分单位元素或关心乘法——它被称为遗忘函子 (forgetful functor)。遗忘函子在范畴理论中经常出现。

我们现在对 **Mon** 有两种不同的看法。我们可以像对待任何其他范畴一样对待它，具有对象和态射。在这种观点中，我们看不到单子的内部结构。我们对 **Mon** 中的特定对象所能说的只是它通过态射连接到自己和其他对象。态射的“乘法”表——组合规则——来源于另一种视角：单子作为集合。通过进入范畴理论，我们并没有完全失去这种视角——我们仍然可以通过我们的遗忘函子访问它。

为了应用泛范式，我们需要定义一个特殊属性，以便我们可以在单子范畴中搜索并选择自由单子的最佳候选者。但自由单子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单子（的列表与 的列表不同）。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！

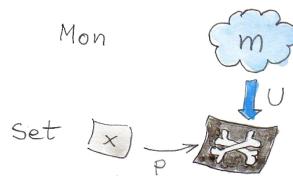
这就是遗忘函子派上用场的地方。我们可以用它来对单子进行 X 光成像。我们可以在这些 blob 的 X 光图像中识别生成元。它是这样工作的：

我们从一组生成元 x 开始。这是 **Set** 中的一个集合。

我们要匹配的模式由单子 m ——**Mon** 的一个对象——和 **Set** 中的一个函数 p 组成：

snippet07 其中 U 是我们的从 **Mon** 到 **Set** 的遗忘函子。这是一个奇怪的异质模式——一半在 **Mon** 中，一半在 **Set** 中。

这个想法是函数 p 将在 m 的 X 光图像中识别生成元。函数可能在识别集合中的点时表现不佳（它们可能会坍缩它们）。这一切都将通过泛范式解决，泛范式将选择该模式的最佳代表。

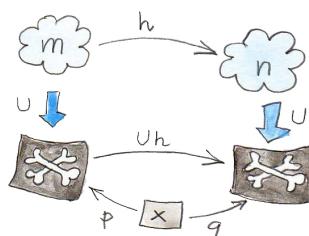


我们还必须定义候选者之间的排序。假设我们有另一个候选者：单子 n 及其 X 光图像中的生成元识别函数：

snippet08 我们将说 m 比 n 更好，如果存在一个单子态射（这是一个保持结构的同态）：

snippet09 它在 U 下的像（记住， U 是一个函子，所以它将态射映射为函数）通过 p 因式分解：

snippet10 如果你将 p 视为在 m 中选择生成元，并将 q 视为在 n 中选择“相同”的生成元，那么你可以将 h 视为在两个单子之间映射这些生成元。记住， h 按定义保持单子结构。这意味着一个单子中两个生成元的积将被映射为第二个单子中相应两个生成元的积，依此类推。



这个排序可用于找到最佳候选者——自由单子。定义如下：

我们将说 m （连同函数 p ）是具有生成元 x 的自由单子，当且仅当存在一个从 m 到任何其他单子 n （连同函数 q ）的唯一态射 h 满足上述因式分解属性。

顺便说一下，这回答了我们第二个问题。函数 Uh 有能力将 Um 的多个元素坍缩为 Un 的单个元素。这种坍缩对应于识别自由单子中的某些元

素。因此，任何具有生成元 x 的单子都可以通过识别一些元素从基于 x 的自由单子中获得。自由单子是一个只做了最小量识别的单子。

当我们讨论伴随 (adjunction) 时，我们将回到自由单子。

13.3 挑战 (Challenges)

1. 你可能会认为（正如我最初所想）保持单子同态的单位的要求是多余的。毕竟，我们知道对于所有 a 都有

因此 he 作为右单位（同理作为左单位）起作用。问题是，所有 a 的 ha 可能只覆盖目标单子的一个子单子。可能存在一个“真正的”单位在 h 的像之外。证明保持乘法的单子之间的同构必须自动保持单位。

2. 考虑一个从整数列表（连接）到整数（乘法）的单子同态。空列表 的像是什么？假设所有单一列表都被映射为它们包含的整数，即 被映射为 3 等。的像是什么？有多少个不同的列表映射到整数 12？是否存在这两个单子之间的其他同态？
3. 由一个元素集生成的自由单子是什么？你能看到它同构于什么吗？

14

Representable Functors

现在是时候谈谈集合了。数学家与集合论（set theory）有一种爱恨交织的关系。集合论曾是数学的汇编语言——至少以前是。范畴论（category theory）在某种程度上试图远离集合论。例如，众所周知，所有集合的集合不存在，但所有集合的范畴 Set 却存在。这是个好消息。另一方面，我们假设一个范畴中任意两个对象之间的态射（morphisms）形成一个集合。我们甚至称之为同态集（hom-set）。公平地说，范畴论有一个分支，其中态射不构成集合，而是另一个范畴中的对象。那些使用同态对象而非同态集的范畴被称为充实范畴（enriched categories）。在接下来的讨论中，我们将坚持使用老式的同态集范畴。

集合是范畴对象之外最接近于无特征斑点的东西。集合有元素，但你不能对这些元素说太多。如果你有一个有限集合，你可以数元素。你也可以用基数（cardinal numbers）“数”无限集合的元素。例如，自然数集合比实数集合要小，尽管两者都是无限的。但也许令人惊讶的是，有理数集合与自然数集合一样大。

除此之外，有关集合的所有信息都可以通过它们之间的函数来编码——特别是可逆的函数，称为同构（isomorphisms）。在所有意图和目的上，同构的集合是相同的。在我招致基础数学家们的愤怒之前，让我解释一下，等同和同构之间的区别是根本性的。事实上，这是最新数学分支之一，同伦类型论（Homotopy Type Theory, HoTT）的主要关注点之一。我提到 HoTT 是因为它是一种纯粹的数学理论，受到计算的启发，而其主要倡导者之一 Vladimir Voevodsky 在研究 Coq 定理证明器时获得了重大启发。数学与编程之间的互动是双向的。

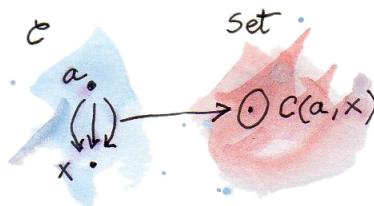
关于集合的重要教训是，可以比较不同元素的集合。例如，我们可以说某个自然变换（natural transformations）集合与某个态射集合是同构的，因为集合只是一个集合。同构在这种情况下仅意味着，对于一个集合中的每个自然变换，另一个集合中都有一个唯一的态射与之对应，

反之亦然。它们可以一一对应。如果它们是不同范畴的对象，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为我们提供必要的洞察，甚至让我们证明有价值的定理。

14.1 同态函子 (The Hom Functor)

每个范畴都配备了一组规范的映射到 **Set**。这些映射实际上是函子 (functors)，因此它们保留了范畴的结构。让我们构建一个这样的映射。

让我们固定范畴 **C** 中的一个对象 a ，然后选择另一个对象 x ，同样在 **C** 中。同态集 $\mathbf{C}(a, x)$ 是一个集合，一个 **Set** 中的对象。当我们在保持 a 不变的情况下改变 x 时， $\mathbf{C}(a, x)$ 也会在 **Set** 中变化。因此，我们得到了从 x 到 **Set** 的映射。



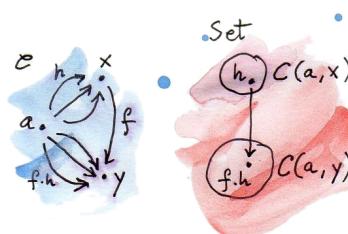
如果我们想强调我们正在考虑的同态集是其第二个参数中的映射，我们使用符号 $\mathbf{C}(a, -)$ ，其中破折号用作参数的占位符。

这个对象的映射可以轻松扩展到态射的映射。让我们在 **C** 中取一个态射 f ，它在两个任意对象 x 和 y 之间。对象 x 映射到集合 $\mathbf{C}(a, x)$ ，对象 y 映射到 $\mathbf{C}(a, y)$ ，在我们刚刚定义的映射下。如果这个映射是一个函子，那么 f 必须被映射到两个集合之间的函数： $\mathbf{C}(a, x) \rightarrow \mathbf{C}(a, y)$

让我们逐点定义这个函数，即分别为每个参数定义。对于参数，我们应该选择 $\mathbf{C}(a, x)$ 的任意元素——我们称之为 h 。态射是可组合的，如果它们端到端匹配。碰巧的是， h 的目标与 f 的源匹配，因此它们的组合：

$$f \circ h :: a \rightarrow y$$

是一个从 a 到 y 的态射。因此它是 $\mathbf{C}(a, y)$ 的成员。



我们刚刚找到了从 $\mathbf{C}(a, x)$ 到 $\mathbf{C}(a, y)$ 的函数，它可以作为 f 的像。如果没有混淆的危险，我们会将这个提升的函数写为： $\mathbf{C}(a, f)$ ，并将其在态射 h 上的作用写为：

$$\mathbf{C}(a, f)h = f \circ h$$

由于这个构造在任何范畴中都有效，因此它也必须在 Haskell 类型的范畴中有效。在 Haskell 中，同态函子更为人所知的是 函子：

snippet01

snippet02 现在让我们考虑，如果不是固定同态集的源，而是固定目标，会发生什么。换句话说，我们在问，映射 $\mathbf{C}(-, a)$ 是否也是一个函子。它是一个函子，但不是协变的（covariant），而是逆变的（contravariant）。这是因为相同的端到端匹配的态射组合结果是在 f 后组合（postcomposition）；而不是像 $\mathbf{C}(a, -)$ 那样是前组合（precomposition）。

我们已经在 Haskell 中看到了这个逆变函子。我们称之为：

snippet03

snippet04 最后，如果我们让两个对象都变化，我们得到一个双函子（profunctor） $\mathbf{C}(-, =)$ ，它在第一个参数中是逆变的，而在第二个参数中是协变的（为了强调两个参数可以独立变化，我们在第二个占位符上使用双破折号）。当我们谈论函子性时，我们已经见过这个双函子：

snippet05 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。由于逆变性等同于从对偶范畴的映射，我们可以简洁地陈述这一事实：

$$\mathbf{C}(-, =) :: \mathbf{C}^{op} \times \mathbf{C} \rightarrow \mathbf{Set}$$

14.2 可表示函子 (Representable Functors)

我们已经看到，对于 \mathbf{C} 中的每个对象 a ，我们都可以得到一个从 \mathbf{C} 到 \mathbf{Set} 的函子。这种结构保留映射到 \mathbf{Set} 的结构通常称为表示（representation）。我们将 \mathbf{C} 的对象和态射表示为 \mathbf{Set} 中的集合和函数。

函子 $\mathbf{C}(a, -)$ 本身有时被称为可表示的。更一般地，任何对某个 a 的选择自然同构于同态函子的函子 F 都称为可表示的。这样的函子必须是 \mathbf{Set} -值的，因为 $\mathbf{C}(a, -)$ 是。

我之前说过，我们经常将同构集合视为相同的。更一般地，我们将范畴中的同构对象视为相同的。这是因为对象除了通过态射与其他对象（以及自身）的关系之外没有其他结构。

例如，我们之前讨论过由集合建模的单子范畴 **Mon**。但我们小心翼翼地选择了那些保持这些集合的单子结构的函数作为态射。因此，如果 **Mon** 中的两个对象是同构的，意味着它们之间存在可逆的态射，那么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，我们会看到一个单子的单位元素映射到另一个单子的单位元素，并且两个元素的积映射到它们的映射的积。

同样的推理可以应用于函子。两个范畴之间的函子形成了一个范畴，在其中自然变换（natural transformations）扮演了态射的角色。因此，如果两个函子是同构的，我们可以将它们视为相同的，如果它们之间存在可逆的自然变换。

让我们从这个角度分析可表示函子的定义。对于 F 是可表示的，我们要求：在 \mathbf{C} 中存在一个对象 a ；从 $\mathbf{C}(a, -)$ 到 F 的自然变换 α ；一个方向相反的自然变换 β ；并且它们的组合是恒等自然变换。

让我们看看 α 在某个对象 x 处的分量。这是一个 **Set** 中的函数：

$$\alpha_x : \mathbf{C}(a, x) \rightarrow Fx$$

这个变换的自然性条件告诉我们，对于从 x 到 y 的任何态射 f ，下图是交换的：

$$Ff \circ \alpha_x = \alpha_y \circ \mathbf{C}(a, f)$$

在 Haskell 中，我们会用多态函数替换自然变换：

snippet06 使用可选的 量词。自然性条件

snippet07 由于参数化（parametricity）的原因自动满足（这是我之前提到的那些免费的定理之一），理解左边的 由函子 F 定义，而右边的由 reader 函子定义。由于 reader 的 只是函数前组合（function precomposition），我们可以更明确一点。作用于 $\mathbf{C}(a, x)$ 的一个元素 h ，自然性条件简化为：

snippet08 另一个变换 走相反的方向：

snippet09 它必须满足自然性条件，并且必须是 的逆：

我们稍后会看到，从 $\mathbf{C}(a, -)$ 到任何 **Set**-值函子的自然变换总是存在的，只要 Fa 非空（Yoneda 引理），但它不一定是可逆的。

让我给你一个 Haskell 中的例子，使用列表函子和 作为。这是一个可以完成任务的自然变换：

snippet10 我任意选择了数字 12，并用它创建了一个单元素列表。然后我可以对这个列表调用 函数，得到一个类型为 返回类型的列表。（实际上，这样的变换与整数列表一样多。）

自然性条件等价于（列表版本的）的可组合性：

snippet11 但如果我们试图找到逆变换，我们需要从一个任意类型的列表到一个返回 的函数：

snippet12 你可能会想到从列表中检索一个，例如，使用，但这对空列表不起作用。注意，这里没有一种类型（代替）可以奏效。所以列表函子不是可表示的。

还记得我们谈到 Haskell 的（自）函子有点像容器吗？同样，我们可以将可表示函子视为存储函数调用结果的缓存容器（在 Haskell 中，同态集的成员只是函数）。表示对象， $\mathbf{C}(a, -)$ 中的类型 a ，被认为是键类型，我们可以用它来访问函数的表格化值。我们称为 的变换称为，它的逆变换 称为。以下是一个（略微简化的）类的定义：

snippet13 注意，表示类型，即我们所说的 a ，这里称为，是 定义的一部分。星号只是意味着 是一个类型（而不是类型构造器或其他更奇特的种类）。

无限列表或流 (streams)，不能是空的，是可表示的。

snippet14 你可以将它们视为接受 作为参数的函数的缓存值。（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。）

要 这样的函数，你需要创建一个无限的值流。当然，这只能在 Haskell 是惰性求值 (lazy evaluation) 的情况下实现。值是在需求时计算的。你可以使用 访问缓存的值：

snippet15 有趣的是，你可以实现一个单一的缓存机制来覆盖一整个家族的函数，返回类型任意。

逆变函子的可表示性类似定义，不同之处在于我们保持 $\mathbf{C}(-, a)$ 的第二个参数固定。或者，等效地，我们可以考虑从 \mathbf{C}^{op} 到 \mathbf{Set} 的函子，因为 $\mathbf{C}^{\text{op}}(a, -)$ 与 $\mathbf{C}(-, a)$ 相同。

可表示性有一个有趣的转折。记住，同态集可以在内部视为指数对象，在笛卡尔闭范畴 (Cartesian closed categories) 中。 $\mathbf{C}(a, x)$ 的同态集等价于 x^a ，对于一个可表示函子 F ，我们可以写作： $-^a = F$ 。

让我们做一个形式上的变换： $a = \mathbf{log}F$

当然，这只是一个纯粹的形式变换，但如果你知道一些对数的性质，它会非常有帮助。特别是，基于乘积类型的函子可以用和类型来表示，而和类型函子通常不可表示（例如：列表函子）。

最后，注意一个可表示函子为我们提供了两个不同的实现——一个是函数，一个是数据结构。它们有完全相同的内容——使用相同的键检索相同的值。这就是我所说的“相同”的含义。对于它们的内容，两个自然同构的函子是相同的。另一方面，这两种表示通常是不同的实现，可能具有不同的性能特性。缓存用作性能增强，可能显著减少运行时间。能够生成相同基础计算的不同表示在实践中非常有价值。因此，令人惊讶的是，尽管范畴论完全不关注性能，但它提供了许多探索具有实际价值的替代实现的机会。

14.3 挑战 (Challenges)

1. 证明同态函子将范畴 C 中的恒等态射映射到 \mathbf{Set} 中相应的恒等函数。
2. 证明 不是可表示的。
3. 函子是可表示的吗？
4. 使用 表示，缓存一个平方其参数的函数。
5. 证明 和 对于 确实是彼此的逆。（提示：使用归纳法。）
6. 函子：

是可表示的。你能猜出表示它的类型吗？实现 和 。

14.4 参考文献 (Bibliography)

1. Catsters 视频关于 可表示函子¹。

¹

15

The Yoneda Lemma

大 多数范畴论中的构造都是从数学中其他更具体的领域中推广而来的结果。像积 (product)、余积 (coproduct)、幺半群 (monoids)、指数 (exponentials) 等概念早在范畴论之前就已经为人所知了。它们在不同的数学分支中可能有不同的名称。例如，在集合论中是笛卡尔积 (Cartesian product)、在序理论中是交 (meet)、在逻辑中是合取 (conjunction) —— 这些都是范畴积这一抽象概念的具体例子。

在这方面，**Yoneda** 引理格外引人注目，因为它是关于范畴的一个广泛声明，在数学的其他分支中几乎没有前例。有人说它最接近的类比是群论中的凯莱定理 (Cayley's theorem)，即每个群都同构于某个集合的置换群。

Yoneda 引理的背景是一个任意的范畴 C ，以及从 C 到 Set 的一个函子 F 。我们在前一节中看到，一些 Set 值函子是可表示的，也就是说，它们同构于某个同态函子 (hom-functor)。**Yoneda** 引理告诉我们，所有的 Set 值函子都可以通过自然变换 (natural transformations) 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具有约束性。当你在一个对象上定义了自然变换的一个分量时，自然性可能足够强大，可以将这个分量“传递”到通过态射与之连接的另一个对象中。在源范畴和目标范畴中的对象之间的箭头越多，你就有越多的约束条件来传递自然变换的分量。 Set 恰好是一个箭头非常丰富的范畴。

Yoneda 引理告诉我们，两个同态函子和任意其他函子 F 之间的自然变换完全由在一个点上的单个分量的值决定！其余的自然变换仅由自然性条件决定。

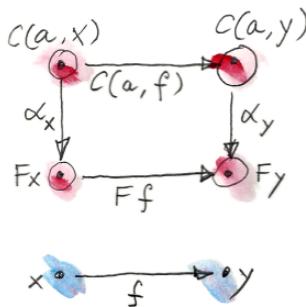
因此，让我们回顾一下 **Yoneda** 引理中涉及的两个函子之间的自然性条件。第一个函子是同态函子。它将 C 中的任意对象 x 映射为 $C(a, x)$

的态射集，其中 a 是 \mathbf{C} 中的一个固定对象。我们还看到它将从 $x \rightarrow y$ 的任何态射 f 映射为 $\mathbf{C}(a, f)$ 。

第二个函子是任意的 **Set** 值函子 F 。

我们将这两个函子之间的自然变换称为 α 。由于我们在 **Set** 中操作，自然变换的分量，如 α_x 或 α_y ，只是集合之间的常规函数：

$$\begin{aligned}\alpha_x &:: \mathbf{C}(a, x) \rightarrow Fx \\ \alpha_y &:: \mathbf{C}(a, y) \rightarrow Fy\end{aligned}$$



因为这些只是函数，我们可以观察它们在特定点上的值。但是在集合 $\mathbf{C}(a, x)$ 中的一个点是什么？这是关键的观察点：集合 $\mathbf{C}(a, x)$ 中的每个点也是从 a 到 x 的态射 h 。

因此，自然性方块 α ：

$$\alpha_y \circ \mathbf{C}(a, f) = Ff \circ \alpha_x$$

在作用于 h 时，逐点变为：

$$\alpha_y(\mathbf{C}(a, f)h) = (Ff)(\alpha_x h)$$

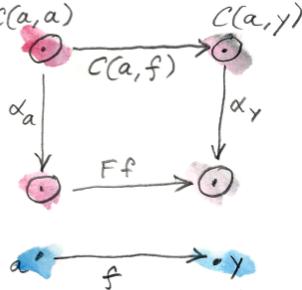
你可能还记得在前一节中，同态函子 $\mathbf{C}(a, -)$ 作用于态射 f 时的操作被定义为前组合：

$$\mathbf{C}(a, f)h = f \circ h$$

这就导致了：

$$\alpha_y(f \circ h) = (Ff)(\alpha_x h)$$

这种条件的强大之处在于将其特化为 $x = a$ 的情况。



在这种情况下， h 变成了从 a 到 a 的态射。我们知道至少有一个这样的态射，即 $h = \mathbf{id}_a$ 。让我们将其代入：

$$\alpha_y f = (Ff)(\alpha_a \mathbf{id}_a)$$

注意到刚刚发生的事情：左侧是自然变换 α_y 对 $\mathbf{C}(a, y)$ 中任意元素 f 的作用。它完全由 α_a 在 \mathbf{id}_a 上的单一值决定。我们可以选择任意这样的值，并且它将生成一个自然变换。由于 α_a 的值位于集合 Fa 中，因此 Fa 中的任何点都将定义一些 α 。

反之，给定从 $\mathbf{C}(a, -)$ 到 F 的任意自然变换 α ，你可以在 \mathbf{id}_a 处评估它以获得 Fa 中的一个点。

我们刚刚证明了 Yoneda 引理：

从 $\mathbf{C}(a, -)$ 到 F 的自然变换与 Fa 的元素之间存在一一对应关系。

换句话说，

$$\mathbf{Nat}(\mathbf{C}(a, -), F) \cong Fa$$

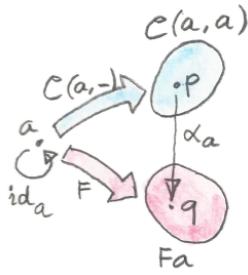
或者，如果我们使用符号 $[\mathbf{C}, \mathbf{Set}]$ 表示 \mathbf{C} 和 \mathbf{Set} 之间的函子范畴，则自然变换的集合只是该范畴中的一个同态集（hom-set），我们可以写作：

$$[\mathbf{C}, \mathbf{Set}](\mathbf{C}(a, -), F) \cong Fa$$

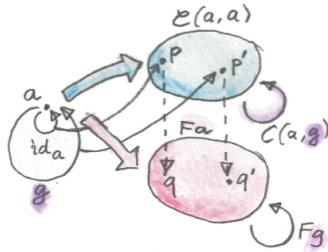
稍后我会解释这个对应关系实际上是一个自然同构。

现在让我们试着对这个结果获得一些直觉。最令人惊奇的是，整个自然变换从一个成核点开始结晶：我们在 \mathbf{id}_a 处赋予它的值。它从那个点开始扩展，遵循自然性条件。它渗透到 \mathbf{Set} 中 \mathbf{C} 的像中。因此，让我们首先考虑 \mathbf{C} 在 $\mathbf{C}(a, -)$ 下的像是什么。

让我们从 a 本身的像开始。在同态函子 $\mathbf{C}(a, -)$ 下， a 被映射为 $\mathbf{C}(a, a)$ 的集合。在函子 F 下，它被映射为集合 Fa 。自然变换 α_a 的分量是从 $\mathbf{C}(a, a)$ 到 Fa 的某个函数。让我们关注 $\mathbf{C}(a, a)$ 集合中的一个点，该点对应于态射 \mathbf{id}_a 。为了强调它只是集合中的一个点，我们称它为 p 。 α_a 的分量应该将 p 映射为 Fa 中的某个点 q 。我会向你展示，任何 q 的选择都会导致一个唯一的自然变换。



第一个主张是，一个点 q 的选择唯一地决定了其余的函数 α_a 。确实，让我们选择 $C(a, a)$ 中的另一个点 p' ，该点对应于从 a 到 a 的某个态射 g 。这是 Yoneda 引理的魔力发生的地方： g 可以看作是 $C(a, a)$ 集合中的一个点 p' 。同时，它选择了集合之间的两个函数。实际上，在同态函子下，态射 g 被映射为函数 $C(a, g)$ ；在 F 下，它被映射为 Fg 。



现在让我们考虑 $C(a, g)$ 作用于我们原来的 p 的动作，正如你所记得的那样，它对应于 \mathbf{id}_a 。它被定义为前组合 $g \circ \mathbf{id}_a$ ，等于 g ，即对应于我们的点 p' 。因此，态射 g 被映射为一个函数，当作用于 p 时产生 p' ，即 g 。我们已经完成了完整的循环！

现在考虑 Fg 对 q 的作用。它是 Fa 中的一个点 q' 。为了完成自然性方块， p' 必须在 α_a 下被映射为 q' 。我们选择了任意的 p' （任意的 g ），并推导出它在 α_a 下的映射。函数 α_a 因此完全确定。

第二个主张是，对于 C 中与 a 相连的任何对象 x ， α_x 是唯一确定的。推理是类似的，只不过现在我们有了两个更多的集合 $C(a, x)$ 和 Fx ，从 a 到 x 的态射 g 在同态函子下被映射为：

$$C(a, g) :: C(a, a) \rightarrow C(a, x)$$

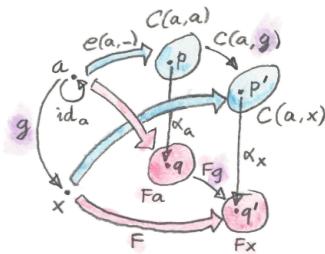
在 F 下则被映射为：

$$Fg :: Fa \rightarrow Fx$$

同样地， $C(a, g)$ 作用于我们的 p 是通过前组合给出的： $g \circ \mathbf{id}_a$ ，对应于 $C(a, x)$ 中的一个点 p' 。自然性决定了作用于 p' 的 α_x 的值是：

$$q' = (Fg)q$$

由于 p' 是任意的，因此整个函数 α_x 因此确定。



如果 \mathbf{C} 中有与 a 无连接的对象怎么办？它们都在 $\mathbf{C}(a, -)$ 下映射为单个集合——空集。回想一下，空集是集合范畴中的始对象（initial object）。这意味着从该集合到任何其他集合都有唯一的函数。我们称这个函数为。因此，在这里，自然变换的分量没有选择，它只能是。

理解 Yoneda 引理的一种方法是认识到，**Set** 值函子之间的自然变换只是函数的家族，而函数通常是有损的（lossy）。函数可能会丢失信息，并且可能只覆盖其值域的一部分。唯一不丢失信息的函数是可逆的——即同构（isomorphisms）。因此，最好的结构保留 **Set** 值函子是可表示的函子。它们要么是同态函子，要么是自然同构于同态函子的函子。任何其他函子 F 都是通过有损变换从同态函子中获得的。这样的变换不仅可能丢失信息，而且可能只覆盖函子 F 在 **Set** 中的像的一小部分。

15.1 Haskell 中的 Yoneda 引理

我们已经在 Haskell 中以 `reader` 函子的形式遇到过同态函子：

snippet01 `reader` 函子通过前组合映射态射（这里的函数）：

snippet02 Yoneda 引理告诉我们，`reader` 函子可以自然地映射到任何其他函子。

自然变换是多态函数。因此，给定一个函子，我们可以从 `reader` 函子映射到它：

snippet03 和往常一样，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 的元素是一一对应的：

\equiv

这个等式的右侧是我们通常认为的数据结构。记住函子作为广义容器的解释？是一个 的容器。但左侧是一个多态函数，它接受一个函数作为参数。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。

换句话说：给我一个类型为：

snippet04 的多态函数，我将生成一个 的容器。诀窍是我们在证明 Yoneda 引理时使用的：我们用 调用这个函数来获得 的一个元素：

snippet05 反之亦然：给定一个类型为 的值：

snippet06 可以定义一个类型正确的多态函数：

snippet07 你可以轻松地在两种表示之间来回转换。

拥有多种表示的好处是，其中一种可能比另一种更容易组合，或者在某些应用中更高效。

这一原理的最简单示例是编译器构建中常用的代码转换：续延传递风格（continuation-passing style, cps）。这是 Yoneda 引理对恒等函子的最简单应用。将 替换为恒等函子得到：

\cong

这一公式的解释是，任何类型 都可以被一个接受 的“处理器”的函数替换。处理器是接受 并执行其余计算的函数——续延。（类型 通常封装某种状态代码。）

这种编程风格在用户界面（UIs）、异步系统和并发编程中非常常见。CPS 的缺点是它涉及控制的反转。代码分为生产者和消费者（处理器），并且不易组合。任何做过不小的 Web 编程的人都熟悉从交互式有状态处理器中产生的面条代码噩梦。正如我们稍后将看到的那样，合理使用函子和单子可以恢复 CPS 的某些组合属性。

15.2 对偶 Yoneda 引理 (Co-Yoneda)

像往常一样，我们通过逆转箭头的方向获得了一个额外的构造。Yoneda 引理可以应用于对偶范畴 \mathbf{C}^{op} ，为我们提供了逆变函子之间的映射。

同样地，我们可以通过固定同态函子的目标对象而不是源对象来推导对偶 Yoneda 引理。我们得到了从 \mathbf{C} 到 \mathbf{Set} 的逆变同态函子： $\mathbf{C}(-, a)$ 。Yoneda 引理的逆变版本建立了从这个函子到任何其他逆变函子 F 的自然变换与集合 Fa 的元素之间的一一对应关系：

$$\mathbf{Nat}(\mathbf{C}(-, a), F) \cong Fa$$

这是 Haskell 版本的对偶 Yoneda 引理：

\cong

请注意，在某些文献中，这个逆变版本被称为 Yoneda 引理。

15.3 挑战 (Challenges)

1. 证明构成 Haskell 中 Yoneda 同构的两个函数 和 是彼此的逆。

2. 离散范畴是指只有对象而没有态射的范畴，除了恒等态射之外。对于从这样一个范畴来的函子，Yoneda 引理如何运作？
3. 单位类型的列表 不包含除长度之外的其他信息。因此，作为一种数据类型，它可以被视为整数的编码。一个空列表编码为零，一个单元素列表（值，而不是类型）编码为一，依此类推。使用 Yoneda 引理为列表函子构造这种数据类型的另一种表示。

15.4 参考文献 (Bibliography)

1. Catsters¹ 视频。

¹

16

Yoneda Embedding

我们之前已经看到，当我们在范畴 \mathbf{C} 中固定一个对象 a 时，映射 $\mathbf{C}(a, -)$ 是一个从 \mathbf{C} 到 \mathbf{Set} 的（协变）函子。

$$x \rightarrow \mathbf{C}(a, x)$$

（由于同态集（hom-set） $\mathbf{C}(a, x)$ 是一个集合（set），因此对域是 \mathbf{Set} 。）我们称这个映射为同态函子（hom-functor）——我们之前已经定义了它在态射（morphisms）上的作用。

现在让我们变化 a 在这个映射中的值。我们得到一个新的映射，将同态函子（functor） $\mathbf{C}(a, -)$ 赋给任何 a 。

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

这是一个从范畴 \mathbf{C} 的对象到函子的映射，这些函子是函子范畴（functor category）中的对象（参见关于自然变换（Natural Transformations）的章节）。我们用符号 $[\mathbf{C}, \mathbf{Set}]$ 来表示从 \mathbf{C} 到 \mathbf{Set} 的函子范畴。你可能还记得，同态函子是可表函子（representable functors）的原型。

每当我们在两个范畴之间有对象的映射时，自然会问这种映射是否也是一个函子。换句话说，我们能否将一个范畴中的态射提升为另一个范畴中的态射。 \mathbf{C} 中的一个态射只是 $\mathbf{C}(a, b)$ 的一个元素，但在函子范畴 $[\mathbf{C}, \mathbf{Set}]$ 中的态射是一个自然变换（natural transformation）。所以我们在寻找将态射映射到自然变换的映射。

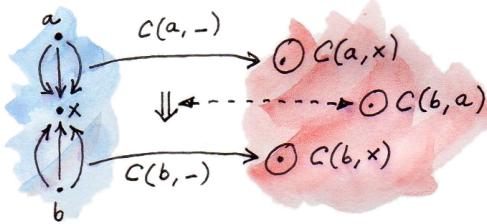
让我们看看是否能找到与态射 $f :: a \rightarrow b$ 对应的自然变换。首先，让我们看看 a 和 b 被映射到了哪里。它们被映射到两个函子： $\mathbf{C}(a, -)$ 和 $\mathbf{C}(b, -)$ 。我们需要在这两个函子之间找到一个自然变换。

这就是诀窍：我们使用 Yoneda 引理（Yoneda lemma）：

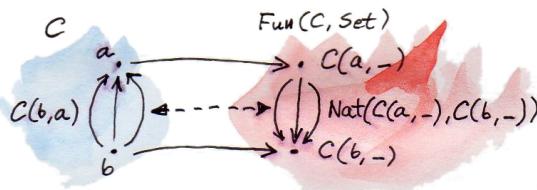
$$[\mathbf{C}, \mathbf{Set}](\mathbf{C}(a, -), F) \cong Fa$$

并将一般的 F 替换为同态函子 $\mathbf{C}(b, -)$ 。我们得到：

$$[\mathbf{C}, \mathbf{Set}](\mathbf{C}(a, -), \mathbf{C}(b, -)) \cong \mathbf{C}(b, a)$$



这正是我们在寻找的两个同态函子之间的自然变换，只是有一个小转折：我们有一个自然变换和态射之间的映射—— $\mathbf{C}(b, a)$ 的一个元素——它朝着“错误”的方向移动。但这没关系；这这意味着我们正在看的函子是逆变函子（contravariant）。



实际上，我们得到了比预期更多的东西。 \mathbf{C} 到 $[\mathbf{C}, \mathbf{Set}]$ 的映射不仅仅是一个逆变函子——它是一个完全忠实（fully faithful）的函子。完全性（fullness）和忠实性（faithfulness）是描述函子如何映射同态集的属性。

一个忠实（faithful）函子在同态集中是单射（injective）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。

一个完全（full）函子在同态集中是满射（surjective）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。

一个完全忠实函子 F 是同态集上的双射（bijection）——一个一对一匹配的所有元素。对于源范畴 \mathbf{C} 中的每一对对象 a 和 b ， $\mathbf{C}(a, b)$ 与 F 的目标范畴 \mathbf{D} 中的 $\mathbf{D}(Fa, Fb)$ 之间存在一个双射。在我们的情况下， \mathbf{D} 是函子范畴 $[\mathbf{C}, \mathbf{Set}]$ 。注意，这并不意味着 F 是对象上的双射。 \mathbf{D} 中可能存在不在 F 映像中的对象，我们无法对这些对象的同态集发表任何意见。

16.1 嵌入 (The Embedding)

我们刚刚描述的（逆变）函子，函子将 \mathbf{C} 中的对象映射到 $[\mathbf{C}, \mathbf{Set}]$ 中的函子：

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

定义了 Yoneda 嵌入 (Yoneda embedding)。它将范畴 \mathbf{C} (严格来说是逆范畴 \mathbf{C}^{op} , 因为它是逆变的) 嵌入到函子范畴 $[\mathbf{C}, \mathbf{Set}]$ 中。它不仅将 \mathbf{C} 中的对象映射为函子，还忠实地保留了它们之间的所有联系。

这是一个非常有用的结果，因为数学家们对函子范畴，尤其是那些对域为 \mathbf{Set} 的函子范畴了解很多。通过将任意范畴嵌入到函子范畴中，我们可以获得很多关于这个范畴的洞见。

当然，Yoneda 嵌入还有一个对偶版本，有时被称为 co-Yoneda 嵌入。注意，我们本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。这将给我们一个逆变同态函子。逆变函子从 \mathbf{C} 到 \mathbf{Set} 是我们熟悉的预层 (presheaves) (参见，例如，极限与余极限 (Limits and Colimits))。co-Yoneda 嵌入定义了将范畴 \mathbf{C} 嵌入到预层范畴中的嵌入。它在态射上的作用是：

$$[\mathbf{C}^{op}, \mathbf{Set}](\mathbf{C}(-, a), \mathbf{C}(-, b)) \cong \mathbf{C}(a, b)$$

同样，数学家们对预层范畴了解很多，所以能够将任意范畴嵌入其中是一个巨大的胜利。

16.2 在 Haskell 中的应用 (Application to Haskell)

在 Haskell 中，Yoneda 嵌入可以表示为读者函子 (reader functors) 之间的自然变换与函数 (逆向进行) 之间的同构：

\cong

(请记住，读者函子相当于 λ 。)

这个等式的左侧是一个多态函数，给定从 \mathbf{C} 到 \mathbf{Set} 的一个函数和一个类型为 $\mathbf{C}(a, -)$ 的值，可以生成一个类型为 $\mathbf{C}(a, b)$ 的值 (我正在展开——去掉了函数周围的括号)。唯一能对所有 \mathbf{C} 执行此操作的方法是，如果我们的函数知道如何将 $\mathbf{C}(a, -)$ 转换为 $\mathbf{C}(a, b)$ 。它必须秘密访问一个从 \mathbf{C} 到 \mathbf{Set} 的函数。

给定这样一个转换器，我们可以定义左侧，称其为 reader ，如下所示：

`snippet01` 相反，给定一个 $\mathbf{C}(a, -)$ 函数，我们可以通过调用`reader` 和`runReader` 来恢复转换器：

`snippet02` 这建立了 $\mathbf{C}(a, -)$ 和 $\mathbf{C}(a, b)$ 类型的函数之间的双射。

另一种看待这种同构的方法是，它是从 \mathbf{C} 到 \mathbf{Set} 的函数的 CPS 编码。参数 $\mathbf{C}(a, -)$ 是一个延续 (continuation) (处理程序)。结果是一个从 \mathbf{C} 到 \mathbf{Set} 的函数，当它被调用并传递一个类型为 $\mathbf{C}(a, -)$ 的值时，将执行与编码函数预组合的延续。

Yoneda 嵌入还解释了 Haskell 中某些数据结构的替代表示。特别是，它为 库中的镜头（lenses）提供了非常有用表示（very useful representation）¹。

16.3 预序示例 (Preorder Example)

这个示例由 Robert Harper 提出。它是 Yoneda 嵌入应用于由预序 (preorder) 定义的范畴。预序是一个具有元素间顺序关系的集合，传统上写作 \leq (小于或等于)。“预 (pre)”在预序中是因为我们只要求关系是传递的和自反的，而不一定是反对称的（所以有可能出现循环）。

具有预序关系的集合会产生一个范畴。对象是这个集合的元素。从对象 a 到 b 的态射要么不存在，如果对象不可比较或 $a \leq b$ 不成立；要么存在，如果 $a \leq b$ ，并且它从 a 指向 b 。在这种范畴中，从一个对象到另一个对象的态射最多只有一个。因此，在这种范畴中，任何同态集要么是一个空集，要么是一个单元素集。这样的范畴被称为瘦 (thin) 范畴。

很容易让自己相信这种结构确实是一个范畴：箭头是可组合的，因为如果 $a \leq b$ 和 $b \leq c$ ，那么 $a \leq c$ ；并且组合是结合的。我们还拥有身份箭头，因为每个元素（小于或）等于它自己（关系的自反性）。

现在我们可以将 co-Yoneda 嵌入应用于预序范畴。特别是，我们感兴趣的是它对态射的作用：

$$[\mathbf{C}, \mathbf{Set}](\mathbf{C}(-, a), \mathbf{C}(-, b)) \cong \mathbf{C}(a, b)$$

右侧的同态集当且仅当 $a \leq b$ 时非空——在这种情况下，它是一个单元素集。因此，如果 $a \leq b$ ，则左侧存在单个自然变换。否则不存在自然变换。

那么，在预序中，同态函子之间的自然变换是什么？它应该是集合 $\mathbf{C}(-, a)$ 和 $\mathbf{C}(-, b)$ 之间的一组函数。在预序中，每个这些集合要么为空，要么是单元素集合。让我们看看有什么样的函数可以供我们使用。

存在一个从空集到自身的函数（作用于空集的身份），一个从空集到单元素集合的函数（它什么也不做，因为它只需要为空集中的元素定义，而空集中没有元素），以及一个从单元素集合到自身的函数（作用于单元素集合的身份）。唯一被禁止的组合是将单元素集合映射到空集合的组合（当对单个元素作用时，这种函数的值将是什么？）。

因此，我们的自然变换永远不会将单元素同态集连接到空同态集。换句话说，如果 $x \leq a$ (单元素同态集 $\mathbf{C}(x, a)$)，那么 $\mathbf{C}(x, b)$ 不能为空。 $\mathbf{C}(x, b)$ 非空意味着 x 小于或等于 b 。因此，所讨论的自然变换的存在要求，对于每个 x ，如果 $x \leq a$ ，则 $x \leq b$ 。

对所有 $x \sqsubseteq x \leq a \Rightarrow x \leq b$

¹

另一方面，co-Yoneda 告诉我们，这种自然变换的存在等价于 $\mathbf{C}(a, b)$ 非空，或者 $a \leq b$ 。综合起来，我们得到：

$$a \leq b \text{ 当且仅当对于所有 } x \in a \Rightarrow x \leq b$$

我们本可以直接得出这个结论。直觉是，如果 $a \leq b$ ，那么所有低于 a 的元素也必须低于 b 。反过来，当你在右侧将 a 替换为 x 时，得出 $a \leq b$ 。但是你必须承认，通过 Yoneda 嵌入得出这个结果要更加令人兴奋。

16.4 自然性 (Naturality)

Yoneda 引理建立了自然变换集与 **Set** 中的对象之间的同构。自然变换是函子范畴 $[\mathbf{C}, \mathbf{Set}]$ 中的态射。任何两个函子之间的自然变换集是该范畴中的同态集。Yoneda 引理是一个同构：

$$[\mathbf{C}, \mathbf{Set}](\mathbf{C}(a, -), F) \cong Fa$$

这个同构在 F 和 a 中都具有自然性 (natural)。换句话说，它在 (F, a) 上是自然的，这是从积范畴 $[\mathbf{C}, \mathbf{Set}] \times \mathbf{C}$ 中选取的一对。注意，我们现在将 F 视为函子范畴中的对象 (object)。

让我们稍微想一下这意味着什么。自然同构是两个函子之间可逆的自然变换。而且，确实，我们同构的右侧是一个函子。它是从 $[\mathbf{C}, \mathbf{Set}] \times \mathbf{C}$ 到 **Set** 的函子。它在一对 (F, a) 上的作用是一个集合——评估函子 F 在对象 a 处的结果。这个函子被称为评估函子 (evaluation functor)。

左侧也是一个函子，它将 (F, a) 映射到自然变换的集合 $[\mathbf{C}, \mathbf{Set}](\mathbf{C}(a, -), F)$ 。

为了表明这些确实是函子，我们还应该定义它们在态射上的作用。但在一对 (F, a) 和 (G, b) 之间的态射是什么？它是一对态射 (Φ, f) ；第一个是函子之间的态射——自然变换；第二个是 \mathbf{C} 中的常规态射。

评估函子将这一对 (Φ, f) 映射到两个集合 Fa 和 Gb 之间的一个函数。我们可以很容易地从 Φ 在 a 处的分量（将 Fa 映射到 Ga ）和 f 被 G 提升的态射构建这样一个函数：

$$(Gf) \circ \Phi_a$$

注意，由于 Φ 的自然性，这与以下内容相同：

$$\Phi_b \circ (Ff)$$

我不会证明整个同构的自然性——一旦你确定了函子是什么，证明就相当机械化。因为我们的同构是由函子和自然变换构建的，所以它没有出错的可能。

16.5 挑战 (Challenges)

1. 用 Haskell 表达 co-Yoneda 嵌入。
2. 证明我们在 和 之间建立的双射是一个同构（两个映射互为逆）。
3. 计算出单子 (monoid) 的 Yoneda 嵌入。哪个函子对应于单子的单个对象？哪些自然变换对应于单子的态射？
4. 协变 Yoneda 嵌入在预序中的应用是什么？（问题由 Gershon Bazerman 提出。）
5. 可以使用 Yoneda 嵌入将任意函子范畴 $[\mathbf{C}, \mathbf{D}]$ 嵌入到函子范畴 $[[\mathbf{C}, \mathbf{D}], \mathbf{Set}]$ 中。弄清楚它如何在态射上工作（在这种情况下，这些态射是自然变换）。

Part Three

17

It's All About Morphisms

如果我还没有让你相信范畴论 (category theory) 完全是关于态射 (morphisms) 的，那么说明我还没有做好我的工作。因为下一个主题是伴随 (adjunctions)，它是以同态集 (hom-sets) 的同构 (isomorphisms) 来定义的，因此回顾一下我们对同态集构建块的直觉是有意义的。此外，你会看到伴随提供了一种更通用的语言来描述我们之前学习的许多构造，所以复习它们也可能有所帮助。

17.1 函子 (Functors)

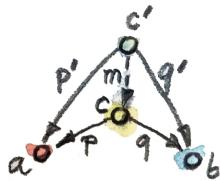
首先，你确实应该把函子 (functors) 看作是态射的映射——这是 Haskell 中 类型类的定义所强调的观点，该定义围绕 展开。当然，函子也映射对象——态射的端点——否则我们将无法讨论保持组合性。对象告诉我们哪些态射对可以组合。如果我们希望态射的组合被映射到“提升” (lifted) 态射的组合，那么它们端点的映射几乎是确定的。

17.2 交换图 (Commuting Diagrams)

许多态射的性质都是用交换图来表达的。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，那么我们就有一个交换图。

特别地，交换图构成了几乎所有泛性质构造的基础（初始对象和终端对象是显著的例外）。我们已经在积 (product)、余积 (coproduct)、各种其他 (余) 极限 ((co-)limits)、指数对象 (exponential objects)、自由幺半群 (free monoids) 等的定义中看到了这一点。

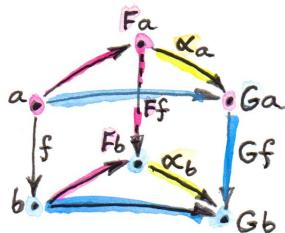
积是一个简单的泛性质构造的例子。我们选择两个对象 a 和 b ，看看是否存在一个对象 c 以及一对态射 p 和 q ，它具有作为它们积的泛性质。



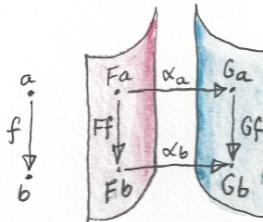
积是极限的一个特殊情况。极限是用圆锥（cones）来定义的。一般的圆锥是由交换图构成的。那些图的交换性可能被适当的自然性条件替代，用于函子的映射。这样，交换性被简化为自然变换高级语言的汇编语言。

17.3 自然变换 (Natural Transformations)

通常，当我们需要从态射映射到交换方块时，自然变换非常方便。自然性方块的两个对立边是某个态射 f 在两个函子 F 和 G 下的映射。其他边是自然变换的分量（它们也是态射）。



自然性意味着当你移动到“邻近”组件时（通过态射连接的邻近），你并没有违背范畴或函子的结构。无论你是先使用自然变换的分量来弥合对象之间的差距，然后再使用函子跳到邻居，还是反过来，都是无所谓的。两个方向是正交的。自然变换使你向左和向右移动，而函子使你上下移动，或者说前后移动——可以这么说。你可以将函子的像（image）可视化为目标范畴中的一张纸。自然变换将对应于 F 的一张纸映射到对应于 G 的另一张纸。



我们已经在 Haskell 中看到了这种正交性的例子。在那里，函子的作用是修改容器的内容而不改变其形状，而自然变换则将未更改的内容重新打包到另一个容器中。这些操作的顺序并不重要。

我们已经在极限的定义中看到了圆锥被自然变换替换的情况。自然性确保了每个圆锥的边都是交换的。然而，极限仍然是以圆锥之间的映射来定义的。这些映射也必须满足交换性条件。（例如，积的定义中的三角形必须是交换的。）

这些条件也可以用自然性来替代。你可能还记得泛（universal）圆锥，或极限，被定义为（逆变）同态函子之间的自然变换：

$$F :: c \rightarrow \mathbf{C}(c, \mathbf{Lim}D)$$

以及映射范畴 \mathbf{C} 中对象到圆锥的（也是逆变的）函子，而这些圆锥本身就是自然变换：

$$G :: c \rightarrow \mathbf{Nat}(\Delta_c, D)$$

其中， Δ_c 是常函子， D 是在 \mathbf{C} 中定义图的函子。两个函子 F 和 G 在 \mathbf{C} 中的态射上具有明确的作用。事实上，这个特殊的自然变换 F 和 G 之间是一个同构（isomorphism）。

17.4 自然同构 (Natural Isomorphisms)

自然同构——即每个分量都是可逆的自然变换——是范畴论中表示“两物相同”的方式。此类变换的分量必须是对象之间的同态态射——即具有逆的态射。如果你将函子像可视化为纸张，自然同构是一对一的可逆映射。

17.5 同态集 (Hom-Sets)

但是态射是什么呢？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趣的集合（至少对于局部小范畴来说是这样）。我们可以给这个集合中的元素起名字，比如 f 或 g ，以区分一个态射和另一个态射——但是什么使它们不同呢？

在给定同态集中，态射之间的本质区别在于它们与其他态射（来自相邻同态集的态射）的组合方式。如果存在一个态射 h ，其与 f 的组合（无论是前组合还是后组合）与与 g 的组合不同，例如：

$$h \circ f \neq h \circ g$$

那么我们可以直接“观察”到 f 和 g 之间的差异。但即使这种差异不是直接可观察到的，我们也可以用函子来放大同态集。函子 F 可能将两个态射映射到不同的态射：

$$Ff \neq Fg$$

在一个更丰富的范畴中，其中相邻的同态集提供了更多的分辨率，例如，

$$h' \circ Ff \neq h' \circ Fg$$

其中， h' 不在 F 的像中。

17.6 同态集同构 (Hom-Set Isomorphisms)

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，集合之间的普通同构并不能告诉你很多信息。对于有限集合，同构只说明它们具有相同数量的元素。如果集合是无限的，那么它们的基数必须相同。但是，任何有意义的同态集同构都必须考虑组合。组合涉及的不仅仅是一个同态集。我们需要定义跨越整个同态集集合的同构，并且我们需要施加一些与组合相互操作的兼容性条件。而自然同构完全符合要求。

但是，同态集的自然同构是什么呢？自然性是函子之间的映射的性质，而不是集合的。所以我们实际上是在谈论同态集值函子之间的自然同构。这些函子不仅仅是集合值的函子。它们在态射上的作用是由适当的同态函子诱导的。态射通过同态函子用前组合或后组合（取决于函子的协变性）来进行规范映射。

Yoneda 嵌入就是这种同构的一个例子。它将 \mathbf{C} 中的同态集映射到函子范畴中的同态集，并且它是自然的。Yoneda 嵌入中的一个函子是 \mathbf{C} 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的集合。

极限的定义也是同态集之间的自然同构（第二个也是在函子范畴中）：

$$\mathbf{C}(c, \mathbf{Lim}D) \simeq \mathbf{Nat}(\Delta_c, D)$$

事实证明，我们对指数对象的构造或对自由幺半群的构造也可以重写为同态集之间的自然同构。

这并非巧合——我们接下来会看到，这些只是伴随的不同例子，而伴随被定义为同态集的自然同构。

17.7 同态集的不对称性 (Asymmetry of Hom-Sets)

还有一个观察将帮助我们理解伴随。一般来说，同态集并不是对称的。同态集 $C(a, b)$ 通常与同态集 $C(b, a)$ 非常不同。部分顺序作为范畴时，是这种不对称性的最终证明。在部分顺序中，当且仅当 a 小于或等于 b 时，从 a 到 b 的态射才存在。如果 a 和 b 是不同的，那么就不能存在从 b 到 a 的态射。所以，如果同态集 $C(a, b)$ 是非空的，在这种情况下意味着它是单元素集合，那么 $C(b, a)$ 必须是空的，除非 $a = b$ 。在这个范畴中，箭头的流动方向是确定的。

预序 (preorder)，基于不一定是反对称的关系，也是“主要”方向性的，除了偶尔的循环。将任意范畴视为预序的广义化是很方便的。

预序是一个稀薄范畴——所有同态集要么是单元素的，要么是空的。我们可以将一般范畴可视化为“厚”预序。

17.8 挑战 (Challenges)

1. 考虑自然性条件的一些退化情况并绘制相应的图。例如，如果函子 F 或 G 将对象 a 和 b （即态射 $f :: a \rightarrow b$ 的两端）映射到同一对象，例如 $Fa = Fb$ 或 $Ga = Gb$ ，会发生什么？（注意，通过这种方式你会得到一个圆锥或余圆锥）。然后，考虑 $Fa = Ga$ 或 $Fb = Gb$ 的情况。最后，如果你从一个循环自身的态射开始，即 $f :: a \rightarrow a$ ，会发生什么？

18

Adjunctions

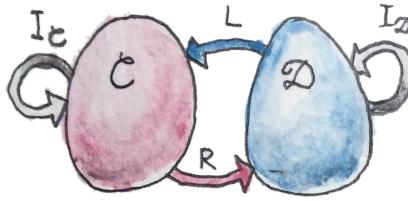
在数学中，我们有多种方式来表达一件事物与另一件事物相似。最严格的方式是相等（equality）。如果没有任何方式可以区分两个事物，那么它们就是相等的。在任何可以想象的情况下，一个可以被替换为另一个。例如，你是否注意到，每当我们谈到交换图（commuting diagrams）时，我们使用的是态射（morphisms）的相等性？这是因为态射形成一个集合（同态集（hom-set）），而集合的元素可以进行相等性比较。

但相等性往往过于严格。有许多事例表明，事物在所有实际用途上都是相同的，但实际上并不相等。例如，类型对严格来说并不等于，但我们理解它们包含相同的信息。这个概念最好用两个类型之间的同构（isomorphism）来描述——这是一个可逆的态射。既然它是一个态射，它就保留了结构；而“同构”意味着它是一个往返行程的一部分，无论从哪一边开始，都会回到原来的位置。在成对类型的情况下，这个同构称为：

snippet01 恰好是它自己的逆。

18.1 伴随和单位/余单位对 (Adjunction and Unit/-Counit Pair)

当我们谈论范畴是同构的时，我们用范畴之间的映射来表达，也就是函子（functors）。我们希望能够说两个范畴 **C** 和 **D** 是同构的，如果存在一个从 **C** 到 **D** 的函子 R （“右”），它是可逆的。换句话说，存在另一个从 **D** 回到 **C** 的函子 L （“左”），当它与 R 组合时，等于恒等函子 I 。有两种可能的组合， $R \circ L$ 和 $L \circ R$ ；以及两个可能的恒等函子：一个在 **C** 中，另一个在 **D** 中。



但这里有一个棘手的问题：两个函子相等是什么意思？我们所说的相等是指：

$$R \circ L = I_D$$

或者这个：

$$L \circ R = I_C$$

将函子的相等性定义为对象的相等性是合理的。当两个函子作用于相等的对象时，它们应该产生相等的对象。但在一般情况下，我们没有在任意范畴中定义对象相等性的概念。它不属于范畴定义的一部分。（如果我们深入探讨“相等性到底是什么”这个问题，我们将进入同伦类型论（Homotopy Type Theory）这个深渊。）

你可能会认为函子是范畴范畴中的态射，因此它们应该是可以进行相等性比较的。确实，只要我们讨论的是小范畴，其中的对象形成一个集合，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。

但请记住，**Cat** 实际上是一个 2-范畴。在一个 2-范畴中，同态集（hom-sets）具有附加结构——在 1-态射之间存在 2-态射。在 **Cat** 中，1-态射是函子，而 2-态射是自然变换。因此，当谈到函子时，考虑自然同构作为相等的替代是更自然的（避免不了这个双关语！）。

所以，与其考虑范畴的同构，不如考虑更广泛的等价（equivalence）概念。如果我们能找到两个函子在它们之间来回映射，并且它们的组合（任意一种方式）自然同构于恒等函子，我们就说两个范畴 **C** 和 **D** 是等价的。换句话说，在 $R \circ L$ 与恒等函子 I_D 之间存在一个双向自然变换，在 $L \circ R$ 与恒等函子 I_C 之间也存在一个双向自然变换。

伴随（Adjunction）甚至比等价更弱，因为它不要求两个函子的组合同构于恒等函子。相反，它规定存在从 I_D 到 $R \circ L$ 的单向自然变换，以及从 $L \circ R$ 到 I_C 的另一个自然变换。以下是这两个自然变换的形式：

$$\eta : I_D \rightarrow R \circ L$$

$$\varepsilon : L \circ R \rightarrow I_C$$

η 被称为单位（unit），而 ε 被称为余单位（counit）。

注意这两个定义之间的不对称性。一般来说，我们没有两个剩余的映射：

$$R \circ L \rightarrow I_D \quad \text{不一定}$$

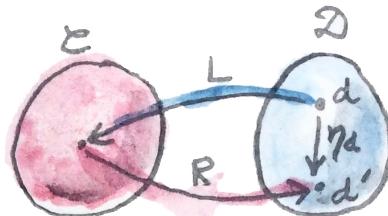
$$I_C \rightarrow L \circ R \quad \text{不一定}$$

由于这种不对称性，函子 L 被称为 R 的左伴随，而函子 R 是 L 的右伴随 (right adjoint)。当然，左和右只有当你以特定方式绘制图表时才有意义。

伴随的紧凑表示是：

$$L \dashv R$$

为了更好地理解伴随，让我们更详细地分析单位和余单位。

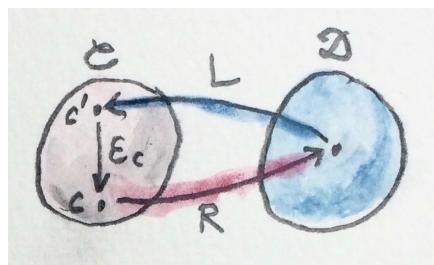


让我们从单位开始。它是一个自然变换，因此它是一组态射的集合。给定一个在 \mathbf{D} 中的对象 d , η 的分量是 Id 和 $(R \circ L)d$ 之间的态射；在图中，它被称为 d' ：

$$\eta_d :: d \rightarrow (R \circ L)d$$

注意，组合 $R \circ L$ 是在 \mathbf{D} 中的一个自函子 (endofunctor)。

这个方程告诉我们，我们可以选择 \mathbf{D} 中的任何对象 d 作为起点，并使用往返函子 $R \circ L$ 选择目标对象 d' 。然后我们发射一支箭——态射 η_d ——到达目标。



同样地，余单位 ε 的分量可以描述为：

$$\varepsilon_c :: (L \circ R)c \rightarrow c$$

它告诉我们，我们可以选择 \mathbf{C} 中的任何对象 c 作为目标，并使用往返函子 $L \circ R$ 选择源对象 $c' = (L \circ R)c$ 。然后我们发射一支箭——态射 ε_c ——从源对象到目标对象。

另一种看待单位和余单位的方式是，单位让我们在 \mathbf{D} 上的任何地方引入组合 $R \circ L$ ，而余单位让我们消除组合 $L \circ R$ ，用 \mathbf{C} 上的恒等函子

替换它。这导致了一些“显而易见”的一致性条件，这些条件确保引入和消除的顺序不会改变任何东西：

$$L = L \circ I_D \rightarrow L \circ R \circ L \rightarrow I_C \circ L = L$$

$$R = I_D \circ R \rightarrow R \circ L \circ R \rightarrow R \circ I_C = R$$

这些被称为三角恒等式（triangular identities），因为它们使以下图表交换：

$$\begin{array}{ccc} L & \xrightarrow{L \circ \eta} & L \circ R \circ L \\ & \searrow \quad \swarrow & \downarrow \epsilon \circ L \\ & & L \end{array} \qquad \begin{array}{ccc} R & \xrightarrow{\eta \circ R} & R \circ L \circ R \\ & \searrow \quad \swarrow & \downarrow R \circ \epsilon \\ & & R \end{array}$$

这些是在函子范畴中的图：箭头是自然变换，它们的组合是自然变换的水平组合。用分量来表示，这些恒等式变为：

$$\varepsilon_{Ld} \circ L\eta_d = \mathbf{id}_{Ld}$$

$$R\varepsilon_c \circ \eta_{Rc} = \mathbf{id}_{Rc}$$

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（或在 定义中称为）：

snippet02 而余单位则称为：

snippet03 在这里，是对应于 $R \circ L$ 的（自）函子，而 是对应于 $L \circ R$ 的（自）函子。正如我们稍后会看到的，它们分别是单子（monad）和余单子（comonad）定义的一部分。

如果你将自函子视为一个容器，单位（或）是一个多态函数，它为任意类型的值创建一个默认的盒子。余单位（或）则是相反的：它从容器中检索或生成单一值。

我们稍后会看到，每对伴随函子都会定义一个单子和一个余单子。反过来，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一对伴随函子，主要是因为这些函子通常会将我们带出 **Hask**。

然而，我们可以在 Haskell 中定义自函子的伴随关系。以下是取自的部分定义：

snippet04 此定义需要一些解释。首先，它描述了一个多参数类型类——这两个参数是 和。它建立了 之间的关系。

纵向条件（在竖线之后）指定了功能依赖性。例如，意味着 是由决定的（ 和 之间的关系是一个函数，这里是在类型构造函数上）。相反，意味着如果我们知道，那么 是唯一确定的。

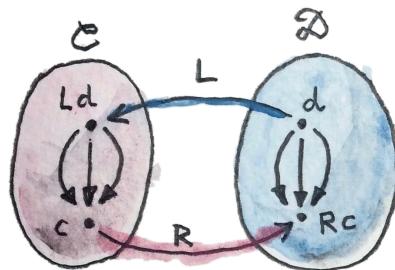
稍后我将解释为什么在 Haskell 中，我们可以施加条件使得右伴随是一个可表函子。

18.2 伴随和同态集 (Adjunctions and Hom-Sets)

有一个等价的定义是基于同态集自然同构的伴随定义。这个定义与我们到目前为止研究的通用构造非常吻合。每次你听到某个唯一态射的声明，它因某种结构而因子化时，你应该将其视为从某个集合到同态集的映射。这就是“选择唯一态射”的意义。

此外，因子化通常可以通过自然变换来描述。因子化涉及交换图——某个态射等于两个态射（因子）的组合。自然变换将态射映射到交换图上。因此，在通用构造中，我们从态射到交换图，然后到唯一态射。我们最终得到的是从态射到态射的映射，或者从一个同态集到另一个（通常是在不同范畴中）。如果这个映射是可逆的，并且它可以自然地扩展到所有同态集上，我们就有了一个伴随。

通用构造和伴随之间的主要区别在于后者是全局定义的——适用于所有同态集。例如，使用通用构造你可以定义两个特定对象的积，即使在该范畴中的其他对象对之间不存在积。正如我们即将看到的，如果在一个范畴中任意一对对象的积存在，它也可以通过伴随来定义。



这是使用同态集定义伴随的另一种方式。与之前一样，我们有两个函子 $L : \mathbf{D} \rightarrow \mathbf{C}$ 和 $R : \mathbf{C} \rightarrow \mathbf{D}$ 。我们选择两个任意的对象：在 \mathbf{D} 中的源对象 d 和在 \mathbf{C} 中的目标对象 c 。我们可以使用 L 将源对象 d 映射到 \mathbf{C} 。现在我们在 \mathbf{C} 中有两个对象， Ld 和 c 。它们定义了一个同态集：

$$\mathbf{C}(Ld, c)$$

类似地，我们可以使用 R 将目标对象 c 映射到 \mathbf{D} 。现在我们在 \mathbf{D} 中有两个对象， d 和 Rc 。它们也定义了一个同态集：

$$\mathbf{D}(d, Rc)$$

我们说 L 是 R 的左伴随当且仅当存在一个同态集之间的同构：

$$\mathbf{C}(Ld, c) \cong \mathbf{D}(d, Rc)$$

这个同构在 d 和 c 上都是自然的。自然性意味着源对象 d 可以在 \mathbf{D} 中平滑变化；目标对象 c 可以在 \mathbf{C} 中平滑变化。更精确地说，我们有一个

自然变换 φ 在以下两个（协变）函子之间，从 \mathbf{C} 到 \mathbf{Set} 。这是这些函子在对象上的作用：

$$\begin{aligned} c &\rightarrow \mathbf{C}(Ld, c) \\ c &\rightarrow \mathbf{D}(d, Rc) \end{aligned}$$

另一个自然变换 ψ 作用在以下（逆变）函子之间：

$$\begin{aligned} d &\rightarrow \mathbf{C}(Ld, c) \\ d &\rightarrow \mathbf{D}(d, Rc) \end{aligned}$$

这两个自然变换都必须是可逆的。

很容易证明这两个伴随定义是等价的。例如，让我们从同态集同构开始推导单位变换：

$$\mathbf{C}(Ld, c) \cong \mathbf{D}(d, Rc)$$

由于这个同构适用于任何对象 c ，它也必须适用于 $c = Ld$ ：

$$\mathbf{C}(Ld, Ld) \cong \mathbf{D}(d, (R \circ L)d)$$

我们知道，左边至少必须包含一个态射，恒等态射。自然变换将这个态射映射到 $\mathbf{D}(d, (R \circ L)d)$ 中的一个元素，或者插入恒等函子 I ，即：

$$\mathbf{D}(Id, (R \circ L)d)$$

我们得到一个由 d 参数化的态射族。它们构成了从函子 I 到函子 $R \circ L$ 之间的自然变换（自然性条件很容易验证）。这正是我们的单位 η 。

反过来，从单位和余单位的存在开始，我们可以定义同态集之间的变换。例如，我们选择一个在同态集 $\mathbf{C}(Ld, c)$ 中的任意态射 f 。我们想定义一个 φ ，它作用在 f 上，生成一个在 $\mathbf{D}(d, Rc)$ 中的态射。

实际上我们没有太多选择。我们可以尝试使用 R 提升 f 。这将生成一个态射 Rf ，从 $R(Ld)$ 到 Rc ——这是 $\mathbf{D}((R \circ L)d, Rc)$ 中的一个态射。

我们需要的是一个 φ 的分量，一个从 d 到 Rc 的态射。这不是问题，因为我们可以使用 η_d 的一个分量从 d 到 $(R \circ L)d$ 。我们得到：

$$\varphi_f = Rf \circ \eta_d$$

另一个方向类似， ψ 的推导也是如此。

回到 Haskell 中的定义，自然变换 φ 和 ψ 被分别替换为多态的（在和上）函数 和。函子 L 和 R 分别被称为 和：

snippet05 在 / 表达和 / 表达之间的等价性由以下映射见证：

snippet06 从伴随的范畴描述到 Haskell 代码的转换是非常有启发性的。我强烈建议将其作为练习。

我们现在准备解释为什么在 Haskell 中右伴随自动是一个可表函子。原因是，在第一近似下，我们可以将 Haskell 类型的范畴视为集合的范畴。

当右范畴 **D** 是 **Set** 时，右伴随 R 是一个从 **C** 到 **Set** 的函子。如果我们可以找到 **C** 中的一个对象 rep ，使得同态函子 $\mathbf{C}(rep, _)$ 自然同构于 R ，则该函子是可表的。事实证明，如果 R 是某个函子 L 从 **Set** 到 **C** 的右伴随，这样的对象总是存在——它是单集合 $()$ 在 L 下的像：

$$rep = L()$$

确实，伴随告诉我们，以下两个同态集是自然同构的：

$$\mathbf{C}(L(), c) \cong \mathbf{Set}(\(), Rc)$$

对于给定的 c ，右侧是从单集合 $()$ 到 Rc 的函数集。我们前面已经看到，每个这样的函数从 Rc 中选择一个元素。这些函数的集合与 Rc 集合同构。因此我们有：

$$\mathbf{C}(L(), -) \cong R$$

这表明 R 确实是可表的。

18.3 从伴随到积 (Product from Adjunction)

我们之前已经使用通用构造介绍了几个概念。当这些概念在全局定义时，使用伴随表达起来更容易。最简单的非平凡例子是积 (product)。积的通用构造的要点是能够通过通用积因子化任何积状候选对象。

更确切地说，两个对象 a 和 b 的积是对象 $(a \times b)$ (或在 Haskell 中表示为)，配有两个态射 fst 和 snd ，使得对于任何其他候选对象 c ，配有两个态射 $p :: c \rightarrow a$ 和 $q :: c \rightarrow b$ ，存在一个唯一的态射 $m :: c \rightarrow (a, b)$ ，它因子化了 p 和 q 通过 fst 和 snd 。

如我们前面所见，在 Haskell 中，我们可以实现一个，它从两个投影生成这个态射：

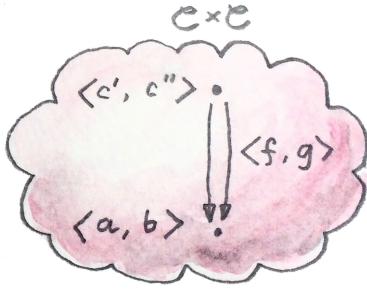
snippet07 验证因子化条件成立是很容易的：

snippet08 我们有一个映射，它接受一对态射 和 并生成另一个态射。

我们如何将其转化为定义伴随所需的同态集之间的映射？诀窍是走出 **Hask**，并将态射对视为在积范畴中的单个态射。

让我提醒你，什么是积范畴。取两个任意范畴 **C** 和 **D**。积范畴 **C × D** 中的对象是对象对，一个来自 **C**，另一个来自 **D**。态射是态射对，一个来自 **C**，一个来自 **D**。

为了在某个范畴 **C** 中定义积，我们应该从积范畴 **C × C** 开始。来自 **C** 的态射对是在积范畴 **C × C** 中的单个态射。



起初使用积范畴来定义积可能有点令人困惑。然而，这些积是非常不同的。我们不需要通用构造来定义积范畴。我们只需要对象对和态射对的概念。

然而，来自 \mathbf{C} 的对象对不是 \mathbf{C} 中的对象。它是一个不同范畴 $\mathbf{C} \times \mathbf{C}$ 中的对象。我们可以将该对形式化为 $\langle a, b \rangle$ ，其中 a 和 b 是 \mathbf{C} 的对象。另一方面，通用构造是必要的，以便在相同的范畴 \mathbf{C} 中定义对象 $a \times b$ （或在 Haskell 中表示为）。该对象应根据通用构造表示对 $\langle a, b \rangle$ 。它不总是存在，并且即使它存在于某些情况下，也可能不适用于 \mathbf{C} 中的其他对象对。

现在让我们将 看作同态集的映射。第一个同态集在积范畴 $\mathbf{C} \times \mathbf{C}$ 中，第二个在 \mathbf{C} 中。积范畴中的一般态射是态射对 $\langle f, g \rangle$ ：

$$\begin{aligned} f &:: c' \rightarrow a \\ g &:: c'' \rightarrow b \end{aligned}$$

其中 c'' 可能不同于 c' 。但为了定义积，我们对 $\mathbf{C} \times \mathbf{C}$ 中的一个特殊态射感兴趣，这对 p 和 q 共享相同的源对象 c 。这没问题：在伴随的定义中，左同态集的源不是任意对象——它是左函子 L 在右范畴的某个对象上作用的结果。符合条件的函子很容易猜测——它是从 \mathbf{C} 到 $\mathbf{C} \times \mathbf{C}$ 的对角函子（diagonal functor） Δ ，其对对象的作用是：

$$\Delta c = \langle c, c \rangle$$

我们伴随中的左侧同态集应为：

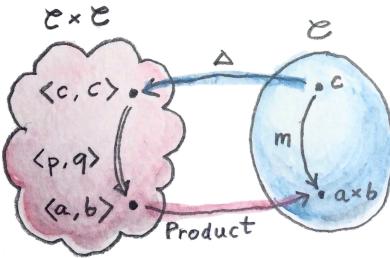
$$(\mathbf{C} \times \mathbf{C})(\Delta c, \langle a, b \rangle)$$

它是积范畴中的同态集。其元素是我们识别为 参数的态射对：

$$(c \rightarrow a) \rightarrow (c \rightarrow b) \dots$$

右侧同态集存在于 \mathbf{C} 中，它在源对象 c 和某些函子 R 的结果之间映射。该函子将对 $\langle a, b \rangle$ 映射到我们的积对象 $a \times b$ 。我们将该同态集的元素识别为 的结果：

$$\dots \rightarrow (c \rightarrow (a, b))$$



我们仍然没有完整的伴随。为此，我们首先需要是可逆的——我们正在构建同态集之间的同构。逆的应该从某个对象 c 到积对象 $a \times b$ 的态射 m 开始。换句话说， m 应该是：

$$C(c, a \times b)$$

逆因子应该将 m 映射到 $C \times C$ 中的态射对 $\langle p, q \rangle$ ，该对从 $\langle c, c \rangle$ 到 $\langle a, b \rangle$ ；换句话说，该态射是：

$$(C \times C)(\Delta c, \langle a, b \rangle)$$

如果该映射存在，则我们得出结论对角函子的右伴随存在。该函子定义一个积。

在 Haskell 中，我们可以通过分别与 和 组合构造 的逆函数。

要完成两种定义积方式的等价性证明，我们还需要证明同态集之间的映射在 a 、 b 和 c 上是自然的。我将此留作给有志读者的练习。

总结我们所做的：一个范畴积可以全局定义为对角函子的右伴随：

$$(C \times C)(\Delta c, \langle a, b \rangle) \cong C(c, a \times b)$$

这里 $a \times b$ 是我们右伴随函子 *Product* 对对象对 $\langle a, b \rangle$ 的作用结果。请注意，任何从 $C \times C$ 到 C 的函子都是一个双函子 (bifunctor)，因此 *Product* 是一个双函子。在 Haskell 中，*Product* 双函子简单地表示为。你可以将它应用于两个类型并得到它们的积类型，例如：

snippet09

18.4 从伴随到指数对象 (Exponential from Adjunction)

指数对象 b^a 或函数对象 $a \Rightarrow b$ 可以使用通用构造来定义。如果这种构造存在于所有对象对中，它可以看作一个伴随。再一次，诀窍是集中注意以下声明：

对于任何其他对象 z 及其态射 $g :: z \times a \rightarrow b$, 存在一个唯一的态射 $h :: z \rightarrow (a \Rightarrow b)$

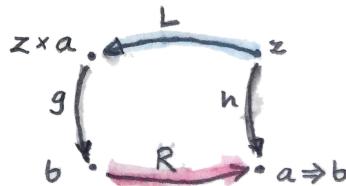
该声明建立了同态集之间的映射。

在这种情况下, 我们处理的是同一范畴中的对象, 因此两个伴随函子是自函子。左(自)函子 L 作用在对象 z 上生成 $z \times a$ 。这是一个对应于固定某个 a 的积函子。

右(自)函子 R 作用在对象 b 上生成函数对象 $a \Rightarrow b$ (或 b^a)。再一次, a 是固定的。该伴随函子通常写为:

$$- \times a \dashv (-)^a$$

该伴随函子下同态集的映射在我们用通用构造的图中最好体现出来。



注意到 eval 态射¹其实就是这个伴随的余单位:

$$(a \Rightarrow b) \times a \rightarrow b$$

其中:

$$(a \Rightarrow b) \times a = (L \circ R)b$$

我之前提到过, 通用构造定义了一个唯一的对象, 至多是同构的。这就是为什么我们有“积”和“指数对象”。这个属性也适用于伴随: 如果一个函子有一个伴随, 这个伴随至多是同构的。

18.5 挑战

- 推导出自然变换 ψ 的自然性方框, 该变换作用于以下两个(逆变)函子之间:

$$a \rightarrow \mathbf{C}(La, b)$$

$$a \rightarrow \mathbf{D}(a, Rb)$$

- 从伴随的第二个定义中同态集同构开始推导余单位 ε 。
- 完成伴随的两种定义等价性的证明。
- 证明可以通过一个伴随来定义余积。从余积的因子化定义开始。
- 证明余积是对角函子的左伴随。
- 在 Haskell 中定义一个积与函数对象之间的伴随关系。

¹参见第 9 章 通用构造。

19

Free/Forgetful Adjunctions

由构造是伴随子的一个强大应用。一个自由函子定义为遗忘函子的左伴随。遗忘函子通常是一个非常简单的函子，它遗忘了一些结构。例如，许多有趣的范畴都是基于集合构建的。但是，抽象这些集合的范畴对象没有内部结构——它们没有元素。然而，这些对象通常保留集合的记忆，即存在一个从给定范畴 \mathbf{C} 到 \mathbf{Set} 的映射——一个函子。对应于 \mathbf{C} 中某个对象的集合称为它的底层集合。

幺半群（monoids）是这样的对象，它们有底层集合——元素的集合。存在一个从幺半群范畴 \mathbf{Mon} 到集合范畴 \mathbf{Set} 的遗忘函子 U ，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）映射为集合之间的函数。

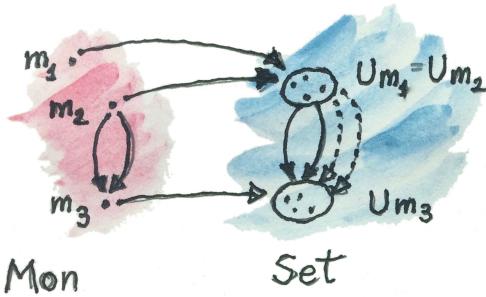
我喜欢把 \mathbf{Mon} 看作有双重人格。一方面，它是一堆带有乘法和单位元素的集合。另一方面，它是一个范畴，具有无结构的对象，这些对象的唯一结构编码在它们之间的态射中。每一个保留乘法和单位的集合函数都会产生一个 \mathbf{Mon} 中的态射。

需要记住的几点：

- 可能有许多幺半群映射到同一个集合，并且
- 幺半群态射的数量比它们底层集合之间的函数少（或至多相等）。

作为遗忘函子 U 的左伴随的函子 F 是构建自由幺半群的自由函子。这个伴随关系来源于我们之前讨论过的自由幺半群的通用构造。¹

¹参见第 13 章 自由幺半群。



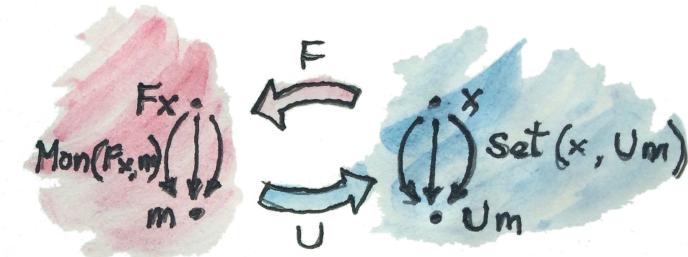
么半群 m_1 和 m_2 具有相同的底层集合。 m_2 和 m_3 的底层集合之间的函数比它们之间的态射要多。

在同态集的术语中，我们可以将这个伴随子写为：

$$\mathbf{Mon}(Fx, m) \cong \mathbf{Set}(x, Um)$$

这个（在 x 和 m 上是自然的）同构告诉我们：

- 对于自由么半群 Fx 和任意么半群 m 之间的每一个么半群同态映射，存在一个唯一的函数将生成元集合 x 嵌入到 m 的底层集合中。它是 $\mathbf{Set}(x, Um)$ 中的一个函数。
- 对于将 x 嵌入某个 m 的底层集合中的每一个函数，存在一个唯一的么半群态射在自由么半群 Fx 和么半群 m 之间。（这是我们在通用构造中称为 h 的那个态射。）



直观上， Fx 是可以基于 x 构建的“最大”么半群。如果我们可以看到么半群的内部，我们会发现任何属于 $\mathbf{Mon}(Fx, m)$ 的态射都将这个自由么半群嵌入到某个其他么半群 m 中。它可能通过标识一些元素来实现这一点。特别是，它将 Fx 的生成元（即 x 的元素）嵌入 m 中。这个伴随关系表明，右侧由 $\mathbf{Set}(x, Um)$ 中的函数给出的 x 的嵌入唯一决定了左侧么半群的嵌入，反之亦然。

在 Haskell 中，列表数据结构是一个自由么半群（有一些警告：参见 Dan Doel 的博客文章²）。列表类型 是一个自由么半群，其中类型

²

代表生成元的集合。例如，类型 包含单位元素——空列表，以及单元素列表如，——这些是自由幺半群的生成元。其余部分通过应用“乘积”生成。在这里，两个列表的乘积只是将一个附加到另一个。附加是结合律和单位律（即这里有一个中立元素——空列表）成立的。由 生成的自由幺半群不过是所有字符列表的集合。在 Haskell 中，它被称为：

snippet01 (定义了一个类型同义词——一个现有类型的不同名称。)

另一个有趣的例子是从单一生成元构建的自由幺半群。它是由 组成的列表类型。它的元素是,, , 等等。每个这样的列表都可以由一个自然数描述——它的长度。列表中的单位并没有更多的信息。附加两个这样的列表会产生一个新列表，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）同构。以下是两个互为逆函数，见证了这种同构：

snippet02 为简单起见，我使用了类型 而不是，但原理是相同的。函数 创建一个长度为 的列表，预先填充了给定的值——在这里是单位值。

19.1 一些直观的理解

接下来的内容是一些随意的推理。这些推理远不严格，但有助于形成直观理解。

要获得对自由/遗忘伴随子的直观理解，帮助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捆绑在一起。此外，它们的像可能只覆盖其陪域的一部分。

在 **Set** 中的“平均”同态集将包含从最不损失的函数（例如，注入函数或可能的同构）到将整个定义域折叠为单一元素的常量函数（如果有的话）的整个谱系。

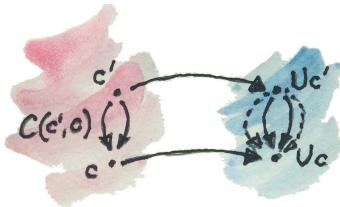
我倾向于将任意范畴中的态射也视为有损失的。这只是一种心理模型，但它是有用的，特别是在考虑伴随子时——尤其是其中一个范畴是 **Set** 的情况。

形式上，我们只能谈论可逆的（同构的）或不可逆的态射。后者的态射可以被认为是有损失的。还有一种单态（mono）和满态（epi）态射的概念，它们概括了注入（非合并）和满射（覆盖整个陪域）的函数的概念，但仍然可能有一个既是单态又是满态但仍然不可逆的态射。

在自由 \dashv 遗忘伴随子中，我们在左侧有一个更受约束的范畴 **C**，在右侧有一个不太受约束的范畴 **D**。**C** 中的态射“更少”，因为它们必须保留一些额外的结构。而 **D** 中的态射不必保留那么多结构，因此它们“更多”。

当我们将遗忘函子 U 应用于 **C** 中的一个对象 c 时，我们认为它揭示了 c 的“内部结构”。事实上，如果 **D** 是 **Set**，我们认为 U 是定义 c 的内部结构——它的底层集合。（在任意范畴中，我们无法谈论对象的内部，除了通过它与其他对象的连接，但这里我们只是在随意推理。）

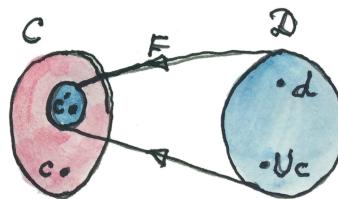
如果我们使用 U 映射两个对象 c' 和 c , 我们通常期望同态集 $\mathbf{C}(c', c)$ 的映射只覆盖 $\mathbf{D}(Uc', Uc)$ 的一个子集。这是因为 $\mathbf{C}(c', c)$ 中的态射必须保留额外的结构, 而 $\mathbf{D}(Uc', Uc)$ 中的态射则不必。



但是由于伴随子被定义为特定同态集的同构, 我们必须非常仔细地选择 c' 。在伴随子中, c' 不是从 \mathbf{C} 中的任意位置选取的, 而是从自由函子 F 的 (可能较小的) 像中选取的:

$$\mathbf{C}(Fd, c) \cong \mathbf{D}(d, Uc)$$

因此, F 的像必须由有许多态射可以通向任意 c 的对象组成。事实上, 必须有尽可能多的结构保留态射从 Fd 到 c , 因为有不保留结构的态射从 d 到 Uc 。这意味着 F 的像必须由本质上没有结构的对象组成 (因此没有结构需要态射保留)。这样的“无结构”对象称为自由对象。



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, 所有的乘法都会生成全新的元素。

在自由幺半群中, $2 * 3$ 不是 6 ——它是一个新元素 $[2, 3]$ 。由于没有 $[2, 3]$ 和 6 的同一性, 从这个自由幺半群到任何其他幺半群 m 的态射被允许将它们单独映射。但是也可以将 $[2, 3]$ 和 6 (它们的积) 映射到 m 的同一个元素中。或者在一个加法幺半群中, 将 $[2, 3]$ 和 5 (它们的和) 标识为相同的元素, 依此类推。不同的标识会给你不同的幺半群。

这引出了另一个有趣的直觉: 自由幺半群不是执行幺半群操作, 而是积累传递给它的参数。它不是将 2 和 3 相乘, 而是记住 2 和 3 在一个列表中。这种方案的优点是我们不必指定我们将使用哪种幺半群操作。我们可以继续积累参数, 只有在最后才对结果应用操作符。此时我们可以选择应用哪个操作符。我们可以加这些数字, 或将它们相乘, 或进行模 2 加法, 等等。自由幺半群将表达式的创建与其求值分开。当我们讨论代数时, 我们会再次看到这一想法。

这种直觉可以推广到其他更复杂的自由构造。例如，我们可以在求值之前积累整个表达式树。这样做的好处是我们可以转换这些树，使求值更快或更少消耗内存。例如，在实现矩阵计算时，这样做可以避免为存储中间结果而分配大量临时数组。

19.2 挑战

1. 考虑一个从单一生成元构建的自由幺半群。证明从这个自由幺半群到任何幺半群 m 的态射与从单一集合到 m 的底层集合的函数之间存在一一对应关系。

20

Monads: Programmer's Definition

程序员们围绕 MONAD 发展出了一整套神话。它被认为是编程中最抽象和最难理解的概念之一。有些人“明白了”，而有些人则不然。对于许多人来说，理解 monad 的那一刻就像是一种神秘的体验。Monad 抽象了许多不同构造的本质，以至于我们在日常生活中没有一个好的比喻来描述它。我们只能像那些盲人摸象一样，触摸大象的不同部分，然后胜利地宣称：“它是一根绳子”、“它是一棵树干”或“它是一个墨西哥卷饼！”

让我把事情说清楚：围绕 monad 的所有神秘感其实是由于误解。Monad 是一个非常简单的概念。造成混淆的是 monad 的应用多种多样。

作为这篇文章的研究内容的一部分，我查阅了胶带（又称鸭绒胶带）及其应用。以下是你可以用它做的一些事情的小样本：

- 密封管道
- 修理阿波罗 13 号上的 CO₂ 清洁器
- 疣的治疗
- 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，如果你不知道胶带是什么，并试图根据这份清单弄清楚它是什么。祝你好运！

因此，我想补充一项内容到“monad 就像……”的陈词滥调中：monad 就像胶带。它的应用范围广泛，但其原理非常简单：它将事物粘合在一起。更确切地说，它将事物组合起来。

这在一定程度上解释了许多程序员，尤其是那些来自命令式背景的程序员，理解 monad 的困难。问题在于我们不习惯于将编程视为函数组合的过程。这是可以理解的。我们经常给中间值命名，而不是直接将它们从一个函数传递到另一个函数。我们也会内联短段的胶合代码，

而不是将其抽象为辅助函数。以下是用 C 语言编写的矢量长度函数的命令式实现：

将此与 Haskell 的一个（风格化的）版本相比，该版本明确了函数组合：

snippet01 (这里，为了让事情变得更加神秘，我通过将第二个参数设置为 来部分应用指数运算符。)

我并不是在争论 Haskell 的无点风格总是更好，只是说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 确实尽了很大的努力，提供了名为 语法糖的命令式风格语法，用于 `monad` 组合。稍后我们将看到它的使用。但首先，让我解释为什么我们首先需要 `monad` 组合。

20.1 Kleisli 范畴 (The Kleisli Category)

我们之前通过装饰普通函数得到了 `writer monad`。特别的装饰是通过将它们的返回值与字符串或更一般的，单态元素配对完成的。现在我们可以认识到这种装饰是一个函子：

snippet02 随后我们找到了组合装饰函数或 Kleisli 箭头的方法，Kleisli 箭头是形如：

snippet03 的函数。我们在组合内部实现了日志的累积。

我们现在准备好给出 Kleisli 范畴的更一般定义。我们从一个范畴 **C** 和一个自函子 m 开始。对应的 Kleisli 范畴 **K** 具有与 **C** 相同的对象，但它的态射不同。在 **K** 中两个对象 a 和 b 之间的态射由原范畴 **C** 中的态射 $a \rightarrow m b$ 实现。需要注意的是，我们将 **K** 中的 Kleisli 箭头视为 a 和 b 之间的态射，而不是 a 和 $m b$ 之间的态射。

在我们的例子中， m 被专门化为某个固定的幺半群 的。

只有当我们能为 Kleisli 箭头定义合适的组合时，它们才能形成一个范畴。如果存在一个组合，它是结合的，并且每个对象都有一个单位箭头，那么函子 m 就称为 `monad` (单子)，结果范畴称为 Kleisli 范畴。

在 Haskell 中，Kleisli 组合使用鱼算子 定义，单位箭头是一个多态函数，称为。以下是使用 Kleisli 组合定义 `monad` 的方法：

snippet04 请记住，有许多等效的方式来定义 `monad`，而这并不是 Haskell 生态系统中的主要定义。我喜欢它的概念简单性和它提供的直觉，但还有其他定义在编程时更为方便。我们稍后会讨论它们。

在这种形式中，`monad` 定律很容易表达。它们不能在 Haskell 中强制执行，但它们可以用于等式推理。它们只是 Kleisli 范畴的标准组合定律：

这种定义还表达了 `monad` 的真正含义：它是一种组合装饰函数的方法。这与副作用或状态无关。它关乎组合。正如我们稍后将看到的，装饰函

数可以用来表达各种效应或状态，但这不是 monad 的用途。monad 是将一个装饰函数的末端与另一个装饰函数的末端捆绑在一起的粘性胶带。

回到我们的例子：记录函数（函子的 Kleisli 箭头）形成一个范畴，因为是一个 monad：

snippet05 只要的幺半群定律得到满足，的 monad 定律也就满足了（这些定律在 Haskell 中也不能强制执行）。

对于 monad，有一个名为的有用的 Kleisli 箭头。它的唯一目的是将其参数添加到日志中：

snippet06 我们稍后将其作为其他 monadic 函数的构建块使用。

20.2 鱼的解剖 (Fish Anatomy)

在为不同的 monad 实现鱼算子时，你会很快意识到许多代码是重复的，可以很容易地提取出来。首先，两个函数的 Kleisli 组合必须返回一个函数，因此它的实现不妨从一个接受类型为的参数的 lambda 开始：

snippet07 我们能对这个参数做的唯一事情是将其传递给：

snippet08 在这一点上，我们必须在拥有类型为的对象和函数的情况下，生成类型为的结果。让我们定义一个为我们完成此任务的函数。这个函数称为 bind（绑定），通常以中缀运算符的形式书写：

snippet09 对于每个 monad，我们可以不定义鱼算子，而是定义 bind。实际上，Haskell 的标准 monad 定义使用 bind：

snippet10 以下是 monad 的 bind 定义：

snippet11 它确实比鱼算子的定义短。

进一步拆解 bind 是可能的，利用是一个函子的事实。我们可以使用将函数应用于的内容。这将把变成。应用的结果因此是类型。这不是我们想要的结果——我们需要类型为的结果——但我们已经很接近了。我们所需要的只是一个可以压缩或展平的双重应用的函数。这个函数称为：

snippet12 使用，我们可以将 bind 重写为：

snippet13 这引导我们到定义 monad 的第三种选择：

snippet14 在这里，我们显式要求是一个（函子）。在之前的两个 monad 定义中，我们不必这样做。这是因为任何支持鱼算子或 bind 运算符的类型构造函数都自动是一个函子。例如，可以用 bind 和来定义：

snippet15 为了完整性，这里是 monad 的：

snippet16

20.3 语法 (The Notation)

一种使用 monad 编写代码的方法是使用 Kleisli 箭头——使用鱼算子组合它们。这种编程模式是无点风格的泛化。无点风格的代码紧凑而且通常相当优雅。但总的来说，它可能难以理解，几乎是神秘的。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 monad 时，这意味着更喜欢 bind 运算符而不是鱼算子。Bind 接受一个 monadic 值并返回一个 monadic 值。程序员可以选择为这些值命名。但这几乎没有改进。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 monadic 容器。这就是命令式代码的工作方式——副作用，如更新全局日志，通常是隐藏的。这就是 Haskell 中 语法的仿效方式。

你可能会想，那么为什么要使用 monad 呢？如果我们想让副作用不可见，为什么不坚持使用命令式语言呢？答案是 monad 给了我们对副作用的更好控制。例如，monad 中的日志从一个函数传递到另一个函数，从未在全局范围内暴露。不存在混淆日志或创建数据竞争的可能性。此外，monadic 代码清晰地与程序的其余部分隔离开来。

语法只是 monadic 组合的语法糖。从表面上看，它看起来很像命令式代码，但它直接转化为一系列 bind 和 lambda 表达式。

例如，使用我们之前用于说明 monad 中 Kleisli 箭头组合的示例。使用我们当前的定义，它可以重写为：

snippet17 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，同时记录它的操作。

在 语法中，它看起来像这样：

snippet18 在这里，只是一个，尽管 生成了一个：

snippet19 这是因为 块由编译器解糖为：

snippet20 的 monadic 结果被绑定到一个 lambda，该 lambda 接受一个。这个字符串的名字出现在 块中。当读取这一行时：

snippet21 我们说 获得了 的结果。

当我们内联 时，伪命令式风格更加明显。我们用调用 替换它，后者记录字符串，然后调用 并将结果拆分为字符串 使用。请注意，是一个作用于字符串的常规函数。

snippet22 在这里，do 块中的每一行都会在解糖后的代码中引入一个新的嵌套 bind：

snippet23 请注意，生成一个单位值，因此不必将其传递给后面的 lambda。忽略 monadic 结果的内容（但不忽略其效果——此处对日志的贡献）是相当常见的，因此有一个特殊运算符可以在这种情况下替换 bind：

snippet24 我们代码的实际解糖如下所示：

snippet25 一般来说，块由行（或子块）组成，这些行（或子块）要么使用左箭头引入新名称，然后这些名称在代码的其余部分中可用，要么仅用于副作用而执行。行之间的 bind 运算符是隐式的。顺便说一下，

在 Haskell 中，可以用大括号和分号替换 块中的格式。这为将 monad 描述为重载分号的方式提供了依据。

请注意，在解糖 语法时，lambda 和 bind 运算符的嵌套对 do 块其余部分的执行产生了影响，具体取决于每一行的结果。这一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，语法的等价物在命令式语言中得到了应用，特别是在 C++ 中。我指的是可恢复函数或协程。C++ futures 是一个 monad¹ 并不是秘密。这是 continuation monad 的一个例子，我们将在稍后讨论。continuation 的问题是它们非常难以组合。在 Haskell 中，我们使用 语法将“我的处理程序会调用你的处理程序”的意大利面条式代码转化为非常像顺序代码的东西。可恢复函数使这种转换在 C++ 中成为可能。相同的机制也可以用于将 嵌套循环的意大利面条式代码² 转化为列表推导式或“生成器”，它们本质上是列表 monad 的 语法。没有 monad 的统一抽象，每个问题通常通过提供语言的自定义扩展来解决。在 Haskell 中，这些都通过库来处理。

¹

²

21

Monads and Effects

现在我们知道 monad 的作用是什么了——它让我们可以组合装饰过的函数——真正有趣的问题是，为什么装饰过的函数在函数式编程中如此重要。我们已经看到了一个例子，即 monad，其中装饰使我们能够在多个函数调用中创建和累积日志。本来需要使用不纯函数（例如，通过访问和修改某个全局状态）来解决的问题，现在通过纯函数解决了。

21.1 问题 (The Problem)

这里是一个类似问题的简短列表，摘自 Eugenio Moggi 的开创性论文¹，所有这些问题传统上都是通过放弃函数的纯性来解决的。

- 不完全性：可能不会终止的计算
- 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，或称为环境
 - 只写状态，或称为日志
 - 读/写状态
- 异常：可能失败的部分函数
- Continuations：保存程序状态并在需要时恢复的能力
- 交互式输入
- 交互式输出

真正令人惊讶的是，所有这些问题都可以用同一个巧妙的技巧解决：转向装饰过的函数。当然，每种情况下的装饰将完全不同。

¹

你必须意识到，在这个阶段，并没有要求装饰是 monadic 的。只有当我们坚持组合——能够将一个装饰过的函数分解为更小的装饰过的函数时——我们才需要 monad。再一次，由于每种装饰都不同，monadic 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：具有结合性和单位元的组合。

下一节有大量 Haskell 示例。如果你迫切想回到范畴论，或者你已经熟悉 Haskell 的 monad 实现，可以随意略读甚至跳过它。

21.2 解决方案 (The Solution)

首先，让我们分析一下我们如何使用 monad。我们从一个执行特定任务的纯函数开始——给定参数，它会生成特定的输出。我们用另一个函数替换了这个函数，该函数通过将原始输出与字符串配对来装饰它。这就是我们解决日志记录问题的方法。

我们不能止步于此，因为通常我们不想处理单一的解决方案。我们需要能够将一个生成日志的函数分解为多个更小的生成日志的函数。正是这些较小函数的组合导致了 monad 概念的出现。

真正令人惊叹的是，装饰函数返回类型的相同模式适用于各种通常需要放弃纯性的复杂问题。让我们逐一浏览我们的列表，并确定适用于每个问题的装饰。

21.2.1 不完全性 (Partiality)

我们通过将每个可能不会终止的函数的返回类型转化为“提升的”类型来修改它——一种包含原始类型的所有值加上特殊的“底”值 \perp 的类型。例如，类型作为一个集合将包含两个元素： 和 。提升后的 包含 三个元素。返回提升后的 的 函数可能会生成 或 ，也可能会永远执行下去。

有趣的是，在像 Haskell 这样的惰性语言中，一个永无止境的函数实际上可能会返回一个值，并且这个值可能会被传递给下一个函数。我们称这个特殊的值为底值。只要不显式需要这个值（例如，用于模式匹配或生成输出），它就可以被传递而不会阻碍程序的执行。因为每个 Haskell 函数可能都是非终止的，Haskell 中的所有类型都被假定为提升的。这就是为什么我们经常谈论 Haskell（提升的）类型和函数的范畴 **Hask**，而不是更简单的 **Set**。不过，是否可以明确 **Hask** 是一个真正的范畴仍不清楚（参见这篇 Andrej Bauer 的文章²）。

21.2.2 非确定性 (Nondeterminism)

如果一个函数可以返回多个不同的结果，那么它也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一个返回结果列表

²

的函数。在像 Haskell 这样的惰性垃圾回收语言中，这很有意义。例如，如果你只需要一个值，你可以只取列表的头部，其余部分永远不会被计算。如果你需要一个随机值，使用随机数生成器选择列表中的第 n 个元素。惰性甚至允许你返回一个无限的结果列表。

在列表 monad——Haskell 的非确定性计算实现中——实现为。请记住，应该将容器的容器压平——将一个列表的列表连接成一个单一的列表。创建一个单元素列表：

snippet01 列表 monad 的 bind 运算符由通用公式给出：先，然后，在此情况下为：

snippet02 在这里，函数 本身生成一个列表，并应用于列表 的每个元素。结果是一个列表的列表，它通过 进行压平。

从程序员的角度来看，处理一个列表比在循环中调用非确定性函数或实现一个返回迭代器的函数要容易得多（尽管在现代 C++³ 中，返回一个惰性范围几乎等同于在 Haskell 中返回一个列表）。

在游戏编程中创造性地使用非确定性是一个很好的例子。例如，当计算机与人类下棋时，它无法预测对手的下一步棋。但它可以生成所有可能的走法列表，并逐一分析它们。同样，非确定性解析器可以为给定表达式生成所有可能的解析列表。

尽管我们可以将返回列表的函数解释为非确定性的，但列表 monad 的应用范围要广得多。这是因为连接生成列表的计算是迭代结构（循环）的一个完美的函数式替代方案，循环通常在命令式编程中使用。单个循环通常可以使用 重新编写，以将循环体应用于列表的每个元素。列表 monad 中的 语法可以用来替换复杂的嵌套循环。

我最喜欢的的例子是生成勾股数三元组的程序——可以构成直角三角形边的正整数三元组。

snippet03 第一行告诉我们，从正整数的无限列表 中获取一个元素。然后，从列表 中获取一个元素，该列表包含从 1 到 的数字。最后，从 和 之间的数字列表中获取一个元素。我们可以使用三个数字 $1 \leq x \leq y \leq z$ 。函数 接受一个 表达式，并返回一个单位列表：

snippet04 此函数（它是更大类 的成员）用于过滤掉非勾股三元组。实际上，如果你查看 bind 运算符（或相关的 运算符）的实现，你会注意到，当给定一个空列表时，它会生成一个空列表。另一方面，当给定一个非空列表（此处是包含单位 的单例列表）时，bind 将调用继续体，即，它生成一个验证的勾股三元组的单例列表。所有这些单例列表将通过包含的 binds 连接起来生成最终的（无限）结果。当然，的调用者永远无法消费整个列表，但这并不重要，因为 Haskell 是惰性的。

通常需要一组三重嵌套循环的问题在列表 monad 和 语法的帮助下得到了极大的简化。如果这还不够，Haskell 允许你使用列表推导进一步简化此代码：

snippet05 这只是列表 monad 的进一步语法糖（严格来说，）。

³

你可能会在其他函数式或命令式语言中看到类似的构造，它们伪装成生成器和协程。

21.2.3 只读状态 (Read-Only State)

一个具有只读访问某些外部状态或环境的函数总是可以替换为一个以该环境作为额外参数的函数。一个纯函数（其中 `Env` 是环境的类型）乍一看不像是 Kleisli 箭头。但只要我们将它柯里化为，我们就会认识到这种装饰是我们老朋友 `reader` 函数：

`snippet06` 你可以将返回 `Env` 的函数解释为生成一个小的可执行文件：一个给定环境生成所需结果的动作。有一个辅助函数 来执行此类操作：

`snippet07` 它可以为不同的环境值生成不同的结果。

请注意，返回 `Env` 的函数和 操作本身都是纯的。

要为 `monad` 实现 `bind`，首先请注意，你必须生成一个接受环境 并生成 的函数：

`snippet08` 在 `lambda` 内部，我们可以执行操作 生成一个：

`snippet09` 然后我们可以将 传递给继续体 以获得一个新的操作：

`snippet10` 最后，我们可以使用环境 运行操作：

`snippet11` 要实现，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，经过一些简化后，我们得到以下定义：

`snippet12`

21.2.4 只写状态 (Write-Only State)

这只是我们最初的日志记录示例。装饰由 函数给出：

`snippet13` 为了完整性，还有一个简单的辅助函数 用于解包数据构造器：

`snippet14` 如前所见，为了使 可组合，必须是一个 `monoid`。这是使用 `bind` 运算符编写的 `monad` 实例：

`snippet15`

21.2.5 状态 (State)

具有读/写状态访问权限的函数结合了 和 的装饰。你可以将它们视为以状态作为额外参数的纯函数，并生成一个对值/状态的结果对：。在柯里化后，我们将它们转化为 Kleisli 箭头形式，装饰抽象在 函数中：

`snippet16` 再次强调，我们可以将 Kleisli 箭头视为返回一个动作，可以使用辅助函数执行此动作：

`snippet17` 不同的初始状态不仅可能生成不同的结果，还可能生成不同的最终状态。

`monad` 的 `bind` 实现与 `monad` 的实现非常相似，只不过在每一步都要注意传递正确的状态：

`snippet18` 这是完整的实例：

snippet19 还有两个辅助 Kleisli 箭头可以用来操作状态。其中一个检索状态进行检查：

snippet20 另一个则用一个全新的状态替换它：

snippet21

21.2.6 异常 (Exceptions)

抛出异常的命令式函数实际上是一个部分函数——这是一个不适用于某些参数值的函数。使用纯全函数来实现异常的最简单方法是使用 函子。部分函数扩展为一个总函数，在合理的情况下返回，而在不合理的情况下返回。如果我们还想返回一些关于失败原因的信息，我们可以改用 函子（第一个类型固定，例如为）。

这是 的 实例：

snippet22 请注意， 的 monadic 组合正确地在检测到错误时短路计算（不调用继续体）。这是我们期望从异常中得到的行为。

21.2.7 Continuations

这是你在面试后可能遇到的情况：“不要打给我们，我们会打给你！”你不再直接得到答案，而是需要提供一个处理函数，在结果可用时调用它。这种编程风格在结果尚未确定时特别有用，例如因为它正在由另一个线程评估或从远程网站获取。Kleisli 箭头在这种情况下返回一个接受处理函数的函数，该函数表示“剩余的计算”：

snippet23 处理函数 在最终调用时生成类型为 的结果，并在最后返回该结果。continuation 是由结果类型参数化的。（实际上，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。它接受处理函数并将其传递给 continuation：

snippet24 continuations 的组合非常困难，因此通过 monad 处理它们，特别是通过 语法，具有极大的优势。

让我们研究一下 bind 的实现。首先让我们看一下精简后的签名：

snippet25 我们的目标是创建一个函数，该函数接受处理函数 并生成结果。所以这是我们的起点：

snippet26 在 lambda 内，我们希望使用适当的处理函数调用函数，该处理函数表示剩余的计算。我们将这个处理函数实现为一个 lambda：

snippet27 在这种情况下，剩余的计算包括首先调用 和，然后将传递给结果操作：

snippet28 如你所见，continuations 是从内到外组合的。最终的处理函数 是从计算的最内层调用的。这是完整的实例：

snippet29

21.2.8 交互式输入 (Interactive Input)

这是最棘手的问题，也是许多困惑的来源。显然，一个像这样的函数，如果它返回键盘上输入的字符，就不能是纯的。但是，如果它返回字符放在容器中呢？只要没有办法从这个容器中提取字符，我们就可以声称这个函数是纯的。每次你调用 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字符的叠加态。

如果你熟悉量子力学，你应该能够理解这个类比。它就像装有 Schrödinger 的猫的盒子——只不过没有办法打开或窥视盒子。该盒子使用特殊的内置 函子定义。在我们的例子中，可以声明为一个 Kleisli 箭头：

snippet30 (实际上，由于从单位类型的函数等效于选择返回类型的一个值，的声明简化为。)

作为一个函子，允许你使用 操作其内容。并且，作为一个函子，它可以存储任何类型的内容，不仅仅是字符。当你考虑到在 Haskell 中，是一个 monad 时，这种方法的实际效用就显现出来了。这意味着你可以组合生成 对象的 Kleisli 箭头。

你可能认为 Kleisli 组合允许你窥探 对象的内容（如果我们继续量子类比，就会“塌缩波函数”）。确实，你可以将 与另一个 Kleisli 箭头组合，该箭头接受一个字符，例如将其转换为整数。问题在于第二个 Kleisli 箭头只能以 的形式返回该整数。再一次，你最终会得到所有可能整数的叠加态。以此类推。Schrödinger 的猫永远不会出袋。一旦你进入 monad，就没有办法退出它。对于 monad 没有 或 的等效物。没有！

那么你能用 Kleisli 箭头的结果 对象做什么，除了与另一个 Kleisli 箭头组合？好吧，你可以从 返回它。在 Haskell 中，的签名为：

snippet31 你可以将其视为 Kleisli 箭头：

snippet32 从这个角度来看，Haskell 程序只是 monad 中的一个大 Kleisli 箭头。你可以使用 monadic 组合从更小的 Kleisli 箭头中组合它。如何处理生成的 对象（也称为 操作）取决于运行时系统。

请注意，箭头本身是一个纯函数——一直都是纯函数。肮脏的工作则留给系统。当它最终执行从 返回的 操作时，它会做各种肮脏的事情，例如读取用户输入、修改文件、打印讨厌的信息、格式化磁盘等等。Haskell 程序永远不会弄脏自己的手（嗯，除了它调用 时，但这是另一个故事）。

当然，由于 Haskell 是惰性的，几乎立即返回，肮脏的工作会立即开始。正是在执行 操作期间，纯计算的结果被请求并按需计算。因此，实际上，程序的执行是纯 (Haskell) 代码和肮脏 (系统) 代码的交错进行。

对于 monad，有一种更为离奇但在数学模型上非常合理的解释。它将整个宇宙视为程序中的一个对象。注意，概念上，命令式模型将宇宙视为一个外部的全局对象，因此通过与该对象交互，执行 I/O 的过程具有副作用。它们可以读取和修改宇宙的状态。

我们已经知道如何在函数式编程中处理状态——我们使用 state monad。然而，宇宙的状态不像简单状态那样可以使用标准数据结构轻松描述。但是，只要我们从不直接与它交互，我们也不必这样做。只要我们假设存在一种类型，并通过一些宇宙工程奇迹，运行时能够提供这种类型的对象就足够了。操作只是一个函数：

snippet33 或者，在 monad 的术语中：

snippet34 不过，monad 的 和 必须内置于语言中。

21.2.9 交互式输出 (Interactive Output)

相同的 monad 用于封装交互式输出。应该包含所有输出设备。你可能想知道，为什么我们不能直接从 Haskell 调用输出函数，并假装它们什么都不做。例如，为什么我们有：

snippet35 而不是更简单的：

snippet36 有两个原因：Haskell 是惰性的，所以它永远不会调用一个其输出——在此处为单位对象——未被用于任何用途的函数。而且，即使它不是惰性的，它仍然可以自由更改这些调用的顺序，从而使输出混乱。强制 Haskell 中两个函数顺序执行的唯一方法是通过数据依赖性。一个函数的输入必须依赖于另一个函数的输出。在 操作之间传递强制执行了顺序。

从概念上讲，在这个程序中：

snippet37 打印“World! ”的操作接收到的是一个宇宙，其中“Hello”已经显示在屏幕上。它输出一个新的宇宙，其中“Hello World!”显示在屏幕上。

21.3 结论 (Conclusion)

当然，我只是浅尝了 monadic 编程的表面。Monads 不仅用纯函数完成了在命令式编程中通常用副作用完成的工作，而且它们还做到了高度的控制和类型安全。不过，它们也不是没有缺点的。关于 monads 的主要抱怨是它们不容易相互组合。当然，你可以使用 monad transformer 库来组合大多数基本 monads。创建一个结合状态和异常的 monad 堆栈相对容易，但没有公式可以将任意 monads 叠加在一起。

22

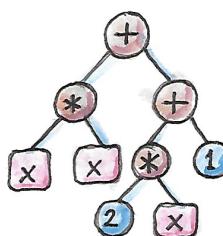
Monads Categorically

如果你向程序员提到单子(monads)，你可能会最终讨论效应(effects)。对于数学家来说，单子是关于代数(algebras)的。我们稍后会讨论代数——它们在编程中起着重要作用——但首先我想给你一些关于它们与单子关系的直觉。现在，这有点像是空谈，但请耐心听我讲下去。

代数涉及到创建、操作和求值表达式。表达式是使用运算符构建的。考虑这个简单的表达式：

$$x^2 + 2x + 1$$

这个表达式是使用变量如 x 和常数如 1 或 2 以及运算符如加号或乘号构成的。作为程序员，我们通常将表达式视为树结构。



树是一种容器，因此更普遍地说，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(endofunctors)。如果我们将类型 a 分配给变量 x ，我们的表达式将具有类型 $m a$ ，其中 m 是一个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来创建)。

对表达式可以执行的最常见操作是什么？是替换：用表达式替换变量。例如，在我们的例子中，我们可以用 $y - 1$ 替换 X ，得到：

$$(y - 1)^2 + 2(y - 1) + 1$$

这里发生了什么：我们获取了一个类型为 $m a$ 的表达式，并应用了一个类型为 $a \rightarrow m b$ 的变换（ b 代表 y 的类型）。结果是一个类型为 $m b$ 的表达式。让我详细说明：

$$m a \rightarrow (a \rightarrow m b) \rightarrow m b$$

是的，这是单子绑定（monadic bind）的签名。

这是一些动机性的介绍。现在让我们进入单子的数学定义。数学家使用的记号与程序员不同。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， η 表示。这两个都是多态函数，所以我们可以猜测它们对应于自然变换。

因此，在范畴论中，单子定义为一个自函子 T ，配备一对自然变换 μ 和 η 。

μ 是从函子 T^2 到 T 的自然变换。 T^2 简单来说就是自函子自身的复合 $T \circ T$ （这种平方运算只能对自函子进行）。

$$\mu :: T^2 \rightarrow T$$

这个自然变换在对象 a 处的分量是态射：

$$\mu_a :: T(Ta) \rightarrow Ta$$

在 **Hask** 中，这直接转化为我们对 的定义。

η 是恒等函子 I 和 T 之间的自然变换：

$$\eta :: I \rightarrow T$$

考虑到 I 在对象 a 上的作用只是 a ，因此 η 的分量由态射给出：

$$\eta_a :: a \rightarrow Ta$$

这直接转化为我们对 的定义。

这些自然变换必须满足一些附加的规律。一种理解方式是，这些规律允许我们为自函子 T 定义一个克莱斯里范畴（Kleisli category）。记住， a 和 b 之间的克莱斯里箭头定义为态射 $a \rightarrow Tb$ 。两个这样的箭头的组合（我将其写作带下标 T 的圆圈）可以使用 μ 实现：

$$g \circ_T f = \mu_c \circ (Tg) \circ f$$

其中

$$\begin{aligned} f &:: a \rightarrow Tb \\ g &:: b \rightarrow Tc \end{aligned}$$

这里 T 作为一个函子，可以应用于态射 g 。在 Haskell 记号中，这个公式可能更容易识别：

snippet01 或者，在组件中：

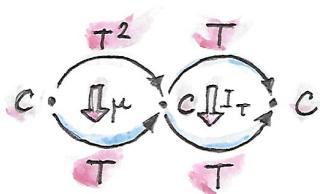
snippet02 在代数解释中，我们只是组合了两个连续的替换。

为了使克莱斯里箭头形成一个范畴，我们希望它们的组合是结合的，并且 η_a 是 a 上的恒等克莱斯里箭头。这个要求可以转化为 μ 和 η 的单子定律。但是还有另一种推导这些定律的方法，使它们更像幺半群定律。事实上， μ 通常被称为“乘法”(multiplication)， η 被称为“单位”(unit)。

粗略地说，结合律规定，将 T 的立方 T^3 降至 T 的两种方式必须得出相同的结果。两个单位律（左单位律和右单位律）规定，当 η 应用于 T 然后由 μ 归约时，我们回到了 T 。

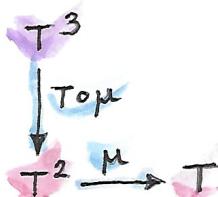
由于我们在组合自然变换和函子，因此这些有点棘手。因此，有必要复习一下横向组合。例如， T^3 可以看作 T 作用于 T^2 之后的组合。我们可以将两个自然变换的横向组合应用于它：

$$I_T \circ \mu$$

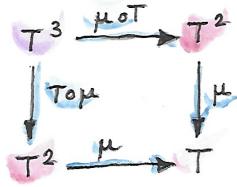


并得到 $T \circ T$ ；然后可以通过应用 μ 将其进一步简化为 T 。 I_T 是从 T 到 T 的恒等自然变换。在这种情况下，你会经常看到 $I_T \circ \mu$ 的横向组合记法缩写为 $T \circ \mu$ 。这种记法是明确的，因为将函子与自然变换组合没有意义，因此在这种上下文中 T 必须表示 I_T 。

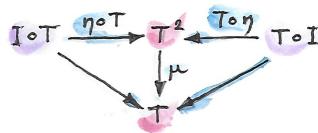
我们还可以在（自）函子范畴 $[C, C]$ 中绘制图表：



或者，我们可以将 T^3 看作 $T^2 \circ T$ 的组合，并对其应用 $\mu \circ T$ 。结果也是 $T \circ T$ ，然后可以用 μ 归约为 T 。我们要求这两条路径产生相同的结果。



类似地，我们可以对 T 之后的恒等函子的组合应用横向组合 $\eta \circ T$ ，得到 T^2 ，然后可以用 μ 归约。结果应该与我们直接将恒等自然变换应用于的结果相同。同样的， $T \circ \eta$ 也是如此。



你可以确信这些定律保证了克莱斯里箭头的组合确实满足范畴的定律。

单子和幺半群之间的相似性是显而易见的。我们有乘法 μ ，单位 η ，结合律和单位律。但是我们对幺半群的定义过于狭窄，无法将单子描述为幺半群。因此，让我们将幺半群的概念推广。

22.1 幺半群范畴 (Monoidal Categories)

让我们回到幺半群的传统定义。幺半群是一个集合，带有一个二元运算和一个称为单位元的特殊元素。在 Haskell 中，这可以用一个类型类来表达：

`snippet03` 二元运算 必须是结合的，并且具有单位性（即乘以单位元 不会改变任何元素）。

注意，在 Haskell 中， 的定义是柯里化的。它可以解释为将 的每个元素映射到一个函数：

`snippet04` 正是这种解释导致了幺半群被定义为一个单对象范畴，其中的态射 代表幺半群的元素。但由于柯里化是 Haskell 的内置特性，我们也可以从另一种乘法定义开始：

`snippet05` 在这里，笛卡尔积 成为待乘元素对的来源。

这个定义暗示了另一种推广的路径：用范畴积 (categorical product) 替换笛卡尔积。我们可以从一个在全局定义积的范畴开始，选择其中的一个对象，并定义乘法为态射：

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

但是我们有一个问题：在任意范畴中，我们无法窥探对象的内部，所以我们如何选择单位元呢？有一个技巧可以解决这个问题。记得元素选择

等价于从单集合到某个集合的函数吗？在 Haskell 中，我们可以将 的定义替换为一个函数：

snippet06 单集合是 **Set** 中的终对象，因此将这个定义推广到任何具有终对象的范畴是自然的：

$$\eta : t \rightarrow m$$

这使我们可以选择单位“元素”而无需讨论元素。

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是自动满足的——我们必须强加它们。但是，为了表述它们，我们必须建立底层范畴积本身的幺半群结构。让我们首先回顾 Haskell 中的幺半群结构是如何工作的。

我们从结合性开始。在 Haskell 中，相应的等式定律是：

snippet07 在将其推广到其他范畴之前，我们必须将其重写为函数（态射）的相等性。我们必须将其抽象化，不再关注其对单个变量的作用——换句话说，我们必须使用点自由记法（point-free notation）。知道笛卡尔积是一个双函子，我们可以将左侧写为：

snippet08 右侧写为：

snippet09 这几乎是我们想要的。不幸的是，笛卡尔积并不是严格结合的——不等同于——所以我们不能直接写点自由的：

snippet10 另一方面，这两种配对嵌套是同构的。存在一个可逆的函数，称为结合子（associator），可以在它们之间转换：

snippet11 借助结合子，我们可以写出点自由的结合律：

snippet12 我们可以对单位律进行类似的处理，在新记法中，它们的形式为：

snippet13 它们可以重写为：

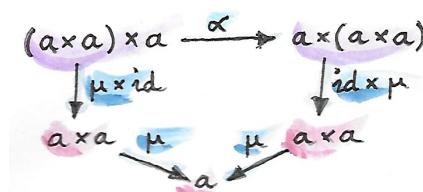
snippet14 同构 和 分别称为左单位元（left unit）和右单位元（right unit）。它们见证了单位 是笛卡尔积的单位元，至多同构：

snippet15

snippet16 因此，单位律的点自由版本为：

snippet17 我们使用了底层笛卡尔积本身在类型范畴中充当幺半群乘法这一事实，得到了 和 的点自由幺半群定律。但请记住，笛卡尔积的结合性和单位律仅在同构意义上成立。

事实证明，这些定律可以推广到任何具有积和终对象的范畴。范畴积确实在同构意义上是结合的，终对象也是单位元，同样在同构意义上。结合子和两个单位元是自然同构。这些定律可以用交换图表示。



注意，由于积是一个双函子，它可以提升一对态射——在 Haskell 中，这是通过 完成的。

我们可以在此停下来，说我们可以在任何具有范畴积和终对象的范畴上定义幺半群。只要我们能选择一个对象 m 以及两个满足幺半群定律的态射 μ 和 η ，我们就有幺半群。但我们可以做得更好。我们不需要完整的范畴积来表述 μ 和 η 的定律。回想一下积是通过一个使用投影的泛构造定义的。在我们的幺半群定律表述中并没有使用任何投影。

一种表现为积但并不是积的双函子称为张量积 (tensor product)，通常记作中缀运算符 \otimes 。一般情况下，张量积的定义有点棘手，但我们不必担心这个。我们只列出它的性质——最重要的是它在同构意义上的结合性。

类似地，我们不需要对象 t 是终对象。我们从未使用过它的终性质——即从任意对象到它的唯一态射的存在性。我们需要的是它能与张量积很好地配合工作。也就是说，我们希望它是张量积的单位元，同样是在同构意义上。让我们把所有这些拼在一起：

一个幺半群范畴是一个范畴 C ，它配备了一个称为张量积的双函子：

$$\otimes : C \times C \rightarrow C$$

以及一个特殊的对象 i ，称为单位对象 (unit object)，并配有三个自然同构，分别称为结合子 (associator) 和左单位元 (left unitor) 以及右单位元 (right unitor)：

$$\begin{aligned}\alpha_{abc} &:: (a \otimes b) \otimes c \rightarrow a \otimes (b \otimes c) \\ \lambda_a &:: i \otimes a \rightarrow a \\ \rho_a &:: a \otimes i \rightarrow a\end{aligned}$$

(还有一个用于简化四重张量积的相干性条件。)

重要的是，张量积描述了许多熟悉的双函子。特别是它适用于积、余积，并且正如我们将很快看到的，它适用于自函子的组合（以及一些更晦涩的积，如 Day convolution）。幺半群范畴在丰富范畴 (enriched categories) 的表述中起着至关重要的作用。

22.2 幺半群范畴中的幺半群 (Monoid in a Monoidal Category)

现在我们准备好在更广泛的幺半群范畴中定义幺半群。我们首先选择一个对象 m 。使用张量积，我们可以形成 m 的幂。 m 的平方是 $m \otimes m$ 。有两种形成 m 的立方的方法，但它们通过结合子是同构的。对于 m 的更高幂也是如此（这就是我们需要相干性条件的地方）。为了形成幺半

群，我们需要选择两个态射：

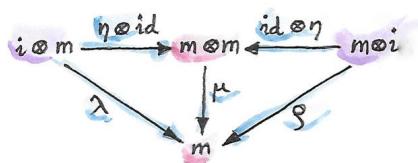
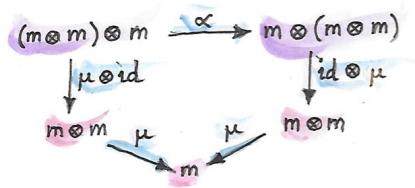
$$\mu :: m \otimes m \rightarrow m$$

$$\eta :: i \rightarrow m$$

其中 i 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，它们可以用以下交换图表示：



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如 $\mu \otimes id$ 或 $\eta \otimes id$ 的积。这些图表只是我们之前对范畴积结果的直接推广。

22.3 单子作为幺半群 (Monads as Monoids)

幺半群结构出现在意想不到的地方。其中一个地方是函子范畴。如果你稍微眯起眼睛，你可能会看到函子组合是一种乘法形式。问题在于并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。这只是态射组合的通常规则——我们知道，函子确实是范畴 Cat 中的态射。但是，就像自态射（从同一对象循环回的态射）总是可

组合的一样，自函子也是如此。对于任意给定的范畴 C ，从 C 到 C 的自函子形成了函子范畴 $[C, C]$ 。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选择任意两个对象，比如自函子 F 和 G ，并生成第三个对象 $F \circ G$ ——一个它们的组合自函子。

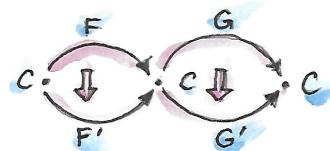
自函子组合是张量积的一个好候选者吗？首先，我们必须确定它是一个双函子。它可以用于提升态射对——这里是自然变换吗？张量积的类似物的签名看起来像这样：

$$bimap :: (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow (a \otimes c \rightarrow b \otimes d)$$

如果你将对象替换为自函子，将箭头替换为自然变换，并将张量积替换为组合，你会得到：

$$(F \rightarrow F') \rightarrow (G \rightarrow G') \rightarrow (F \circ G \rightarrow F' \circ G')$$

你可能会认识到这是横向组合的特例。



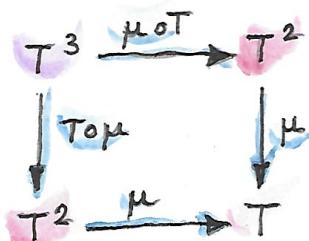
我们还可以使用恒等自函子 I 作为自函子组合的恒等元——我们新的张量积。此外，函子组合是结合的。事实上，结合律和单位律是严格的——不需要结合子或两个单位元。因此，自函子通过函子组合形成一个严格的幺半群范畴。

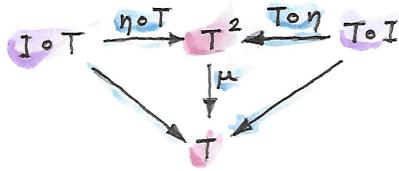
这个范畴中的幺半群是什么？它是一个对象——即自函子 T ；以及两个态射——即自然变换：

$$\mu :: T \circ T \rightarrow T$$

$$\eta :: I \rightarrow T$$

不仅如此，这里是幺半群定律：





它们正是我们之前看到的单子定律。现在你明白了 Saunders Mac Lane 的那句著名的话：

总而言之，单子只是自函子范畴中的幺半群。

你可能在函数式编程会议上看到过这句话印在 T 恤上。

22.4 从伴随子得出单子 (Monads from Adjunctions)

一个伴随关系¹ $L \dashv R$ ，是一个在两个范畴 **C** 和 **D** 之间来回的函子对。有两种组合它们的方法产生两个自函子， $R \circ L$ 和 $L \circ R$ 。根据伴随关系，这些自函子通过两个称为单位元 (unit) 和余单位元 (counit) 的自然变换与恒等函子相关联：

$$\begin{aligned}\eta &:: I_D \rightarrow R \circ L \\ \varepsilon &:: L \circ R \rightarrow I_C\end{aligned}$$

我们立即看到，伴随关系的单位元看起来就像单子的单位元。事实证明，自函子 $R \circ L$ 确实是一个单子。我们只需要定义与 η 对应的 μ 。这是一个自函子的平方与自函子本身之间的自然变换，或者换句话说，是伴随函子的关系：

$$R \circ L \circ R \circ L \rightarrow R \circ L$$

事实上，我们可以使用余单位元来消除中间的 $L \circ R$ 。 μ 的精确公式由横向组合给出：

$$\mu = R \circ \varepsilon \circ L$$

单子定律从伴随关系的单位元和余单位元的恒等式以及互换律中得出。

在 Haskell 中，我们不常见到从伴随关系派生的单子，因为伴随关系通常涉及两个范畴。然而，指数对象或函数对象的定义是一个例外。这里是形成这种伴随关系的两个自函子：

$$Lz = z \times s$$

$$Rb = s \Rightarrow b$$

¹参见第 18 章关于伴随的内容。

你可能会认出它们的组合是我们熟悉的状态单子：

$$R(Lz) = s \Rightarrow (z \times s)$$

我们之前在 Haskell 中见过这个单子：

snippet18 让我们也将伴随关系翻译成 Haskell。左函子是积函子：

snippet19 右函子是读者函子：

snippet20 它们形成了伴随关系：

snippet21 你可以轻松地验证读者函子与积函子的组合确实等价于状态函子：

snippet22 如预期的那样，伴随关系的 等价于状态单子的 函数。余单位元通过求值一个函数作用于其参数来起作用。这可以识别为函数的未柯里化版本：

snippet23 (未柯里化，因为在 中它作用于一个对。)

现在我们可以将状态单子的 定义为自然变换 μ 的一个分量。为此，我们需要将三个自然变换进行横向组合：

$$\mu = R \circ \varepsilon \circ L$$

换句话说，我们需要通过读者函子的一个级别来偷偷地将余单位元 ε 跨过。我们不能直接调用，因为编译器会选择适用于 函子的版本，而不是 函子的版本。但请记住，读者函子的 只是左函数组合。因此，我们将直接使用函数组合。

我们首先需要剥离数据构造器 以暴露 函子内部的函数。这是通过完成的：

snippet24 然后我们将它与余单位元左组合，后者由 定义。最后，我们将其重新包装回 数据构造器中：

snippet25 这确实是状态单子的 实现。

事实证明，不仅每个伴随关系都产生一个单子，反过来也成立：每个单子都可以分解为两个伴随函子的组合。然而，这种分解不是唯一的。

我们将在下一节讨论另一个自函子 $L \circ R$ 。

23

Comonads

现 在我们已经介绍了 monad (单子)，我们可以利用对偶性来免费获得 comonad (余单子)，只需反转箭头并在对偶范畴中工作。

回想一下，在最基本的层面上，monad 是关于组合 Kleisli 箭头的： snippet01 其中，是一个函子，该函子是一个 monad。如果我们使用字母（倒转的）来表示 comonad，我们可以将 co-Kleisli 箭头定义为以下类型的态射：

snippet02 对于 co-Kleisli 箭头来说，其对应的鱼操作符的定义为：

snippet03 为了使 co-Kleisli 箭头形成一个范畴，我们还必须有一个身份 co-Kleisli 箭头，这被称为：

snippet04 这是 的对偶。我们还必须施加结合律以及左、右单位律。将这些全部结合起来，我们可以在 Haskell 中定义一个 comonad 如下：

snippet05 实际上，我们使用的基本函数略有不同，稍后我们会看到。

问题是，在编程中 comonad 的用途是什么？

23.1 使用 Comonad 编程

让我们比较一下 monad 和 comonad。monad 提供了一种使用 将值放入容器的方法。它并不提供访问容器内存储值的方式。当然，许多实现了 monad 的数据结构可能会提供访问其内容的方法，但这被视为一种额外功能。对于 monad，没有通用的接口来提取值。我们已经看到 monad 的例子，它以从不暴露其内容为傲。

另一方面，comonad 提供了一种从其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 comonad 视为一个容器，那么它总是预先填充了内容，并允许你窥探其中的内容。

就像 Kleisli 箭头接受一个值并产生一些带有上下文的结果——它将其与上下文一起装饰，co-Kleisli 箭头则接受一个值和整个上下文并产生一个结果。这是上下文计算的体现。

23.2 Product Comonad (乘积余单子)

还记得 reader monad 吗？我们引入它是为了应对实现需要访问某些只读环境 的计算的问题。这类计算可以表示为以下形式的纯函数：

snippet06 我们使用柯里化将它们转化为 Kleisli 箭头：

snippet07 但请注意，这些函数已经具有 co-Kleisli 箭头的形式。让我们将它们的参数调整为更方便的函子形式：

snippet08 我们可以轻松定义组合操作符，通过使我们要组合的箭头都可以访问相同的环境：

snippet09 的实现简单地忽略了环境：

snippet10 不出所料，product comonad 可以用于执行与 reader monad 完全相同的计算。在某种程度上，comonad 的环境实现更为自然——它遵循了“上下文中的计算”的精神。另一方面，monad 具有 记法的便捷语法糖。

reader monad 与 product comonad 之间的联系更加深入，因为 reader 函子是 product 函子的右伴随。不过，一般来说，comonad 涵盖了与 monad 不同的计算概念。稍后我们将看到更多的例子。

将 comonad 推广到任意乘积类型，包括元组和记录类型是很容易的。

23.3 解剖组合

继续对偶化的过程，我们可以继续对偶化 monad 的 bind 和 join 操作。或者，我们可以重复使用 monad 时的过程，即研究鱼操作符的解剖。这种方法似乎更具启发性。

起点是意识到组合操作符必须生成一个 co-Kleisli 箭头，它接受 并生成。生成 的唯一方法是将第二个函数应用于类型为 的参数：

snippet11 但我们如何生成类型为 的值，以便传递给？我们手头有一个类型为 的参数和一个函数。解决方案是定义 bind 的对偶，它被称为：

snippet12 使用 我们可以实现组合：

snippet13 接下来，我们能解剖 吗？你可能会说，为什么不直接将函数应用于参数，但你很快就会意识到你无法将结果 转换为。请记住，comonad 不提供提升值的方法。在类似的 monad 构造中，我们使用了。在这里我们只能使用，前提是手头有类型为 的东西。如果我们可以将 转换为。方便的是，这正是 的对偶。我们称之为：

snippet14 因此，就像对 monad 的定义一样，我们对 comonad 有三种等效的定义：使用 co-Kleisli 箭头、或。以下是直接取自 库的 Haskell 定义：

snippet15 提供了 基于 的默认实现，反之亦然，所以你只需覆盖其中之一。

这些函数背后的直觉基于这样一个想法：一般来说，comonad 可以被看作是一个充满类型为 的值的容器（product comonad 是只有一个值的特例）。有一个“当前”值的概念，它可以通过 轻松获取。co-Kleisli 箭头执行一些专注于当前值的计算，但它可以访问所有周围的值。想想康威的生命游戏。每个单元格都包含一个值（通常只是 或）。一个对应于生命游戏的 comonad 将是一个集中在“当前”单元格的单元格网格。

那么 做了什么？它接受一个 comonad 容器 并生成一个容器的容器。其想法是这些容器中的每一个都集中在 内的不同 上。在生命游戏中，你会得到一个网格的网格，外部网格的每个单元格都包含一个内部网格，该网格集中在不同的单元格上。

现在看看。它接受一个 co-Kleisli 箭头和一个充满 的 comonad 容器。它将计算应用于所有这些，并将它们替换为。结果是一个充满 的 comonad 容器。通过将焦点从一个 转移到另一个并依次应用 co-Kleisli 箭头来实现这一点。在生命游戏中，co-Kleisli 箭头将计算当前单元格的新状态。为此，它将查看其上下文——可能是最近的邻居。默认的 实现说明了这一过程。首先我们调用 生成所有可能的焦点，然后将 应用于它们。

23.4 Stream Comonad (流余单子)

这个从容器的一个元素转移焦点到另一个元素的过程最适合用无限流的例子来说明。这样的流就像一个列表，只是它没有空构造函数：

snippet16 它显然是一个：

snippet17 流的焦点是它的第一个元素，所以这是 的实现：

snippet18 生成一个流的流，每个流都集中在不同的元素上。

snippet19 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，以此类推，无穷无尽。

这是完整的实例：

snippet20 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 开始，该方法将流移位一个位置。这里，一次性生成所有移位流。Haskell 的惰性计算使这成为可能，并且甚至是可取的。当然，为了使 实际可用，我们还会实现 的类似物：

snippet21 但这永远不是 comonad 接口的一部分。

如果你有任何数字信号处理的经验，你会立即看到流的 co-Kleisli 箭头只是一个数字滤波器，而 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。这里是一个求和流中 n 个元素的函数：

snippet22 这里是计算流的前 n 个元素的平均值的函数：

snippet23 部分应用的 是一个 co-Kleisli 箭头，因此我们可以在整个流上 它：

snippet24 结果是运行平均值的流。

流是一个单向、一维 comonad 的例子。它可以轻松扩展为双向或扩展为二维或多维。

23.5 Comonad 的范畴论定义

在范畴论中定义 comonad 是一个对偶性的简单练习。与 monad 一样，我们从一个端函子 开始。定义 monad 的两个自然变换 η 和 μ 仅需对 comonad 反转：

$$\varepsilon : T \rightarrow I$$

$$\delta : T \rightarrow T^2$$

这些变换的分量对应于 和 。comonad 定律是 monad 定律的镜像。这里没有什么大惊喜。

然后还有从伴随中推导出 monad。对偶性反转了一个伴随：左伴随变成右伴随，反之亦然。而且，由于组合 $R \circ L$ 定义了一个 monad， $L \circ R$ 必须定义一个 comonad。伴随的 counit：

$$\varepsilon : L \circ R \rightarrow I$$

确实是我们 在 comonad 定义中看到的 ε ——或者在 Haskell 的 中作为组件。我们还可以使用伴随的 unit：

$$\eta : I \rightarrow R \circ L$$

将一个 $R \circ L$ 插入到 $L \circ R$ 中间，并生成 $L \circ R \circ L \circ R$ 。使 T^2 成为 T 的定义了 δ ，从而完成了 comonad 的定义。

我们还看到 monad 是一个 monoid。这个陈述的对偶性需要使用 comonoid，那么 comonoid 是什么？monoid 作为单对象范畴的原始定义并没有对偶化为任何有趣的东西。当你反转所有同态射的方向时，你得到另一个 monoid。然而，回想一下，在我们的方法中，monad 是作为 monoidal 范畴中的一个 monoid 被定义的。这个构造基于两个态射：

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

$$\eta : i \rightarrow m$$

这些态射的反转生成了 monoidal 范畴中的一个 comonoid：

$$\delta : m \rightarrow m \otimes m$$

$$\varepsilon : m \rightarrow i$$

可以在 Haskell 中编写 comonoid 的定义：

snippet25 但它相当简单。显然，忽略了它的参数。

snippet26 只是一个函数对：

snippet27 现在考虑对偶于 monoid 单位律的 comonoid 定律。

snippet28 这里， η 和 δ 分别是左和右单位元（参见 monoidal 范畴的定义）。替换定义后，我们得到：

snippet29 这证明了。同样，第二个定律展开为。结论是：

snippet30 这表明在 Haskell 中（一般来说，在范畴 **Set** 中）每个对象都是一个简单的 comonoid。

幸运的是，还有其他更有趣的 monoidal 范畴可以用来定义 comonoid。其中之一是端函子的范畴。事实证明，正如 monad 是端函子范畴中的 monoid，

comonad 是端函子范畴中的 comonoid。

23.6 Store Comonad (存储余单子)

另一个重要的 comonad 例子是 state monad 的对偶。它被称为 costate comonad 或者称为 store comonad。

我们之前已经看到 state monad 是由定义指数对象的伴随生成的：

$$Lz = z \times s$$

$$Ra = s \Rightarrow a$$

我们将使用相同的伴随来定义 costate comonad。一个 comonad 由组合 $L \circ R$ 定义：

$$L(Ra) = (s \Rightarrow a) \times s$$

将其转换为 Haskell，我们从左侧的 函子和右侧的 函子之间的伴随开始。组合 在 之后等效于以下定义：

snippet31 伴随在对象 a 处的 counit 是态射：

$$\varepsilon_a :: ((s \Rightarrow a) \times s) \rightarrow a$$

或者在 Haskell 记法中：

snippet32 这成为我们的：

snippet33 伴随的 unit：

snippet34 可以重写为部分应用的数据构造函数：

snippet35 我们将 构造为水平组合：

$$\delta :: L \circ R \rightarrow L \circ R \circ L \circ R$$

$$\delta = L \circ \eta \circ R$$

我们必须通过最左侧的 L 来偷渡 η , 这是 函子。这意味着对对偶化的左侧部分应用 η 或 (这就是 的 将要做的)。我们得到:

snippet36 (请记住, 在 的公式中, L 和 R 代表分量为身份态射的自然变换。)

以下是 comonad 的完整定义:

snippet37 你可以将 部分的 看作是一个以 类型的元素作为键的类型的通用容器。例如, 如果 是 , 是一个 类型的双向无限流。将此容器与一个键类型的值配对。例如, 与一个 配对。在这种情况下, 使用这个整数来索引到无限流中。你可以将 的第二个分量视为当前位置。

继续这个例子, 创建一个由 索引的新无限流。这个流包含流作为它的元素。特别是在当前位置, 它包含原始流。但是, 如果你使用其他(正数或负数)作为键, 你会得到一个在该新索引位置移位的流。

一般来说, 你可以说服自己, 当 作用于 d 的 时, 它会生成原始的(事实上, comonad 的单位定律指出)。

comonad 在 库中起着重要的理论基础作用。概念上, comonad 封装了“聚焦”(如同镜头)于数据类型 的特定子结构的思想, 使用类型作为索引。特别是, 以下类型的函数:

snippet38 等价于一对函数:

snippet39 如果 是一个乘积类型, 可以实现为设置 内部 类型的字段, 同时返回修改后的 版本。同样, 可以实现为从 中读取 字段的值。我们将在下一节中进一步探讨这些思想。

23.7 挑战

1. 使用 comonad 实现 Conway 的生命游戏。提示: 你为 选择什么类型?

24

F-Algebras

我们已经看到了 monoid 的几种形式：作为一个集合，作为一个单对象范畴，作为 monoidal 范畴中的一个对象。我们还能从这个简单的概念中榨取出多少“果汁”呢？

让我们试试看。将 monoid 定义为一个集合 m 以及一对函数：

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

$$\eta :: 1 \rightarrow m$$

这里， 1 是 **Set** 中的终对象——单元素集合。第一个函数定义了乘法（它接收一对元素并返回它们的乘积），第二个选择 m 中的单位元。并不是所有具有这些签名的函数对都会产生一个 monoid。为此，我们需要施加附加条件：结合律和单位律。但让我们暂时忘记这些，只考虑“潜在的 monoid”。一对函数是两个函数集的笛卡尔积的元素。我们知道，这些集合可以表示为指数对象：

$$\mu \in m^{m \times m}$$

$$\eta \in m^1$$

这两个集合的笛卡尔积为：

$$m^{m \times m} \times m^1$$

利用一些高中代数（这在每个笛卡尔闭范畴中都适用），我们可以将其重写为：

$$m^{m \times m + 1}$$

加号表示 **Set** 中的余积。我们刚刚用一个函数代替了一对函数——即集合的一个元素：

$$m \times m + 1 \rightarrow m$$

这个函数集中的任何元素都是一个潜在的 monoid。

这种表述的美妙之处在于它引出了有趣的推广。例如，我们如何用这种语言描述群（group）呢？群是一个具有附加函数的 monoid，该函数为每个元素分配逆元。后者是类型 $m \rightarrow m$ 的函数。举个例子，整数构成了一个加法群，零是单位元，取反是逆元。为了定义一个群，我们将从一组三元函数开始：

$$\begin{aligned}m \times m &\rightarrow m \\m &\rightarrow m \\1 &\rightarrow m\end{aligned}$$

和以前一样，我们可以将这些三元组合成一个函数集：

$$m \times m + m + 1 \rightarrow m$$

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（单位元——这里是零）开始。我们将它们组合成一个函数。所有具有这个签名的函数都定义了潜在的群。

我们可以继续这样做。例如，要定义一个环（ring），我们可以添加另一个二元运算符和一个零元运算符，依此类推。每次我们都以一个函数类型结束，其左侧是幂的总和（可能包括零次幂——即终对象），而右侧是集合本身。

现在我们可以进行广泛的推广。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 元乘积出发的态射。这意味着我们需要一个支持有限积的范畴。对于零元运算符，我们要求终对象的存在。因此我们需要一个笛卡尔范畴。为了组合这些运算符，我们需要指数对象，所以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诡计。

或者，我们可以忘记我们推导公式的方式，专注于最终结果。态射左侧的乘积之和定义了一个端函子。如果我们选择一个任意的端函子 F ，会发生什么呢？在这种情况下，我们不需要对我们的范畴施加任何约束。我们得到的就是所谓的 F -代数。

一个 F -代数是由一个端函子 F 、一个对象 a 以及一个态射组成的三元组

$$Fa \rightarrow a$$

这个对象通常被称为载体（carrier）、基础对象或在编程上下文中称为载体类型。态射通常被称为求值函数或结构映射。可以将函子 F 看作是形成表达式的结构，而态射则用于对它们求值。

以下是 Haskell 中对 F -代数的定义：

snippet01 它将代数与其求值函数等同。

在 monoid 示例中，所讨论的函子为：

snippet02 这在 Haskell 中表示 $1 + a \times a$ （请记住 代数数据结构）。

一个环将使用以下函子定义：

snippet03 这在 Haskell 中表示 $1 + 1 + a \times a + a \times a + a$ 。

整数集是环的一个例子。我们可以选择 作为载体类型，并定义求值函数为：

snippet04 基于同一函子，还有更多的 F-代数。例如，多项式构成了一个环，方阵也是如此。

如你所见，函子的作用是生成可以使用代数求值器求值的表达式。到目前为止，我们只看到了非常简单的表达式。我们通常对可以使用递归定义的更复杂的表达式感兴趣。

24.1 递归 (Recursion)

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。例如，一个环中的任意表达式由这个类树状的数据结构生成：

snippet05 我们可以将原始的环求值器替换为其递归版本：

snippet06 这仍然不太实用，因为我们被迫将所有整数表示为 1 的和，但在紧要关头可以使用。

但是，我们如何使用 F-代数的语言描述表达式树呢？我们必须以某种形式化替换我们的函子定义中自由类型变量的过程，递归地用替换结果填充这个类型变量。想象分步进行这个过程。首先，定义一个深度为 1 的树为：

snippet07 我们将 的定义中用 生成的深度为 0 的树填入空缺中。类似地，深度为 2 的树也可以这样得到：

snippet08 我们还可以写成：

snippet09 继续这个过程，我们可以写出一个符号方程：

n+1n

从概念上讲，在重复这个过程无穷次之后，我们最终得到。注意 不依赖于。我们旅程的起点无关紧要，最终总是到达同一个地方。这并不总是适用于任意范畴中的任意端函子，但在 Set 范畴中情况较好。

当然，这只是一个牵强的论点，我稍后会对其进行更严格的论证。

无限次地应用一个端函子生成一个不动点 (fixed point)，定义为：

$$\text{Fix } f = f(\text{Fix } f)$$

这个定义背后的直觉是，由于我们已经将 f 应用无穷多次以得到 $\text{Fix } f$ ，再应用一次也不会改变任何东西。在 Haskell 中，不动点的定义为：

snippet10 可以说，如果构造器的名称与定义的类型不同，可能会更易读，如：

snippet11 但我会坚持使用公认的符号。构造器（或，如果你喜欢这样）可以看作是一个函数：

snippet12 还有一个函数可以剥去一个层次的函子应用：

snippet13 这两个函数是彼此的逆运算。我们稍后会用到这些函数。

24.2 F-代数的范畴 (Category of F-Algebras)

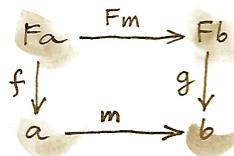
这是书中最古老的技巧之一：每当你想出一种构建新对象的方法时，看看它们是否构成了一个范畴。不出所料，给定端函子 F 的代数构成了一个范畴。该范畴中的对象是代数——由一个载体对象 a 和一个态射 $Fa \rightarrow a$ 组成的二元组，两者都来自原范畴 \mathbf{C} 。

为了完成这个图景，我们必须定义 F -代数范畴中的态射。态射必须将一个代数 (a, f) 映射到另一个代数 (b, g) 。我们将其定义为一个态射 m ，它映射载体——即从 a 到 b 在原范畴中的映射。并不是任何态射都可以做到：我们希望它与两个求值器兼容（我们称这种保持结构的态射为同态（homomorphism））。下面是 F -代数同态的定义。首先，注意我们可以将 m 提升为映射：

$$Fm :: Fa \rightarrow Fb$$

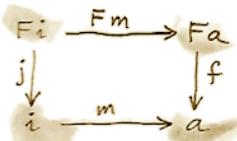
然后我们可以用 g 来得到 b 。或者，我们可以用 f 从 Fa 到 a ，然后跟随它使用 m 。我们希望这两条路径相等：

$$g \circ Fm = m \circ f$$



你很容易说服自己这确实是一个范畴（提示： \mathbf{C} 中的恒等态射工作得很好，且同态的组合仍是同态）。

F -代数范畴中的初始对象（如果存在）称为初始代数（initial algebra）。我们称这个初始代数的载体为 i ，其求值器为 $j :: Fi \rightarrow i$ 。事实证明，初始代数的求值器 j 是一个同构（isomorphism）。这一结果称为 Lambek 定理。证明依赖于初始对象的定义，它要求存在唯一的同态 m 从它映射到任何其他 F -代数。因为 m 是同态，下面的图必须交换：



现在让我们构造一个载体为 Fi 的代数。此类代数的求值器必须是从 $F(Fi)$ 到 Fi 的态射。我们可以通过提升 j 来轻松构造这样的求值器：

$$Fj :: F(Fi) \rightarrow Fi$$

由于 (i, j) 是初始代数，必须存在一个唯一的同态 m 从它映射到 (F_i, F_j) 。以下图必须交换：

$$\begin{array}{ccc} F_i & \xrightarrow{F_m} & F(F_i) \\ j \downarrow & & \downarrow F_j \\ i & \xrightarrow{m} & F_i \end{array}$$

但我们还有这个可交换的图（两条路径是相同的！）：

$$\begin{array}{ccc} F(F_i) & \xrightarrow{F_j} & F_i \\ F_j \downarrow & & \downarrow j \\ F_i & \xrightarrow{j} & i \end{array}$$

这可以解释为 j 是一个将 (F_i, F_j) 映射到 (i, j) 的代数同态。我们可以将这两个图拼接在一起：

$$\begin{array}{ccccc} F_i & \xrightarrow{F_m} & F(F_i) & \xrightarrow{F_j} & F_i \\ j \downarrow & & F_j \downarrow & & \downarrow j \\ i & \xrightarrow{m} & F_i & \xrightarrow{j} & i \end{array}$$

这个图可以解释为 $j \circ m$ 是代数同态。在这种情况下，这两个代数是相同的。此外，因为 (i, j) 是初始对象，只能存在一个从它映射到自身的同态，那就是恒等态射 id_i ——我们知道这是代数同态。因此 $j \circ m = \text{id}_i$ 。利用这个事实和左侧图的可交换性质，我们可以证明 $m \circ j = \text{id}_{F_i}$ 。这表明 m 是 j 的逆运算，因此 j 是 F_i 和 i 之间的同构：

$$F_i \cong i$$

这仅仅表明 i 是 F 的一个不动点 (fixed point)。这是对原先牵强论点的正式证明。

回到 Haskell：我们将 i 识别为我们的，将 j 识别为我们的构造器，将其逆运算识别为。Lambek 定理中的同构告诉我们，为了得到初始代数，我们取函子 f 并用 替换其参数 a 。我们还看到了为什么不动点不依赖于 a 。

24.3 自然数 (Natural Numbers)

自然数也可以定义为一个 F-代数。起点是两个态射的对：

$$\text{zero} :: 1 \rightarrow N$$

$$\text{succ} :: N \rightarrow N$$

第一个选择零，第二个将所有数字映射到其后继数。和以前一样，我们可以将两者组合成一个：

$$1 + N \rightarrow N$$

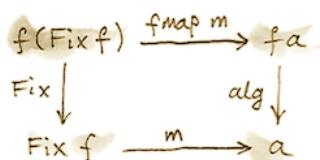
左侧定义了一个函子，在 Haskell 中可以写成这样：

snippet14 这个函子的固定点（即它生成的初始代数）可以在 Haskell 中编码为：

snippet15 自然数要么是零，要么是另一个数字的后继数。这就是所谓的 Peano 表示法。

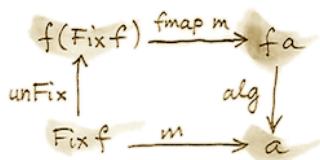
24.4 Catamorphisms

让我们使用 Haskell 符号重写初始性条件。我们将初始代数称为。它的求值器是构造器。从初始代数到任何其他代数的唯一态射是。



顺便说一句，注意 是什么：它是整个递归表达式树的求值器。让我们找到一种实现它的通用方法。

Lambek 定理告诉我们，构造器 是同构。我们称其逆运算为。因此，我们可以翻转这个图中的一条箭头得到：



让我们写下这个图的可交换性条件：

我们可以将这个方程解释为 的递归定义。对于由函子 创建的任何有限树，这个递归必然会终止。我们可以通过注意到 在函子 的顶层之下操作来看到这一点。换句话说，它作用于原始树的子节点。子节点总是比原始树浅一层。

当我们将 应用于用 构造的树时会发生什么。操作 剥去了构造器，暴露了树的顶层。然后我们将 应用于顶节点的所有子节点。这会产生类型为 的结果。最后，我们通过应用非递归求值器 来组合这些结果。关键点是我们的求值器 是一个简单的非递归函数。

既然我们可以对任何代数 执行此操作，那么定义一个高阶函数是有意义的，该函数以代数为参数并为我们提供我们称之为 的函数。这个高阶函数称为 **catamorphism**：

snippet16 让我们看看一个例子。采用定义自然数的函子：

snippet17 让我们选择 作为载体类型，并定义我们的代数为：

snippet18 你很容易说服自己，这个代数的 **catamorphism**，，计算斐波那契数。

一般来说， 的代数定义了递推关系：当前元素的值以之前的元素为依据。然后， **catamorphism** 计算该序列的第 n 个元素。

24.5 折叠 (Folds)

的列表是以下函子的初始代数：

snippet19 实际上，将变量 替换为递归的结果，我们称之为 ，我们得到：

snippet20 列表函子的代数为一个特定的载体类型选择定义一个函数，该函数对两个构造器进行模式匹配。它对 的值告诉我们如何计算空列表，对 的值告诉我们如何将当前元素与先前累积的值组合。

例如，下面是一个可以用于计算列表长度的代数（载体类型为）：

snippet21 实际上， **catamorphism** 计算列表的长度。请注意，求值器是 (1) 一个接受列表元素和累加器并返回新累加器的函数与 (2) 一个起始值，这里为零的组合。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：

snippet22 的两个参数正好是代数的两个组成部分。

让我们尝试另一个例子：

snippet23 再次比较：

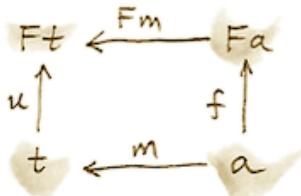
snippet24 正如你所见，只是一个对列表的 **catamorphism** 的便利特化。

24.6 F-Coalgebras

一如既往，我们有一个 F-余代数 (F-coalgebra) 的对偶构造，其中态射的方向是相反的：

$$a \rightarrow Fa$$

给定函子的余代数也构成一个范畴，具有保持余代数结构的同态。该范畴中的终端对象 (t, u) 称为终端余代数。对于每个其他代数 (a, f) ，存在唯一的同态 m 使以下图交换：



终端余代数是函子的一个不动点，因为态射 $u :: t \rightarrow Ft$ 是一个同构 (Lambek 的余代数定理)：

$$Ft \cong t$$

终端余代数通常在编程中解释为生成（可能是无限的）数据结构或转移系统的配方。

就像 catamorphism 可以用于求值初始代数一样，anamorphism 可以用于余求值终端余代数：

snippet25 余代数的一个典型例子基于一个固定点为元素类型 的无限流 (stream) 的函子。这个函子为：

snippet26 其固定点为：

snippet27 的余代数是一个函数，它接收类型为 的种子并生成一个对（是对的别名），包括一个元素和下一个种子。

你可以轻松生成简单的余代数来产生无限序列，例如平方数列表或倒数。

一个更有趣的例子是一个余代数，它产生一个素数列表。技巧是使用无限列表作为载体。我们的起始种子将是列表。下一个种子将是此列表的尾部，所有 2 的倍数都被移除。这是以 3 开始的奇数列表。在下一步中，我们将取此列表的尾部并移除所有 3 的倍数，依此类推。你可能认出这是埃拉托色尼筛法的雏形。这个余代数由以下函数实现：

snippet28 这个余代数的 anamorphism 生成素数列表：

snippet29 流是无限列表，因此应该可以将其转换为 Haskell 列表。为此，我们可以使用相同的函子 形成一个代数，并可以在其上运行一个 catamorphism。例如，这是一个将流转换为列表的 catamorphism：

snippet30 在这里，同一个不动点同时是同一端函子的初始代数和终端余代数。在任意范畴中情况并非总是如此。通常，端函子可能有很

多（或没有）不动点。初始代数是所谓的最小不动点，终端余代数是最大的固定点。然而，在 Haskell 中，两者由相同的公式定义，并且它们重合。

列表的 anamorphism 称为 unfold。为了创建有限列表，该函子被修改为生成一个对：

snippet31 的值将终止列表的生成。

与镜头（lens）相关的余代数是一个有趣的例子。镜头可以表示为一个 getter 和 setter 的对：

snippet32 这里，通常是某种具有类型 α 的字段的产品数据类型。getter 获取该字段的值，setter 用一个新值替换此字段。这两个函数可以组合成一个：

snippet33 我们可以进一步重写此函数为：

snippet34 其中我们定义了一个函子：

snippet35 请注意，这不是由乘积和余积构造的简单代数函子。它涉及一个指数 a^s 。

镜头是这个函子的余代数，载体类型为 α 。我们之前看到过也是一个余单子（comonad）。事实证明，一个良好行为的镜头对应于一个与余单子结构兼容的余代数。在下一节中，我们将讨论这一点。

24.7 挑战 (Challenges)

1. 实现一个单变量多项式环的求值函数。你可以将多项式表示为一个系数列表，其前面是 x 的幂。例如， $4x^2 - 1$ 可以表示为（从零次幂开始）。
2. 将前面的构造推广到多变量多项式，如 $x^2y - 3y^3z$ 。
3. 实现 2×2 矩阵环的代数。
4. 定义一个余代数，其 anamorphism 生成自然数的平方列表。
5. 使用生成前 n 个素数的列表。

25

Algebras for Monads

如果我们将自函子（endofunctors）解释为定义表达式的方法，那么代数（algebras）允许我们对它们进行求值，而 Monad 则允许我们构造和操作它们。通过将代数与 Monad 结合，我们不仅获得了许多功能，还可以解答一些有趣的问题。

其中一个问题涉及 Monad 和伴随（adjunctions）之间的关系。正如我们所见，每个伴随关系都会定义一个 Monad（以及一个 comonad）。问题是：是否每个 Monad（或 comonad）都可以从伴随关系中导出？答案是肯定的。有一整组伴随关系可以生成给定的 Monad。我将向你展示两种这样的伴随关系。



让我们回顾一下定义。Monad 是一个自函子 m ，配备了两个满足某些一致性的自然变换（natural transformations）。这些变换在 a 处的分量是：

$$\begin{aligned}\eta_a &:: a \rightarrow m a \\ \mu_a &:: m(m a) \rightarrow m a\end{aligned}$$

同一个自函子的代数是选择一个特定对象——载体 a ，以及一个态射：

$$alg :: m a \rightarrow a$$

首先要注意的是，代数的方向与 η_a 相反。直观上讲， η_a 从类型为 a 的值创建一个平凡的表达式。使代数与 Monad 兼容的第一个一致性条件确保使用载体为 a 的代数对该表达式进行求值时，我们会得到原始值：

$$alg \circ \eta_a = \mathbf{id}_a$$

第二个条件来自于这样一个事实，即有两种方法可以对双重嵌套的表达式 $m(m a)$ 进行求值。我们可以先应用 μ_a 来展平表达式，然后使用代数的求值器；或者我们可以先应用提升的求值器来求值内部表达式，然后将求值器应用于结果。我们希望这两种策略是等价的：

$$alg \circ \mu_a = alg \circ m \ alg$$

这里， alg 是通过使用函子 m 提升 alg 而得到的态射。以下交换图描述了这两个条件（我将 m 替换为 T ，以预示接下来的内容）：

$$\begin{array}{ccc} a & \xrightarrow{\eta_a} & Ta \\ & \searrow \text{alg} & \downarrow \\ & a & \end{array} \quad \begin{array}{ccc} T(Ta) & \xrightarrow{T \ alg} & Ta \\ \downarrow \mu_a & & \downarrow \\ Ta & \xrightarrow{alg} & a \end{array}$$

我们也可以在 Haskell 中表达这些条件：

`snippet01` 让我们看一个小例子。列表自函子的代数包括某个类型和一个从 列表生成 的函数。我们可以通过选择元素类型和累加器类型都为，使用 来表达这个函数：

`snippet02` 这个特定的代数由一个两参数函数（我们称之为）和一个值 来指定。列表函子恰好也是一个 Monad，其中 将值转化为一个单元素列表。代数（这里是）在 之后的组合将 转换为：

`snippet03` 这里 的操作以中缀表示。代数与 Monad 兼容的条件是，对于每个，以下一致性条件必须得到满足：

`snippet04` 如果我们将 视为一个二元操作符，这个条件告诉我们它是右单位元。

第二个一致性条件作用于列表的列表。 的作用是将各个列表连接起来。然后我们可以折叠（fold）结果列表。另一种方式是先折叠各个列表，然后折叠结果列表。同样，如果我们将 解释为一个二元操作符，这个条件告诉我们这个二元操作是结合律的。当 是一个 Monoid 时，这些条件肯定得到了满足。

25.1 T-代数 (T-algebras)

由于数学家更喜欢将他们的 Monad 称为 T ，他们将与之兼容的代数称为 T-代数。给定 Monad T 的 T-代数在范畴 C 中形成一个范畴，称

为 Eilenberg-Moore 范畴，通常表示为 \mathbf{C}^T 。该范畴中的态射是代数的同态。这些与我们之前为 F-代数定义的同态是一样的。

T -代数是一个载体对象和一个求值器（即对的 (a, f) ）。从 \mathbf{C}^T 到 \mathbf{C} 存在一个明显的遗忘函子（forgetful functor） U^T ，它将 (a, f) 映射到 a 。它还将 T -代数同态映射到 \mathbf{C} 中载体对象之间的相应态射。你可能还记得我们讨论伴随关系时提到过，遗忘函子的左伴随称为自由函子（free functor）。

U^T 的左伴随称为 F^T 。它将范畴 \mathbf{C} 中的一个对象 a 映射到 \mathbf{C}^T 中的一个自由代数。这个自由代数的载体是 Ta 。它的求值器是从 $T(Ta)$ 返回到 Ta 的态射。由于 T 是一个 Monad，我们可以使用 Monad 的 μ_a （在 Haskell 中是）作为求值器。

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：

$$\begin{aligned} alg \circ \eta_{Ta} &= \mathbf{id}_{Ta} \\ alg \circ \mu_a &= alg \circ T alg \end{aligned}$$

但如果将 μ 插入代数，这些条件就是 Monad 的定律。

你可能还记得，每个伴随关系都会定义一个 Monad。事实证明， F^T 和 U^T 之间的伴随关系定义了用于构造 Eilenberg-Moore 范畴的 Monad T 。由于我们可以对每个 Monad 执行此构造，因此我们得出结论：每个 Monad 都可以从一个伴随关系中生成。稍后我将向你展示另一种生成相同 Monad 的伴随关系。

这是计划：首先我将向你展示 F^T 确实是 U^T 的左伴随。我将通过定义这个伴随的单元（unit）和余单元（counit），并证明相应的三角恒等式得以满足。然后我将向你展示由此伴随关系生成的 Monad 确实是我们原来的 Monad。

伴随的单元是自然变换：

$$\eta : I \rightarrow U^T \circ F^T$$

让我们计算这个变换的 a 分量。恒等函子给我们 a 。自由函子产生自由代数 (Ta, μ_a) ，而遗忘函子将其简化为 Ta 。总之，我们得到一个从 a 到 Ta 的映射。我们将简单地使用 Monad T 的单元作为此伴随的单元。

让我们看看余单元：

$$\varepsilon : F^T \circ U^T \rightarrow I$$

让我们计算其在某个 T -代数 (a, f) 处的分量。遗忘函子遗忘了 f ，自由函子生成对 (Ta, μ_a) 。因此，为了定义余单元 ε 在 (a, f) 处的分量，我们需要在 Eilenberg-Moore 范畴中合适的态射，或者是 T -代数的同态：

$$(Ta, \mu_a) \rightarrow (a, f)$$

这种同态应该将载体 Ta 映射到 a 。让我们复活被遗忘的求值器 f 。这次我们将其用作 T-代数的同态。事实上，使 f 成为 T-代数的同一交换图可以重新解释为表明它是 T-代数的同态：

$$\begin{array}{ccc} T(Ta) & \xrightarrow{Tf} & Ta \\ \downarrow \mu_a & & \downarrow f \\ Ta & \xrightarrow{f} & a \end{array}$$

因此，我们定义了自然变换 ε 在 (a, f) (T-代数范畴中的一个对象) 处的分量为 f 。

要完成这个伴随关系，我们还需要证明单元和余单元满足三角恒等式。这些是：

$$\begin{array}{ccc} Ta & \xrightarrow{T\eta_a} & T(Ta) \\ & \searrow & \downarrow \mu_a \\ & & Ta \end{array} \quad \begin{array}{ccc} a & \xrightarrow{\eta_a} & Ta \\ & \swarrow & \downarrow f \\ & & a \end{array}$$

第一个恒等式成立是因为 Monad T 的单位律。第二个恒等式只是 T-代数 (a, f) 的定律。

我们已经证明了这两个函子构成了一个伴随关系：

$$F^T \dashv U^T$$

每个伴随关系都会产生一个 Monad。环形往返

$$U^T \circ F^T$$

是范畴 C 中产生相应 Monad 的自函子。让我们看看它对对象 a 的作用。由 F^T 创建的自由代数是 (Ta, μ_a) 。遗忘函子 U^T 丢弃求值器。因此，确实如此：

$$U^T \circ F^T = T$$

正如预期的那样，此伴随关系的单元是 Monad T 的单元。

你可能还记得，伴随关系的余单元通过以下公式产生 Monad 乘法：

$$\mu = R \circ \varepsilon \circ L$$

这是三个自然变换的水平组合，其中两个是将 L 映射到 L 和将 R 映射到 R 的恒等自然变换。中间的那个，余单元，是一个自然变换，其在代数 (a, f) 处的分量是 f 。

让我们计算 μ_a 的分量。我们首先在 F^T 之后水平组合 ε , 结果是 ε 在 $F^T a$ 处的分量。由于 F^T 将 a 映射到代数 (Ta, μ_a) , 而 ε 选择求值器, 我们最终得到 μ_a 。在左侧与 U^T 的水平组合不会改变任何内容, 因为 U^T 对态射的作用是平凡的。所以, 确实由此伴随关系得到的 μ 与原始 Monad T 的 μ 是相同的。

25.2 Kleisli 范畴 (The Kleisli Category)

我们之前已经见过 Kleisli 范畴。这是从另一个范畴 C 和一个 Monad T 构建的范畴。我们将此范畴称为 C_T 。Kleisli 范畴 C_T 中的对象是 C 的对象, 但态射有所不同。Kleisli 范畴中从 a 到 b 的态射 f_K 对应于原始范畴中从 a 到 Tb 的态射 f 。我们称这个态射为从 a 到 b 的 Kleisli 箭头。

Kleisli 范畴中态射的组合是通过 Kleisli 箭头的 Monad 组合定义的。例如, 让我们在 Kleisli 范畴中组合 g_K 和 f_K 。在 Kleisli 范畴中我们有:

$$\begin{aligned} f_K &:: a \rightarrow b \\ g_K &:: b \rightarrow c \end{aligned}$$

这在范畴 C 中对应于:

$$\begin{aligned} f &:: a \rightarrow Tb \\ g &:: b \rightarrow Tc \end{aligned}$$

我们将组合定义为:

$$h_K = g_K \circ f_K$$

作为 C 中的一个 Kleisli 箭头

$$\begin{aligned} h &:: a \rightarrow Tc \\ h &= \mu \circ (Tg) \circ f \end{aligned}$$

在 Haskell 中我们可以这样写:

snippet05 从 C 到 C_T 存在一个函子 F , 它在对象上是平凡的。在态射上, 它将 C 中的态射 f 映射到 C_T 中的态射, 方法是创建一个将 f 的返回值装饰 (embellish) 的 Kleisli 箭头。给定一个态射:

$$f :: a \rightarrow b$$

它在 C_T 中创建一个具有相应 Kleisli 箭头的态射:

$$\eta \circ f$$

在 Haskell 中我们可以这样写:

snippet06 我们还可以定义一个从 C_T 回到 C 的函子 G 。它将 Kleisli 范畴中的对象 a 映射到 C 中的对象 Ta 。它对 Kleisli 箭头 f_K (对应的态射为:

$$f :: a \rightarrow Tb$$

) 在 C 中的态射是:

$$Ta \rightarrow Tb$$

给定的定义是先提升 f , 然后应用 μ :

$$\mu_{Tb} \circ Tf$$

在 Haskell 表示中, 这样写:

T

你可能认出这是用 定义 Monad bind 的方式。

很容易看出这两个函子形成了一个伴随关系:

$$F \dashv G$$

并且它们的组合 $G \circ F$ 再现了原始 Monad T 。

因此, 这是第二个生成相同 Monad 的伴随关系。实际上, 有一个范畴 $\text{Adj}(C, T)$, 其中的伴随关系都产生相同的 Monad T 在范畴 C 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 伴随是终端对象。

25.3 Comonad 的余代数 (Coalgebras for Comonads)

类似的构造可以应用于任何 comonad W 。我们可以定义一个与 comonad 兼容的余代数 (coalgebra) 范畴。它们使以下交换图成立:

$$\begin{array}{ccc} a & \xleftarrow{\epsilon_a} & Wa \\ \nearrow a & \uparrow coa & \uparrow \delta_a \\ W(Wa) & \xleftarrow{W \text{ coa}} & Wa \\ & \uparrow coa & \uparrow coa \\ & Wa & a \end{array}$$

其中, coa 是载体为 a 的余代数的余求值态射:

$$coa :: a \rightarrow Wa$$

ϵ 和 δ 是定义 comonad 的两个自然变换 (在 Haskell 中, 它们的分量分别称为 和)。

从这些余代数范畴到 \mathbf{C} 存在一个明显的遗忘函子 U^W 。它只会遗忘余求值态射。我们将考虑它的右伴随 F^W 。

$$U^W \dashv F^W$$

遗忘函子的右伴随称为余自由函子 (cofree functor)。 F^W 生成余自由余代数。它将 \mathbf{C} 中的对象 a 分配到余代数 (Wa, δ_a) 。该伴随关系通过组合 $U^W \circ F^W$ 再现了原始 comonad。

类似地，我们可以构建一个带有 co-Kleisli 箭头的 co-Kleisli 范畴，并通过相应的伴随关系重新生成 comonad。

25.4 Lenses

让我们回到对 lenses 的讨论。一个 lens 可以写作一个余代数：

$$\text{coalg}_s :: a \rightarrow \text{Store } s \ a$$

对于函子 $\text{Store } s$ ：

snippet07 这个余代数也可以表示为一对函数：

$$\begin{aligned} \text{set} &:: a \rightarrow s \rightarrow a \\ \text{get} &:: a \rightarrow s \end{aligned}$$

(可以将 a 理解为“全体”(all)，将 s 理解为其“小”部分。) 在这对函数的术语中，我们有：

$$\text{coalg}_s \ a = \text{Store} (\text{set } a) (\text{get } a)$$

这里， a 是类型 a 的一个值。注意部分应用的 是一个从 s 到 a 的函数。

我们还知道 $\text{Store } s$ 是一个 comonad：

snippet08 问题是：在什么条件下，lens 是这个 comonad 的余代数？第一个一致性条件：

$$\varepsilon_a \circ \text{coalg} = \mathbf{id}_a$$

可以翻译为：

$$\text{set } a (\text{get } a) = a$$

这是 lens 定律，它表达了这样一个事实：如果你将结构 a 的字段设置为其先前的值，那么什么也不会改变。

第二个条件：

$$\text{fmap coalg} \circ \text{coalg} = \delta_a \circ \text{coalg}$$

需要更多的工作。首先，回忆 函子的 定义：

snippet09 将 应用于 的结果会给出：

snippet10 另一方面，将 应用于 的结果会生成：

snippet11 要使这两个表达式相等，下面的两个函数在作用于任意时必须相等：

snippet12 展开，我们得到：

snippet13 这相当于剩下的两个 lens 定律。第一个是：

snippet14 告诉我们，设置字段值两次与设置一次是相同的。第二个定律：

snippet15 告诉我们，获取字段的值将字段设置为 s 会返回 s 。

换句话说，一个良构的 lens 确实是 函子的 comonad 余代数。

25.5 Challenges

1. 讨论自由函子 $F :: C \rightarrow C^T$ 在态射上的作用。提示：使用 Monad μ 的自然性条件。
2. 定义伴随关系：

$$U^W \dashv F^W$$

3. 证明上述伴随关系再现了原始 comonad。

26

Ends and Coends

许多直觉可以应用于范畴中的态射，但我们可以同意，如果对象 a 和对象 b 之间存在态射，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。在任何偏序集范畴中，这一点都很明显，其中态射本身就是一种关系。一般来说，在两个对象之间可能有许多关于同一关系的“证明”。这些证明构成了我们称之为态射集（hom-set）的集合。当我们改变对象时，我们得到一个从对象对到“证明”集合的映射。这个映射是函子性的——在第一个参数中是反变的，在第二个参数中是协变的。我们可以将其视为在范畴中建立的一种全局关系。这种关系由态射函子（hom-functor）描述：

$$C(-, =) :: C^{op} \times C \rightarrow \mathbf{Set}$$

通常，任何这样的函子都可以解释为在范畴中建立一种关系。这种关系还可以涉及两个不同的范畴 C 和 D 。描述这种关系的函子具有以下签名，并称为预函子（profunctor）：

$$p :: D^{op} \times C \rightarrow \mathbf{Set}$$

数学家们说它是从 C 到 D 的预函子（注意符号的倒置），并使用带斜线的箭头作为它的符号：

$$C \nrightarrow D$$

你可以将预函子视为 C 和 D 中对象之间的证明相关关系，其中集合的元素象征着这种关系的证明。每当 $p a b$ 为空时， a 和 b 之间就不存在关系。请记住，关系不必是对称的。

另一种有用的直觉是将 endofunctor 视为容器的想法进行推广。类型 $p a b$ 的预函子值可以被认为 b 的容器，其键由类型 a 的元素表示。特别是，态射预函子的一个元素是从 a 到 b 的函数。

在 Haskell 中，预函子被定义为一个带有 方法的二参类型构造器，该方法提升了一对函数，第一个函数是以“错误”的方向作用的：

snippet01 预函子的函子性告诉我们，如果我们有证明 与 相关的证明，那么只要存在从 到 的态射和从 到 的另一个态射，我们就可以得到证明 与 相关的证明。或者，我们可以将第一个函数视为将新键转换为旧键，第二个函数则修改容器的内容。

对于在一个范畴内作用的预函子，我们可以从类型 $p \ a \ a$ 的对角元素中提取出大量信息。我们可以证明 b 与 c 相关，只要我们有一对态射 $b \rightarrow a$ 和 $a \rightarrow c$ 。更好的是，我们可以使用单个态射来到达非对角线值。例如，如果我们有一个态射 $f :: a \rightarrow b$ ，我们可以提升对 $\langle f, \text{id}_b \rangle$ 从 $p \ b \ b$ 到 $p \ a \ b$ ：

snippet02 或者我们可以提升对 $\langle \text{id}_a, f \rangle$ 从 $p \ a \ a$ 到 $p \ a \ b$ ：

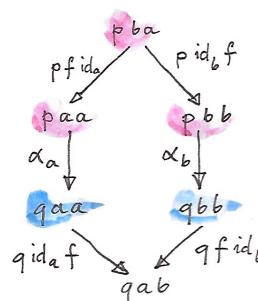
snippet03

26.1 双自然变换 (Dinatural Transformations)

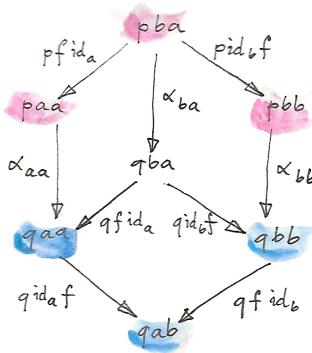
由于预函子是函子，我们可以按照标准方式定义它们之间的自然变换。然而，在许多情况下，仅定义两个预函子对角线元素之间的映射就足够了。这样的变换称为双自然变换，只要它满足反映我们可以将对角线元素连接到非对角线元素的两种方式的交换条件。两个预函子 p 和 q 之间的双自然变换，是函子范畴 $[\mathbf{C}^{\text{op}} \times \mathbf{C}, \mathbf{Set}]$ 中的态射族：

$$\alpha_a :: p \ a \ a \rightarrow q \ a \ a$$

对于任意 $f :: a \rightarrow b$ ，下面的图表是交换的：



请注意，这比自然性条件要弱一些。如果 α 是 $[\mathbf{C}^{\text{op}} \times \mathbf{C}, \mathbf{Set}]$ 中的自然变换，那么上面的图表可以由两个自然性方块和一个函子性条件（预函子 q 保持组合性）构成：



请注意，自然变换 α 的分量在 $[C^{op} \times C, \text{Set}]$ 中是由一对对象 α_{ab} 索引的。而双自然变换则只对一个对象索引，因为它只映射预函子的对角线元素。

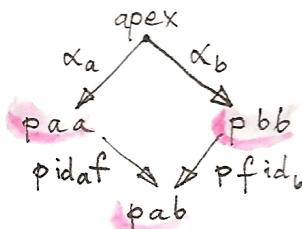
26.2 Ends

我们现在准备从“代数”进阶到范畴论的“微积分”。端的微积分 (ends 和 coends) 借用了传统微积分中的一些思想，甚至包括一些符号。特别是，coend 可以理解为一个无穷和或积分，而 end 类似于无穷乘积。甚至还有一些类似于狄拉克 δ 函数的东西。

end 是极限的推广，函数子被预函子所取代。代替锥体，我们有了楔形。楔形的底部由预函子 p 的对角线元素构成。楔形的顶点是一个对象（在这里，由于我们考虑的是 Set 值的预函子，所以是一个集合），而边是将顶点映射到基中的集合的函数族。你可以将这个族视为一个多态函数——一个在返回类型上多态的函数：

$$\alpha :: \forall a . apex \rightarrow p a a$$

与锥体不同，在楔形中我们没有连接基顶点的函数。然而，正如我们前面所看到的，给定范畴 C 中的任意态射 $f :: a \rightarrow b$ ，我们可以将 $p a a$ 和 $p b b$ 都连接到公共集合 $p a b$ 。因此，我们坚持要求以下图表交换：



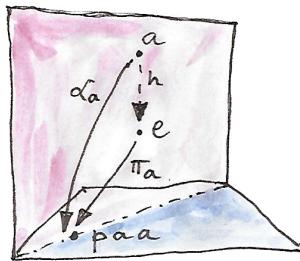
这被称为楔形条件。它可以写成：

$$p \mathbf{id}_a f \circ \alpha_a = p f \mathbf{id}_b \circ \alpha_b$$

或者，使用 Haskell 表示法：

snippet04 现在我们可以进行通用构造，并将 p 的 end 定义为通用楔形——一个集合 e 和一个函数族 π ，使得对于任何顶点为 a 的楔形和一个函数族 α ，都存在一个唯一的函数 $h :: a \rightarrow e$ 使得所有的三角形都交换：

$$\pi_a \circ h = \alpha_a$$



end 的符号是积分符号，“积分变量”位于下标位置：

$$\int_c p c c$$

π 的分量称为 end 的投影映射：

$$\pi_a :: \int_c p c c \rightarrow p a a$$

请注意，如果 C 是离散范畴（除了恒等态射外没有其他态射），那么 end 只是预函子 p 在整个范畴 C 中所有对角线项的全局积。稍后我将向你展示，在更一般的情况下，end 与通过等化子（equalizer）获得的积之间的关系。

在 Haskell 中，end 公式直接翻译为通用量词：

snippet05 严格来说，这只是 p 的所有对角线元素的乘积，但楔形条件由于参数性¹的原因自动满足。对于任何函数 $f :: a \rightarrow b$ ，楔形条件表示为：

snippet06 或者，带有类型注释：

bbaa

¹

这两个公式的类型都是：

snippet07 而 是多态投影：

snippet08 在这里，类型推断自动选择 的正确分量。

正如我们能够将锥体的所有交换条件表达为一个自然变换一样，我们也可以将所有楔形条件组合成一个双自然变换。为此，我们需要将常量函子 Δ_c 推广为常量预函子，将所有对象对映射到一个单一对象 c ，并将所有态射对映射到该对象的恒等态射。楔形是从该函子到预函子 p 的双自然变换。实际上，当我们意识到 Δ_c 将所有态射提升为一个恒等函数时，双自然性六边形会缩小为楔形菱形。

end 也可以为其他目标范畴定义，但在这里我们只考虑 **Set** 值的预函子及其 end。

26.3 Ends 作为等化子

end 定义中的交换条件可以使用等化子来书写。首先，让我们定义两个函数（我使用 Haskell 表示法，因为数学符号在这种情况下似乎不太友好）。这些函数对应于楔形条件的两个汇聚分支：

snippet09[b] 两个函数都将预函子 的对角线元素映射到多态函数类型：

snippet10 这些函数具有不同的类型。然而，如果我们形成一个大乘积类型，将所有的对角线元素聚集在一起，我们可以统一它们的类型：

snippet11 函数 和 诱导了从这个乘积类型的两个映射：

snippet12 的 end 是这两个函数的等化子。请记住，等化子选择了两个函数相等的最大子集。在这种情况下，它选择了 的所有对角线元素的乘积，其中楔形图交换。

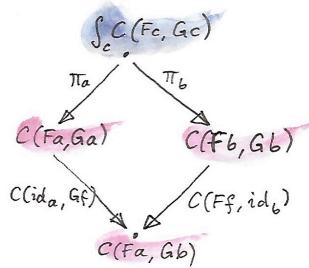
26.4 自然变换作为 Ends

end 最重要的一个例子是自然变换的集合。两个函子 F 和 G 之间的自然变换是从形如 $C(Fa, Ga)$ 的态射集选择的态射族。如果没有自然性条件，自然变换的集合只是这些态射集的乘积。实际上，在 Haskell 中就是这样：

snippet13 之所以在 Haskell 中起作用，是因为自然性是从参数性推导出来的。然而，在 Haskell 之外，并不是所有跨越这些态射集的对角线部分都能生成自然变换。但请注意，映射：

$$\langle a, b \rangle \rightarrow C(Fa, Gb)$$

是一个预函子，因此研究其 end 是有意义的。这是楔形条件：



让我们从集合 $\int_c \mathbf{C}(Fc, Gc)$ 中选择一个元素。这两个投影将此元素映射到一个特定变换的两个分量上，让我们称它们为：

$$\begin{aligned}\tau_a &:: Fa \rightarrow Ga \\ \tau_b &:: Fb \rightarrow Gb\end{aligned}$$

在左分支中，我们提升了一个态射对 (\mathbf{id}_a, Gf) ，使用态射函子。当作用于 τ_a 时，提升的对给我们：

$$Gf \circ \tau_a \circ \mathbf{id}_a$$

图的另一个分支给我们：

$$\mathbf{id}_b \circ \tau_b \circ Ff$$

它们的相等性由楔形条件要求，这正是 τ 的自然性条件。

26.5 Coends

如预期的那样，end 的对偶称为 coend。它是由楔形的对偶构建的，称为反楔形（cowedge，发音为 co-wedge，而不是 cow-edge）。



“反楔形”

coend 的符号是积分符号，“积分变量”位于上标位置：

$$\int^c p c c$$

就像 `end` 与乘积有关一样，`coend` 与并集或求和有关（在这方面，它类似于积分，它是和的极限）。与投影不同，`coend` 有从预函子的对角线元素到 `coend` 的注入。如果没有反楔形条件，我们可以说，预函子 p 的 `coend` 可能是 $p \alpha a$ 、 $p \beta b$ 、 $p \gamma c$ 等等。或者我们可以说，存在这样的 a ，使得 `coend` 只是集合 $p \alpha a$ 。我们在 `end` 定义中使用的通用量词在 `coend` 中变为存在量词。

这就是为什么，在伪 Haskell 中，我们会将 `coend` 定义为：

在 Haskell 中编码存在量词的标准方法是使用通用量词的数据构造器。因此我们可以定义：

snippet14 其逻辑是，应该可以使用 $p \alpha a$ 家族中的任何类型的值来构造一个 `coend`，无论我们选择了哪个 a 。

就像 `end` 可以用等化子来定义一样，`coend` 可以用反等化子 (*co-equalizer*) 来描述。所有的反楔形条件可以通过取 $p \alpha b$ 的一个巨大的并集，并为所有可能的 $b \rightarrow a$ 函数而满足。在 Haskell 中，可以表示为存在类型：

snippet15 有两种方法来计算这个和类型，即通过提升函数并应用于预函子 p ：

snippet16 其中，是预函子 p 的对角线元素的并集：

snippet17 `coend` 是这两个函数的反等化子。反等化子通过识别由同一参数应用 或 获得的值，自动满足反楔形条件。这里，参数是一个包含函数 $b \rightarrow a$ 和 $p \alpha b$ 元素的对。应用 和 生成的可能不同的 值。在 `coend` 中，这两个值被识别出来，使得反楔形条件自动满足。

在集合中识别相关元素的过程正式称为取商 (taking a quotient)。要定义商，我们需要一个等价关系 \sim ，它是自反的、对称的和传递的关系：

$$a \sim a$$

如果 $a \sim b$ 则 $b \sim a$

如果 $a \sim b$ 且 $b \sim c$ 则 $a \sim c$

这种关系将集合分割成等价类。每个类由彼此相关的元素组成。通过从每个类中选择一个代表来形成商集。一个经典的例子是有理数的定义，它们是由整数对构成，并具有以下等价关系：

$$(a, b) \sim (c, d) \text{ 当且仅当 } a * d = b * c$$

很容易验证这是一种等价关系。对 (a, b) 被解释为分数 $\frac{a}{b}$ ，分子和分母具有公约数的分数被识别为相同的有理数。一个有理数是这样的分数的等价类。

你可能还记得我们早先讨论极限和余极限时，态射函子是连续的，也就是说，它保持极限。对偶地，反变态射函子将余极限转换为极限。

这些属性可以推广到 ends 和 coends，它们分别是极限和余极限的推广。特别是，我们得到一个非常有用的恒等式，用于将 coends 转换为 ends：

$$\text{Set}\left(\int^x p \ x \ x, c\right) \cong \int_x \text{Set}(p \ x \ x, c)$$

让我们在伪 Haskell 中看看它：

\cong

它告诉我们，接受存在类型的函数等价于一个多态函数。这非常有意义，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一种类型。与此相同的原理告诉我们，接受和类型的函数必须作为 case 语句实现，其中包含一个处理程序元组，每种类型都有一个处理程序。在这里，和类型被 coend 替代，而处理程序族则成为 end 或多态函数。

26.6 忍者 Yoneda 引理

在 Yoneda 引理中出现的自然变换集合可以使用 end 进行编码，得到以下公式：

$$\int_z \text{Set}(\mathbf{C}(a, z), Fz) \cong Fa$$

还有一个对偶公式：

$$\int^z \mathbf{C}(z, a) \times Fz \cong Fa$$

这一定律与狄拉克 δ 函数的公式有着强烈的相似性（函数 $\delta(a - z)$ ，或更确切地说是一个分布，在 $a = z$ 时有一个无限的峰值）。在这里，态射函子起到了 δ 函数的作用。

这两个公式合在一起，有时被称为忍者 Yoneda 引理。

要证明第二个公式，我们将使用 Yoneda 镶嵌的结果，该结果指出，如果两个对象的态射函子同构，那么它们是同构的。换句话说， $a \cong b$ 如果且仅有且仅有一个类型为：

$$[\mathbf{C}, \text{Set}](\mathbf{C}(a, -), \mathbf{C}(b, =))$$

的自然变换，并且这个自然变换是一个同构。

我们首先将我们想要证明的恒等式的左侧插入到一个态射函子中，该态射函子将映射到某个任意对象 c ：

$$\text{Set}\left(\int^z \mathbf{C}(z, a) \times Fz, c\right)$$

使用连续性论证，我们可以将 coend 替换为 end：

$$\int_z \text{Set}(\mathbf{C}(z, a) \times Fz, c)$$

现在，我们可以利用乘积和指数之间的对合关系：

$$\int_z \mathbf{Set}(\mathbf{C}(z, a), c^{(Fz)})$$

我们可以使用 Yoneda 引理进行“积分”以得到：

$$c^{(Fa)}$$

(注意，我们使用了 Yoneda 引理的反变版本，因为函子 $c^{(Fz)}$ 是 z 的反变函子。) 该指数对象与态射集同构：

$$\mathbf{Set}(Fa, c)$$

最后，我们利用 Yoneda 镶嵌得出同构：

$$\int^z \mathbf{C}(z, a) \times Fz \cong Fa$$

26.7 预函子的组合

让我们进一步探讨预函子描述关系的想法——更确切地说，证明相关关系，这意味着集合 $p \ a \ b$ 代表证明 a 与 b 相关的证明集。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。我们说，通过组合 q 和 p ， a 与 b 相关，如果存在一个中介对象 c 使得 $q \ b \ c$ 和 $p \ c \ a$ 都不为空。这个新关系的证明是所有单个关系证明的对。因此，在理解存在量词对应于 `coend` 和两个集合的笛卡尔积对应于“证明对”的前提下，我们可以使用以下公式定义预函子的组合：

$$(q \circ p) \ a \ b = \int^c p \ c \ a \times q \ b \ c$$

以下是 中的 Haskell 定义，经过一些重命名：

snippet18 这是使用广义代数数据类型 (GADT 语法，其中一个自由类型变量 (此处为) 自动存在量化。数据构造器 因此等价于：

组合的单位是态射函子——这直接来自于忍者 Yoneda 引理。因此，有意义的是，问是否存在一个范畴，其中预函子作为态射。答案是肯定的，需要注意的是，预函子组合的结合律和单位律仅在自然同构的意义上成立。这样的范畴，法律在同构的基础上成立，被称为双范畴 (bicategory)，它比 2-范畴更为普遍。因此我们有一个双范畴 **Prof**，其中对象是范畴，态射是预函子，而态射之间的态射 (即所谓的二胞元) 是自然变换。事实上，我们甚至可以更进一步，因为除了预函子，我们还有常规函子作为范畴之间的态射。具有两种类型态射的范畴称为双范畴。

预函子在 Haskell 镜头库和箭头库中扮演着重要角色。

27

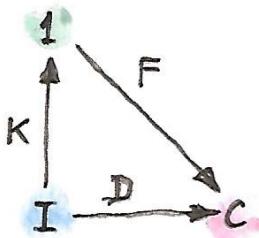
Kan Extensions

到

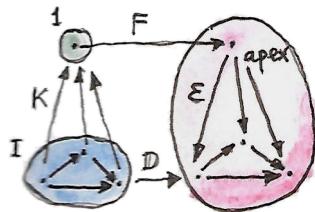
目前为止，我们主要在处理单一范畴（category）或一对范畴。在某些情况下，这有些过于限制。

例如，当我们在一个范畴 C 中定义极限（limit）时，我们引入了一个索引范畴（index category） I 作为模板，这个模板将构成我们圆锥体（cone）基础的模式。此时，引入另一个范畴（一个简单的范畴）来作为圆锥体顶点的模板是有意义的。然而，我们使用了从 I 到 C 的常量函子（constant functor） Δ_c 来解决这个问题。

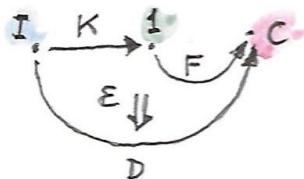
现在是时候修正这种尴尬了。让我们使用三个范畴来定义极限。首先，我们从索引范畴 I 到范畴 C 的函子 D 开始。这个函子选择了圆锥体的基础——即图函子（diagram functor）。



新的添加是范畴 1 ，它只包含一个对象（以及一个单一的恒等态射）。从 I 到这个范畴唯一可能的函子 K 将所有对象映射到 1 中唯一的对象，并将所有态射映射到恒等态射。任何从 1 到 C 的函子 F 选择了我们圆锥体的潜在顶点。

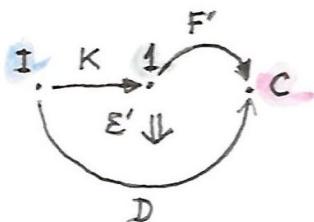


圆锥体是从 $F \circ K$ 到 D 的自然变换 (natural transformation) ε 。注意, $F \circ K$ 做的与我们最初 Δ_c 是一样的事情。下图展示了这一变换。



我们现在可以定义一个普遍性质 (universal property), 来选择最“优”的函子 F 。这个 F 会将 1 映射到对象 D 在范畴 C 中的极限上, 并且自然变换 ε 将提供相应的投影。这一普遍函子被称为函子 D 沿着 K 的右 Kan 扩展 (right Kan extension), 并用 $\mathbf{Ran}_K D$ 表示。

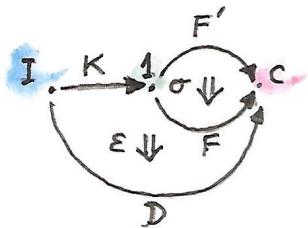
我们来表述一下这个普遍性质。假设我们有另一个圆锥体, 即另一个函子 F' 及其与 $F' \circ K$ 到 D 的自然变换 ε' 。



如果右 Kan 扩展 $F = \mathbf{Ran}_K D$ 存在, 那么必定存在一个从 F' 到它的唯一自然变换 σ , 使得 ε' 通过 ε 分解, 即:

$$\varepsilon' = \varepsilon \cdot (\sigma \circ K)$$

这里, $\sigma \circ K$ 是两个自然变换的水平组合 (其中一个是 K 上的恒等自然变换)。然后将此变换与 ε 垂直组合。



在分量上，作用在范畴 \mathbf{I} 中的对象 i 上时，我们得到：

$$\varepsilon'_i = \varepsilon_i \circ \sigma_{Ki}$$

在我们的例子中， σ 只有一个对应于 $\mathbf{1}$ 中单一对象的分量。所以，这的確是从 F' 定义的圆锥体顶点到 $\mathbf{Ran}_K D$ 定义的普遍圆锥体顶点的唯一态射。所需的交换条件正是极限定义所要求的。

但重要的是，我们可以将简单范畴 $\mathbf{1}$ 替换为任意范畴 \mathbf{A} ，右 Kan 扩展的定义依然有效。

27.1 右 Kan 扩展 (Right Kan Extension)

函子 $D :: \mathbf{I} \rightarrow \mathbf{C}$ 沿着函子 $K :: \mathbf{I} \rightarrow \mathbf{A}$ 的右 Kan 扩展是一个函子 $F :: \mathbf{A} \rightarrow \mathbf{C}$ (记作 $\mathbf{Ran}_K D$)，并伴随着一个自然变换

$$\varepsilon :: F \circ K \rightarrow D$$

使得对于任何其他函子 $F' :: \mathbf{A} \rightarrow \mathbf{C}$ 和自然变换

$$\varepsilon' :: F' \circ K \rightarrow D$$

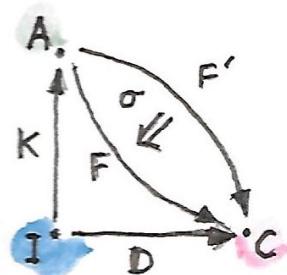
都有一个唯一的自然变换

$$\sigma :: F' \rightarrow F$$

使得 ε' 通过 ε 分解：

$$\varepsilon' = \varepsilon \cdot (\sigma \circ K)$$

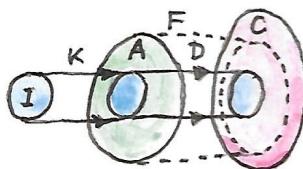
这听起来有点复杂，但可以用下图来直观地表示：



一个有趣的观点是，注意到在某种意义上，Kan 扩展起到了“函子乘法”逆操作的作用。有些作者甚至使用 D/K 来表示 $\mathbf{Ran}_K D$ 。确实，用这种表示法，右 Kan 扩展的定义 ε （也称为右 Kan 扩展的余单元（counit））看起来像是简单的消去：

$$\varepsilon :: D/K \circ K \rightarrow D$$

对 Kan 扩展还有另一种解释。考虑函子 K 嵌入范畴 \mathbf{I} 到 \mathbf{A} 中。在最简单的情况下， \mathbf{I} 可能只是 \mathbf{A} 的一个子范畴。我们有一个函子 D 将 \mathbf{I} 映射到 \mathbf{C} 。我们能否将 D 扩展为定义在整个 \mathbf{A} 上的函子 F ? 理想情况下，这样的扩展会使得 $F \circ K$ 与 D 同构。换句话说， F 会将 D 的定义域扩展到 \mathbf{A} 上。但通常来说，完全的同构往往过于苛刻，我们可以只要求一个方向的自然变换 ε 从 $F \circ K$ 到 D （左 Kan 扩展取的是另一方向）。



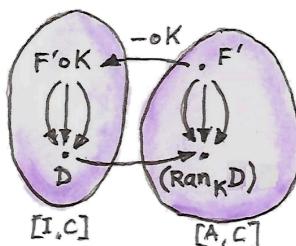
当然，当函子 K 在对象上不单射或在同态集（hom-sets）上不忠实时，嵌入的概念就会失效，例如在极限的例子中。此时，Kan 扩展会尽其所能地推测丢失的信息。

27.2 Kan 扩展作为伴随 (Adjunction)

现在假设右 Kan 扩展对于任何 D （和一个固定的 K ）都存在。在这种情况下， \mathbf{Ran}_{K-} （用破折号替代 D ）是一个从函子范畴 $[\mathbf{I}, \mathbf{C}]$ 到函子范畴 $[\mathbf{A}, \mathbf{C}]$ 的函子。事实证明，这个函子是前合成函子（precomposition functor） $- \circ K$ 的右伴随。后者将 $[\mathbf{A}, \mathbf{C}]$ 中的函子映射到 $[\mathbf{I}, \mathbf{C}]$ 中。这个伴随关系是：

$$[\mathbf{I}, \mathbf{C}](F' \circ K, D) \cong [\mathbf{A}, \mathbf{C}](F', \mathbf{Ran}_K D)$$

这只是对我们称为 ε' 的自然变换对应唯一的自然变换 σ 这一事实的重述。



此外，如果我们选择范畴 \mathbf{I} 与 \mathbf{C} 相同，我们可以用恒等函子 $I_{\mathbf{C}}$ 替换 D 。我们得到如下恒等式：

$$[\mathbf{C}, \mathbf{C}](F' \circ K, I_{\mathbf{C}}) \cong [\mathbf{A}, \mathbf{C}](F', \mathbf{Ran}_K I_{\mathbf{C}})$$

我们现在可以选择 F' 与 $\mathbf{Ran}_K I_{\mathbf{C}}$ 相同。在这种情况下，右侧包含恒等自然变换，对应的左侧给出了以下自然变换：

$$\varepsilon :: \mathbf{Ran}_K I_{\mathbf{C}} \circ K \rightarrow I_{\mathbf{C}}$$

这非常类似于一个伴随的余单元：

$$\mathbf{Ran}_K I_{\mathbf{C}} \dashv K$$

确实，恒等函子 $I_{\mathbf{C}}$ 沿着一个函子 K 的右 Kan 扩展可以用来计算 K 的左伴随。为此，还需要一个条件：右 Kan 扩展必须被函子 K 所保持。保持扩展意味着，如果我们计算与 K 前合成后的函子的 Kan 扩展，我们应该得到与前合成原始 Kan 扩展的结果相同。在我们的例子中，这个条件简化为：

$$K \circ \mathbf{Ran}_K I_{\mathbf{C}} \cong \mathbf{Ran}_K K$$

注意，使用除以 K 的表示法，伴随关系可以写为：

$$I/K \dashv K$$

这证实了我们对伴随描述某种逆操作的直觉。保持条件变为：

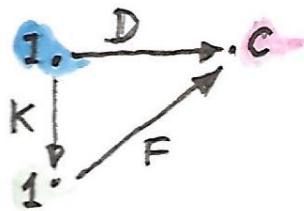
$$K \circ I/K \cong K/K$$

一个函子沿自身的右 Kan 扩展， K/K ，被称为余密（codensity）单子（monad）。

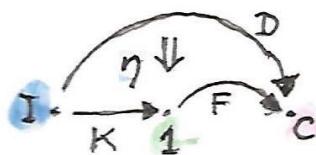
伴随公式是一个重要的结果，因为正如我们很快会看到的，我们可以使用积（end）（或余积（coend））来计算 Kan 扩展，从而为我们提供了实际计算右（和左）伴随的方法。

27.3 左 Kan 扩展 (Left Kan Extension)

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（colimit）的定义开始，并将其重新构建为使用单例范畴 $\mathbf{1}$ 。我们使用函子 $D :: \mathbf{I} \rightarrow \mathbf{C}$ 形成基底，并使用函子 $F :: \mathbf{1} \rightarrow \mathbf{C}$ 选择顶点，来构建一个余锥体（cocone）。

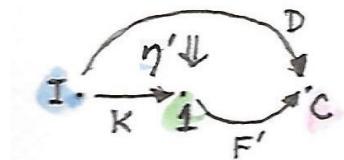


余锥体的侧边，即插入 (injections)，是从 D 到 $F \circ K$ 的自然变换 η 的分量。

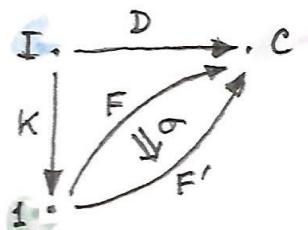


余极限是普遍余锥体。所以对于任何其他函子 F' 和自然变换

$$\eta' :: D \rightarrow F' \circ K$$



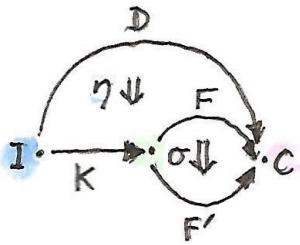
都有一个从 F 到 F' 的唯一自然变换 σ :



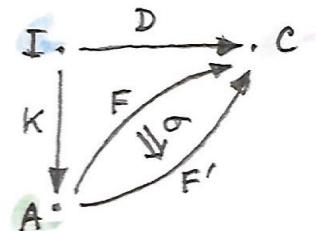
使得:

$$\eta' = (\sigma \circ K) . \eta$$

如以下图所示:



将单例范畴 $\mathbf{1}$ 替换为 \mathbf{A} , 这个定义自然推广到左 Kan 扩展的定义, 记作 $\mathbf{Lan}_K D$ 。



自然变换:

$$\eta :: D \rightarrow \mathbf{Lan}_K D \circ K$$

被称为左 Kan 扩展的单元 (unit)。

和之前一样, 我们可以将自然变换 $\eta' = (\sigma \circ K) . \eta$ 的一对一对应关系重新表述为伴随:

$$[\mathbf{A}, \mathbf{C}](\mathbf{Lan}_K D, F') \cong [\mathbf{I}, \mathbf{C}](D, F' \circ K)$$

换句话说, 左 Kan 扩展是左伴随, 而右 Kan 扩展是前合成函子的右伴随。

就像恒等函子的右 Kan 扩展可以用来计算 K 的左伴随一样, 恒等函子的左 Kan 扩展最终成为 K 的右伴随 (其中 η 是伴随的单元):

$$K \dashv \mathbf{Lan}_K I_{\mathbf{C}}$$

结合这两个结果, 我们得到:

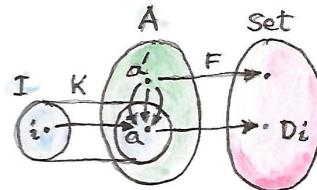
$$\mathbf{Ran}_K I_{\mathbf{C}} \dashv K \dashv \mathbf{Lan}_K I_{\mathbf{C}}$$

27.4 Kan 扩展作为积 (Ends)

Kan 扩展的真正威力在于它们可以使用积 (end) (和余积 (coend)) 来计算。为了简单起见, 我们将注意力限制在目标范畴 \mathbf{C} 为 \mathbf{Set} 的情况下, 但这些公式可以推广到任何范畴。

让我们重新审视一个 Kan 扩展可以用来扩展函子的作用范围的想法。假设 K 嵌入 \mathbf{I} 到 \mathbf{A} 中。函子 D 将 \mathbf{I} 映射到 \mathbf{Set} 。我们可以简单地说，对于 K 的像中的任何对象 a ，即 $a = Ki$ ，扩展后的函子将 a 映射到 Di 。问题是，如何处理那些在 K 的像之外的 \mathbf{A} 中的对象？想法是每个这样的对象都有可能通过大量的态射与 K 的像中的每个对象连接在一起。一个函子必须保持这些态射的结构。一个对象 a 到 K 的像的态射的总和由同态函子 (hom-functor) 表征：

$$\mathbf{A}(a, K-)$$



注意到这个同态函子是两个函子的合成：

$$\mathbf{A}(a, K-) = \mathbf{A}(a, -) \circ K$$

右 Kan 扩展是函子合成的右伴随：

$$[\mathbf{I}, \mathbf{Set}](F' \circ K, D) \cong [\mathbf{A}, \mathbf{Set}](F', \mathbf{Ran}_K D)$$

让我们看看当我们用同态函子替换 F' 时会发生什么：

$$[\mathbf{I}, \mathbf{Set}](\mathbf{A}(a, -) \circ K, D) \cong [\mathbf{A}, \mathbf{Set}](\mathbf{A}(a, -), \mathbf{Ran}_K D)$$

然后将合成公式内联化：

$$[\mathbf{I}, \mathbf{Set}](\mathbf{A}(a, K-), D) \cong \mathbf{Ran}_K Da$$

我们现在可以将自然变换集表示为一个积，从而得到这个非常方便的右 Kan 扩展公式：

$$\mathbf{Ran}_K Da \cong \int_i \mathbf{Set}(\mathbf{A}(a, Ki), Di)$$

左 Kan 扩展也有一个类似的用余积表示的公式：

$$\mathbf{Lan}_K Da = \int^i \mathbf{A}(Ki, a) \times Di$$

为了证明这一点，我们将展示这确实是前合成函子的左伴随：

$$[\mathbf{A}, \mathbf{Set}](\mathbf{Lan}_K D, F') \cong [\mathbf{I}, \mathbf{Set}](D, F' \circ K)$$

让我们在左侧公式中代入我们的公式：

$$[\mathbf{A}, \mathbf{Set}]\left(\int^i \mathbf{A}(Ki, -) \times Di, F'\right)$$

这是一组自然变换，所以它可以被重写为一个积：

$$\int_a \mathbf{Set}\left(\int^i \mathbf{A}(Ki, a) \times Di, F'a\right)$$

利用同态函子的连续性，我们可以将余积替换为积：

$$\int_a \int_i \mathbf{Set}(\mathbf{A}(Ki, a) \times Di, F'a)$$

我们可以使用乘积-指数（product-exponential）伴随：

$$\int_a \int_i \mathbf{Set}(\mathbf{A}(Ki, a), (F'a)^{Di})$$

指数同构于相应的同态集：

$$\int_a \int_i \mathbf{Set}(\mathbf{A}(Ki, a), \mathbf{Set}(Di, F'a))$$

有一个叫做 Fubini 定理的定理允许我们交换两个积：

$$\int_i \int_a \mathbf{Set}(\mathbf{A}(Ki, a), \mathbf{Set}(Di, F'a))$$

内部积表示两个函子之间的自然变换集，所以我们可以使用 Yoneda 引理：

$$\int_i \mathbf{Set}(Di, F'(Ki))$$

这确实是我们要证明的伴随右侧的自然变换集：

$$[\mathbf{I}, \mathbf{Set}](D, F' \circ K)$$

这些使用积、余积和 Yoneda 引理的计算对于“积的微积分”来说是非常典型的。

27.5 Haskell 中的 Kan 扩展

Kan 扩展的积/余积公式可以很容易地翻译成 Haskell。让我们从右 Kan 扩展开始：

$$\mathbf{Ran}_K Da \cong \int_i \mathbf{Set}(\mathbf{A}(a, Ki), Di)$$

我们用通用量词替代积，并用函数类型替代同态集：

snippet01 从这个定义来看，很明显 必须包含一个可以应用于函数的类型，以及两个函子 和 之间的自然变换。例如，假设 是树函子 (tree functor)， 是列表函子 (list functor)，你被给了一个。如果你传递给它一个函数：

snippet02 你将得到一个 列表，等等。右 Kan 扩展将使用你的函数生成一棵树，然后将其重新打包成一个列表。例如，你可以传递一个解析器，该解析器从字符串生成解析树，你将得到一个对应于该树的深度优先遍历的列表。

右 Kan 扩展可以用来计算给定函子的左伴随，通过用恒等函子替换函子。这导致的结果是函子 的左伴随由以下类型的多态函数集表示：

snippet03 假设 是从幺半群 (monoids) 范畴的遗忘函子 (forgetful functor)。通用量词将遍历所有幺半群。当然，在 Haskell 中我们无法表达幺半群的定律，但以下是一个生成的自由函子的相应近似 (遗忘函子是对对象的恒等)：

snippet04 正如预期的那样，它生成自由幺半群，或者 Haskell 列表：

snippet05 左 Kan 扩展是一个余积：

$$\mathbf{Lan}_K Da = \int^i A(Ki, a) \times Di$$

因此，它翻译为一个存在量词。符号上表示为：

这可以在 Haskell 中使用 GADT 进行编码，或者使用一个通用量词的数据构造器：

snippet06 这个数据结构的解释是，它包含一个函数，该函数接受某些未指定的 容器并生成一个 。它还包含了这些 容器。由于你不知道 是什么，这个数据结构唯一能做的事情就是检索 的容器，将其重新打包成由函子 定义的容器，并调用该函数以获得。例如，如果 是一棵树， 是一个列表，你可以序列化树，用生成的列表调用函数并得到一个 。

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：

snippet07 这确实与函数类型同构，如以下一对函数所示：

snippet08 正如前面所描述的一般情况一样，我们执行了以下步骤：

1. 检索到 的容器（在这里，它只是一个简单的恒等容器），以及函数 。
2. 使用恒等函子和对函子对的自然变换，将容器重新打包。
3. 调用函数 。

27.6 自由函子 (Free Functor)

Kan 扩展的一个有趣应用是自由函子的构造。它解决了以下实际问题：假设你有一个类型构造器 (type constructor) ——即对象之间的映射。是否有可能基于这个类型构造器定义一个函子？换句话说，我们能否定义一个态射映射，从而将这个类型构造器扩展为一个完备的自函子 (endofunctor)？

关键的观察是，一个类型构造器可以被描述为一个其定义域是离散范畴 (discrete category) 的函子。离散范畴除了恒等态射外没有其他态射。给定一个范畴 \mathbf{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 $|\mathbf{C}|$ 。从 $|\mathbf{C}|$ 到 \mathbf{C} 的函子 F 是一个简单的对象映射，或者我们在 Haskell 中称为类型构造器。还有一个典型的函子 J 将 $|\mathbf{C}|$ 嵌入到 \mathbf{C} 中：它在对象和恒等态射上是恒等的。 F 沿 J 的左 Kan 扩展 (如果存在) 就是一个从 \mathbf{C} 到 \mathbf{C} 的函子：

$$\mathbf{Lan}_J F a = \int^i |\mathbf{C}(Ji, a) \times Fi|$$

它被称为基于 F 的自由函子。

在 Haskell 中，我们会这样写：

snippet09 确实，对于任何类型构造器，是一个函子：

snippet10 如你所见，自由函子通过记录函数及其参数来模拟函数的提升 (lifting)。它通过记录它们的组合来累积提升的函数。函子规则自动得到满足。这种构造在论文Freer Monads, More Extensible Effects¹中被使用。

或者，我们可以使用右 Kan 扩展来达到相同目的：

snippet11 很容易验证这确实是一个函子：

snippet12

¹

28

Enriched Categories

一个范畴（CATEGORY）是小的（SMALL），如果它的对象构成一个集合。但是我们知道，有些事物比集合更大。众所周知，在标准集合论（Zermelo-Fraenkel 理论，通常附加选择公理）的框架下，不能形成所有集合的集合。因此，所有集合的范畴必须是大的。数学上有一些技巧，如 Grothendieck 宇宙，可以用来定义超越集合的集合。这些技巧使我们可以讨论大范畴（large categories）。

一个范畴是局部小的（locally small），如果任何两个对象之间的态射（morphism）形成一个集合。如果它们不能形成一个集合，我们就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态射，那么复合态射（composition of morphisms）又意味着什么？解决方案是通过用来自其他某个范畴 V 的对象替换同态集（hom-sets）来引导我们自己。区别在于，一般来说，对象没有元素，所以我们不再允许讨论单个态射。我们必须用可以整体操作的同态对象（hom-objects）的术语来定义一个加厚范畴（enriched category）的所有性质。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——它必须是一个单积范畴（monoidal category）。如果我们称这个单积范畴为 V ，我们可以讨论一个范畴 C 加厚于 V 的情形。

除了规模上的原因，我们可能还对将同态集推广为比单纯集合更具结构的事物感兴趣。例如，传统范畴没有定义对象之间距离（distance）的概念。两个对象要么通过态射连接，要么不连接。与给定对象相连接的所有对象都是它的邻居。与现实生活不同，在一个范畴中，一个朋友的朋友的朋友与我的知心好友一样接近。在一个适当加厚的范畴中，我们可以定义对象之间的距离。

还有一个非常实际的理由来了解加厚范畴，因为一个非常有用在线范畴知识来源，nLab¹，主要是以加厚范畴的术语编写的。

28.1 为什么选择单积范畴？

Why Monoidal Category?

当构造一个加厚范畴 (enriched category) 时，我们必须记住，当我们用 **Set** 替换单积范畴 (monoidal category) 并用同态集 (hom-sets) 替换同态对象 (hom-objects) 时，我们应该能够恢复通常的定义。实现这一目标的最好方法是从通常的定义开始，并不断以无元素 (point-free) 方式重新表述它们——即不命名集合的元素。

让我们从复合的定义开始。通常，它接受一对态射，一个来自 $\mathbf{C}(b, c)$ ，另一个来自 $\mathbf{C}(a, b)$ ，并将其映射到来自 $\mathbf{C}(a, c)$ 的态射。换句话说，它是一个映射：

$$\mathbf{C}(b, c) \times \mathbf{C}(a, b) \rightarrow \mathbf{C}(a, c)$$

这是一个集合之间的函数——其中一个是两个同态集的笛卡尔积 (Cartesian product)。通过用更一般的笛卡尔积替换笛卡尔积，这个公式可以很容易地推广。范畴积 (categorical product) 可以起作用，但我们可以更进一步，使用一个完全通用的张量积 (tensor product)。

接下来是恒等态射 (identity morphisms)。我们可以用从单元素集合 **1** 到 $\mathbf{C}(a, a)$ 的函数来定义它们，而不是从同态集中选择单个元素：

$$j_a :: \mathbf{1} \rightarrow \mathbf{C}(a, a)$$

同样，我们可以用终对象 (terminal object) 替换单元素集合，但我们可以更进一步，用张量积的单位元 i 替换它。

正如你所见，从某个单积范畴 (monoidal category) \mathbf{V} 取出的对象是替换同态集的好候选者。

28.2 单积范畴

Monoidal Category

我们之前讨论过单积范畴 (monoidal category)，但值得重新说明其定义。一个单积范畴定义了一个张量积，它是一个双函子：

$$\otimes :: \mathbf{V} \times \mathbf{V} \rightarrow \mathbf{V}$$

我们希望张量积是结合的，但满足结合律的自然同构就足够了。这种同构称为结合子 (associator)。它的分量是：

$$\alpha_{abc} :: (a \otimes b) \otimes c \rightarrow a \otimes (b \otimes c)$$

¹

它必须在所有三个参数中自然。

一个单积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元; 同样是通过自然同构。这两个同构分别称为左结合子 (left unit) 和右结合子 (right unit), 它们的分量是:

$$\lambda_a :: i \otimes a \rightarrow a$$

$$\rho_a :: a \otimes i \rightarrow a$$

结合子和单位元必须满足一致性条件:

$$\begin{array}{ccc}
 ((a \otimes b) \otimes c) \otimes d & \xrightarrow{\alpha_{abc} \otimes \mathbf{id}_d} & (a \otimes (b \otimes c)) \otimes d \\
 \downarrow \alpha_{(a \otimes b)cd} & & \downarrow \alpha_{a(b \otimes c)d} \\
 (a \otimes b) \otimes (c \otimes d) & & a \otimes ((b \otimes c) \otimes d) \\
 & \searrow \alpha_{ab(c \otimes d)} & \swarrow \mathbf{id}_a \otimes \alpha_{bcd} \\
 & a \otimes (b \otimes (c \otimes d)) &
 \end{array}$$

$$\begin{array}{ccc}
 (a \otimes i) \otimes b & \xrightarrow{\alpha_{aib}} & a \otimes (i \otimes b) \\
 \searrow \rho_a \otimes \mathbf{id}_b & & \swarrow \mathbf{id}_a \otimes \lambda_b \\
 a \otimes b & &
 \end{array}$$

一个单积范畴被称为对称的 (symmetric), 如果它有一个自然同构, 其分量为:

$$\gamma_{ab} :: a \otimes b \rightarrow b \otimes a$$

且其“平方为一”:

$$\gamma_{ba} \circ \gamma_{ab} = \mathbf{id}_{a \otimes b}$$

并且它与单积结构是一致的。

关于单积范畴的一个有趣之处是, 你可以定义内部同态 (internal hom, 也就是函数对象) 作为张量积的右伴随 (right adjoint)。你可能还记得, 函数对象或指数 (exponential) 的标准定义是通过范畴积的右伴随来实现的。一个范畴, 如果这样的对象存在于任何一对对象之间, 就被称为笛卡尔闭范畴 (Cartesian closed)。这里是一个定义单积范畴内部同态的伴随关系:

$$\mathbf{V}(a \otimes b, c) \sim \mathbf{V}(a, [b, c])$$

参照G. M. Kelly²，我使用符号 $[b, c]$ 表示内部同态。这一伴随关系的余单元（counit）是一个自然变换，其分量被称为评估态射（evaluation morphisms）：

$$\varepsilon_{ab} :: ([a, b] \otimes a) \rightarrow b$$

注意，如果张量积不是对称的，我们可以定义另一个内部同态，记为 $[[a, c]]$ ，使用以下伴随关系：

$$\mathbf{V}(a \otimes b, c) \sim \mathbf{V}(b, [[a, c]])$$

一个单积范畴，如果这两者都被定义出来，则称为双闭的（biclosed）。一个非双闭范畴的例子是 **Set** 中的端函子范畴（endofunctors），其中函子合成作为张量积。我们用这个范畴来定义 Monad。

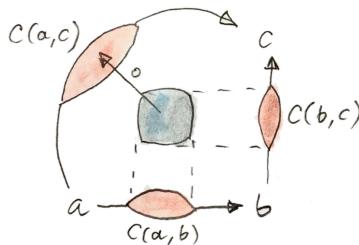
28.3 加厚范畴

Enriched Category

一个加厚于单积范畴 \mathbf{V} 的范畴 \mathbf{C} 用同态对象替换了同态集。对于 \mathbf{C} 中的每一对对象 a 和 b ，我们关联一个对象 $\mathbf{C}(a, b)$ 在 \mathbf{V} 中。我们对同态对象使用与同态集相同的符号，理解为它们不包含态射。另一方面， \mathbf{V} 是一个常规的（非加厚的）具有同态集和态射的范畴。所以我们并没有完全摆脱集合——我们只是将它们隐藏了起来。

既然我们不能讨论 \mathbf{C} 中的单个态射，那么复合态射的操作就由 \mathbf{V} 中的态射家族替代：

$$\circ :: \mathbf{C}(b, c) \otimes \mathbf{C}(a, b) \rightarrow \mathbf{C}(a, c)$$

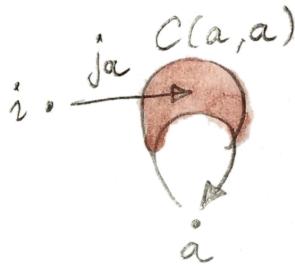


同样，恒等态射被替换为 \mathbf{V} 中的一个态射家族：

$$j_a :: i \rightarrow \mathbf{C}(a, a)$$

这里的 i 是 \mathbf{V} 中的张量单位元。

²



复合态射的结合性由 \mathbf{V} 中的结合子定义：

$$\begin{array}{ccc}
 (\mathbf{C}(c,d) \otimes \mathbf{C}(b,c)) \otimes \mathbf{C}(a,b) & \xrightarrow{\circ \otimes \text{id}} & \mathbf{C}(b,d) \otimes \mathbf{C}(a,b) \\
 \downarrow \alpha & & \searrow \circ \\
 \mathbf{C}(c,d) \otimes (\mathbf{C}(b,c) \otimes \mathbf{C}(a,b)) & \xrightarrow{\text{id} \otimes \circ} & \mathbf{C}(c,d) \otimes \mathbf{C}(a,c)
 \end{array}$$

单位律同样以单位元的形式表达：

$$\begin{array}{ccc}
 \mathbf{C}(a,b) \otimes i & \xrightarrow{\text{id} \otimes j_a} & \mathbf{C}(a,b) \otimes \mathbf{C}(a,a) \\
 & \swarrow \rho & \searrow \circ \\
 & \mathbf{C}(a,b) &
 \end{array}$$

$$\begin{array}{ccc}
 i \otimes \mathbf{C}(a,b) & \xrightarrow{j_b \otimes \text{id}} & \mathbf{C}(b,b) \otimes \mathbf{C}(a,b) \\
 & \swarrow \lambda & \searrow \circ \\
 & \mathbf{C}(a,b) &
 \end{array}$$

28.4 预序 (Preorders)

Preorders

预序 (preorder) 定义为一个稀薄范畴 (thin category)，其中每个同态集要么为空，要么是单元集。我们将一个非空的 $\mathbf{C}(a,b)$ 集解释为 a 小于或等于 b 的证明。这样的范畴可以解释为加厚于一个非常简单的单积范畴，该范畴只包含两个对象，0 和 1 (有时称为 *False* 和 *True*)。除了强制性的恒等态射外，这个范畴还有一个从 0 到 1 的态射，称为 $0 \rightarrow 1$ 。

一个简单的单积结构可以在其中建立，张量积模拟 0 和 1 的简单算术运算（即，唯一的非零积是 $1 \otimes 1$ ）。这个范畴中的单位对象是 1。这是一个严格的单积范畴，即结合子和单位元是恒等态射。

由于在预序中，同态集要么为空，要么是单元集，我们可以很容易地用我们这个微小范畴中的同态对象替换它。加厚预序 \mathbf{C} 对于任意一对对象 a 和 b 有一个同态对象 $\mathbf{C}(a, b)$ 。如果 a 小于或等于 b ，这个对象是 1；否则它是 0。

让我们来看一下复合态射。任何两个对象的张量积是 0，除非它们都是 1，在这种情况下它是 1。如果它是 0，那么我们有两个选择用于复合态射：它可以是 \mathbf{id}_0 或 $0 \rightarrow 1$ 。但是如果它是 1，那么唯一的选择是 \mathbf{id}_1 。将此翻译回关系中，这表示如果 $a \leq b$ 且 $b \leq c$ 那么 $a \leq c$ ，这正是我们所需的传递律。

恒等态射如何？它是从 1 到 $\mathbf{C}(a, a)$ 的态射。唯一从 1 出发的态射是 \mathbf{id}_1 ，所以 $\mathbf{C}(a, a)$ 必须是 1。这意味着 $a \leq a$ ，这是预序的反身性定律。因此，如果我们将预序实现为加厚范畴，传递性和反身性都会自动强制执行。

28.5 度量空间

Metric Spaces

一个有趣的例子来自 William Lawvere³。他注意到度量空间（metric space）可以用加厚范畴来定义。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。将无穷大作为一个可能的值是方便的。如果距离是无穷大的，就无法从起点对象到达目标对象。

距离必须满足的一些显而易见的属性之一是，距离从一个对象到它自身必须为零。另一个是三角不等式：直接距离不大于带有中间站点的距离之和。我们不要求距离是对称的，这在开始时可能看起来很奇怪，但正如 Lawvere 解释的那样，你可以想象在一个方向上你在上坡，而在另一个方向上你在下坡。无论如何，对称性可以稍后作为附加约束强加。

那么，如何将度量空间转换为范畴语言？我们必须构造一个范畴，在其中同态对象是距离。注意，距离不是态射，而是同态对象。同态对象如何成为一个数字？只有当我们可以构造一个单积范畴 \mathbf{V} ，在其中这些数字是对象。非负实数（加上无穷大）形成一个全序，所以它们可以被视为稀薄范畴。在两个这样的数字 x 和 y 之间存在一个态射，当且仅当 $x \geq y$ （注意：这是与传统预序定义相反的方向）。单积结构由加法给出，零作为单位对象。换句话说，两个数字的张量积是它们的和。

一个度量空间是加厚于这样的单积范畴的范畴。从对象 a 到对象 b 的同态对象 $\mathbf{C}(a, b)$ 是一个非负（可能是无穷大的）数，我们称之为从 a

³

到 b 的距离。让我们看看在这样的范畴中，我们为恒等态射和复合态射得到了什么。

根据我们的定义，从张量单位（即数字零）到同态对象 $\mathbf{C}(a, a)$ 的态射是关系：

$$0 \geq \mathbf{C}(a, a)$$

由于 $\mathbf{C}(a, a)$ 是一个非负数，这个条件告诉我们从 a 到 a 的距离总是零。检查！

现在让我们讨论复合态射。我们从两个相邻同态对象 $\mathbf{C}(b, c) \otimes \mathbf{C}(a, b)$ 的张量积开始。我们已经定义了张量积作为两个距离的和。复合态射是 \mathbf{V} 中从这个积到 $\mathbf{C}(a, c)$ 的态射。 \mathbf{V} 中的态射定义为大于或等于关系。换句话说，从 a 到 b 和从 b 到 c 的距离之和大于或等于从 a 到 c 的距离。但这正是标准的三角不等式。检查！

通过将度量空间重新构造成一个加厚范畴，我们“免费”获得了三角不等式和零自距离。

28.6 加厚函子

Enriched Functors

函子（functor）的定义涉及态射的映射。在加厚环境中，我们没有单个态射的概念，所以我们必须整体处理同态对象。同态对象是单积范畴 \mathbf{V} 中的对象，并且我们可以使用它们之间的态射。因此，在加厚于同一个单积范畴 \mathbf{V} 的范畴之间定义加厚函子是有意义的。然后我们可以使用 \mathbf{V} 中的态射来映射两个加厚范畴之间的同态对象。

一个加厚函子（enriched functor） F 在两个范畴 \mathbf{C} 和 \mathbf{D} 之间，除了将对象映射到对象，还为 \mathbf{C} 中的每一对对象分配一个 \mathbf{V} 中的态射：

$$F_{ab} : \mathbf{C}(a, b) \rightarrow \mathbf{D}(Fa, Fb)$$

函子是一个保持结构的映射。对于常规函子，这意味着保持复合和恒等态射。在加厚环境中，保持复合意味着以下图表交换：

$$\begin{array}{ccc} \mathbf{C}(b, c) \otimes \mathbf{C}(a, b) & \xrightarrow{\quad \circ \quad} & \mathbf{C}(a, c) \\ \downarrow F_{bc} \otimes F_{ab} & & \downarrow F_{ac} \\ \mathbf{D}(Fb, Fc) \otimes \mathbf{D}(Fa, Fb) & \xrightarrow{\quad \circ \quad} & \mathbf{D}(Fa, Fc) \end{array}$$

保持恒等性被替换为保持选择恒等态射的 \mathbf{V} 中的态射：

$$\begin{array}{ccc}
 & i & \\
 j_a \swarrow & & \searrow j_{Fa} \\
 \mathbf{C}(a, a) & \xrightarrow{F_{aa}} & \mathbf{D}(Fa, Fa)
 \end{array}$$

28.7 自加厚

Self Enrichment

一个封闭对称单积范畴 (closed symmetric monoidal category) 可以通过用内部同态 (internal homs) 替换同态集来自加厚。为了使其工作，我们必须为内部同态定义复合法。换句话说，我们必须实现具有以下签名的态射：

$$[b, c] \otimes [a, b] \rightarrow [a, c]$$

这与其他任何编程任务没有太大不同，只是，在范畴论中，我们通常使用无元素实现。我们首先指定其元素应该是什么集合。在这种情况下，它是同态集的一个成员：

$$\mathbf{V}([b, c] \otimes [a, b], [a, c])$$

这个同态集与以下同构：

$$\mathbf{V}(([b, c] \otimes [a, b]) \otimes a, c)$$

我刚刚使用了定义内部同态 $[a, c]$ 的伴随。如果我们能在这个新集合中构造一个态射，伴随将指向原始集合中的态射，我们可以将其用作复合。我们通过组合一些我们可以使用的态射来构建这个态射。首先，我们可以使用结合子 $\alpha_{[b,c][a,b]a}$ 来重新关联左边的表达式：

$$([b, c] \otimes [a, b]) \otimes a \rightarrow [b, c] \otimes ([a, b] \otimes a)$$

我们可以接着用伴随的余单元 ε_{ab} ：

$$[b, c] \otimes ([a, b] \otimes a) \rightarrow [b, c] \otimes b$$

然后再次使用余单元 ε_{bc} 到 c 。我们就这样构造了一个态射：

$$\varepsilon_{bc} \cdot (\mathbf{id}_{[b,c]} \otimes \varepsilon_{ab}) \cdot \alpha_{[b,c][a,b]a}$$

这是同态集的一个元素：

$$\mathbf{V}(([b, c] \otimes [a, b]) \otimes a, c)$$

伴随将给我们所寻找的复合法。

同样地，恒等态射：

$$j_a :: i \rightarrow [a, a]$$

是以下同态集的一个成员：

$$\mathbf{V}(i, [a, a])$$

它通过伴随同构于：

$$\mathbf{V}(i \otimes a, a)$$

我们知道这个同态集中包含左恒等 λ_a 。我们可以将 j_a 定义为其在伴随下的像。

一个实际的自加厚例子是 **Set** 范畴，它是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封闭单积范畴。在 **Set** 中，任意两个集合之间的同态集本身就是一个集合，所以它是 **Set** 中的一个对象。我们知道它与指数集合同构，所以外部同态和内部同态是等价的。现在我们也知道，通过自加厚，我们可以使用指数集合作为同态对象，并通过笛卡尔积的指数对象来表示复合。

28.8 与 2-范畴的关系

Relation to 2-Categories

我在谈论 **Cat** 范畴（小范畴的范畴）时谈到过 2-范畴（2-categories）。范畴之间的态射是函子，但还有一个附加结构：函子之间的自然变换。在一个 2-范畴中，对象通常称为零胞（zero-cells）；态射称为 1-胞（1-cells）；态射之间的态射称为 2-胞（2-cells）。在 **Cat** 中，0-胞是范畴，1-胞是函子，2-胞是自然变换。

但是请注意，两个范畴之间的函子本身也形成一个范畴；因此，在 **Cat** 中，我们实际上有一个同态范畴（hom-category），而不是同态集。这意味着，正如 **Set** 可以视为加厚于 **Set** 的范畴，**Cat** 可以视为加厚于 **Cat** 的范畴。更广泛地讲，就像每个范畴可以视为加厚于 **Set**，每个 2-范畴也可以视为加厚于 **Cat**。

29

Topoi

我意识到我们可能正在偏离编程，进入硬核数学的领域。但是你永远不知道下一场编程革命会带来什么样的新变革，以及理解这些变革需要什么样的数学。目前有一些非常有趣的想法在流行，比如具有连续时间的函数式反应式编程（functional reactive programming）、Haskell 类型系统的依赖类型扩展，或者在编程中探索同伦类型论（homotopy type theory）。

到目前为止，我一直随意地将类型与值的集合（sets）等同起来。这并不完全正确，因为这种方法没有考虑到在编程中，我们是计算值的，而计算是一个需要时间的过程，在极端情况下，可能永远不会终止。发散计算（divergent computations）是每种图灵完备语言的一部分。

也有一些基础原因说明集合论可能并不是计算机科学甚至数学本身的最佳基础。一个好的类比是，将集合论视为与特定架构相关的汇编语言。如果你想在不同的架构上运行数学，你必须使用更通用的工具。

一种可能性是用空间（spaces）代替集合。空间带有更多的结构，并且可以在不依赖集合的情况下定义。通常与空间相关联的是拓扑学（topology），它是定义连续性等概念所必需的。而拓扑学的传统方法正如你猜到的那样，是通过集合论来实现的。特别是子集的概念在拓扑学中起着核心作用。毫不奇怪，范畴论学者将这一思想推广到了 **Set** 以外的范畴中。具有适当性质、可以替代集合论的那种范畴被称为拓扑斯（topos，复数：topoi），它提供了一个广义的子集概念。

29.1 子对象分类器

Subobject Classifier

让我们从尝试用函数而不是元素来表达子集的概念开始。任何从集合 a 到 b 的函数 f 都定义了 b 的一个子集——即 f 下 a 的像。但是有许多函数定义了相同的子集。我们需要更具体一点。首先，我们可以关注单射 (injective) 的函数——那些不会将多个元素压缩为一个的函数。单射函数将一个集合“注入”到另一个集合中。对于有限集合，你可以将单射函数形象地理解为连接一个集合与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大，否则这些箭头必然会收敛。仍然存在一些模糊性：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' ，它们选取了相同的子集。但你可以轻易地证明，这样的集合必须与 a 同构。我们可以利用这一事实将子集定义为由其定义域的同构关系所关联的一组单射函数。更准确地说，我们说两个单射函数：

$$\begin{aligned} f &:: a \rightarrow b \\ f' &:: a' \rightarrow b \end{aligned}$$

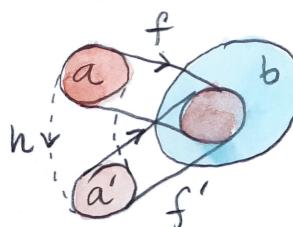
是等价的，如果存在一个同构：

$$h :: a \rightarrow a'$$

使得：

$$f = f' . h$$

这类等价的单射集合定义了 b 的一个子集。



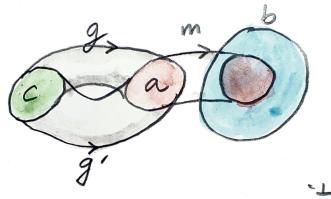
如果我们将单射函数替换为单态射 (monomorphism)，这个定义可以提升到任意范畴。提醒一下，从 a 到 b 的单态射 m 是由其泛性质定义的。对于任意对象 c 和任意一对态射：

$$\begin{aligned} g &:: c \rightarrow a \\ g' &:: c \rightarrow a \end{aligned}$$

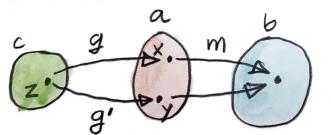
如果：

$$m \cdot g = m \cdot g'$$

那么必然有 $g = g'$ 。



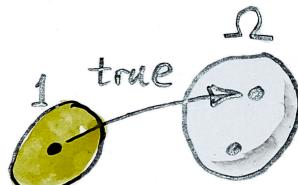
在集合上，这一定义更容易理解，如果我们考虑一个函数 m 不是单态射会意味着什么。它会将 a 的两个不同元素映射到 b 的同一个元素上。我们可以找到两个函数 g 和 g' ，它们只在这两个元素上有所不同。然后与 m 的后合成将掩盖这一差异。



还有另一种定义子集的方法：使用一个称为特征函数（characteristic function）的单一函数。这是一个从集合 b 到两元素集合 Ω 的函数。这个集合的一个元素被指定为“真”（true），另一个为“假”（false）。这个函数将 b 中的那些属于子集的元素映射为“真”，而将其他元素映射为“假”。

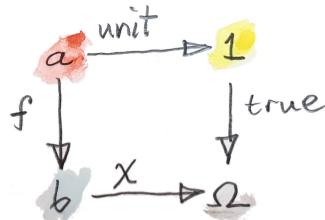
我们还需要指定将 Ω 的一个元素指定为“真”的含义。我们可以使用一个标准技巧：使用从单元素集合到 Ω 的函数。我们将这个函数称为 *true*：

$$\text{true} :: 1 \rightarrow \Omega$$



这些定义可以结合在一起，不仅定义了什么是子对象，还定义了特殊对象 Ω ，而无需讨论元素。这个想法是我们希望态射 *true* 表示一个“通用”的子对象。在 **Set** 中，它从两元素集合 Ω 中选择了一个单元素子集。这是最通用的形式。显然这是一个适当的子集，因为 Ω 有一个不在该子集中的元素。

在更一般的环境中，我们将 *true* 定义为从终对象到分类对象（classifying object） Ω 的单态射。但是我们必须定义分类对象。我们需要一个泛性质，将这个对象与特征函数联系起来。事实证明，在 **Set** 中，特征函数 χ 与 *true* 的纤维积（pullback）同时定义了子集 a 和嵌入到 b 的单射函数。以下是纤维积图：



让我们分析这个图表。纤维积方程是：

$$\text{true} \cdot \text{unit} = \chi \cdot f$$

函数 $\text{true} \cdot \text{unit}$ 将 a 的每个元素映射为“真”。因此 f 必须将 a 的所有元素映射到 b 中的那些 χ 为“真”的元素上。这些元素根据定义，构成了由特征函数 χ 指定的子集。因此 f 的像确实是所讨论的子集。纤维积的泛性保证了 f 是单射。

这个纤维积图可以用来在 **Set** 以外的范畴中定义分类对象。这样的范畴必须具有一个终对象，它将使我们能够定义单态射 *true*。它还必须有纤维积——实际的要求是它必须具有所有有限极限（纤维积是有限极限的一个例子）。在这些假设下，我们通过如下性质定义分类对象 Ω ：对于每个单态射 f ，都存在一个唯一的态射 χ ，使纤维积图完成。

让我们分析最后这句话。当我们构造纤维积时，我们给定了三个对象 Ω 、 b 和 1 ；以及两个态射 *true* 和 χ 。纤维积的存在意味着我们可以找到最合适对象 a ，并配备两个态射 f 和 *unit*（后者由终对象的定义唯一确定），使得图表交换。

在这里，我们正在求解另一个方程组。我们正在解 Ω 和 *true*，同时改变 a 和 b 。对于给定的 a 和 b ，可能有也可能没有单态射 $f :: a \rightarrow b$ 。但是如果存在，我们希望它是某个 χ 的纤维积。此外，我们希望这个 χ 由 f 唯一确定。

我们不能说单态射 f 和特征函数 χ 之间存在一对一对应关系，因为纤维积在同构意义上是唯一的。但是请记住我们早先将子集定义为一组等价单射的定义。我们可以通过将 b 的子对象定义为一组等价的到 b 的单态射来推广它。这个单态射的集合与我们图表的等价纤维积集合一一对应。

因此，我们可以将 b 的子对象集合 $\text{Sub}(b)$ 定义为一组单态射，并看到它与从 b 到 Ω 的态射集合同构：

$$\text{Sub}(b) \cong \mathbf{C}(b, \Omega)$$

这恰好是两个函子的自然同构。换句话说， $\text{Sub}(-)$ 是一个可表（反变）函子，其表示是对象 Ω 。

29.2 拓扑斯 Topos

一个拓扑斯是一个满足以下条件的范畴：

1. 是笛卡尔闭的：它有所有积、终对象和指数（定义为积的右伴随），
2. 对所有有限图有极限，
3. 有一个子对象分类器 Ω 。

这一组性质使得拓扑斯在大多数应用中成为 **Set** 的理想替代品。它还具有从其定义中得出的其他性质。例如，拓扑斯具有所有有限余极限，包括初始对象。

我们可能会认为可以将子对象分类器定义为两个终对象的余积（和），——在 **Set** 中确实如此——但我们希望比这更普遍。在这种情况下成立的拓扑斯称为布尔（Boolean）拓扑斯。

29.3 拓扑斯与逻辑 Topoi and Logic

在集合论中，特征函数可以解释为定义了集合元素的性质——一个对某些元素为真的谓词，对其他元素为假的谓词。谓词 *isEven* 从自然数集合中选择偶数的子集。在拓扑斯中，我们可以将谓词的概念推广为从对象 a 到 Ω 的态射。这就是为什么 Ω 有时被称为真值对象。

谓词是逻辑的构建块。一个拓扑斯包含了研究逻辑所需的所有工具。它有对应逻辑合取（逻辑“与”的积，有对应逻辑析取（逻辑“或”）的余积，还有对应蕴涵的指数。除去排中律（或者等价地，双重否定律）的情况外，所有标准的逻辑公理在拓扑斯中都成立。这就是为什么拓扑斯的逻辑对应于构造性或直觉主义逻辑。

直觉主义逻辑正在稳步发展，并且在计算机科学中得到了意外的支持。经典的排中律基于这样一种信念，即存在绝对真理：任何陈述要么为真要么为假，正如古罗马人所说的那样，*tertium non datur*（无第三选项）。但我们唯一能知道某件事是否为真或为假的方法是证明或反驳它。证明是一个过程，是一个计算——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。如果我们不能在有限的时间内证明某件事，那么声称它为真是没有意义的。一个拓扑斯通过其更为细致的真值对象，提供了一个更为普遍的框架来建模有趣的逻辑。

29.4 挑战

Challenges

1. 证明函数 f 是沿特征函数的 $true$ 的纤维积，必须是单射。

30

Lawvere Theories

现在，你几乎不能谈论函数式编程而不提到幺半群（monads）。但在另一个平行宇宙中，尤金尼奥·莫吉（Eugenio Moggi）可能关注了劳维尔理论（Lawvere theories）而不是幺半群。让我们探索一下那个宇宙。

30.1 泛代数

Universal Algebra

有很多方法可以在各种抽象层次上描述代数。我们尝试找到一种通用语言来描述诸如幺半群（monoids）、群（groups）或环（rings）之类的结构。在最简单的层次上，所有这些结构都定义了集合上元素的运算，以及必须满足的定律。例如，幺半群可以通过一个满足结合律的二元运算来定义。我们还需要一个单位元素和单位定律。但只要稍加想象，我们就可以将单位元素转变为一个零元运算——一个不需要任何参数并返回集合中特殊元素的运算。如果我们想讨论群，我们会添加一个一元运算符，该运算符接受一个元素并返回其逆元。还有相应的左逆定律和右逆定律。环定义了两个二元运算符以及更多的定律。等等。

大体来看，一个代数是通过对不同 n 值的 n 元运算以及一组方程等式来定义的。这些等式都是全称量化的。结合律等式必须对所有可能的三个元素组合成立，等等。

顺便提一下，这就排除了对域的考虑，原因很简单，零（加法的单位）对于乘法没有逆元。对于域的逆元定律不能全称量化。

如果我们将运算（函数）替换为态射，这种泛代数的定义可以扩展到 **Set** 以外的范畴。我们选择一个对象 a （称为泛对象）而不是集合。一个一元运算就是 a 的一个自同态射（endomorphism）。但对于其他元数

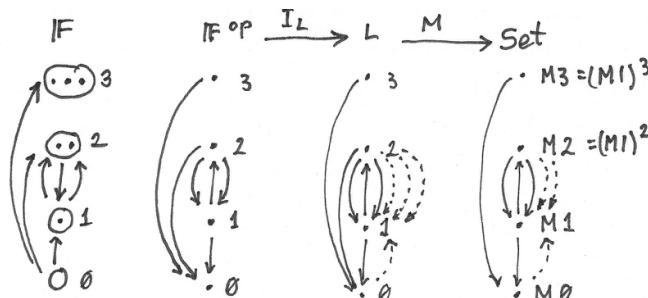
呢？（元数（arity）是给定运算的参数个数。）一个二元运算（元数为 2）可以定义为从积 $a \times a$ 到 a 的态射。一般的 n 元运算是从 a 的 n 次幂到 a 的态射：

$$\alpha_n : a^n \rightarrow a$$

一个零元运算是从终对象（ a 的零次幂）到 a 的态射。因此，为了定义任何代数，我们只需要一个范畴，其对象是某个特殊对象 a 的幂。特定的代数被编码在这个范畴的态射集中。这就是劳维尔理论的简要概述。

劳维尔理论的推导经过了许多步骤，以下是路线图：

1. 有限集范畴 **FinSet**。
2. 其骨架 **F**。
3. 其对立范畴 **F^{op}**。
4. 劳维尔理论 **L**：在 **Law** 范畴中的一个对象。
5. 劳维尔范畴的模型 **M**：在范畴 **Mod(Law, Set)** 中的一个对象。



30.2 劳维尔理论

Lawvere Theories

所有劳维尔理论共享一个共同的骨架。劳维尔理论中的所有对象都是通过对一个对象进行积运算（实际上是幂运算）生成的。但我们如何在一般范畴中定义这些积？事实证明，我们可以通过从一个更简单的范畴映射来定义积。实际上，这个简单的范畴可以定义余积而不是积，而我们会使用一个反变函子将它们嵌入我们的目标范畴中。反变函子将余积转换为积，并将嵌入态射转换为投影态射。

劳维尔范畴骨架的自然选择是有限集范畴 **FinSet**。它包含空集 \emptyset ，一个单元素集合 1 ，一个双元素集合 2 ，等等。这个范畴中的所有对象都可以通过使用余积从单元素集合生成（将空集视为零元余积的特例）。例如，双元素集合是两个单元素集合的和， $2 = 1 + 1$ ，如下所示的 Haskell 表达式：

snippet01 然而，即使我们自然地认为只有一个空集，但可能有许多不同的单元素集合。特别是，集合 $1 + \emptyset$ 与集合 $\emptyset + 1$ 是不同的，也与 1 不同——尽管它们都是同构的。集合范畴中的余积不是结合的。我们可以通过构建一个范畴来解决这种情况，该范畴识别所有同构的集合。这种范畴称为骨架（skeleton）。换句话说，任何劳维尔理论的骨架是 **FinSet** 的骨架 **F**。这个范畴中的对象可以与自然数（包括零）相对应，这些自然数对应于 **FinSet** 中的元素数量。余积在此处扮演加法的角色。**F** 中的态射对应于有限集之间的函数。例如，从 \emptyset 到 n 存在唯一的态射（空集是初对象），从 n 到 \emptyset 没有态射（除了 $\emptyset \rightarrow \emptyset$ ），从 1 到 n 有 n 个态射（即嵌入态射），从 n 到 1 有一个态射，等等。这里， n 表示 **F** 中的一个对象，它对应于通过同构识别的 **FinSet** 中的所有 n 元集合。

使用范畴 **F**，我们可以正式定义劳维尔理论（Lawvere theory）为一个带有特殊函子的范畴 **L**：

$$I_L : \mathbf{F}^{op} \rightarrow \mathbf{L}$$

这个函子必须在对象上是双射，并且必须保持有限积（ \mathbf{F}^{op} 中的积与 **F** 中的余积相同）：

$$I_L(m \times n) = I_L m \times I_L n$$

你有时可能会看到这个函子被描述为对象上的恒等，这意味着 **F** 和 **L** 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——我们将它们用自然数表示。请记住，**F** 中的对象与集合不同（它们是同构集合的类）。

L 中的态射集通常比 \mathbf{F}^{op} 中的更丰富。它们可能包含除那些对应于 **FinSet** 中函数的态射（后者有时称为基本积运算）以外的态射。劳维尔理论的等式定律编码在这些态射中。

关键的一点是，**F** 中的单元素集合 1 被映射到 **L** 中的某个对象，我们也称其为 1 ，并且 **L** 中的所有其他对象都是这个对象的幂。例如，**F** 中的双元素集合 2 是 $1+1$ 的余积，因此它必须映射到 **L** 中的积 1×1 （或 1^2 ）。从这个意义上说，范畴 **F** 的行为就像是 **L** 的对数。

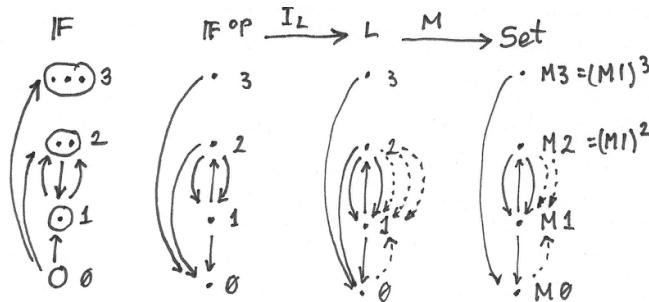
在 **L** 中，我们有那些通过函子 I_L 从 **F** 转移来的态射。它们在 **L** 中起结构性作用。特别地，余积嵌入 i_k 变成积投影 p_k 。一个有用的直觉是将投影：

$$p_k : 1^n \rightarrow 1$$

看作一个 n 元函数的原型，它忽略了除第 k 个变量之外的所有变量。相反，**F** 中的常值态射 $n \rightarrow 1$ 在 **L** 中变成对角态射 $1 \rightarrow 1^n$ 。它们对应于变量的重复。

L 中有趣的态射是定义非投影 n 元运算的那些态射。正是这些态射区分了一个劳维尔理论与另一个。这些是乘法、加法、单位元素的选择等等，这些定义了代数。但是，为了使 **L** 成为一个完整的范畴，我们还需要复合运算 $n \rightarrow m$ （或等价地 $1^n \rightarrow 1^m$ ）。由于范畴的简单结构，它们最终是 $n \rightarrow 1$ 类型的简单态射的积。这是前面我们已经见过的一个泛

化，即返回积的函数是函数的积（或者我们已经看到过的同态函子是连续的）。



劳维尔理论 \mathbf{L} 基于 \mathbf{F}^{op} ，它从中继承了定义积的“无趣”态射。它添加了描述 n 元运算的“有趣”态射（虚线箭头）。

劳维尔理论形成一个范畴 **Law**，其中的态射是保持有限积并与函子 I 交换的函子。给定两个这样的理论， $(\mathbf{L}, I_{\mathbf{L}})$ 和 $(\mathbf{L}', I'_{\mathbf{L}'})$ 之间的态射是一个函子 $F : \mathbf{L} \rightarrow \mathbf{L}'$ ，使得：

$$\begin{aligned} F(m \times n) &= Fm \times Fn \\ F \circ I_{\mathbf{L}} &= I'_{\mathbf{L}'} \end{aligned}$$

劳维尔理论之间的态射封装了将一个理论解释为另一个理论的思想。例如，如果我们忽略逆元，群的乘法可以被解释为幺半群的乘法。

劳维尔范畴最简单的例子是 \mathbf{F}^{op} 本身（对应于 $I_{\mathbf{L}}$ 的恒等函子的选择）。这个没有运算或定律的劳维尔理论恰好是 **Law** 中的初对象。

此时，如果我们没有先了解模型，提供一个非平凡的劳维尔理论例子会非常困难。

30.3 劳维尔理论的模型

Models of Lawvere Theories

理解劳维尔理论的关键是要意识到这样的理论概括了很多具有相同结构的个别代数。例如，幺半群的劳维尔理论描述了成为幺半群的本质。它必须对所有幺半群有效。特定的幺半群成为这种理论的模型。模型被定义为从劳维尔理论 \mathbf{L} 到集合范畴 **Set** 的函子（有劳维尔理论的推广使用其他范畴作为模型，但在这里我只集中在 **Set** 上）。由于 \mathbf{L} 的结构高度依赖于积，我们要求这样的函子保持有限积。 \mathbf{L} 的模型（也称为劳维尔理论 \mathbf{L} 上的代数）因此由一个函子定义：

$$\begin{aligned} M : \mathbf{L} &\rightarrow \mathbf{Set} \\ M(a \times b) &\cong Ma \times Mb \end{aligned}$$

注意，我们只要求积保持到同构。这一点非常重要，因为严格保持积会排除大多数有趣的理论。

模型保持积意味着 M 在 **Set** 中的像是由集合 $M1$ (\mathbf{L} 中 1 的像) 生成的一系列集合。我们称这个集合为 a 。在特别的情况下， \mathbf{L} 中的二元运算被映射到函数：

$$a \times a \rightarrow a$$

和任何函子一样， \mathbf{L} 中的多个态射可能会折叠为 **Set** 中的同一个函数。

顺便提一下，由于所有定律都是全称量化的等式，这意味着每个劳维尔理论都有一个平凡模型：一个将所有对象映射到单元素集合的常值函子，并将所有态射映射为该集合上的恒等函数。

在 \mathbf{L} 中，形式为 $m \rightarrow n$ 的一般态射被映射到一个函数：

$$a^m \rightarrow a^n$$

如果我们有两个不同的模型， M 和 N ，它们之间的自然变换是一个由 n 索引的函数族：

$$\mu_n :: Mn \rightarrow Nn$$

或者等价地：

$$\mu_n :: a^n \rightarrow b^n$$

其中 $b = N1$ 。

请注意，自然性条件保证了 n 元运算的保持：

$$Nf \circ \mu_n = \mu_1 \circ Mf$$

其中 $f :: n \rightarrow 1$ 是 \mathbf{L} 中的一个 n 元运算。

定义模型的函子形成了模型的一个范畴，**Mod(L, Set)**，其态射是自然变换。

考虑劳维尔范畴 \mathbf{F}^{op} 的模型。这样的模型完全由其在 1 处的值 $M1$ 决定。因为 $M1$ 可以是任意集合，所以有与 **Set** 中集合一样多的模型。此外，**Mod(F^{op}, Set)** 中的每个态射（即 M 和 N 之间的自然变换）都由其在 $M1$ 的分量唯一确定。反过来， $M1 \rightarrow N1$ 的每个函数都会在两个模型 M 和 N 之间诱导一个自然变换。因此，**Mod(F^{op}, Set)** 与 **Set** 是等价的。

30.4 么半群理论

The Theory of Monoids

最简单的非平凡劳维尔理论例子描述了么半群的结构。它是一个单一理论，提炼了所有可能么半群的结构，因为该理论的模型涵盖了整个 **Mon** 范畴的么半群。我们已经看到一个通用构造，它表明每个么半群都可以通过识别适当的自由么半群的态射来获得。因此，一个自由么

半群已经概括了许多幺半群。然而，自由幺半群有无数个。幺半群的劳维尔理论 $\mathbf{L}_{\mathbf{Mon}}$ 将它们组合成一个优雅的构造。

每个幺半群必须有一个单位，所以我们必须在 $\mathbf{L}_{\mathbf{Mon}}$ 中有一个从 0 到 1 的特殊态射 η 。注意，在 \mathbf{F} 中没有相应的态射。这样的态射会朝相反的方向，从 1 到 0，在 \mathbf{FinSet} 中，这将是一个从单元素集合到空集的函数。不存在这样的函数。

接下来，考虑 $\mathbf{L}_{\mathbf{Mon}}(2, 1)$ 中的态射 $2 \rightarrow 1$ ，它们必须包含所有二元运算的原型。在 $\mathbf{Mod}(\mathbf{L}_{\mathbf{Mon}}, \mathbf{Set})$ 中构建模型时，这些态射将被映射为从笛卡尔积 $M1 \times M1$ 到 $M1$ 的函数。换句话说，二元函数。

问题是：仅使用幺半群运算符可以实现多少个二元函数。我们将两个参数称为 a 和 b 。有一个函数忽略这两个参数并返回幺半群的单位元素。然后有两个投影函数，分别返回 a 和 b 。然后是返回 ab 、 ba 、 aa 、 bb 、 aab 等等的函数……事实上，存在与具有生成元 a 和 b 的自由幺半群的元素数量相同的二元函数。注意， $\mathbf{L}_{\mathbf{Mon}}(2, 1)$ 必须包含所有这些态射，因为自由幺半群是其中一个模型。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 $\mathbf{L}_{\mathbf{Mon}}(2, 1)$ 中的多个态射折叠为一个函数，但不是自由幺半群。

如果我们用 n 个生成元的自由幺半群表示为 n^* ，我们可以将 $\mathbf{L}(2, 1)$ 的态射集识别为 $\mathbf{Mon}(1^*, 2^*)$ 在 \mathbf{Mon} 范畴中的态射集。一般来说，我们选择 $\mathbf{L}_{\mathbf{Mon}}(m, n)$ 为 $\mathbf{Mon}(n^*, m^*)$ 。换句话说， $\mathbf{L}_{\mathbf{Mon}}$ 是自由幺半群范畴的对偶范畴。

幺半群劳维尔理论的模型的范畴

$\mathbf{Mod}(\mathbf{L}_{\mathbf{Mon}}, \mathbf{Set})$ 与所有幺半群的范畴 \mathbf{Mon} 是等价的。

30.5 劳维尔理论与幺半群

Lawvere Theories and Monads

你可能记得，代数理论可以使用幺半群来描述——特别是幺半群的代数。因此，不应感到惊讶，劳维尔理论与幺半群之间存在联系。

首先，让我们看看一个劳维尔理论如何诱导一个幺半群。它通过一个伴随来实现，在遗忘函子和自由函子之间。遗忘函子 U 将一个集合分配给每个模型。这个集合是通过在 \mathbf{L} 中对对象 1 求值的函子 M 给出的。

另一种导出 U 的方法是利用 \mathbf{F}^{op} 是 \mathbf{Law} 中的初对象这一事实。这意味着，对于任何劳维尔理论 \mathbf{L} ，存在一个唯一的函子 $\mathbf{F}^{op} \rightarrow \mathbf{L}$ 。这个函子在模型上诱导了一个相反的函子（因为模型是从理论到集合的函子）：

$$\mathbf{Mod}(\mathbf{L}, \mathbf{Set}) \rightarrow \mathbf{Mod}(\mathbf{F}^{op}, \mathbf{Set})$$

但是，正如我们讨论的， \mathbf{F}^{op} 的模型范畴等价于 \mathbf{Set} ，因此我们得到了遗忘函子：

$$U :: \mathbf{Mod}(\mathbf{L}, \mathbf{Set}) \rightarrow \mathbf{Set}$$

可以证明，如此定义的 U 始终有一个左伴随，即自由函子 F 。

对于有限集合，这很容易看出。自由函子 F 产生自由代数。自由代数是从有限生成元集 n 生成的 $\mathbf{Mod}(\mathbf{L}, \mathbf{Set})$ 中的特定模型。我们可以将 F 实现为可表函子：

$$\mathbf{L}(n, -) :: \mathbf{L} \rightarrow \mathbf{Set}$$

要证明它确实是自由的，我们需要做的只是证明它是遗忘函子的左伴随：

$$\mathbf{Mod}(\mathbf{L}(n, -), M) \cong \mathbf{Set}(n, U(M))$$

让我们简化右侧：

$$\mathbf{Set}(n, U(M)) \cong \mathbf{Set}(n, M1) \cong (M1)^n \cong Mn$$

(我使用了这样一个事实，即态射集同构于指数集，在这种情况下，这只是迭代积。) 伴随关系是 Yoneda 引理的结果：

$$[\mathbf{L}, \mathbf{Set}](\mathbf{L}(n, -), M) \cong Mn$$

因此，遗忘函子和自由函子一起定义了 \mathbf{Set} 上的幺半群 $T = U \circ F$ 。因此，每个劳维尔理论都生成一个幺半群。

事实证明，此幺半群的幺半群代数范畴与模型的范畴是等价的。

你可能还记得，幺半群代数定义了使用幺半群形成的表达式的评估方法。劳维尔理论定义了可用于生成表达式的 n 元运算。模型提供了评估这些表达式的方法。

劳维尔理论与幺半群之间的联系并不是双向的。只有有限的幺半群会导致劳维尔理论。有限幺半群基于有限函子。 \mathbf{Set} 上的有限函子完全由其在有限集上的作用决定。它在任意集合 a 上的作用可以使用以下共端来求值：

$$Fa = \int^n a^n \times (Fn)$$

由于共端推广了余积或和，因此该公式是幂级数展开的推广。或者我们可以使用函子是广义容器的直觉。在这种情况下， a 的有限容器可以描述为形状和内容的和。这里， Fn 是存储 n 个元素的形状的集合，内容是 n 元组的元素本身，是 a^n 的一个元素。例如，列表（作为函子）是有限的，每个元数只有一个形状。树在每个元数上有更多的形状，等等。

首先，从劳维尔理论生成的所有幺半群都是有限的，它们可以表示为共端：

$$T_{\mathbf{L}} a = \int^n a^n \times \mathbf{L}(n, 1)$$

反过来，给定 \mathbf{Set} 上的任意有限幺半群 T ，我们可以构造一个劳维尔理论。我们首先构造一个 T 的 Kleisli 范畴。正如你可能记得的，在 Kleisli 范畴中，从 a 到 b 的态射由基础范畴中的一个态射给出：

$$a \rightarrow Tb$$

当限制为有限集时，这变为：

$$m \rightarrow Tn$$

这个 Kleisli 范畴的对立范畴 \mathbf{Kl}_T^{op} 限制为有限集就是所讨论的劳维尔理论。特别地，描述 \mathbf{L} 中 n 元运算的态射集 $\mathbf{L}(n, 1)$ 由 $\mathbf{Kl}_T(1, n)$ 中的态射集给出。

事实证明，我们在编程中遇到的大多数幺半群都是有限的，唯一的例外是延续幺半群。可以扩展劳维尔理论的概念以超越有限元运算。

30.6 幺半群作为共端

Monads as Coends

让我们更详细地探索共端公式。

$$T_{\mathbf{L}} a = \int^n a^n \times \mathbf{L}(n, 1)$$

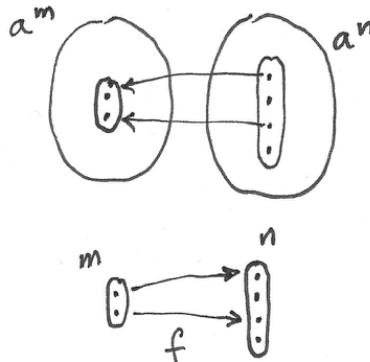
首先，这个共端是在 \mathbf{F} 中的一个伴随函子 P 上取的，定义为：

$$Pnm = a^n \times \mathbf{L}(m, 1)$$

这个伴随函子在第一个参数 n 中是反变的。考虑它如何提升态射。 \mathbf{FinSet} 中的一个态射是一个有限集 $f :: m \rightarrow n$ 的映射。这样的映射描述了从一个 n 元集合中选择 m 个元素（允许重复）。它可以提升为 a 幂的映射，注意方向：

$$a^n \rightarrow a^m$$

提升只是从 n 元组 (a_1, a_2, \dots, a_n) 中选择 m 个元素（可能有重复）。



例如，让我们取 $f_k :: 1 \rightarrow n$ ——从 n 元集合中选择第 k 个元素。它提升为一个函数，该函数接受 n 元组的 a 元素并返回第 k 个元素。

或者让我们取 $f :: m \rightarrow 1$ ——将所有 m 个元素映射为一个常值的函数。它的提升是一个函数，该函数接受 a 的单个元素并将其复制 m 次：

$$\lambda x \rightarrow \underbrace{(x, x, \dots, x)}_m$$

你可能注意到，立刻看出这个伴随函子在第二个参数中是协变的并不容易。同态函子 $\mathbf{L}(m, 1)$ 实际上在 m 上是反变的。然而，我们并不是在 \mathbf{L} 范畴中取共端，而是在 \mathbf{F} 范畴中。共端变量 n 遍历有限集（或其骨架）。 \mathbf{L} 包含 \mathbf{F} 的对偶，所以在 \mathbf{F} 中的态射 $m \rightarrow n$ 是 \mathbf{L} 中的 $\mathbf{L}(n, m)$ 的成员（嵌入由函子 \mathbf{L}_* 给出）。

让我们检查一下 $\mathbf{L}(m, 1)$ 作为从 \mathbf{F} 到 \mathbf{Set} 的函子的函子性。我们希望提升一个函数 $f :: m \rightarrow n$ ，因此我们的目标是从 $\mathbf{L}(m, 1)$ 到 $\mathbf{L}(n, 1)$ 实现一个函数。与函数 f 对应的 \mathbf{L} 中的态射是从 n 到 m （注意方向）。与 $\mathbf{L}(m, 1)$ 组合这个态射给我们 $\mathbf{L}(n, 1)$ 的一个子集。

$$\mathbf{L}(m, 1) \longrightarrow \mathbf{L}(n, 1)$$

$${}^m \bullet \xrightarrow[f]{} \bullet^n$$

注意，通过提升函数 $1 \rightarrow n$ 我们可以从 $\mathbf{L}(1, 1)$ 到 $\mathbf{L}(n, 1)$ 。我们稍后会用到这个事实。

一个反变函子 a^n 和一个协变函子 $\mathbf{L}(m, 1)$ 的乘积是一个伴随函子 $\mathbf{F}^{op} \times \mathbf{F} \rightarrow \mathbf{Set}$ 。记住，共端可以定义为伴随函子的所有对角成员的并集（不相交和），其中某些元素被识别。这些识别对应于共楔条件。

在这里，共端从对所有 n 的集合 $a^n \times \mathbf{L}(n, 1)$ 的不相交和开始。这些识别可以通过表达共端为共等化来生成。我们从 $a^n \times \mathbf{L}(m, 1)$ 的对角项开始。要达到对角线，我们可以将一个态射 $f :: m \rightarrow n$ 应用于积的第一个或第二个分量。然后将两个结果识别出来。

$$\begin{array}{ccc} & a^n \times \mathbf{L}(m, 1) & \\ \langle f, \mathbf{id} \rangle \swarrow & & \searrow \langle \mathbf{id}, f \rangle \\ a^m \times \mathbf{L}(m, 1) & \sim & a^n \times \mathbf{L}(n, 1) \end{array}$$

$$f :: m \rightarrow n$$

正如我之前展示的，提升 $f :: 1 \rightarrow n$ 生成以下两种变换：

$$a^n \rightarrow a$$

和：

$$L(1, 1) \rightarrow L(n, 1)$$

因此，从 $a^n \times L(1, 1)$ 开始，我们可以到达：

$$a \times L(1, 1)$$

当我们提升 $\langle f, \text{id} \rangle$ 时，以及：

$$a^n \times L(n, 1)$$

当我们提升 $\langle \text{id}, f \rangle$ 时。然而，这并不意味着 $a^n \times L(n, 1)$ 的所有元素都可以与 $a \times L(1, 1)$ 识别。这是因为 $L(n, 1)$ 的所有元素都不能从 $L(1, 1)$ 达到。请记住，我们只能提升来自 \mathbf{F} 的态射。 \mathbf{L} 中的非平凡 n 元运算不能通过提升 $f :: 1 \rightarrow n$ 构造出来。

换句话说，我们只能识别共端公式中的所有加法子集，其中 $L(n, 1)$ 可以通过应用基本态射从 $L(1, 1)$ 达到。它们都等价于 $a \times L(1, 1)$ 。基本态射是 \mathbf{F} 中态射的图像。

让我们看看这在最简单的劳维尔理论 \mathbf{F}^{op} 中是如何工作的。在这样的理论中，每个 $L(n, 1)$ 都可以从 $L(1, 1)$ 达到。这是因为 $L(1, 1)$ 是一个单元素集合，仅包含恒等态射，并且 $L(n, 1)$ 仅包含对应于 \mathbf{F} 中嵌入 $1 \rightarrow n$ 的态射，它们确实是基本态射。因此，余积中的所有加法项都是等价的，我们得到：

$$Ta = a \times L(1, 1) = a$$

这是恒等幺半群。

30.7 带有副作用的劳维尔理论

Lawvere Theory of Side Effects

由于幺半群与劳维尔理论之间有如此紧密的联系，自然会问劳维尔理论是否可以作为编程中幺半群的替代方案。幺半群的主要问题在于它们不能很好地组合。没有通用的构建幺半群变换器的配方。劳维尔理论在这方面有优势：它们可以使用余积和张量积来组合。另一方面，只有有限的幺半群可以轻松转换为劳维尔理论。这里的例外是延续幺半群。这个领域的研究正在进行中（参见参考书目）。

为了让你了解劳维尔理论如何用于描述副作用，我将讨论一个使用幺半群传统实现的简单异常情况。

幺半群由具有单个零元运算 $0 \rightarrow 1$ 的劳维尔理论生成。该理论的模型是一个函子，将 1 映射到某个集合 a ，并将零元运算映射为一个函数：

snippet02 我们可以使用共端公式恢复幺半群。让我们考虑零元运算的添加对态射集 $L(n, 1)$ 的影响。除了创建一个新的 $L(0, 1)$ （在 \mathbf{F}^{op} 中不存在），它还向 $L(n, 1)$ 添加了新的态射。这些态射是由 $n \rightarrow 0$ 的态射

与我们的 $0 \rightarrow 1$ 组合的结果。这些贡献在共端公式中与 $a^0 \times \mathbf{L}(0, 1)$ 相关，因为它们可以通过两种不同的方式从：

$$a^n \times \mathbf{L}(0, 1)$$

通过提升 $0 \rightarrow n$ 获得。

$$\begin{array}{ccc} & a^n \times \mathbf{L}(0, 1) & \\ \langle f, \text{id} \rangle \swarrow & \sim & \searrow \langle \text{id}, f \rangle \\ a^0 \times \mathbf{L}(0, 1) & & a^n \times \mathbf{L}(n, 1) \end{array}$$

$$f :: 0 \rightarrow n$$

共端简化为：

$$T_{\mathbf{L}} a = a^0 + a^1$$

或者，用 Haskell 表示：

snippet03 这相当于：

snippet04 请注意，此劳维尔理论仅支持引发异常，而不支持其处理。

30.8 挑战

Challenges

1. 列举 \mathbf{F} （有限集范畴的骨架）中 2 和 3 之间的所有态射。
2. 证明幺半群劳维尔理论的模型范畴等价于列表幺半群的幺半群代数范畴。
3. 幺半群劳维尔理论生成了列表幺半群。证明其二元运算可以通过相应的 Kleisli 箭头生成。
4. **FinSet** 是 **Set** 的一个子范畴，并且有一个函子将其嵌入 **Set**。**Set** 上的任何函子都可以限制在 **FinSet** 上。证明一个有限函子是其自身限制的左 Kan 延拓。

30.9 进一步阅读

Further Reading

1. Functorial Semantics of Algebraic Theories¹, F. William Lawvere

¹

2. Notions of computation determine monads², Gordon Plotkin and John Power

²

31

Monads, Monoids, and Categories

现 在已经没有一个完美的地方可以结束关于范畴论的书籍。总有更多的内容可以学习。范畴论是一个庞大的学科。同时，很明显的是，相同主题、概念和模式不断地反复出现。有一种说法认为，所有的概念都是 Kan 扩张（Kan extension），事实上，你可以用 Kan 扩张推导极限（limit）、余极限（colimit）、伴随（adjunction）、单子（monad）、Yoneda 引理（Yoneda lemma）以及更多内容。范畴的概念本身在所有抽象层次上都出现，单子和单子的概念也是如此。那么，哪一个是最基础的呢？结果表明，它们彼此关联，相互引导，在一个永无止境的抽象循环中相互作用。我决定展示这些相互联系，可能是结束这本书的一个好方法。

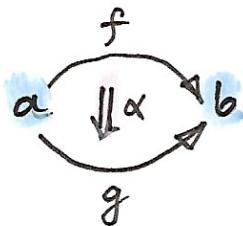
31.1 双范畴 (Bicategories)

范畴论最困难的方面之一就是不断转换视角。以集合范畴 (**Set**) 为例。我们习惯于用元素来定义集合。空集没有元素，单元素集有一个元素，两个集合的笛卡尔积（Cartesian product）是一个对的集合，等等。但是，当我们讨论集合范畴 (**Set**) 时，我让你忘记集合的内容，而是专注于它们之间的态射（箭头）。你可以偶尔窥视一下集合中的具体内容，以便理解范畴中的某个通用构造。终对象（terminal object）原来是一个只有一个元素的集合，等等。但这些只是一些验证检查。

函子（Functor）被定义为范畴之间的映射。自然地，可以将映射视为一个范畴中的态射。函子实际上是范畴范畴（category of categories）中的态射（箭头），如果我们希望避免关于大小的问题，应该说是小范畴（small categories）。通过将函子视为箭头，我们丢弃了关于它在范畴内部（对象和态射）作用的信息，就像我们在集合范畴中将函数视为箭头时丢弃了关于它对集合元素作用的信息一样。然而，两个范畴之间的

函子也构成一个范畴。这一次，我要求你考虑在一个范畴中作为箭头的东西在另一个范畴中作为对象。在函子范畴中，函子是对象，自然变换是态射。我们发现，同样的东西在一个范畴中是箭头，在另一个范畴中是对象。物体作为名词和箭头作为动词的朴素观点并不成立。

我们可以尝试将两种视角合并为一种，这样我们就得到了 2-范畴 (2-category) 的概念，在这种范畴中，对象称为 0-胞元 (0-cells)，态射称为 1-胞元 (1-cells)，态射之间的态射称为 2-胞元 (2-cells)。



0-胞元 a, b ; 1-胞元 f, g ; 以及 2-胞元 α 。

范畴的范畴 **Cat** 是一个直接的例子。我们有范畴作为 0-胞元，函子作为 1-胞元，自然变换作为 2-胞元。2-范畴的规则告诉我们，任何两个 0-胞元之间的 1-胞元构成一个范畴（换句话说， $\mathbf{C}(a, b)$ 是一个同态范畴 (hom-category)，而不是同态集 (hom-set)）。这与我们早先的断言相吻合，即任何两个范畴之间的函子构成一个函子范畴。

特别地，从任何 0-胞元回到自身的 1-胞元也构成一个范畴，即同态范畴 $\mathbf{C}(a, a)$ ，但这个范畴具有更丰富的结构。 $\mathbf{C}(a, a)$ 的成员可以看作是在范畴 **C** 中的箭头，或在 $\mathbf{C}(a, a)$ 中的对象。作为箭头，它们可以相互组合。但当我们将其视为对象时，组合就变成了从一对对象到一个对象的映射。事实上，它看起来非常像一个积——确切地说，是一个张量积 (tensor product)。这个张量积有一个单位元：即 1-胞元中的恒等元。事实证明，在任何 2-范畴中，同态范畴 $\mathbf{C}(a, a)$ 自动成为一个幺半范畴 (monoidal category)，其中张量积由 1-胞元的组合定义。结合律 (associativity) 和单位律 (unit laws) 简单地由相应的范畴法则推导出来。

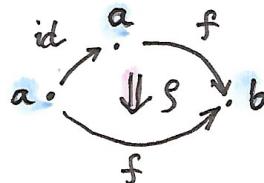
让我们看看这在我们的标准 2-范畴 **Cat** 的例子中意味着什么。 $\mathbf{Cat}(a, a)$ 的同态范畴是关于 A 的自函子 (endofunctor) 范畴，其中自函子组合作为它的张量积，恒等函子是与此张量积相关的单位元。我们之前已经看到，自函子形成一个幺半范畴（我们在定义单子 (monad) 时使用了这个事实），但现在我们看到，这是一个更普遍的现象：任何 2-范畴中的 endo-1-胞元形成一个幺半范畴。稍后我们会回到这一点，当我们推广单子时。

你可能还记得，在一般的幺半范畴中，我们没有坚持必须严格满足幺半群律 (monoid laws)。通常来说，只需在自然同构 (isomorphism)

下满足结合律（associativity）和单位律（unit laws）就可以了。在 2-范畴中， $C(a, a)$ 中的幺半群律直接由 1-胞元的组合法则推导出来。这些法则是严格的，所以我们将始终得到一个严格的幺半范畴。然而，也可以放宽这些法则。我们可以说，例如，恒等 1-胞元 id_a 与另一个 1-胞元 $f :: a \rightarrow b$ 的组合是同构的，而不是相等的。同构的 1-胞元通过 2-胞元定义。换句话说，有一个 2-胞元：

$$\rho :: f \circ \text{id}_a \rightarrow f$$

该 2-胞元具有逆元。



在双范畴中的恒等律在同构（一个可逆的 2-胞元 ρ ）下成立。

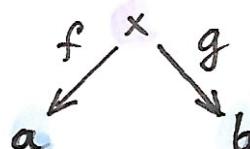
我们可以对左恒等律（left identity）和结合律（associativity laws）做同样的处理。这种放宽后的 2-范畴称为双范畴（bicategory）（还有一些附加的相干律，这里我将省略）。

正如预期的那样，双范畴中的 endo-1-胞元形成一个一般的幺半范畴，具有非严格的法则。

双范畴的一个有趣例子是跨范畴（category of spans）。两个对象 a 和 b 之间的跨（span）是一个对象 x 和一对态射：

$$f :: x \rightarrow a$$

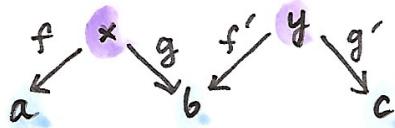
$$g :: x \rightarrow b$$



你可能还记得我们在定义笛卡尔积（categorical product）时使用了跨。在这里，我们希望将跨视为双范畴中的 1-胞元。第一步是定义跨的组合。假设我们有一个相邻的跨：

$$f' :: y \rightarrow b$$

$$g' :: y \rightarrow c$$



组合将是第三个跨，顶点是某个 z 。最自然的选择是 f' 沿 g 的拉回 (pullback)。记住，拉回是对象 z 以及两个态射：

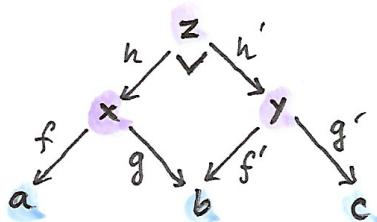
$$h :: z \rightarrow x$$

$$h' :: z \rightarrow y$$

使得：

$$g \circ h = f' \circ h'$$

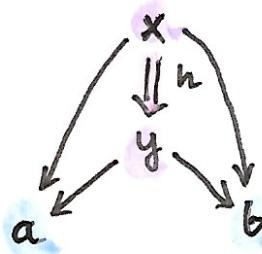
并且在所有这样的对象中是通用的。



目前，我们集中讨论集合范畴中的跨。在这种情况下，拉回只是笛卡尔积 $x \times y$ 中的对的集合 (p, q) ，使得：

$$g p = f' q$$

两个共享相同终点的跨之间的态射被定义为其顶点之间的一个态射 h ，使得适当的三角形交换。



Span 中的 2-胞元。

总结一下，在双范畴 **Span** 中：0-胞元是集合，1-胞元是跨，2-胞元是跨的态射。一个恒等 1-胞元是一个退化的跨，其中所有三个对象都是相同的，两个态射是恒等态射。

我们之前还看到了双范畴的另一个例子：**Prof** 双范畴，其中 0-胞元是范畴，1-胞元是函子，2-胞元是自然变换。函子的组合由余积 (coend) 给出。

31.2 单子 (Monads)

到目前为止，你应该对单子作为自函子范畴中的幺半群的定义非常熟悉了。让我们用新的理解重新审视这个定义，即自函子范畴只是双范畴 **Cat** 中的一个小的 **endo-1**-胞元范畴。我们知道它是一个幺半范畴：张量积来自于自函子的组合。幺半群被定义为幺半范畴中的一个对象——在这里它将是一个自函子 T ——连同两个态射。自函子之间的态射是自然变换。一个态射将幺半元，即恒等自函子，映射到 T ：

$$\eta :: I \rightarrow T$$

第二个态射将 $T \otimes T$ 的张量积映射到 T 。张量积由自函子组合给出，所以我们得到：

$$\mu :: T \circ T \rightarrow T$$

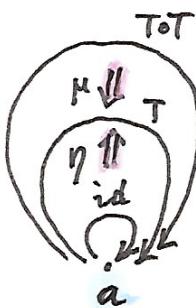
$$\begin{array}{ccc} & T \circ T & \\ & \downarrow \mu & \\ I & \xrightarrow{\eta} & T \end{array}$$

我们将这些视为定义单子的两个操作（在 Haskell 中，它们被称为 和），我们知道幺半群律转化为单子律。

现在让我们从这个定义中删除所有提到自函子的内容。我们从一个双范畴 \mathbf{C} 开始，并选择一个 0-胞元 a 。如前所述，同态范畴 $\mathbf{C}(a, a)$ 是一个幺半范畴。因此，我们可以通过选择一个 1-胞元 T 和两个 2-胞元来定义 $\mathbf{C}(a, a)$ 中的幺半群：

$$\begin{aligned}\eta &: I \rightarrow T \\ \mu &: T \circ T \rightarrow T\end{aligned}$$

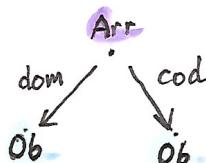
满足幺半群律。我们称之为单子。



这是一个更广义的单子的定义，仅使用 0-胞元、1-胞元和 2-胞元。在应用于双范畴 \mathbf{Cat} 时，它将归结为通常的单子。但是让我们看看在其他双范畴中会发生什么。

让我们在 \mathbf{Span} 中构造一个单子。我们选择一个 0-胞元，它是一个集合，出于将要解释的原因，我将其称为 Ob 。接下来，我们选择一个 endo-1-胞元：从 Ob 返回到 Ob 的一个跨。它在顶点处有一个集合，我将其称为 Ar ，它配备了两个函数：

$$\begin{aligned}dom &: Ar \rightarrow Ob \\ cod &: Ar \rightarrow Ob\end{aligned}$$

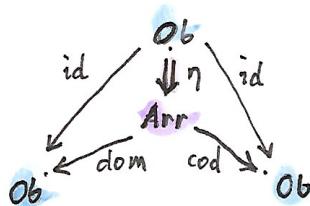


让我们称 Ar 集合中的元素为“箭头”。如果我还告诉你称 Ob 集合中的元素为“对象”，你可能会明白这指向哪里。两个函数 dom 和 cod 为每个“箭头”分配了一个“对象”的域和余域。

为了使我们的跨成为一个单子，我们需要两个 2-胞元 η 和 μ 。在这种情况下，么半元是一个从 Ob 到 Ob 的平凡跨，其顶点位于 Ob ，两个态射是恒等态射。2-胞元 η 是 Ob 和 Ar 之间的一个函数。换句话说， η 为每个“对象”分配了一个“箭头”。Span 中的 2-胞元必须满足交换条件——在这种情况下：

$$dom \circ \eta = \text{id}$$

$$cod \circ \eta = \text{id}$$



在组件中，这变为：

$$dom(\eta ob) = ob = cod(\eta ob)$$

其中 ob 是 Ob 中的一个“对象”。换句话说， η 为每个“对象”分配了一个“箭头”，其域和余域都是该“对象”。我们称这个特殊的“箭头”为“恒等箭头”。

第二个 2-胞元 μ 作用于跨 Ar 与自身的组合。组合定义为拉回 (pull-back)，因此其元素是 Ar 中的一对箭头 (a_1, a_2) ，其中满足拉回条件：

$$cod a_1 = dom a_2$$

我们说 a_1 和 a_2 是“可组合的”，因为一个的余域是另一个的域。

